Part II

End-to-End Internet Packet Dynamics

Chapter 9

Overview of the Packet Dynamics Study

In this part of our study we present our efforts to find convincing answers to questions about end-to-end Internet packet dynamics such as “how often are packets dropped?” As in Part I, we devise a large-scale measurement experiment based on the “Network Probe Daemon” (NPD) measurement framework. Our goal with this part of our study is to develop persuasive characterizations of the dynamics of Internet packet loss and delay. To do so, however, requires a great deal of groundwork in order to assure that the resulting findings are sound.

First, we need to calibrate our basic packet measurements, detecting those that are untrustworthy or inaccurate so that we can discard them to avoid drawing false conclusions. We describe how we do so in Chapter 10. Because we use TCP transfers as our basic “probes” for measuring network paths, our probes have a complicated structure due to the particulars of TCP. In Chapter 11 we discuss our development of an analysis tool, `tcpanaly`, that accounts for the details of the various TCPs in our study, and thus can separate their effects from true networking effects. The development of `tcpanaly` also gives us an opportunity to look at the differences in behavior between the TCP implementations in our study. These turn out to be quite significant, including some sufficiently broken TCPs that, if ubiquitously deployed, would devastate Internet performance due to *congestion collapse*.

Because one of our goals is to characterize one-way packet delays, we must also deal with the problem of calibrating the clocks used in our study. This proves much more difficult than we had originally anticipated. Chapter 12 details our efforts.

In Chapter 13 we turn to examining network “pathologies,” meaning unexpected network behavior. These include out-of-order delivery, in which packets arrive at the receiver in a different order than that in which they were sent; packet replication, in which the network delivers multiple copies of a single packet; and packet corruption, in which the data in the packet delivered by the network differs from that in the packet as originally sent.

In order to then soundly evaluate packet delay and loss, we need to first determine each connection's *bottleneck rate*, i.e., the upper bound imposed by the network path on the connection's throughput. This rate plays a crucial role because it determines when closely-spaced packets must necessarily queue behind each other in the network. Network conditions observed by such packets are *correlated* and must be treated separately from uncorrelated observations. In Chapter 14 we discuss shortcomings of the main existing technique for estimating bottleneck bandwidth, “packet pair,” and develop a robust algorithm, PBM (“packet bunch modes”), to address these problems.

In addition, we characterize the range of bottleneck rates we observed among the various Internet paths, and assess the stability of a path's bottleneck rate over time.

We then proceed in Chapter 15 to an analysis of patterns of Internet packet loss. We look at many different facets of loss, including the differences between loss rates of data packets and acknowledgements; correlations between loss rates along the two directions of a network path; trends in loss rates; differences in loss rates due to geography; the duration of loss "outages"; the location, with respect to the path's bottleneck element, where packet loss occurs; how well a connection's observed packet loss predicts those of future connections; and how well TCP deals with packet loss, in terms of retransmitting only when necessary.

We finish in Chapter 16 with an analysis of patterns of Internet packet delay. We look at variations and extremes of round-trip times (RTTs) and one-way transit times (OTTs); symmetry in OTT variation along the two directions of a network path; correlations between delay variations and loss; how well a connection's delay variations predict those of future connections; the phenomenon of packet timing "compression"; the time scales on which queueing occurs; and the degree of *available bandwidth* present along Internet paths.

Chapter 17 summarizes the findings of both Part I and Part II, and sketches the main themes of the work.

In the remainder of this chapter, we discuss our experimental methodology (§ 9.1); those aspects of the TCP protocol relevant to our study (§ 9.2); and the raw data produced by the experiment (§ 9.3).

9.1 Methodology

In this section we discuss the methodology underlying the packet dynamics experiment. We address two separate issues: how to make the measurements, and how to analyze them.

9.1.1 Measurement considerations

For our packet dynamics study, our measurement "probes" consisted of TCP transfers of 100 Kbyte files over different Internet paths. We discuss in § 9.1.2 the reasoning behind using TCP for the study. The transfers were *unidirectional*: data only flowed along one direction of the path. Such connections are referred to as *bulk transfers* [DJCME92, Pa94a]. There are other classes of traffic in the Internet (such as request/response, interactive, multicast, and real-time). All of these ultimately boil down to dividing data into packets for delivery by the Internet's packet forwarding infrastructure. Our goal is to characterize what happens to packets once they are in the hands of this infrastructure. For this purpose, bulk transfers serve well, as they provide a fairly steady stream of data packets traveling in one direction, and a corresponding stream of ack packets traveling in the other. We can then analyze the fate and timing of the packets to determine how the two directions of the Internet path performed.

Each transfer was traced using the `tcpdump` utility [JLM89] at both the sender and the receiver, resulting in two trace files. We term the combination of the two trace files a "trace pair." Our findings are all based on analyzing trace files and trace pairs.

For security reasons, the NPD transfers used fixed TCP sending and receiving "ports," so `tcpdump` could immediately filter out traffic not related to the transfer. That we did so has two drawbacks. First, it means that the traces lack some network traffic relevant to the transfer, namely any associated Internet Control Message Protocol (ICMP; [Po81b]) messages. We discuss in § 11.3.3 how we inferred the presence of a particular type of ICMP message, termed "source quench." In addition, using fixed ports resulted in our measurements incurring a *minimum separation* between consecutive measurements of the same pair of hosts, because TCP has rules governing how quickly a pair of ports can be reused for a new connection.[1]

As with the routing dynamics experiment, we used exponentially-spaced sampling intervals in order that our measurements might observe an unbiased sample of conditions along the different Internet paths (§ 4.3). We conducted two experimental runs, $\mathcal{N}_1$ and $\mathcal{N}_2$, detailed in § 9.3. For $\mathcal{N}_1$, source hosts were randomly paired with destination hosts, and we conducted a single measurement for each pairing. The drawback of this approach is that, if we want to study how an Internet path's characteristics change (or "evolve") over time, then random pairing results in widely-spaced measurements of individual pairs. For example, in $\mathcal{N}_1$ the mean sampling interval for a given pair was about two days. Consequently, we cannot analyze much finer time scales of evolution.

We addressed this difficulty in the second run, $\mathcal{N}_2$, by randomly pairing source and destinations into *groups* of measurements. Each measurement group consisted of two subgroups. Within a subgroup, we conducted six measurements, separated by 180 sec plus exponentially-distributed intervals with means 30 sec, 60 sec, 120 sec, 240 sec, and 480 sec.[2] These spacings allow us to analyze evolutions over short time intervals.

The two subgroups were then separated by an exponentially-distributed interval with mean 2 hours, allowing us to characterize evolution over medium time intervals. In addition, source/destination pairs would conduct additional groups of measurements separated from the previous group by another exponential interval with a mean of 12 hours. Finally, the pairs would be revisited on the order of a number of days later. These last two groups of measurements allow us to characterize relatively long time intervals, too.

9.1.2 Using TCP

Most previous end-to-end studies have used ICMP "ping" messages [Mi83, CPB93a] or User Datagram Protocol (UDP; [Po80]) "echo" messages for their network probes [Bo93].[3] Both have the considerable advantage of logistical ease: most Internet hosts readily reply to "ping" messages,[4] and activating the UDP echo service is often a one line configuration tweak.

However, these types of probes also incur disadvantages. The most significant of these is that of the *rate* at which the probes are sent. To probe fine time scales requires sending closely-spaced probe packets. Yet, if this is done blindly, say by deciding to send packets 1 msec apart, then depending on the mismatch between the sending rate and the capacity of the network path, the measurement traffic can grossly overload the path. Consequently, both "production" traffic sharing the path suffers, and the measurements are skewed by the abnormal loading. Unfortunately, there is a very wide range in network path capacity (we develop this claim in detail in Chapter 14 and

[1]Nominally, this minimum time is four minutes, twice the "maximum segment lifetime" of two minutes. In practice, it varies between TCP implementations.

[2]The 180 sec constant interval was required to avoid problems with reusing the fixed source and destination ports, discussed above.

[3]An exception is Mogul's study of TCP packet dynamics [Mo92].

[4]This is changing, with the advent of firewalls.

Chapter 16), so there is no *a priori* correct choice to use for the probe spacing.

Furthermore, capacity *changes* over the course of a series of probes, so we cannot determine a single correct choice for a path even after studying the path a bit. Therefore, ICMP- and UDP-based measurement must make a trade-off between possibly overloading the network path, and probing conservatively but with no possibility of analyzing finer time scales. In general, researchers have prudently chosen the latter.

One could devise a probing strategy based on *adapting* the probe transmission rate to the current network conditions. However, to do this properly, one essentially must implement TCP's congestion control. At this point, it becomes easier to just start with TCP in the first place!

Another drawback with echo-based techniques is that the echo services return a full copy of whatever packet they receive. Consequently, the measurement loads the network path both in the forward and the return direction. If the measurement is conducted using "sender-only" techniques (§ 9.1.3), then the reverse-path loading makes it impossible to determine which direction of the path is responsible for what proportion of the phenomena observed. If the echoes are instead *small,* such as are TCP acknowledgements for data packets, then the connection does not load the reverse path,[5] which lessens the conflation of the two directions.

Both of these considerations, particularly the first, argue favorably for using TCP transfers as network probes, since then, by construction, our probes do not load the network any more than does a routine file transfer. Using TCP has one other major advantage: TCP is very widely used. Consequently, the end-to-end performance observed by TCP transfers is a much closer match to the service Internet users actually obtain from the network than are echo-based techniques. We will also see in Chapter 11 that one result of our using TCP is to uncover a large variation in how different TCP implementations perform, some with major performance and congestion implications.

Using TCP, however, also brings with it some serious drawbacks. The first of these is that the TCP protocol behavior is quite complex. When casually inspecting TCP measurements, it can be difficult to determine which facets of the overall connection behavior were due to the state of the network path, and which were due to the behavior of the TCP implementations at the endpoints. If our goal is to characterize the network path, we *must* be able to separate these two, which entails understanding the nitty-gritty details of how different TCP implementations realize the protocol. To do so, for our study we developed a program, `tcpanaly`, which has knowledge of various TCP implementations and can analyze `tcpdump` traces in order to separate TCP endpoint effects from those due to the network path. Writing `tcpanaly` was a significant undertaking, much harder than we had initially anticipated (because we had not realized the wide range of real-world TCP behaviors). We discuss it in detail in Chapter 11.

The other major drawback with using TCP is that often it sends small groups of data packets at rates exceeding that of the network path's capacity (§ 9.2.5). These packets necessarily queue behind one another at the path's bottleneck. Therefore, for measuring the network's state such a group constitutes a *correlated* set of probes. We address this difficulty at length in Chapter 14.

[5]There is one way in which small packets can contribute to load along the reverse path similarly to large packets. If a congested router manages its buffers for queued packets on a *per-packet* basis, rather than allocating the number of bytes required to queue a packet out of a shared pool, then small packets consume the same amount of resource when queued at the router as do large packets. In this regard, small packets can push the congested router to the point of buffer overflow as fast as large packets do. Once, however, the small packets receive service, by transmission across their outbound link, then their contribution to the router's load immediately diminishes, since they require significantly less transmission time.

Furthermore, the TCP sender *adapts* the rate at which it transmits data packets based on previously observed network conditions (in particular, packet loss, per $\S$ 9.2.6). Thus, even when uncorrelated, the data packets do *not* reflect an unbiased measurement process, but rather one that changes its sampling rate in order to try to *minimize* observed packet loss. We discuss this property in Chapter 15.

On the other hand, for a TCP bulk transfer, both of these problems *only occur along the forward path*. The traffic along the reverse path is comprised entirely of small acknowledgement packets. These in general do *not* necessarily queue behind one another at the bottleneck, and, furthermore, their transmission rate is adapted not to conditions along the reverse path, which they observe, but to conditions along the forward path. We show in Chapter 15 that these conditions are generally uncorrelated. Thus, the “ack stream” along the reverse path reflects a much cleaner measurement process.

In summary, by using TCP transfers, we get two basic types of measurements: those that correspond to conditions that TCP data packets encounter (the forward path), and those that tell us about general Internet path properties (the reverse path ack stream). The combination makes for rich analysis.

9.1.3 Tracing at both sender and receiver

End-to-end measurement is often done using what we term “sender-based” or “sender-only” measurement, meaning that probes and their replies are recorded only at the location of the probe sender. Sender-based measurement has the enormous logistical advantage of not requiring access to the remote site in order to instrument the probe arrivals. Such access can be difficult to gain, for administrative and security reasons.

On the other hand, sender-based measurement carries with it the limitation that from it one can say little about how traffic behaves along the path's two different directions. For example, suppose a measurement consists of sending a flight of 20 ICMP “ping” packets from A to B, and timing at A the arrival of their echoes. If only 6 echoes return, we have no way of knowing whether B never sent the 14 others, because their corresponding pings never arrived at B; or if B did send them, but they were lost on their journey from B back to A; or if some combination of loss from A to B and loss from B to A occurred. Consequently, it is difficult to say anything concrete about the nature of the loss event.

This consideration becomes more subtle, but equally important, when applied to analyzing packet delay. A sender-based scheme can only observe round-trip time (RTT) delays. These are perforce the sum of the one-way transit time (OTT) delays in the two directions, plus the (unobserved) delay of the receiver generating its reply. If the goal of the timing measurement is to estimate capacity along the forward path, such as for TCP Vegas [BOP94], then any delay variations incurred on the return path are pure noise, and at best dilute the precision with which the sender can estimate the path capacity.

Because we traced our transfers at both the sender and the receiver, we can fully separate effects due to the forward path, the reverse path, and the processing delays at both the sender and the receiver. Throughout our study we examine issues of path symmetry with an eye to gauging the effectiveness of sender-only measurement. We find, overall, that such measurement is significantly less accurate than receiver-based measurement. Consequently, it behooves us to consider mechanisms for coordinating measurement between sender and receiver.

9.1.4 Analysis strategies

In this section we discuss the principles underlying our analysis of the measurement data. They are all in response to three dominant considerations. The first is that we gathered a very large volume of data: more than 20,000 transfers recorded at both sender and receiver. Each transfer consisted of 100–400 packets, resulting in well over a gigabyte of data. The second consideration is that we lack separate means of *calibrating* the measurements. All we have to work with are the packet traces. It is easy to *assume* that such traces accurately reflect the true number and timing of the packets comprising the traffic we wish to measure, but no large-scale study has been made to test the overall integrity of packet traces, so the validity of this assumption is unproven. The third consideration is that network behavior almost inevitably includes “noise” in a variety of forms and on a variety of scales. We observe “extreme” behavior much more often than we might expect using a traditional statistical framework (such as one based on assumptions of normality and tame correlations).

That we must deal with a large volume of data lies at the heart of our study: the study is interesting precisely because the volume of data is large. By (very careful) analysis of it, we have a hope of capturing a useful description of the immensely diverse behavior of the huge, heterogeneous network that is the Internet. We further argue that future Internet traffic studies must likewise measure on a large scale, otherwise we have little hope of divining from them general results. Thus, a central contribution of our work is the set of approaches we develop to deal with this large, uncalibrated, noisy mass of measurements.

In addition, in the hopes of abetting future studies, we will make our TCP data publicly available via the *Internet Traffic Archive*, sited at:[6]

```
http://www.acm.org/sigcomm/ITA
```

The routing data analyzed in Part I is already available in the Archive, under the name *NPD-Routes*.

Automated analysis

Confronted with 20,000 traces to analyze, it is clear that we cannot hope to individually analyze each trace. We must instead turn to *automated analysis*. That is, we realize part of our analysis in terms of a computer program that has coded into it the different reductions and calculations required by the analysis. We briefly mentioned this program, `tcpanaly`, above. One of its basic tasks is to separate TCP endpoint behavior from network behavior, hence its name. Another is to then characterize the network dynamics reflected in the trace of the connection.

`tcpanaly` undertakes what we might call “micro-analysis.” It is limited in its scope to analyzing single connections. The “macro-analysis,” namely the sifting through the individual micro-analyses in search of unifying observations and themes (much in the sense of “scientific inference,” as discussed in [Cha95]), is then done manually.[7] Both forms of analysis are highly iterative processes, and each gives insight into the other by identifying patterns that merit further investigation.

[6]At the time of this writing, the Archive is moving from its old location to this URL. If the reader has any difficulty accessing the Archive, send email to `vern@ee.lbl.gov`.

[7]We used the *S* statistical environment [BCW88] for the macro-analysis.

Self-consistency checks

To address the second problem—lack of separate calibration—we must turn to "self-calibration" in the form of *self-consistency* checks: testing, to as great a degree as possible, for any ways in which different aspects of the data contradict one another.

Calibration is all about detecting *error*, whether introduced by the measurement process, or by the subsequent analysis. Ideally, all of the effort is for naught; the data and analysis are wholly free of error. Consequently, it can sometimes be tempting to skip calibration or treat it lightly, since it only provides negative results. Doing so, however, undermines the entire validity of the measurement process. Furthermore, our experience in conducting both this study and several other large-scale studies [Pa94a, Pa94b, PF95] is that, when the scale becomes sufficiently large, errors are inevitable, since even rarely observed problems have sufficient opportunity to manifest themselves. Thus, we discuss self-consistency checks throughout our study. (For example, Chapter 12 is almost entirely about developing self-consistency checks for calibrating the timing measurements recorded in our traces.) The degree to which these checks prove persuasive is the degree to which one might accept our findings as well-grounded.

Robust statistics

The final problem we must address with our analysis strategies is that of widespread noise. For example, if we wish to summarize a connection's round trip times (RTTs), we might at first think to express them in terms of their sample *mean* and *variance* (or *standard deviation*, the square root of variance). However, in practice we find that often a connection observes one or two RTTs that are *much* higher than the remainder. These extreme values greatly *skew* the sample mean and variance, so that the resulting summaries do not accurately reflect "typical" behavior.

To address these sorts of problems, statisticians have developed the field of *robust statistics* [HMT83]. These are statistics that remain resilient in the presence of extremes, or "outliers." One example is use of the *median*, or 50th percentile, as a statistic for summarizing a distribution's central location, rather than the mean. Unlike the mean, the median is virtually unaffected by the presence of outliers.

In our study, we make heavy use of medians as robust estimates of central location. To compute a median of n points, $x_i = x_1, \ldots, x_n$, we sort the points to obtain $x_{(1)}, \ldots, x_{(n)}$, and then use:

$$\text{median}(x_i) = x_{(\frac{n+1}{2})},$$

if n is odd, or:

$$\text{median}(x_i) = \frac{1}{2}(x_{(\frac{n}{2})} + x_{(\frac{n+1}{2})}),$$

if n is even.

A robust statistic for measuring variation is the *interquartile range*, or IQR [Ri95]. The IQR is the difference between a distribution's 75th percentile and its 25th percentile. Thus, it characterizes the distribution's "central variation." It is likewise virtually unaffected by the presence of outliers, since these by definition fall outside of the range of the values used to compute the IQR. We likewise in our study often make use of IQR rather than standard deviation.

One other technique we borrow from robust statistics is that of fitting a line to a series of $\langle x, y \rangle$ points. Techniques such as least-squares can be heavily skewed by trying to minimize the distance between the fitted line and any outliers. The technique we use, taken from [HMT83], is to first estimate the slope of the line as the median of all of the pairwise slopes between the different points, and then estimate the intercept as the median of the offset of the y coordinates from a line with the given slope and zero-intercept.

9.2 An overview of TCP

In this section we give an overview of how the Internet's TCP protocol works. We make numerous references to its operation in subsequent chapters. Our presentation is not exhaustive, but confined to those aspects of TCP relevant to our later discussion.

The main protocol used in the Internet for reliable data delivery is the Transmission Control Protocol, or TCP. TCP is specified in [Po81c], with updates and clarifications given by [Br89], as well as several other documents specifying optional extensions [BJ88, BBJ92, Br94, MMFR96]. Stevens gives an excellent, detailed description of how TCP works [St94], and [WS95] analyzes an entire TCP implementation line-by-line.[8] TCP is implemented on top of the Internet Protocol, or IP, described in [Po81a]. The combination is often referred to as “TCP/IP.”

9.2.1 Data delivery goals

TCP is a complex protocol, since it was designed to accomplish a number of objectives:

- *In-order* delivery, meaning that data is presented to the receiving application in the same sequence as transmitted by the sending application.
- A *byte-stream* model, in which the sender and receiver view the data simply as a series of bytes, with no apparent boundary points (such as those introduced by packetization).
- *Reliable* data delivery, meaning that all of the data transmitted ultimately arrives at the receiver with its original contents (i.e., undamaged).

Accomplishing these objectives in an environment where packets can be delayed, dropped, reordered, duplicated, or corrupted is quite challenging. TCP achieves in-order, byte-stream data delivery by assigning each byte of data a *sequence number*, corresponding to its offset from the beginning of the byte stream. It does so efficiently by associating with each data packet a beginning sequence number (i.e., the sequence number of the first byte in the payload) and a length, which then gives the packet's upper sequence number. In subsequent discussion, we will adopt the convention of using upper sequence numbers to distinguish between different data packets. When this identification is not unique, we will also give the time at which the packet was sent or received, to disambiguate.

TCP achieves reliability by having the data receiver return *acknowledgements*, or “acks,” to the data sender.[9] Each ack includes an acknowledged sequence number, which indicates *all* of the in-order data that the receiver has successfully received. For example, if data packets with sequence numbers 1, 2, 3, 5, and 6 arrive at the receiver, then it can acknowledge up to sequence

[8]Both books also discuss other Internet protocols in depth.

[9]It also uses a 16-bit *checksum* to verify data integrity, a point we return to in § 11.4.2.

number 3. It cannot acknowledge 5 or 6, since they are not (yet) in-order. When the receiver subsequently receives sequence number 4, then it can acknowledge all the way up to 6. Such acks are termed “cumulative,” since receipt of any ack serves to acknowledge all of the data correctly received so far. [MMFR96] describes a TCP extension for “selective acks” (SACKs), which allow more detailed feedback of exactly which out-of-order packets have arrived at the receiver so far. In Chapters 13 and 15 we study some aspects of the efficacy of this extension, finding that it has considerable merits.

If a TCP sender does not eventually receive an ack for data it has sent, then it concludes that the data packet was lost (“dropped” or “discarded”) during its journey through the network, and it retransmits the data in a new packet. Such a retransmission is termed a “timeout retransmission,” because it occurs when a timer expires indicating that enough time has elapsed that the packet was presumably lost, since an ack should have been received by now. The amount of time to wait before retransmitting is termed the retransmission timeout (RTO). Choosing a good value for RTO is a major problem, which we discuss in more detail below. We discuss another form of retransmission in § 9.2.7.

9.2.2 Achieving high performance

Achieving these objectives would be considerably simpler if TCP did not have another goal, namely *performance*. Without performance considerations, one can achieve in-order, reliable byte-stream delivery by simply sending one packet at a time until the receiver acknowledges it, and then advancing to the next packet (“stop-and-go”). Stop-and-go can be tremendously inefficient in terms of the performance achieved. If packets are b bytes and the round-trip time (RTT; the interval between when a packet is sent and when the corresponding acknowledgement arrives) is ΔT seconds, then even if the network path is completely unloaded and does not suffer from any undue loss or delay, the maximum achieved throughput is:

$$\rho = \frac{b}{\Delta T}. \tag{9.1}$$

A typical value for b is 512 bytes, and a typical cross-country path in the U.S. has $\Delta T \approx 100$ msec, so $\rho = 5,120$ bytes/sec, even though the path might be capable of transferring megabytes per second.

TCP addresses performance issues in several ways. First, it sends packets that are as large as possible. Each Internet path has a Maximum Transmission Unit (MTU), which is the largest IP packet that can be transmitted along the path without incurring potentially expensive “fragmentation” into smaller packets. An end-to-end path's MTU is the minimum of the MTUs of the various links that comprise the path. When a connection is established between two TCP endpoints, they negotiate a Maximum Segment Size (MSS), which is the largest amount of data each TCP is prepared to receive in a single packet transmitted to it by the other TCP. In general, the MSS is less than the MTU, since the MTU must also include the overhead associated with each packet, namely its protocol header information.[10] Given these considerations, TCP implementations strive to transmit “full-sized” data packets, meaning those that carry MSS bytes of user data. They cannot always do so, if the sending application has not provided them with enough data to completely fill

[10]Some TCPs confuse MSS and MTU, as described in § 11.5.4.

a data packet. For our bulk-transfer connections, however, this is generally not a problem, and the TCPs usually sent full-sized data packets.

Using full-sized data packets helps increase b in Eqn 9.1, but never beyond MSS. Generally, MSS values are on the order of 512 bytes or sometimes 1460 bytes or 4 Kbytes, so this increase alone does not suffice for achieving good performance along a high-speed path. The much larger performance gain comes from having *multiple* packets in flight at one time. If a TCP has k packets in flight, then the potential throughput is:

$$\rho = \frac{kb}{\Delta T},$$

which can in principle match any available path speed ("bandwidth") by using a suitably large k.

The problem then becomes how to choose k. There are two separate concerns: how fast the receiver can accept data, and how fast the network can accept data. The first problem is referred to as "flow control," and the second as "congestion control."

TCP addresses flow control by including in the receiver's acks an "offered window" (also referred to as "advertised window", "receiver window," and, in some contexts, as simply the "window"). The offered window specifies how much new data the receiver promises to accept from the sender. It reflects the buffer available at the receiver, which is used to absorb discrepancies between the rate at which new data arrives at the receiver, and the rate at which the receiving application consumes that data from the receiving TCP. When the available buffer changes, the receiving TCP may send "window updates," which are acknowledgements with revised values for the offered window.

The offered window is expressed in terms of a "credit" beyond the packet acknowledged by the ack. For example, suppose the MSS is 512 bytes and the receiver has 4,096 bytes of buffer available. If data packets with sequence numbers 512, 1024, 2048, and 2560 arrive,[11] then the receiver can acknowledge up to 1024, since the first two packets arrived in sequence. It cannot acknowledge 2048 or 2560 because they arrived "above sequence." So far, the receiving application has not consumed any of the data, even though it could read the first 1,024 bytes if it wished. So the receiving TCP needs to hold the data from all four of the packets in its buffer. Because it has 4,096 bytes of buffer, it can accommodate additional data up to sequence 4096, so in the ack it includes an offered window of 4,096$-$1,024 $=$ 3,072 bytes, instructing the sender that it can accommodate 3,072 bytes beyond what is has now acknowledged receiving. Note that it advertises 3,072 bytes even though it has already committed 4 packets worth of buffer, or 2,048 bytes, leaving it with 2,048 bytes of uncommitted buffer. This works because the sender can only use the 3,072 byte credit as a window beyond the acknowledgement point ("ack point"). So it can only transmit 2,048 bytes' worth of data not already buffered, namely those corresponding to sequence numbers 1536, 3072, 3584, and 4096.

Suppose now the receiving application reads the 1,024 bytes that have been acknowledged. (It cannot yet read the data in the later packets, since they are presently "above sequence," so they cannot be read in-sequence yet.) Now the receiving TCP no longer needs to buffer the first two packets, so it can accommodate an additional 1,024 bytes from the sender. It may at this point send another acknowledgement for sequence 1024, but this time with an offered window of 4,096, allowing the sender to transmit all the way up to sequence $1024 + 4096 = 5120$.

[11] Where we are using the conventions that the sequence number refers to the upper sequence number carried in the packet, and that data packets are always full-sized.

When sequence 1536 arrives, then up to sequence 2560 may be ack'd, since the data up to there can now be delivered in-sequence. When the sender receives the ack of 2560, its window, meaning the range of data it can now send, “advances.” As part of this advance, the “upper edge” of the window, meaning the largest sequence number the sender can transmit (equal to the ack point plus the offered window), “slides” to a new maximum. Consequently, transport protocols using this form of flow control are termed “sliding window” protocols.

In this fashion, the receiving TCP can (if it wishes) assure that it is always able to accommodate data arriving from the sender. If, for example, the receiving application ceases to consume data, then eventually the TCP's buffer will fill. When it does, the TCP will advertise a window of 0 bytes, requiring the sender to cease transmission.

9.2.3 Congestion control

Quite separate from flow control is the vital performance issue of *congestion control*. The limitation on how fast the sender should transmit may derive not from limited buffer at the receiver, but limited capacity inside the network. Originally, TCP dealt with congestion control by setting the RTO (retransmission timeout) to a multiple of the estimated mean RTT (round-trip time). When the RTO expired, unacknowledged packets were retransmitted, and the RTO was doubled (“exponential backoff”), so that during periods of high congestion, the connection would progressively lower its sending rate.

In a landmark paper, Jacobson described the shortcomings of this form of congestion control: in particular, its excessive consumption of resources due to retransmitting multiple packets at one time, and the instability that occurs because it does so precisely when the network has been overloaded to the point of packet loss. He also identified inadequacies in the RTO algorithm, which used only the estimated mean RTT, without including an estimated RTT variance [Ja88]. He addressed these problems by introducing a second window, the *congestion window* (or, *cwnd*), and a modified RTO algorithm that includes the estimated RTT variance, both of which have been incorporated into the TCP specification [Br89, St97]. It is no exaggeration to say that the Internet works today only because of these changes. Without them, the network would inevitably devolve into “congestion collapse” (discussed below). Thus, *proper TCP congestion control is* vital *to the Internet's stability*, a point we return to in Chapter 11, where we find that some TCP implementations fail to follow these requirements.

We will focus on the first of Jacobson's changes, the congestion window. *cwnd* is completely separate from the receiver's offered window. At any time, a TCP sender must not send beyond the *minimum* of the two windows. The offered window governs how much in-flight data the receiver's buffer can accommodate, and *cwnd* governs how much the buffers along the network path can accommodate. The networking infrastructure, however, does not provide this latter information explicitly (nor can it, for scalability reasons). Jacobson's insightful observation was that the network path does, however, provide an *implicit* signal that its buffer resources are scarce: namely, it drops packets. Thus, packet loss is interpreted as a sign of congestion (also observed by Jain in [Jai89]). Such losses are termed “congestive losses.” While packet loss can occur for other reasons, the presumption is that most losses occur due to congestion, and so merit a response by the sending TCP: diminishing the rate at which it transmits packets. It does so by reducing *cwnd*.

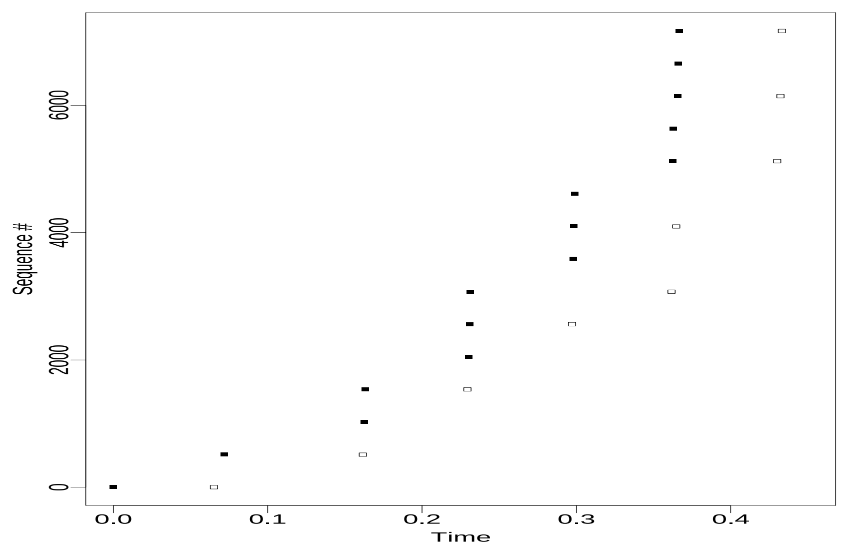

Figure 9.1: Sequence plot of a TCP connection during its “slow start” phase

9.2.4 Slow start

Jacobson discussed two different issues in managing *cwnd*. The first is what value to use for it initially, which we address in this section. The second is how it should be cut upon detecting loss, to adapt to congestion, which we address in § 9.2.6. His scheme addresses the first issue by initializing *cwnd* to one packet (more precisely, to MSS bytes), so connections begin by transmitting just one packet and waiting for an acknowledgement. Each ack that arrives then increases *cwnd* by one packet (again, actually by MSS bytes). Thus, if the receiving TCP sends an ack for every in-sequence packet it receives, then in the absence of loss the congestion window will be 1 packet, 2 packets, 4 packets, 8 packets, and so on, where each increase reflects *cwnd* after the packets in the previous “flight” have been acknowledged. (We use the term “flight” to refer to a set of packets transmitted within a single RTT's worth of time.) Thus, in the absence of loss, *cwnd* increases exponentially quickly. It continues to do so until either it is limited by the receiver's offered window; or the connection suffers a loss, indicating that a network-imposed congestive limit has been reached; or the connection completes before either of these occur.

This form of window increase is called “slow start,” since the window starts at a small value, and hence the TCP transmits slowly at first. Figure 9.1 shows a “sequence plot” of the packets sent and received by a TCP sender during its slow-start phase. We will make extensive use of such plots and so describe them here in detail. The x-axis gives time since the connection was established. The y-axis gives sequence numbers: these are either upper sequence numbers for data packets (shown as solid squares), or acknowledged sequence numbers for acks (hollow squares).

Sequence plots are highly informative illustrations of what happens during a connection. Here, the solid square at $T = 0$ sec with sequence number 1 corresponds to the “initial SYN” packet. Each connection begins with the originator transmitting a packet with the “SYN” flag set in the header to request establishment of a connection (“SYN” is short for “synchronize sequence numbers"). If the connection request is accepted, then the responder replies with a SYN acknowledgement ("SYN-ack") packet. If the sequence numbers in the SYN-ack accord with those that the originator sent, then the sender acknowledges the SYN-ack and the connection has been established. Because establishment entails exchanging three packets—the initial SYN, the SYN-ack, and the final ack of the SYN-ack—it is referred to as a "three-way" handshake.[12] TCP terminates connections in a similar fashion, using an exchange of "FIN" ("finish") packets and a final ack, for another three-way handshake.

The initial SYN carries a sequence number of 1 because the SYN flag conceptually occupies the first sequence position of the byte stream. At about $T = 0.07$ sec, the plot shows the arrival of an acknowledgement for sequence number 1. This is the SYN-ack packet. Shortly after, the sender begins transmitting. It sends a single packet carrying 512 bytes (and with sequence number 513), because *cwnd* has been set to one packet due to slow start. This data packet also carries the ack for the SYN-ack packet, and hence completes the three-way handshake. When 513 is ack'd at $T = 0.17$ sec, the congestion window opens to two packets, and these are promptly sent (sequence numbers 1025 and 1537). Both of these are acknowledged by a single ack at $T = 0.23$ sec, which opens *cwnd* by an additional packet, and three new packets are sent.[13]

At $T = 0.30$ sec, an ack arrives for the first two of the three packets in this flight. It opens the window to four packets. Since one of these four is already in flight, the TCP only transmits the three new ones. At $T = 0.36$ sec an ack arrives for the third packet of the earlier flight (sequence 3073). This is a *delayed* ack, one that the receiver momentarily refrained from sending in hope that more data would arrive and it could ack two packets at once. (The receiver employs an "ack-every-other" policy for sending its acknowledgements, as do many TCPs.) Very shortly after this ack arrives, so does another one, for sequence 4097, corresponding to the first two packets of the most recent flight. Each of these acks advances *cwnd* by one packet, so after both arrive, *cwnd* is 6 packets. One of these is already in flight (and not yet unacknowledged), so the TCP sends the other five.

Note that we can read the RTT directly from the plot: it is the x-axis distance between the transmission of a packet and its acknowledgement, in this case about 70 msec.

The sending TCP continues to open up *cwnd* until loss occurs. If *cwnd* reaches the point where it exceeds the size of the offered window, then the connection becomes *receiver-window limited*. Figure 9.2 shows a sequence plot of the same connection later in its transmission, when this has occurred. We have added circles to the plot indicating the upper "edge" of the window, that is, the sum of the offered window and the sequence number acknowledged by the ack (the "ack point"). We see that the sending TCP closely tracks the upper edge, sending packets up to that limit every time the edge advances.

9.2.5 Self-clocking

Another effect shown in Figure 9.2 is the important phenomenon of *self-clocking*. In the figure, each flight of data packets elicits in response an ack "echo" that preserves the temporal

[12]TCP uses a three-way handshake for reliability concerns that we will not describe further here; see [St94] for a detailed discussion.

[13]One might expect that *cwnd* would open by two additional packets, since the received ack acknowledges receipt of that many packets. However, the TCP standard governing congestion window management [St97] specifies that, during slow start, each ack for new data increases the window by one packet, regardless of how much new data has been ack'd.

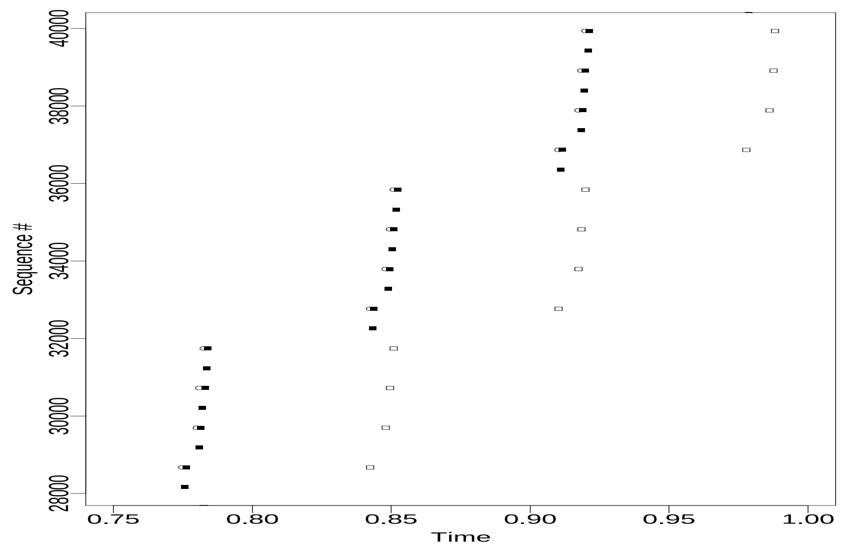

Figure 9.2: Sequence plot of a “window-limited” TCP connection

structure of the flight. When the flight of acks arrives at the sender, the window advances in step with the echo, because the receiving application is consuming the data as fast as it arrives, and hence the offered window remains constant—4,096 bytes, in this case. Because the connection is receiver-window limited, the sending TCP then transmits new data whose temporal structure reflects that of the window advances, and thus ultimately reflects that of the previous flight of data packets, so the cycle continues.

The term “self-clocking” is also due to Jacobson. It comes from the observation that a window-limited TCP connection will over time naturally pace out its data packets to exactly match the bandwidth available along the network path. Figure 9.3, reproduced from [Ja88], illustrates this property. The top “pipe” represents that network path from the sender to the receiver, and the bottom pipe that from the receiver back to the sender. The thickness of each component in a pipe reflects the bandwidth available at that part of the pipe, and horizontal distances correspond to differences in time. Each packet occupies a portion of the pipe, shown as a shaded region. The width of these regions indicates how long it takes the packet to traverse that portion of the pipe, and the height reflects that the packet consumes the region's available bandwidth during the traversal. In the figure, the sender has sent a number of packets back-to-back into the local, high-speed end of the path. These packets travel through the network closely spaced, until they reach the path's *bottleneck* (thin central region), where the available bandwidth sharply diminishes. At this point, it takes much more time to transmit each packet, so they spread out in time.

The key observation underlying self-clocking is that once the packets have been spaced out to a distance P_b by the bottleneck, they *remain* spaced out. That is, P_r in the figure is equal to P_b. There is no mechanism for subsequently recovering their initial spacing.[14] Furthermore, such recovery is not desirable: the distance P_b is in fact the optimal spacing for the connection's

[14]To first order. See § 16.3.

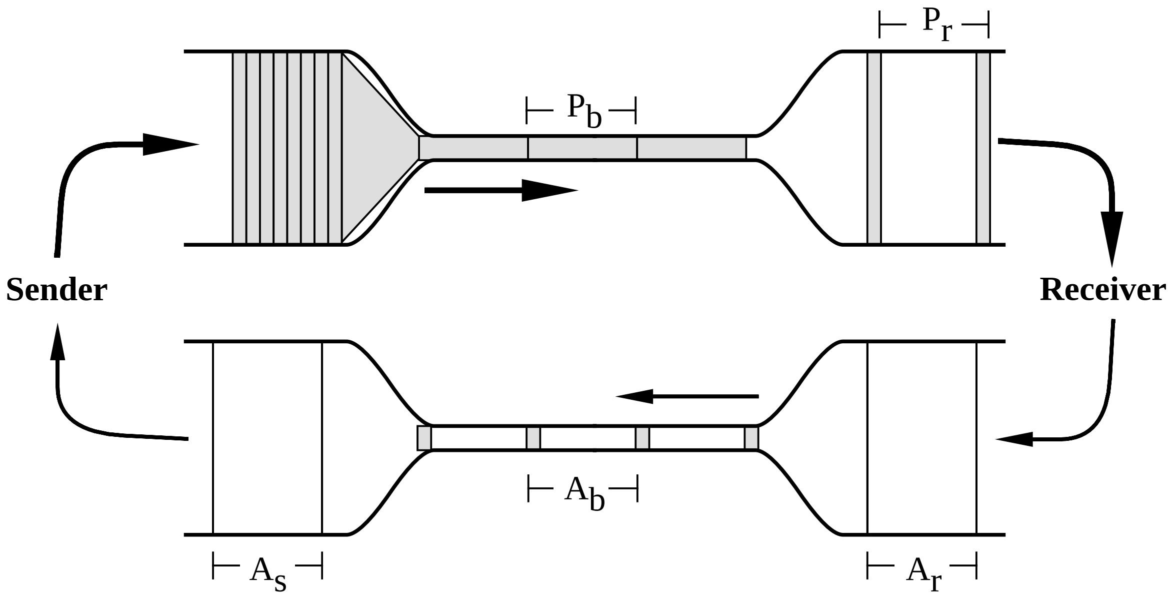

Figure 9.3: TCP "self-clocking." Reprinted with permission from [Ja88], copyright ©1988 Association for Computing Machinery.

packets. If they are transmitted any closer together, they will simply have to *wait* in a queue at the bottleneck anyway, because it cannot accommodate a faster rate. So we *want* packets ideally to be transmitted with a distance P_b between them. Any less and they cause queueing without any gain in performance. Any more and the connection underutilizes the available bandwidth.

The very nice property of window-based flow control is that, as the data packets arrive at the receiver with a spacing of $P_r = P_b$ between them, the receiver generates acks for them and these *also* have a spacing $A_r = P_r = P_b$ between them. Furthermore, the acks are *small* and are *not* spaced out by the bottleneck along the return path, even if it is smaller than that along the forward path. Consequently, the acks arrive at the sender with a spacing A_s between them, and because the timing has not been perturbed, $A_s = P_b$. Finally, these acks then advance the window, and the sender transmits new packets in response to them. The timing of the new packets, however, reflects that of the acks, and hence they have a spacing of P_b, just as we desire. Thus, the connection settles into a state in which it "clocks out" new packets at exactly the proper rate for the available bandwidth.

Receivers that employ ack-every-other policies, such as that shown in Figure 9.2, perturb self-clocking in only a minor fashion. Instead of generating acks with a spacing $A_r = P_b$ between them, they generate acks with a spacing of $A_r = 2P_b$. Consequently, in the absence of extra delays along the return path, they arrive at the sender with that same spacing. Note, however, that these acks advance the window by *two* packets each instead of one packet, so the sender then transmits two packets that are spaced a distance $P_r = 2P_b$ apart from the previous group of two packets. Thus, over intervals of $2P_b$, the connection transmits at exactly the bottleneck rate. On finer time scales, it can transmit faster, but the excess is only one additional packet, so very modest buffer space at the bottleneck can accommodate the burst. In § 11.6.1 we will see other acking policies that involve acking large numbers of packets with single acks. These can lead to highly bursty arrivals at the bottleneck.

Self-clocking is an idealized state. In practice, connections might not self-clock due to delay variations along the network paths in either direction, discussed in depth in Chapter 16. One particular form of delay variation that defeats self-clocking is *timing compression,* which we discuss in § 16.3.

Connections also do not self-clock when in the slow-start phase, since arriving acks do not simply advance the window, but widen it also. Consequently, the average spacing between the packets transmitted by the sender during slow start will be less than P_b. When the network is unloaded, this behavior is not only acceptable but desirable, because our goal is to *continually* send packets with intervals of P_b between them ("filling the pipe"). If we can accomplish this, then the connection sustains transmission at the full available bandwidth, and we have achieved the maximum performance possible along the given path. However, a TCP sender does not know in advance the proper value of P_b (and the proper value might change over the course of a connection). Slow start is a mechanism for *hunting* for the correct spacing, by continually opening up the window until the connection finds itself in the self-clocking regime corresponding to the currently available bandwidth. The difficulty with this hunt is knowing when to stop. TCP currently determines the stopping point when it has driven the network to the point of packet loss (see below). But this point corresponds to having exceeded the available bandwidth by a factor proportional to the available *buffer space*, too, since Internet routers today only drop packets when their buffers are exhausted. Thus, when beginning a connection, using slow start will often *drive the network to the point of loss,* which is excessive, since we instead want them to *drive the network to the limit of available throughput*.

There are proposals for modifying either TCP [WC91, WC92, BOP94] or the drop policy used by routers [RJ90, FJ93] so that connections can find the available bandwidth without unduly stressing the network. We comment on both approaches as we analyze our measurements. The TCP modifications are of particular interest for our study because they rely on accurate packet timing information, which we will find can be elusive.

In Figure 9.2, the connection fails to fill the pipe because it is receiver-window limited. In general, to fill the pipe requires that the window size in bytes, w, exceeds the "bandwidth-delay product," i.e.:

$$w \geq \rho_A \cdot \text{RTT}, \tag{9.2}$$

where ρ_A is the available bandwidth in bytes per second, and RTT is the round-trip time in seconds. We develop this relationship in detail in § 16.1. While estimating RTT is not difficult, estimating ρ_A is, so connections cannot readily use Eqn 9.2 to determine their correct window size. In Chapter 14 we discuss ways of estimating the *bottleneck* bandwidth, ρ_B, which is an upper bound on ρ_A, and in § 16.5 we look at the relationship between the two.

9.2.6 Responding to congestion

The other fundamental component of Jacobson's modifications to TCP concerns how TCP reacts to *congestion,* i.e., periods when some element of the end-to-end chain of routers and links is under stress: a sustained interval during which packets arrive more quickly than the element can service them. During congestion, the unserviced packets are queued at the congested router until they can be serviced. If the congestion lasts long enough, the queues build and build until eventually the queued packets exhaust the router's buffer. At this point, the router must discard incoming packets.

Note that congestion spans a *spectrum* between busy periods during which queues grow large, and periods when no more buffer remains and packets are lost. Thus, packets may or may not be lost during congestion periods, depending on the sizes of the buffers and the duration of the congestion. (We examine the interplay between delay and loss in § 16.2.4.)

Congestion is potentially *lethal* to a network because it can lead to positive feedback that sustains and even magnifies the congestion. In particular, if packet loss leads to retransmissions that are sent at the same rate as the original packets, then the load borne by the network will not diminish, and the congestion sustains itself. Packets from newly-initiated connections add further to the load, leading to even higher levels of congestion.

The positive feedback can thus bring the network to a state of *congestion collapse,* in which the network load stays extremely high but throughput is reduced to close to zero [Na84]. Exactly this happened in the early days of the Internet, and led to Jacobson's work on TCP congestion avoidance [Ja88]. As discussed above, one of the key insights of that work is that the network provides an implicit signal of congestion in the form of packet loss. Barring loss due to causes such as transmission noise on a network link, the network should only discard packets if it no longer has enough buffers to carry them. Consequently, when a TCP sender observes a packet loss, it should infer that the network path is congested, and ease its use of the path by cutting *cwnd* (and hence limiting its transmission rate). It does so as follows.

First, upon retransmitting, *cwnd* is set to one packet, so the connection begins a "slow start" phase in order to hunt for the correct value of the available bandwidth again. Second, the TCP state variable *ssthresh* ("slow start threshold") is set to half of the window in effect at the time of the retransmission (i.e., the smaller of either the offered window, or the value of *cwnd* prior to setting it down to one packet). The intent behind *ssthresh* is to denote the window size beyond which it is likely that there is no more available bandwidth. The sending TCP should only gingerly expand *cwnd* beyond *ssthresh*.

As acks arrive for packets now transmitted by the sender, each increases *cwnd* by one packet, per the usual slow-start increase. Once *cwnd* reaches *ssthresh*, however, then the TCP increases *cwnd* by only one packet per RTT. Thus, the rate at which *cwnd* increases changes from *exponential* during the slow-start phase to *linear* during the "congestion avoidance" phase.

Figure 9.4 illustrates how a TCP timeout retransmission appears on a sequence plot. At $T = 2.3$ sec, data up to 24577 has been acknowledged, and eight more packets are in flight, which equals the offered window. A little later two more acks for 24577 arrive ("duplicates," as discussed in § 9.2.7 below). However, no additional acks are forthcoming. At $T = 3.4$ sec, the RTO expires and the sending TCP retransmits the first unacknowledged packet, 25089. At this point, *cwnd* has been set to 1 packet (which is why only one packet is retransmitted), and *ssthresh* has been set to 4 packets, half of the window in effect at the time of the retransmission. The retransmitted packet elicits an ack for 28673, corresponding to all of the outstanding data. This indicates that only 25089 was dropped by the network—all of the later packets arrived at the receiver, so a retransmission of 25089 was all that was needed to fill the sequence "hole."

The ack for 28673 both advances the window edge and enlarges *cwnd* to 2 packets, so

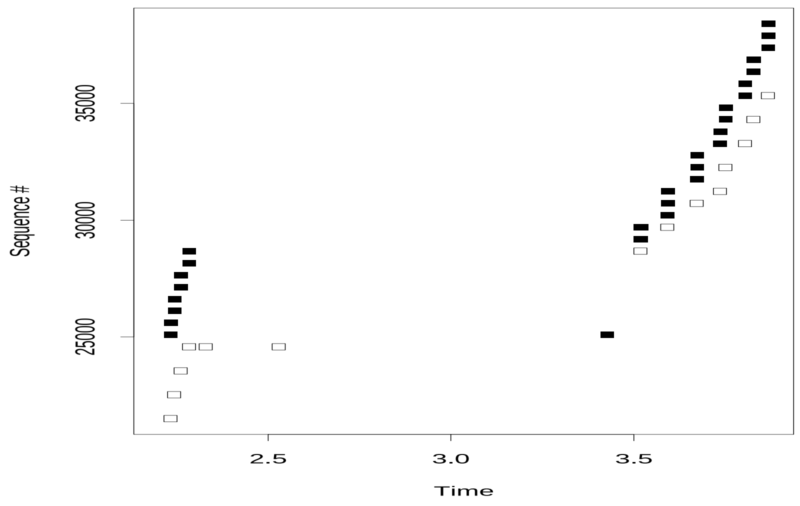

Figure 9.4: Sequence plot showing a TCP timeout retransmission

the sender now transmits two new packets. As acks arrive, the sender continues rapidly increasing *cwnd* in slow-start fashion. However, when the ack for 32257 arrives at $T = 3.75$ sec, *cwnd* does not increase from 5 packets to 6 packets, but remains at 5 packets, because the TCP has now entered congestion avoidance. It is only after the arrival of the ack for 35329, the last in the plot, that *cwnd* increases to six packets.

While the above outlines the congestion avoidance principles, in practice there are many fine points regarding exactly how congestive avoidance is implemented. (For example, why in Figure 9.4 it took more than one RTT during congestion avoidance for *cwnd* to increase by one packet.) We discuss a number of these in Chapter 11.

9.2.7 Fast retransmit and recovery

In addition to timeouts, TCP supports another retransmission mechanism, called “fast retransmit.” It is also due to Jacobson. Although not part of the TCP specification, it is widely implemented. Fast retransmission is an attempt to avoid the sometimes lengthy lulls a connection experiences upon a loss, due to the RTO being much larger than the RTT. Figure 9.4 above illustrates the problem. For this connection, the RTT was about 65 msec, but the RTO wait was 1.2 sec.

In general, RTO should be larger than the *maximum* RTT a connection's packets might experience, in order to allow enough time for acks to arrive. Yet, it is difficult to estimate this maximum due to frequent fluctuations in RTT, and, furthermore, it is important to estimate it conservatively, i.e., overestimate it rather than underestimate it, so that packets are not needlessly retransmitted. (We will see the effects of underestimation in § 11.5.10.) Finally, many TCP implementations have access to only coarse-grained clocks, so it is difficult for them to time small RTOs.

To address this problem, Jacobson noted that TCPs receive an additional, implicit signal when a packet has been lost. This signal comes in the form of the arrival of “duplicate acks.” When

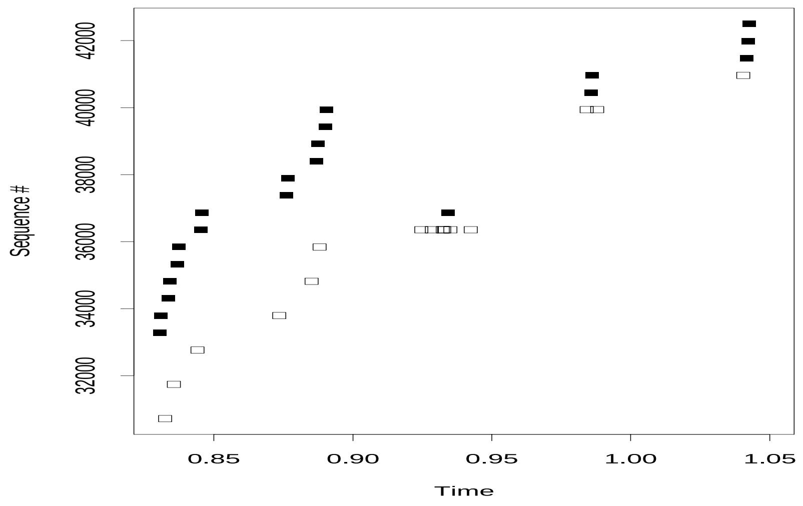

Figure 9.5: Sequence plot showing a TCP "fast retransmission"

above-sequence data arrives at a TCP receiver, the specification states that the TCP should generate a redundant acknowledgement.[15] These are termed "duplicate acks" or "dups." In Figure 9.4 we see two of these arriving at $T = 2.33$ sec and $T = 2.53$ sec. Since all but the first packet beyond the ack point arrived at the receiver, it should have sent 7 dups. From the plot, we cannot tell whether it did so but 5 were lost, or if it failed to do so. (It turns out that, in this case, it failed to do so. The TCP implementation was one that does not include the recommended generation of dups.)

Fast retransmission works by counting duplicate acks, and, if their number reaches a given threshold, N_d, then the sending TCP infers that the packet beyond the ack point was lost, and retransmits it. Current implementations use $N_d = 3$. This value was chosen as a trade-off between not missing fast retransmit opportunities because too few dups arrive, versus not misinterpreting the arrival of dups and retransmitting unnecessarily. The latter can occur when packets arrive out of order. In § 13.1.3 we examine how well $N_d = 3$ performs, and find that it does very well, almost always detecting true loss and not being fooled by reordering; and, further, that $N_d = 2$ would result in TCPs being fooled significantly more often.

Figure 9.5 shows a sequence plot depicting a fast retransmission. Packet 36865, originally transmitted at $T = 0.85$ sec, was lost, but all of its 6 successors arrived successfully. These then elicit six dups, the third of which causes a fast retransmission at $T = 0.93$ sec. At this point, *cwnd* is one packet and *ssthresh* is 4 packets. When the retransmission is ack'd at $T = 0.98$ sec, slow start advances *cwnd* to 2 packets, and then to 3 packets upon receipt of an ack for those two.[16]

Fast retransmit works very well for eliminating the lengthy timeout lull, provided enough above-sequence packets arrive at the receiver to elicit at least 3 dups. (If the receiver's offered

[15] It also does this if below-sequence data arrives, i.e., unnecessarily retransmitted data. We explore the distinction between these two in § 13.1.3.

[16] The apparent duplicate ack for 39937 is in fact a "window update," per § 9.2.2. TCPs are careful to distinguish between window updates and true duplicates, as the former do not indicate the safe arrival of an additional data packet.

window is small, or if *cwnd* is small, then this may be a problem.) Jacobson further refined it with a mechanism termed "fast recovery." The observation underlying fast recovery is that each additional dup beyond the first $N_d = 3$ indicates that another data packet has arrived at the receiver. Thus, it is sound to increase *cwnd* (which was cut to 1 packet upon the fast retransmission) by one packet for each of these, though not to exceed *ssthresh*. Furthermore, it is sound to increase *cwnd* by N_d packets upon a fast retransmission, too, because each of the first N_d dups likewise corresponds to a successfully received packet.

Thus, fast recovery opens *cwnd* more quickly. If this were all that the TCP did, then fast recovery would lead to a large burst when the TCP received an ack for the retransmitted packet ($T = 0.98$ sec in Figure 9.5), because at this point *cwnd* would often be much larger than 1 packet (then increased in Figure 9.5 to 2 packets by the arrival of the ack). To eliminate this burstiness, fast recovery also specifies that, if the TCP receives enough additional dups, it then begins transmitting *new* data, *before* it has received an acknowledgement for the retransmitted data. Thus, the algorithm looks like:

1. Upon receiving 3 dups, set *ssthresh* to half the effective window, set *cwnd* to one packet, and retransmit the first unacknowledged packet.

2. Next, "inflate" *cwnd* using:
$$\textit{cwnd} \leftarrow \textit{ssthresh} + 3,$$
where the constant 3 reflects the three duplicates already received.[17]

3. Whenever another dup arrives, increase *cwnd* by one more packet. If *cwnd* is now large enough to transmit new data, do so.

4. When an ack arrives that advances the ack point to or beyond the last packet that was in flight prior to the fast retransmission, then fast recovery ends. Execute:
$$\textit{cwnd} \leftarrow \textit{ssthresh}$$
to "deflate" the window to its proper post-recovery size, and update *cwnd* from the ack normally.

Figure 9.6 illustrates how fast recovery appears on a sequence plot. A number of dups arrive for 74573, which is retransmitted after the third dup is received (i.e., after four acks for 74573 arrive, the first being the "original" and the others being dups). Prior to the retransmission, the window was 8 packets, so after the retransmission *ssthresh* is 4 packets, and after window inflation, *cwnd* is $4 + 3 = 7$ packets. The next dup advances *cwnd* to 8 packets, but the TCP already has 8 packets' worth of data in flight, so it cannot retransmit at this point. The dup after that, though, arriving at $T = 7.94$ sec, advances *cwnd* to 9 packets, and this is enough to liberate a new data packet, 79361. Two more dups after it advance *cwnd* to 10 and 11 packets, and two more data packets are sent. Then, at $T = 7.96$ sec, all of the data outstanding prior to the retransmission is ack'd (closely followed by a window update, the second ack shown overlapping with the first). At

[17]We have simplified discussion by presenting the algorithms in terms of full-sized *packets*, when in fact they are implemented in terms of *bytes*. Provided all of the packets contain a full MSS' worth of data, these two are equivalent.

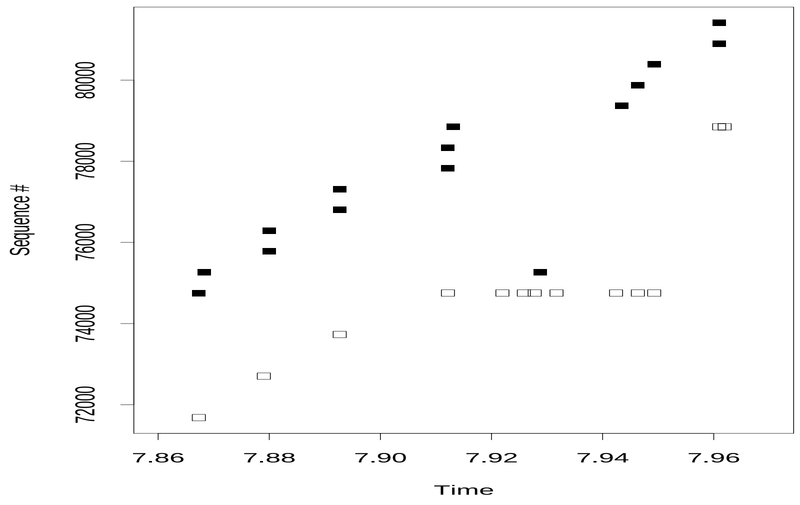

Figure 9.6: Sequence plot showing TCP “fast recovery”

this point, the window deflates back to *ssthresh*, or 4 packets. The ack is then processed, and since this TCP's test for congestion avoidance is

$$cwnd > ssthresh$$

rather than

$$cwnd \geq ssthresh$$

(used by some other TCPs), the connection is deemed still in slow start, so the ack advances *cwnd* to 5 packets. Three of these are already in flight, so the TCP transmits two new packets. Thus, the TCP was able to continue transmitting, and ended the retransmission period with *cwnd* having just entered congestion avoidance, and it did so without generating any unduly large bursts.

We make one final point regarding fast recovery. The window inflation and deflation is subtle (and often confusing). It arises due to conflating the meaning of *cwnd* to be both “how many packets the connection can have in flight” and “how far above the ack point can the connection transmit.” During fast recovery, these notions are separate, since some of the packets above the ack point are indeed no longer in flight (because they are what caused the dups). Because these points are subtle, we should not be too surprised to learn in Chapter 11 that TCPs implementing fast recovery suffer from more than one bug in managing the window deflation.

9.3 The Raw Measurements

Table XIV lists the 35 sites that participated in the two experimental runs, $\mathcal{N}_1$ and $\mathcal{N}_2$. Tables I and II in Part I summarize the sites.

We conducted the first run, $\mathcal{N}_1$, during December 1994, coincident with the routing study. Likewise, we conducted the second, $\mathcal{N}_2$, during November–December 1995. As with the routing

Name	# $\mathcal{N}_1$	# $\mathcal{N}_2$	Tracing machine
`adv`	–	1,244	
`austr`	207	1,036	BSDI 1.1
`austr2`	–	1,259	
`bnl`	307	1,200	
`bsdi`	166	1,374	
`connix`	308	1,474	
`harv`	190	1,061	
`inria`	172	1,180	
`korea`	49	–	HP/UX 9.01
`lbl`	318	1,412	SunOS/BPF
`lbli`	230	1,134	SunOS/BPF
`mid`	–	1,295	
`mit`	308	–	
`near`	–	1,296	SunOS/BPF
`nrao`	301	982	
`oce`	126	838	
`panix`	–	240	
`pubnix`	148	1,085	
`rain`	–	1,289	
`sandia`	–	1,182	
`sdsc`	259	964	
`sintef1`	–	1,469	NetBSD 1.0
`sintef2`	–	1,524	SunOS/BPF
`sri`	194	1,306	
`ucl`	230	1,266	
`ucla`	–	1,397	SunOS 4.1
`ucol`	275	1,208	SunOS 4.1 ($\mathcal{N}_1$)
`ukc`	299	989	SunOS 4.1
`umann`	222	998	
`umont`	144	1,469	SunOS 4.1
`unij`	74	1,412	SunOS 4.1
`usc`	231	–	SunOS 4.1
`ustutt`	240	1,165	
`wustl`	304	1,232	
`xor`	316	–	
Total	2,805	18,490	

Table XIV: Sites participating in the packet dynamics study

study, differences between $\mathcal{N}_1$ and $\mathcal{N}_2$ give us an opportunity to analyze how Internet packet dynamics changed during the course of 1995.

The second and third columns give the number of connections in which the site participated as either sender or receiver. The final column lists the operating system of the machine used to trace the site's TCP traffic, or empty, if the tracing was conducted on the same machine as ran the TCP. Tracing systems listed as "*XYZ*/BPF" had the Berkeley Packet Filter installed [MJ93], which greatly aids with accurate packet measurement. One site, `ucol`, changed its measurement setup between $\mathcal{N}_1$ and $\mathcal{N}_2$, using a separate machine during $\mathcal{N}_1$ but the same machine during $\mathcal{N}_2$.

As discussed above, each measurement was made by instructing the Network Probe Daemons (NPDs) running at two of the sites to send or receive a 100 Kbyte TCP bulk transfer, and to trace the results using `tcpdump`. An important difference between $\mathcal{N}_1$ and $\mathcal{N}_2$ is that in $\mathcal{N}_2$ we used Unix socket options to assure that the sending and receiving TCPs had sufficiently large buffers that they were never "window limited" (§ 9.2.4), to prevent window limitations from throttling the transfer's throughput. This change has a downside, which is that it sometimes clouds apparent trends between the $\mathcal{N}_1$ measurements and the $\mathcal{N}_2$ measurements with questions concerning whether the trends are simply artifacts of using bigger windows in $\mathcal{N}_2$. Nevertheless, the change was worth making, since the bigger windows enabled the $\mathcal{N}_2$ connections to push considerably harder on the network path, with more opportunities to observe the amount of resources the path had available as a result.

Finally, we limited measurements to a total of 10 minutes, as a mechanism to prevent measurement attempts from indefinitely consuming resources at the NPD sites. This limit leads to *under-representation* of those times during which network conditions were poor enough to make it difficult to complete a 100 Kbyte transfer in that much time. Thus, our measurements are *biased* towards more favorable network conditions. In § 15.1 we show that the bias was negligible for North American sites, but noticeable for European sites.