Building Blocks of TCP

NETWORKING 101, CHAPTER 2

At the heart of the Internet are two protocols, IP and TCP. The IP, or Internet Protocol, is what provides the host-to-host routing and addressing, and TCP, or Transmission Control Protocol, is what provides the abstraction of a reliable network running over an unreliable channel. TCP/IP is also commonly referred to as the Internet Protocol Suite and was first proposed by Vint Cerf and Bob Kahn in their 1974 paper titled "A Protocol for Packet Network Intercommunication."

The original proposal (RFC 675) was revised several times, and in 1981 the v4 specification of TCP/IP was published not as one, but as two separate RFCs:

- RFC 791 — Internet Protocol
- RFC 793 — Transmission Control Protocol

Since then, there have been a number of enhancements proposed and made to TCP, but the core operation has not changed significantly. TCP quickly replaced previous protocols and is now the protocol of choice for many of the most popular applications: World Wide Web, email, file transfers, and many others.

TCP provides an effective abstraction of a reliable network running over an unreliable channel, hiding most of the complexity of network communication from our applications: retransmission of lost data, in-order delivery, congestion control and avoidance, data integrity, and more. When you work with a TCP stream, you are guaranteed that all bytes sent will be identical with bytes received and that they will arrive in the same order to the client. As such, TCP is optimized for accurate delivery, rather than a timely one. This, as it turns out, also creates some challenges when it comes to optimizing for web performance in the browser.

The HTTP standard does not mandate TCP as the only transport protocol. If we wanted, we could deliver HTTP via a datagram socket (User Datagram Protocol or UDP), or any other transport protocol of our choice, but in practice all HTTP traffic on the Internet today is delivered via TCP due to the many great and convenient features it provides out of the box.

Because of this, understanding some of the core mechanisms of TCP is essential knowledge for building an optimized web experience. Chances are you won't be working with TCP sockets directly in your application, but the design choices you make at the application layer will dictate the performance of TCP and the underlying network over which your application is delivered.

Intertwined History of TCP and IP Protocols

We are all familiar with IPv4 and IPv6, but what happened to IPv{1,2,3,5}? The 4 in IPv4 stands for the version 4 of the TCP/IP protocol, which was published in September 1981. The original TCP/IP proposal coupled the two protocols, and it was the v4 draft that officially split the two into separate RFCs. Hence, the v4 in IPv4 is a heritage of its relationship to TCP: there are no standalone IPv1, IPv2, or IPv3 protocols.

When the working group began work on "Internet Protocol next generation" (IPng) in 1994, a new version number was needed, but v5 was already assigned to another experimental protocol: Internet Stream Protocol (ST). As it turns out, ST never took off, which is why few ever heard of it. Hence the 6 in IPv6.

Three-Way Handshake §

All TCP connections begin with a three-way handshake (Figure 2-1). Before the client or the server can exchange any application data, they must agree on starting packet sequence numbers, as well as a number of other connection specific variables, from both sides. The sequence numbers are picked randomly from both sides for security reasons.

SYN

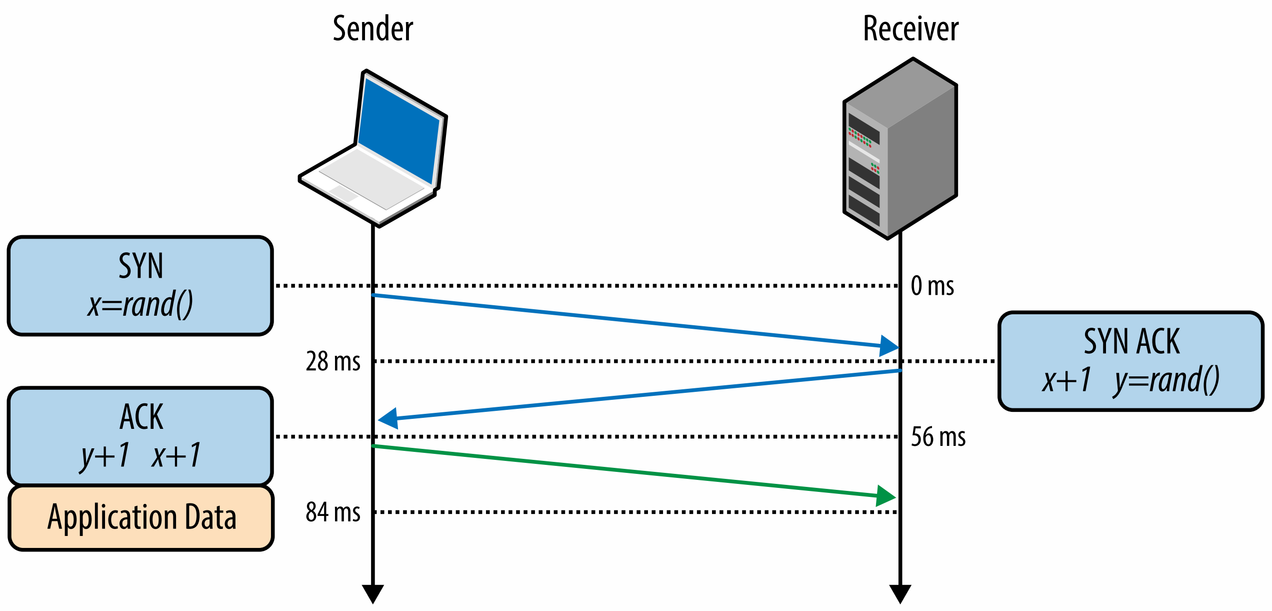
Client picks a random sequence number x and sends a SYN packet, which may also include additional TCP flags and options.

SYN ACK

Server increments x by one, picks own random sequence number y, appends its own set of flags and options, and dispatches the response.

ACK

Client increments both x and y by one and completes the handshake by dispatching the last ACK packet in the handshake.



Once the three-way handshake is complete, the application data can begin to flow between the client and the server. The client can send a data packet immediately after the ACK packet, and the server must wait for the ACK before it can dispatch any data. This startup process applies to every TCP connection and carries an important implication for performance of all network applications using TCP: each new connection will have a full roundtrip of latency before any application data can be transferred.

For example, if our client is in New York, the server is in London, and we are starting a new TCP connection over a fiber link, then the three-way handshake will take a minimum of 56 milliseconds (Table 1-1): 28 milliseconds to propagate the packet in one direction, after which it must return back to New York. Note that bandwidth of the connection plays no role here. Instead, the delay is governed by the latency between the client and the server, which in turn is dominated by the propagation time between New York and London.

The delay imposed by the three-way handshake makes new TCP connections expensive to create, and is one of the big reasons why connection reuse is a critical optimization for any application running over TCP.

TCP Fast Open §

Loading a webpage often requires fetching hundreds of resources from dozens of different hosts. In turn, this might require the browser to establish dozens of new TCP connections, each of which will have to incur the overhead of the TCP handshake. Needless to say, this can be a significant source of web browsing latency, especially on slower mobile networks.

TCP Fast Open (TFO) is a mechanism that aims to eliminate the latency penalty imposed on new TCP connections by allowing data transfer within the SYN packet. However, it does have its own set of limitations: there are limits on the maximum size of the data payload within the SYN packet, only certain types of HTTP requests can be sent, and it works only for repeat connections due to a requirement for a cryptographic cookie. For a detailed discussion on the capabilities and limitations of TFO, check the latest IETF draft of "TCP Fast Open."

Enabling TFO requires explicit support on the client, server, and opt-in from the application. For best results, use the Linux kernel v4.1+ on the server, a compatible client (e.g. Linux, or iOS9+ / OSX 10.11+), and enable the appropriate socket flags within your application.

Based on traffic analysis and network emulation done at Google, researchers have shown that TFO can decrease HTTP transaction network latency by 15%, whole-page load times by over 10% on average, and in some cases by up to 40% in high-latency scenarios!

Congestion Avoidance and Control §

In early 1984, John Nagle documented a condition known as "congestion collapse," which could

> *Congestion control is a recognized problem in complex networks. We have discovered that the Department of Defense's Internet Protocol (IP), a pure datagram protocol, and Transmission Control Protocol (TCP), a transport layer protocol, when used together, are subject to unusual congestion problems caused by interactions between the transport and datagram layers. In particular, IP gateways are vulnerable to a phenomenon we call "congestion collapse", especially when such gateways connect networks of widely different bandwidth…*
>
> *Should the roundtrip time exceed the maximum retransmission interval for any host, that host will begin to introduce more and more copies of the same datagrams into the net. The network is now in serious trouble. Eventually all available buffers in the switching nodes will be full and packets must be dropped. The roundtrip time for packets that are delivered is now at its maximum. Hosts are sending each packet several times, and eventually some copy of each packet arrives at its destination. This is congestion collapse.*
>
> *This condition is stable. Once the saturation point has been reached, if the algorithm for selecting packets to be dropped is fair, the network will continue to operate in a degraded condition.*
>
> *John Nagle, RFC 896*

The report concluded that congestion collapse had not yet become a problem for ARPANET because most nodes had uniform bandwidth, and the backbone had substantial excess capacity. However, neither of these assertions held true for long. In 1986, as the number (5,000+) and the variety of nodes on the network grew, a series of congestion collapse incidents swept throughout the network — in some cases the capacity dropped by a factor of 1,000 and the network became unusable.

To address these issues, multiple mechanisms were implemented in TCP to govern the rate with which the data can be sent in both directions: flow control, congestion control, and congestion avoidance.

Note

Advanced Research Projects Agency Network (ARPANET) was the precursor to the modern Internet and the world's first operational packet-switched network. The project was officially launched in 1969, and in 1983 the TCP/IP protocols replaced the earlier NCP (Network Control Program) as the principal communication protocols. The rest, as they say, is history.

Flow Control §

Flow control is a mechanism to prevent the sender from overwhelming the receiver with data it to allocate a fixed amount of buffer space. To address this, each side of the TCP connection advertises (Figure 2-2) its own receive window (rwnd), which communicates the size of the available buffer space to hold the incoming data.

When the connection is first established, both sides initiate their rwnd values by using their system default settings. A typical web page will stream the majority of the data from the server to the client, making the client's window the likely bottleneck. However, if a client is streaming large amounts of data to the server, such as in the case of an image or a video upload, then the server receive window may become the limiting factor.

If, for any reason, one of the sides is not able to keep up, then it can advertise a smaller window to the sender. If the window reaches zero, then it is treated as a signal that no more data should be sent until the existing data in the buffer has been cleared by the application layer. This workflow continues throughout the lifetime of every TCP connection: each ACK packet carries the latest rwnd value for each side, allowing both sides to dynamically adjust the data flow rate to the capacity and processing speed of the sender and receiver.

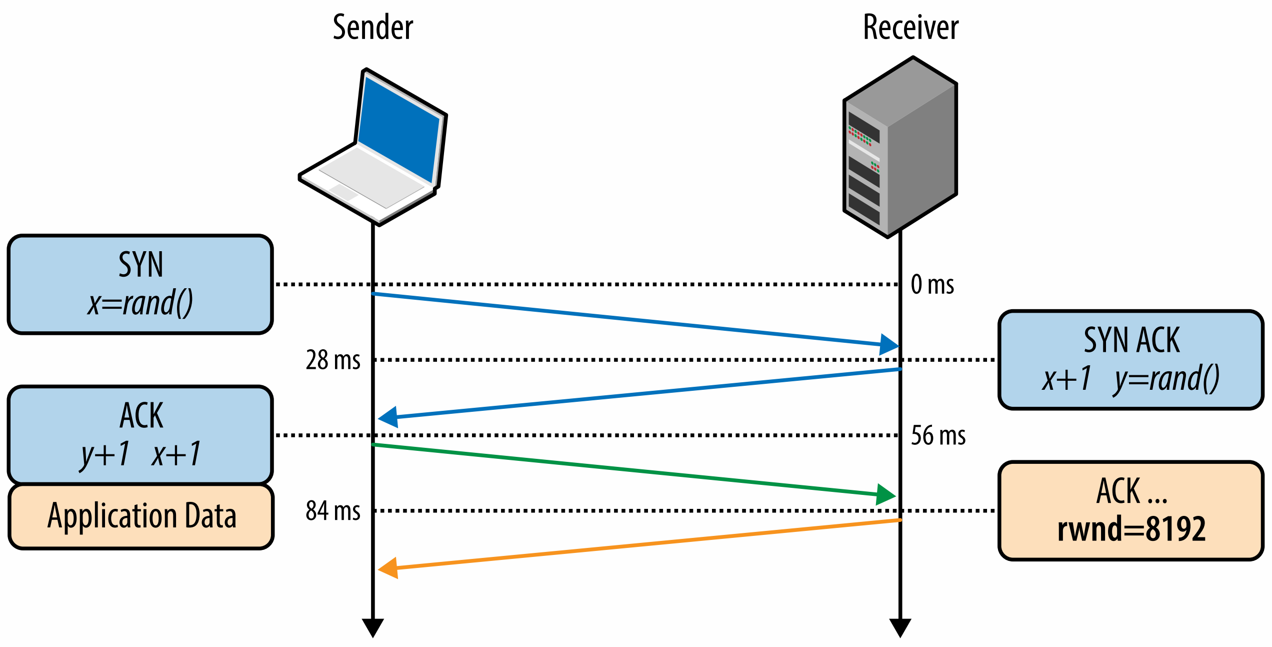


Figure 2-2. Receive window (rwnd) size advertisement

Window Scaling (RFC 1323)

The original TCP specification allocated 16 bits for advertising the receive window size, which places a hard upper bound on the maximum value (2^{16}, or 65,535 bytes) that can be advertised by the sender and receiver. Turns out, this upper bound is often not enough to get optimal performance, especially in networks that exhibit high bandwidth delay product; more on this can be found Bandwidth-Delay Product.

To address this, RFC 1323 was drafted to provide a "TCP window scaling" option, which allows us to raise the maximum receive window size from 65,535 bytes to 1 gigabyte! The window scaling

Today, TCP window scaling is enabled by default on all major platforms. However, intermediate nodes, routers, and firewalls can rewrite or even strip this option entirely. If your connection to the server, or the client, is unable to make full use of the available bandwidth, then checking the interaction of your window sizes is always a good place to start. On Linux platforms, the window scaling setting can be checked and enabled via the following commands:

- `$> sysctl net.ipv4.tcp_window_scaling`
- `$> sysctl -w net.ipv4.tcp_window_scaling=1`

Slow-Start §

Despite the presence of flow control in TCP, network congestion collapse became a real issue in the mid to late 1980s. The problem was that flow control prevented the sender from overwhelming the receiver, but there was no mechanism to prevent either side from overwhelming the underlying network: neither the sender nor the receiver knows the available bandwidth at the beginning of a new connection, and hence need a mechanism to estimate it and also to adapt their speeds to the continuously changing conditions within the network.

To illustrate one example where such an adaptation is beneficial, imagine you are at home and streaming a large video from a remote server that managed to saturate your downlink to deliver the maximum quality experience. Then another user on your home network opens a new connection to download some software updates. All of the sudden, the amount of available downlink bandwidth to the video stream is much less, and the video server must adjust its data rate — otherwise, if it continues at the same rate, the data will simply pile up at some intermediate gateway and packets will be dropped, leading to inefficient use of the network.

In 1988, Van Jacobson and Michael J. Karels documented several algorithms to address these problems: slow-start, congestion avoidance, fast retransmit, and fast recovery. All four quickly became a mandatory part of the TCP specification. In fact, it is widely held that it was these updates to TCP that prevented an Internet meltdown in the '80s and the early '90s as the traffic continued to grow at an exponential rate.

To understand slow-start, it is best to see it in action. So, once again, let us come back to our client, who is located in New York, attempting to retrieve a file from a server in London. First, the three-way handshake is performed, during which both sides advertise their respective receive window (rwnd) sizes within the ACK packets (Figure 2-2). Once the final ACK packet is put on the wire, we can start exchanging application data.

The only way to estimate the available capacity between the client and the server is to measure it by exchanging data, and this is precisely what slow-start is designed to do. To start, the server initializes a new congestion window (cwnd) variable per TCP connection and sets its initial value to a conservative, system-specified value (initcwnd on Linux).

Sender-side limit on the amount of data the sender can have in flight before receiving an acknowledgment (ACK) from the client.

The cwnd variable is not advertised or exchanged between the sender and receiver — in this case, it will be a private variable maintained by the server in London. Further, a new rule is introduced: the maximum amount of data in flight (not ACKed) between the client and the server is the minimum of the rwnd and cwnd variables. So far so good, but how do the server and the client determine optimal values for their congestion window sizes? After all, network conditions vary all the time, even between the same two network nodes, as we saw in the earlier example, and it would be great if we could use the algorithm without having to hand-tune the window sizes for each connection.

The solution is to start slow and to grow the window size as the packets are acknowledged: slow-start! Originally, the cwnd start value was set to 1 network segment; RFC 2581 updated this value to 4 segments in April 1999; most recently the value was increased once more to 10 segments by RFC 6928 in April 2013.

The maximum amount of data in flight for a new TCP connection is the minimum of the rwnd and cwnd values; hence a modern server can send up to ten network segments to the client, at which point it must stop and wait for an acknowledgment. Then, for every received ACK, the slow-start algorithm indicates that the server can increment its cwnd window size by one segment — for every ACKed packet, two new packets can be sent. This phase of the TCP connection is commonly known as the "exponential growth" algorithm (Figure 2-3), as the client and the server are trying to quickly converge on the available bandwidth on the network path between them.

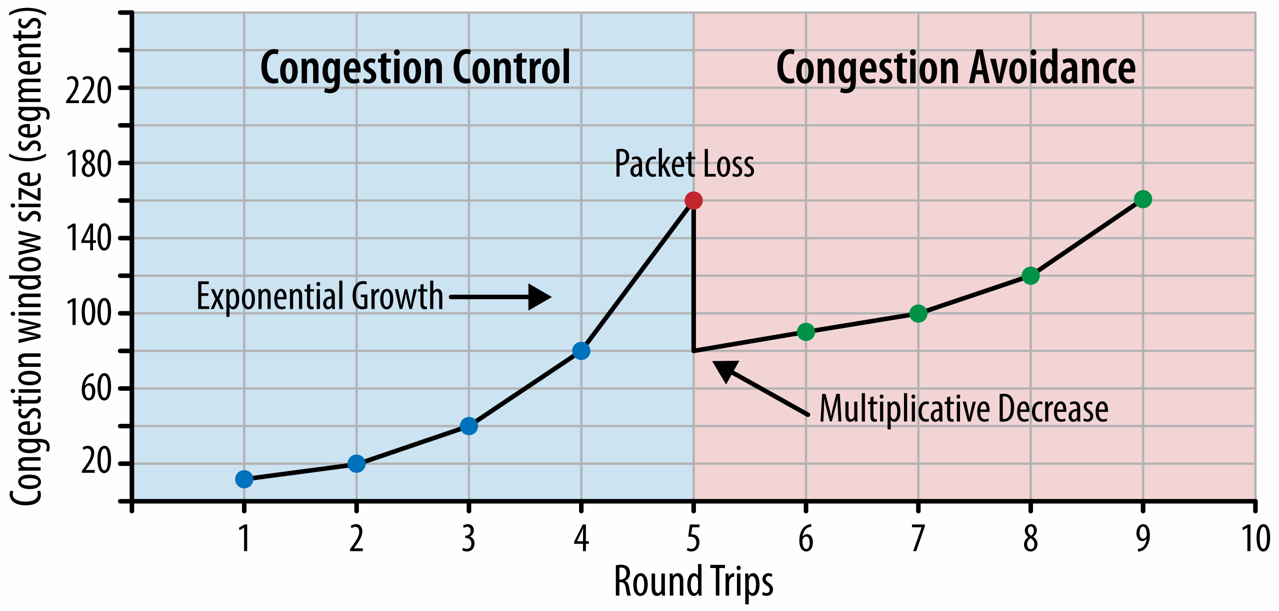


Figure 2-3. Congestion control and congestion avoidance

So why is slow-start an important factor to keep in mind when we are building applications for the browser? Well, HTTP and many other application protocols run over TCP, and no matter the

Instead, we start with a small congestion window and double it for every roundtrip — i.e., exponential growth. As a result, the time required to reach a specific throughput target is a function (Time to reach the cwnd size of size N) of both the roundtrip time between the client and server and the initial congestion window size.

Time to reach the cwnd size of size N

$$\text{Time} = \text{RTT} \times \left\lceil log_2 \left(1 + \frac{\text{N}}{\text{initial cwnd}} \right) \right\rceil$$

For a hands-on example of slow-start impact, let’s assume the following scenario:

- Client and server receive windows: 65,535 bytes (64 KB)
- Initial congestion window: 10 segments (RFC 6928)
- Roundtrip time: 56 ms (London to New York)

Despite the 64 KB receive window size, the throughput of a new TCP connection is initially limited by the size of the congestion window. In fact, to reach the 64 KB receive window limit, we will first need to grow the congestion window size to 45 segments, which will take 168 milliseconds:

$$\frac{65,535 \text{ bytes}}{1,460 \text{ bytes}} \approx 45 \text{ segments}$$

$$56 \text{ ms} \times \left\lceil log_2 \left(1 + \frac{45}{10} \right) \right\rceil = 168 \text{ ms}$$

That’s three roundtrips (Figure 2-4) to reach 64 KB of throughput between the client and server! The fact that the client and server may be capable of transferring at Mbps+ data rates has no effect when a new connection is established — that’s slow-start.

Note

The above example uses the new (RFC 6928) value of ten network segments for the initial congestion window. As an exercise, repeat the same calculation with the older size of four segments — you’ll see that this will add an additional 56 millisecond roundtrip to above result!

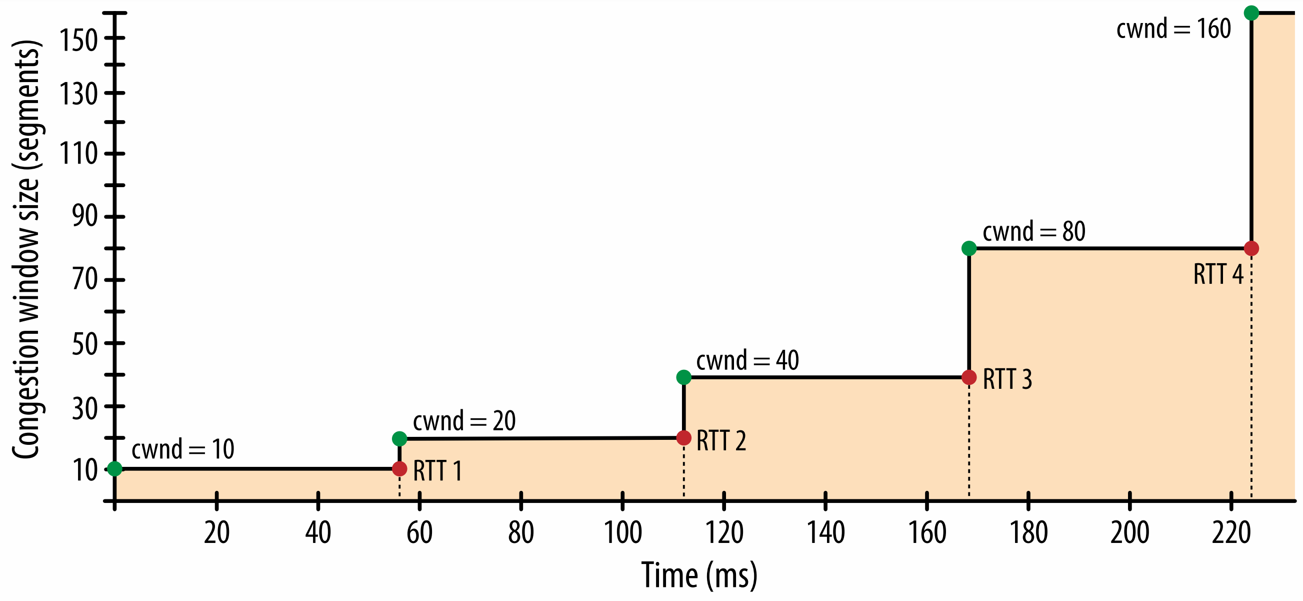


Figure 2-4. Congestion window size growth

To decrease the amount of time it takes to grow the congestion window, we can decrease the roundtrip time between the client and server — e.g., move the server geographically closer to the client. Or we can increase the initial congestion window size to the new RFC 6928 value of 10 segments.

Slow-start is not as big of an issue for large, streaming downloads, as the client and the server will arrive at their maximum window sizes after a few hundred milliseconds and continue to transmit at near maximum speeds — the cost of the slow-start phase is amortized over the lifetime of the larger transfer.

However, for many HTTP connections, which are often short and bursty, it is not unusual for the data transfer to finish before the maximum window size is reached. As a result, the performance of many web applications is often limited by the roundtrip time between server and client: slow-start limits the available bandwidth throughput, which has an adverse effect on the performance of small transfers.

Slow-Start Restart §

In addition to regulating the transmission rate of new connections, TCP also implements a slow-start restart (SSR) mechanism, which resets the congestion window of a connection after it has been idle for a defined period of time. The rationale is simple: the network conditions may have changed while the connection has been idle, and to avoid congestion, the window is reset to a "safe" default.

Not surprisingly, SSR can have a significant impact on performance of long-lived TCP connections that may idle for bursts of time — e.g., due to user inactivity. As a result, it is generally recommended to disable SSR on the server to help improve performance of long-lived HTTP connections. On Linux platforms, the SSR setting can be checked and disabled via the following commands:

```
• $> sysctl net.ipv4.tcp_slow_start_after_idle
• $> sysctl -w net.ipv4.tcp_slow_start_after_idle=0
```

To illustrate the impact of the three-way handshake and the slow-start phase on a simple HTTP transfer, let's assume that our client in New York requests a 64 KB file from the server in London over a new TCP connection (Figure 2-5), and the following connection parameters are in place:

- Roundtrip time: 56 ms
- Client and server bandwidth: 5 Mbps
- Client and server receive window: 65,535 bytes
- Initial congestion window: 10 segments ($10 \times 1460 \text{ bytes} \approx 14 \text{ KB}$)
- Server processing time to generate response: 40 ms
- No packet loss, ACK per packet, GET request fits into single segment

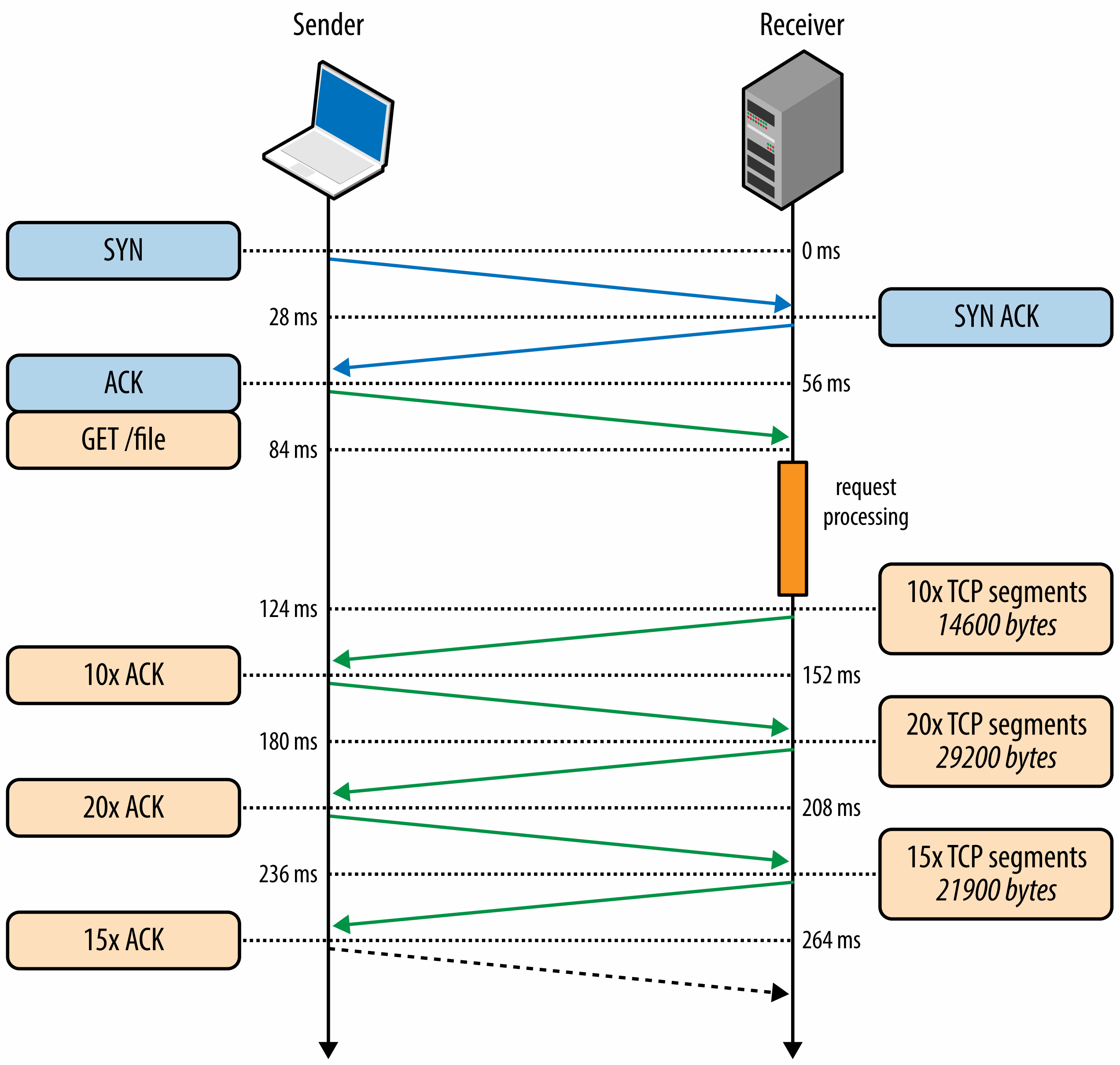


Figure 2-5. Fetching a file over a new TCP connection

`0 ms`

Client begins the TCP handshake with the SYN packet.

`28 ms`

Server replies with SYN-ACK and specifies its rwnd size.

`56 ms`

Client ACKs the SYN-ACK, specifies its rwnd size, and immediately sends the HTTP GET request.

`84 ms`

Server receives the HTTP request.

`124 ms`

Server completes generating the 64 KB response and sends 10 TCP segments before pausing for an ACK (initial cwnd size is 10).

`152 ms`

Client receives 10 TCP segments and ACKs each one.

`180 ms`

Server increments its cwnd for each ACK and sends 20 TCP segments.

`208 ms`

Client receives 20 TCP segments and ACKs each one.

`236 ms`

Server increments its cwnd for each ACK and sends remaining 15 TCP segments.

`264 ms`

Client receives 15 TCP segments, ACKs each one.

264 ms to transfer a 64 KB file on a new TCP connection with 56 ms roundtrip time between the client and server! By comparison, let’s now assume that the client is able to reuse the same TCP connection (Figure 2-6) and issues the same request once more.

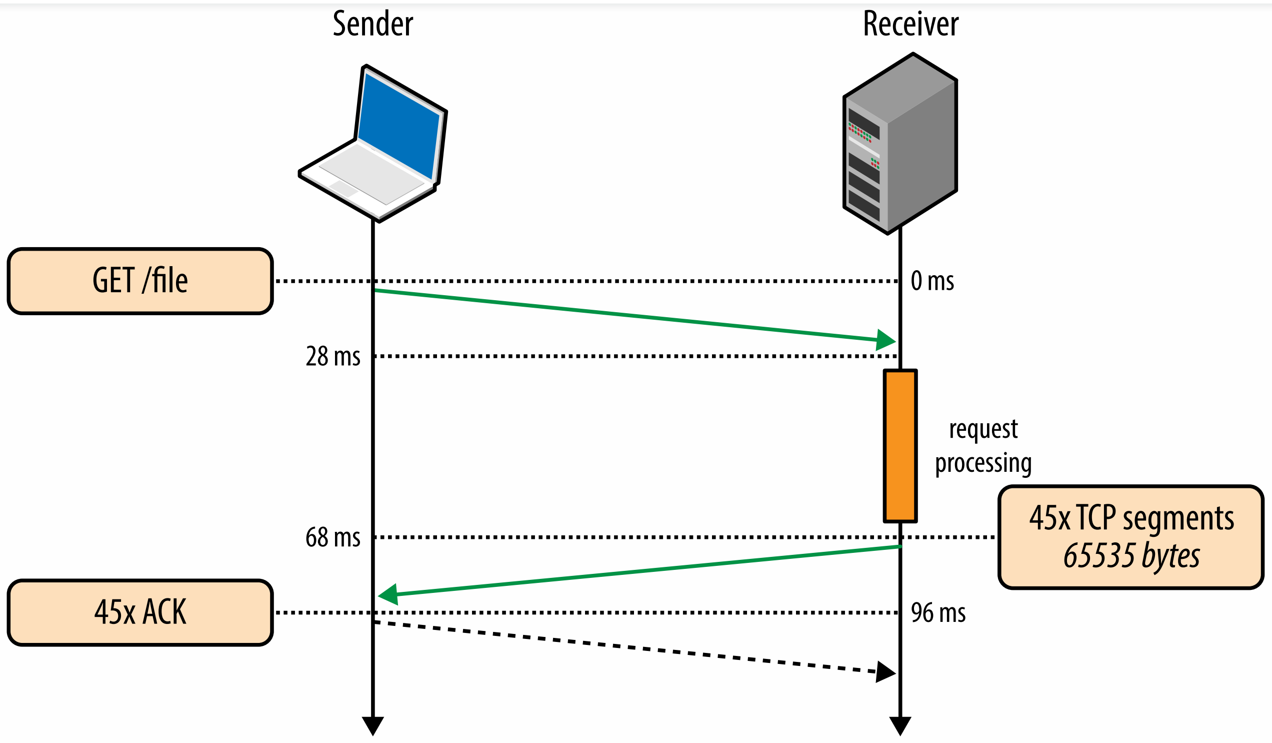


Figure 2-6. Fetching a file over an existing TCP connection

`0 ms`

Client sends the HTTP request.

`28 ms`

Server receives the HTTP request.

`68 ms`

Server completes generating the 64 KB response, but the cwnd value is already greater than the 45 segments required to send the file; hence it dispatches all the segments in one burst.

`96 ms`

Client receives all 45 segments, ACKs each one.

The same request made on the same connection, but without the cost of the three-way handshake and the penalty of the slow-start phase, now took 96 milliseconds, which translates into a 275% improvement in performance!

In both cases, the fact that both the server and the client have access to 5 Mbps of upstream bandwidth had no impact during the startup phase of the TCP connection. Instead, the latency and the congestion window sizes were the limiting factors.

In fact, the performance gap between the first and the second request dispatched over an existing connection will only widen if we increase the roundtrip time; as an exercise, try it with a few different values. Once you develop an intuition for the mechanics of TCP congestion control, dozens of optimizations such as keepalive, pipelining, and multiplexing will require little further motivation.

Increasing TCP's Initial Congestion Window §

Increasing the initial cwnd size on the server to the new RFC 6928 value of 10 segments (IW10) is one of the simplest ways to improve performance for all users and all applications running over TCP. And the good news is that many operating systems have already updated their latest kernels to use the increased value — check the appropriate documentation and release notes.

For Linux, IW10 is the new default for all kernels above 2.6.39. However, don't stop there: upgrade to 3.2+ to also get the benefit of other important updates; see Proportional Rate Reduction for TCP.

Congestion Avoidance §

It is important to recognize that TCP is specifically designed to use packet loss as a feedback mechanism to help regulate its performance. In other words, it is not a question of *if*, but rather of *when* the packet loss will occur. Slow-start initializes the connection with a conservative window and, for every roundtrip, doubles the amount of data in flight until it exceeds the receiver's flow-control window, a system-configured congestion threshold (ssthresh) window, or until a packet is lost, at which point the congestion avoidance algorithm (Figure 2-3) takes over.

The implicit assumption in congestion avoidance is that packet loss is indicative of network congestion: somewhere along the path we have encountered a congested link or a router, which was forced to drop the packet, and hence we need to adjust our window to avoid inducing more packet loss to avoid overwhelming the network.

Once the congestion window is reset, congestion avoidance specifies its own algorithms for how to grow the window to minimize further loss. At a certain point, another packet loss event will occur, and the process will repeat once over. If you have ever looked at a throughput trace of a TCP connection and observed a sawtooth pattern within it, now you know why it looks as such: it is the congestion control and avoidance algorithms adjusting the congestion window size to account for packet loss in the network.

Finally, it is worth noting that improving congestion control and avoidance is an active area both for academic research and commercial products: there are adaptations for different network types, different types of data transfers, and so on. Today, depending on your platform, you will likely run one of the many variants: TCP Tahoe and Reno (original implementations), TCP Vegas, TCP New Reno, TCP BIC, TCP CUBIC (default on Linux), or Compound TCP (default on Windows), among many others. However, regardless of the flavor, the core performance implications of congestion control and avoidance hold for all.

Proportional Rate Reduction for TCP §

Determining the optimal way to recover from packet loss is a nontrivial exercise: if you are too connection, and if you don't adjust quickly enough, then you will induce more packet loss!

Originally, TCP used the Multiplicative Decrease and Additive Increase (AIMD) algorithm: when packet loss occurs, halve the congestion window size, and then slowly increase the window by a fixed amount per roundtrip. However, in many cases AIMD is too conservative, and hence new algorithms were developed.

Proportional Rate Reduction (PRR) is a new algorithm specified by RFC 6937, whose goal is to improve the speed of recovery when a packet is lost. How much better is it? According to measurements done at Google, where the new algorithm was developed, it provides a 3–10% reduction in average latency for connections with packet loss.

PRR is now the default congestion-avoidance algorithm in Linux 3.2+ kernels — another good reason to upgrade your servers!

Bandwidth-Delay Product §

The built-in congestion control and congestion avoidance mechanisms in TCP carry another important performance implication: the optimal sender and receiver window sizes must vary based on the roundtrip time and the target data rate between them.

To understand why this is the case, first recall that the maximum amount of unacknowledged, in-flight data between the sender and receiver is defined as the minimum of the receive (rwnd) and congestion (cwnd) window sizes: the current receive windows are communicated in every ACK, and the congestion window is dynamically adjusted by the sender based on the congestion control and avoidance algorithms.

If either the sender or receiver exceeds the maximum amount of unacknowledged data, then it must stop and wait for the other end to ACK some of the packets before proceeding. How long would it have to wait? That's dictated by the roundtrip time between the two!

Bandwidth-delay product (BDP)

Product of data link's capacity and its end-to-end delay. The result is the maximum amount of unacknowledged data that can be in flight at any point in time.

If either the sender or receiver are frequently forced to stop and wait for ACKs for previous packets, then this would create gaps in the data flow (Figure 2-7), which would consequently limit the maximum throughput of the connection. To address this problem, the window sizes should be made just big enough, such that either side can continue sending data until an ACK arrives back from the client for an earlier packet — no gaps, maximum throughput. Consequently, the optimal window size depends on the roundtrip time! Pick a low window size, and you will limit your connection throughput, regardless of the available or advertised bandwidth between the peers.

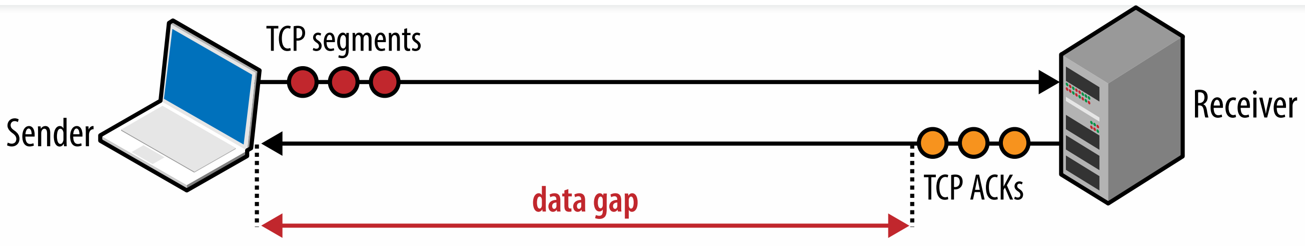


Figure 2-7. Transmission gaps due to low congestion window size

So how big do the flow control (rwnd) and congestion control (cwnd) window values need to be? The actual calculation is a simple one. First, let us assume that the minimum of the cwnd and rwnd window sizes is 16 KB, and the roundtrip time is 100 ms:

$$\begin{aligned} 16\ \text{KB} &= (16 \times 1024 \times 8) = 131,072\ \text{bits} \\ \frac{131,072\ \text{bits}}{0.1\ \text{s}} &= 1,310,720\ \text{bits/s} \\ 1,310,720\ \text{bits/s} = \frac{1,310,720}{1,000,000} &= 1.31\ \text{Mbps} \end{aligned}$$

Regardless of the available bandwidth between the sender and receiver, this TCP connection will not exceed a 1.31 Mbps data rate! To achieve higher throughput we need to raise the window size or lower the roundtrip time.

Similarly, we can compute the optimal window size if we know the roundtrip time and the available bandwidth on both ends. In this scenario, let’s assume that the roundtrip time stays the same (100 ms), but the sender has 10 Mbps of available bandwidth, and the receiver is on a high-throughput 100 Mbps+ link. Assuming there is no network congestion between them, our goal is to saturate the 10 Mbps link available to the client:

$$\begin{aligned} 10\ \text{Mbps} = 10 \times 1,000,000 &= 10,000,000\ \text{bits/s} \\ 10,000,000\ \text{bits/s} = \frac{10,000,000}{8 \times 1024} &= 1,221\ \text{KB/s} \\ 1,221\ \text{KB/s} \times 0.1\ \text{s} &= 122.1\ \text{KB} \end{aligned}$$

The window size needs to be at least 122.1 KB to saturate the 10 Mbps link. Recall that the maximum receive window size in TCP is 64 KB unless window scaling — see Window Scaling (RFC 1323) — is present: double-check your client and server settings!

The good news is that the window size negotiation and tuning is managed automatically by the network stack and should adjust accordingly. The bad news is sometimes it will still be the limiting factor on TCP performance. If you have ever wondered why your connection is transmitting at a fraction of the available bandwidth, even when you know that both the client and the server are capable of higher rates, then it is likely due to a small window size: a saturated peer advertising low receive window, bad network weather and high packet loss resetting the congestion window, or explicit traffic shaping that could have been applied to limit throughput of your connection.

Bandwidth-Delay Product in High-Speed LANs §

BDP is a function of the roundtrip time and the target data rate. Hence, while the roundtrip time is a common bottleneck in cases with high propagation delay, it can also be a bottleneck on a local LAN!

To achieve 1 Gbit/s with 1 ms roundtrip time, we would also need a congestion window of at least 122 KB. The calculation is exactly the same as we saw earlier; we simply added a few zeroes to the target data rate and removed the same amount of zeroes from the roundtrip latency.

Head-of-Line Blocking §

TCP provides the abstraction of a reliable network running over an unreliable channel, which includes basic packet error checking and correction, in-order delivery, retransmission of lost packets, as well as flow control, congestion control, and congestion avoidance designed to operate the network at the point of greatest efficiency. Combined, these features make TCP the preferred transport for most applications.

However, while TCP is a popular choice, it is not the only, nor necessarily the best choice for every occasion. Specifically, some of the features, such as in-order and reliable packet delivery, are not always necessary and can introduce unnecessary delays and negative performance implications.

To understand why that is the case, recall that every TCP packet carries a unique sequence number when put on the wire, and the data must be passed to the receiver in-order (Figure 2-8). If one of the packets is lost en route to the receiver, then all subsequent packets must be held in the receiver's TCP buffer until the lost packet is retransmitted and arrives at the receiver. Because this work is done within the TCP layer, our application has no visibility into the TCP retransmissions or the queued packet buffers, and must wait for the full sequence before it is able to access the data. Instead, it simply sees a delivery delay when it tries to read the data from the socket. This effect is known as TCP head-of-line (HOL) blocking.

The delay imposed by head-of-line blocking allows our applications to avoid having to deal with packet reordering and reassembly, which makes our application code much simpler. However, this is done at the cost of introducing unpredictable latency variation in the packet arrival times, commonly referred to as *jitter*, which can negatively impact the performance of the application.

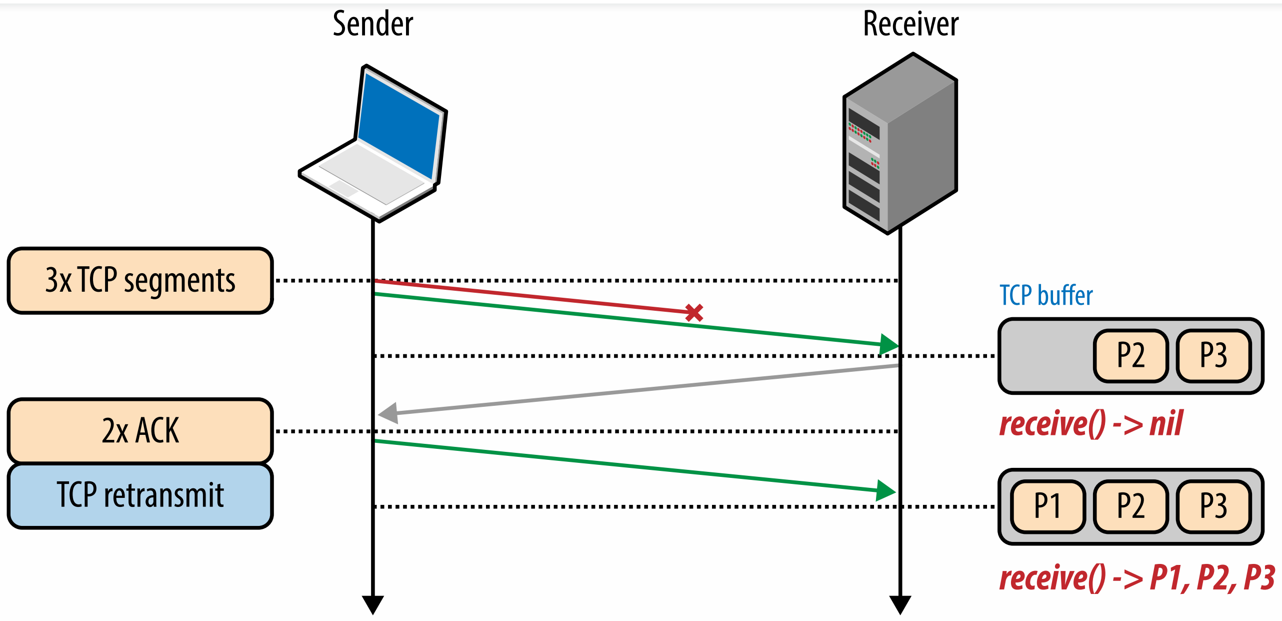


Figure 2-8. TCP Head-of-line blocking

Further, some applications may not even need either reliable delivery or in-order delivery: if every packet is a standalone message, then in-order delivery is strictly unnecessary, and if every message overrides all previous messages, then the requirement for reliable delivery can be removed entirely. Unfortunately, TCP does not provide such configuration — all packets are sequenced and delivered in order.

Applications that can deal with out-of-order delivery or packet loss and that are latency or jitter sensitive are likely better served with an alternate transport, such as UDP.

Packet Loss Is OK §

In fact, packet loss is necessary to get the best performance from TCP! A dropped packet acts as a feedback mechanism, which allows the receiver and sender to adjust their sending rates to avoid overwhelming the network, and to minimize latency; see Bufferbloat in Your Local Router. Further, some applications can tolerate packet loss without adverse effects: audio, video, and game state updates are common examples of application data that do not require either reliable or in-order delivery — incidentally, this is also why WebRTC uses UDP as its base transport.

If a packet is lost, then the audio codec can simply insert a minor break in the audio and continue processing the incoming packets. If the gap is small, the user may not even notice, and waiting for the lost packet runs the risk of introducing variable pauses in audio output, which would result in a much worse experience for the user.

Similarly, if we are delivering game state updates, then waiting for a packet describing its state at time `T-1`, when we already have the packet for time T is often simply unnecessary — ideally, we would receive each and every update, but to avoid gameplay delays, we can accept intermittent loss in favor of lower latency.

Optimizing for TCP §

TCP is an adaptive protocol designed to be fair to all network peers and to make the most efficient use of the underlying network. Thus, the best way to optimize TCP is to tune how TCP senses the current network conditions and adapts its behavior based on the type and the requirements of the layers below and above it: wireless networks may need different congestion algorithms, and some applications may need custom quality of service (QoS) semantics to deliver the best experience.

The close interplay of the varying application requirements, and the many knobs in every TCP algorithm, make TCP tuning and optimization an inexhaustible area of academic and commercial research. In this chapter, we have only scratched the surface of the many factors that govern TCP performance. Additional mechanisms, such as selective acknowledgments (SACK), delayed acknowledgments, and fast retransmit, among many others, make each TCP session much more complicated (or interesting, depending on your perspective) to understand, analyze, and tune.

Having said that, while the specific details of each algorithm and feedback mechanism will continue to evolve, the core principles and their implications remain unchanged:

- TCP three-way handshake introduces a full roundtrip of latency.
- TCP slow-start is applied to every new connection.
- TCP flow and congestion control regulate throughput of all connections.
- TCP throughput is regulated by current congestion window size.

As a result, the rate with which a TCP connection can transfer data in modern high-speed networks is often limited by the roundtrip time between the receiver and sender. Further, while bandwidth continues to increase, latency is bounded by the speed of light and is already within a small constant factor of its maximum value. In most cases, latency, not bandwidth, is the bottleneck for TCP — e.g., see Figure 2-5.

Tuning Server Configuration §

As a starting point, prior to tuning any specific values for each buffer and timeout variable in TCP, of which there are dozens, you are much better off simply upgrading your hosts to their latest system versions. TCP best practices and underlying algorithms that govern its performance continue to evolve, and most of these changes are available only in the latest kernels. In short, keep your servers up to date to ensure the optimal interaction between the sender’s and receiver’s TCP stacks.

Note

On the surface, upgrading server kernel versions seems like trivial advice. However, in practice, *versions, and system administrators are reluctant to perform the upgrade.*

To be fair, every upgrade brings its risks, but to get the best TCP performance, it is also likely the single best investment you can make.

With the latest kernel in place, it is good practice to ensure that your server is configured to use the following best practices:

Increasing TCP's Initial Congestion Window
: A larger starting congestion window allows TCP to transfer more data in the first roundtrip and significantly accelerates the window growth.

Slow-Start Restart
: Disabling slow-start after idle will improve performance of long-lived TCP connections that transfer data in periodic bursts.

Window Scaling (RFC 1323)
: Enabling window scaling increases the maximum receive window size and allows high-latency connections to achieve better throughput.

TCP Fast Open
: Allows application data to be sent in the initial SYN packet in certain situations. TFO is a new optimization, which requires support both on client and server; investigate if your application can make use of it.

The combination of the preceding settings and the latest kernel will enable the best performance — lower latency and higher throughput — for individual TCP connections.

Depending on your application, you may also need to tune other TCP settings on the server to optimize for high connection rates, memory consumption, or similar criteria. Consult your platform documentation and read through "TCP Tuning for HTTP" document maintained by the HTTP Working Group for additional advice.

Note

For Linux users, `ss` is a useful power tool to inspect various statistics for open sockets. From the command line, run `ss --options --extended --memory --processes --info` to see the current peers and their respective connection settings.

Tuning Application Behavior

Tuning performance of TCP allows the server and client to deliver the best throughput and latency for an individual connection. However, how an application uses each new, or established,

- No bit is faster than one that is not sent; send fewer bits.
- We can't make the bits travel faster, but we can move the bits closer.
- TCP connection reuse is critical to improve performance.

Eliminating unnecessary data transfers is, of course, the single best optimization — e.g., eliminating unnecessary resources or ensuring that the minimum number of bits is transferred by applying the appropriate compression algorithm. Following that, locating the bits closer to the client, by geo-distributing servers around the world — e.g., using a CDN — will help reduce latency of network roundtrips and significantly improve TCP performance. Finally, where possible, existing TCP connections should be reused to minimize overhead imposed by slow-start and other congestion mechanisms.

Performance Checklist §

Optimizing TCP performance pays high dividends, regardless of the type of application, for every new connection to your servers. A short list to put on the agenda:

- Upgrade server kernel to latest version.
- Ensure that cwnd size is set to 10.
- Ensure that window scaling is enabled.
- Disable slow-start after idle.
- Investigate enabling TCP Fast Open.
- Eliminate redundant data transfers.
- Compress transferred data.
- Position servers closer to the user to reduce roundtrip times.
- Reuse TCP connections whenever possible.
- Investigate "TCP Tuning for HTTP" recommendations.

« Back to the Table of Contents