CHAPTER 3

Tuning TCP: Transport Layer

This chapter describes some of key Transport Control Protocol (TCP) tunable parameters related to performance tuning. More importantly it describes how these tunables work, how they interact with each other, and how they impact network traffic when they are modified.

Applications often recommend TCP settings for tunable parameters, but offer few details on the meaning of the parameters and adverse effects that might result from the recommended settings. This chapter is intended as a guide to understanding those recommendations. This chapter is intended for network architects and administrators who have an intermediate knowledge of networking and TCP. This is not an introductory chapter on TCP terminology. The concepts discussed in this chapter build on basic terminology concepts and definitions. For an excellent resource, refer to *Internetworking with TCP/IP Volume 1, Principles, Protocols, and Architectures* by Douglas Comer, Prentice Hall, New Jersey.

Network architects responsible for designing optimal backbone and distribution IP network architectures for the corporate infrastructure are primarily concerned with issues at or below the IP layer—network topology, routing, and so on. However, in data center networks, servers connect either to the corporate infrastructure or the service provider networks, which host applications. These applications provide networked application services with additional requirements in the area of networking and computer systems, where the goal is to move data as fast as possible from the application out to the network interface card (NIC) and onto the network. Designing network architectures for performance at the data center includes looking at protocol processing above Layer 3, into the transport and application layers. Further, the problem becomes more complicated because many clients' stateful connections are aggregated onto one server. Each client connection might have vastly different characteristics, such as bandwidth, latencies, or probability of packet loss. You must identify the predominant traffic characteristics and tune the protocol stack for optimal performance. Depending on the server hardware, operating system, and device driver implementations, there could be many possible tuning configurations and recommendations. However, tuning the connection-oriented transport layer protocol is often most challenging.

This chapter includes the following topics:

- "TCP Tuning Domains" on page 38 provides an overview of TCP from a tuning perspective, describing the various components that contain tunable parameters and where they fit together from a high level, thus showing the complexities of tuning TCP.
- "TCP State Model" on page 48 proposes a model of TCP that illustrates the behavior of TCP and the impact of tunable parameters. The system model then projects a network traffic diagram baseline case showing an ideal scenario.
- "TCP Congestion Control and Flow Control – Sliding Windows" on page 53 shows various conditions to help explain how and why TCP tuning is needed and which are the most effective TCP tunable parameters needed to compensate for adverse conditions.
- "TCP and RDMA Future Data Center Transport Protocols" on page 62 describes TCP and RDMA, promising future networking protocols that may overcome the limitations of TCP.

TCP Tuning Domains

Transport Control Protocol (TCP) tuning is complicated because there are many algorithms running and controlling TCP data transmissions concurrently, each with slightly different purposes.

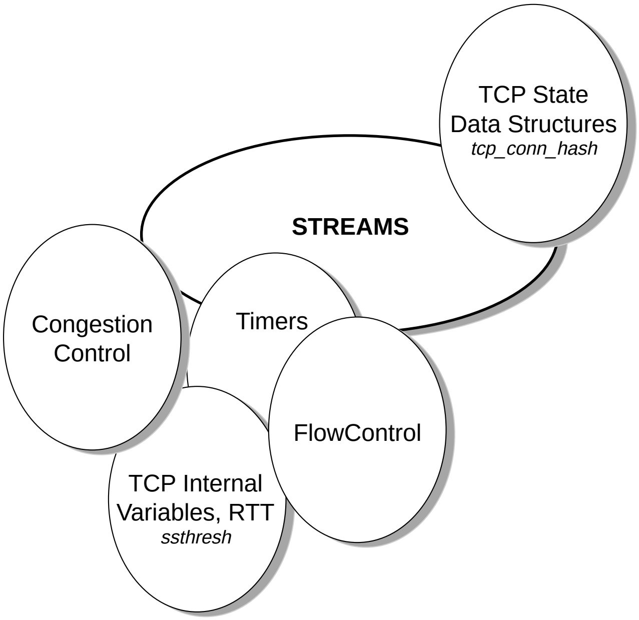


FIGURE 3-1 Overview of Overlapping Tuning Domains

FIGURE 3-1 shows a high-level view of the different components that impact TCP processing and performance. While the components are interrelated, each has its own function and optimization strategy.

- The STREAMS framework looks at raw bytes flowing up and down the streams modules. It has no notion of TCP, congestion in the network, or the client load. It only looks at how congested the STREAMS queues are. It has its own flow control mechanisms.
- TCP-specific control mechanisms are not tunable, but they are computed based on algorithms that are tunable.
- Flow control mechanisms and congestion control mechanisms are functionally completely different. One is concerned with the endpoints, and the other is concerned with the network. Both impact how TCP data is transmitted.
- Tunable parameters control scalability. TCP requires certain static data structures that are backed by non-swappable kernel memory. Avoid the following two scenarios:
  - Allocating large amounts of memory. If the actual number of simultaneous connections is fewer than anticipated, memory that could have been used by other applications is wasted.

- Allocating insufficient memory. If the actual number of connections exceeds the anticipated TCP load, there will not be sufficient free TCP data structures to handle the peak load.

This class of tunable parameters directly impacts the number of simultaneous TCP connections a server can handle at peak load and control scalability.

TCP Queueing System Model

The goal of TCP tuning can be reduced to maximizing the throughput of a closed loop system, as shown in FIGURE 3-2. This system abstracts all the main components of a complete TCP system, which consists of the following components:

- Server—The focus of this chapter.
- Network—The endpoints can only infer the state of the network by measuring and computing various delays, such as round-trip times, timers, receipt of acknowledgments, and so on.
- Client—The remote client endpoint of the TCP connection.

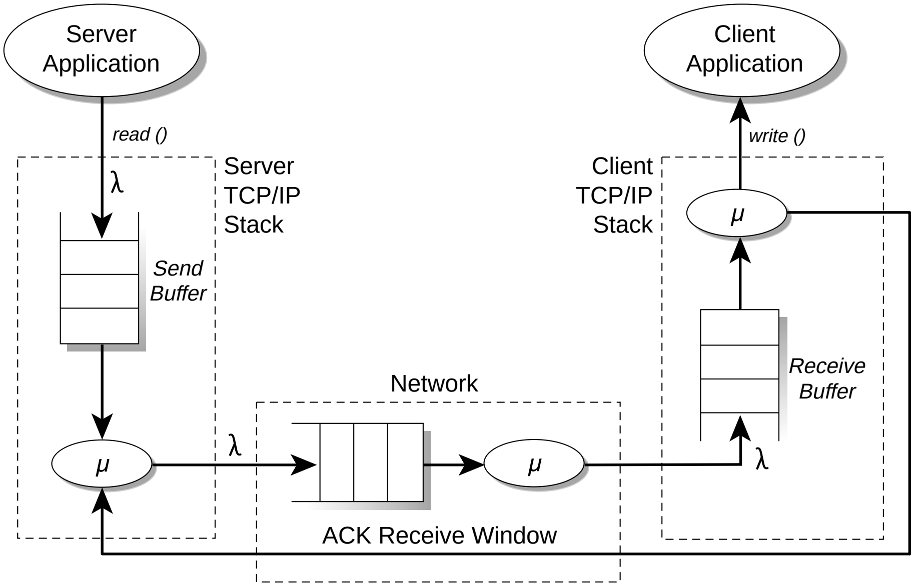


FIGURE 3-2 Closed-Loop TCP System Model

This section requires basic background in queueing theory. For more information, refer to *Queueing Systems, Volume 1*, by Dr. Lenny Kleinrock, 1975, Wiley, New York. In FIGURE 3-2, we model each component as an M/M/1 queue. An M/M/1 queue is a simple queue that has packet arrivals at a certain speed, which we've designated as λ. At the other end of the queue, these packets are processed at a certain speed, which we've designated as μ.

TCP is a full duplex protocol. For the sake of simplicity, only one side of the duplex communication process is shown. Starting from the server side on the left in FIGURE 3-2, the server application writes a byte stream to a TCP socket. This is modeled as messages arriving at the M/M/1 queue at the rate of l. These messages are queued and processed by the TCP engine. The TCP engine implements the TCP protocol and consists of various timers, algorithms, retransmit queues, and so on, modeled as the server process μ, which is also controlled by the feedback loop as shown in FIGURE 3-2. The feedback loop represents acknowledgements (ACKs) from the client side and receive windows. The server process sends packets to the network, which is also modeled as an M/M/1 queue. The network can be congested, hence packets are queued up. This captures latency issues in the network, which are a result of propagation delays, bandwidth limitations, or congested routers. In FIGURE 3-2 the client side is also represented as an M/M/1 queue, which receives packets from the network and the client TCP stack, processes the packets as quickly as possible, forwards them to the client application process, and sends feedback information to the server. The feedback represents the ACK and receive window, which provide flow control capabilities to this system.

Why the Need to Tune TCP

FIGURE 3-3 shows a cross-section view of the sequence of packets sent from the server to the client of an ideally tuned system. Send window-sized packets are sent one after another in a pipelined fashion continuously to the client receiver. Simultaneously, the client sends back ACKs and receive windows in unison with the server. This is the goal we are trying to achieve by tuning TCP parameters. Problems crop up when delays vary because of network congestion, asymmetric network capacities, dropped packets, or asymmetric server/client processing capacities. Hence, tuning is required. To see the TCP default values for your version of Solaris, refer to the Solaris documentation at `docs.sun.com`.

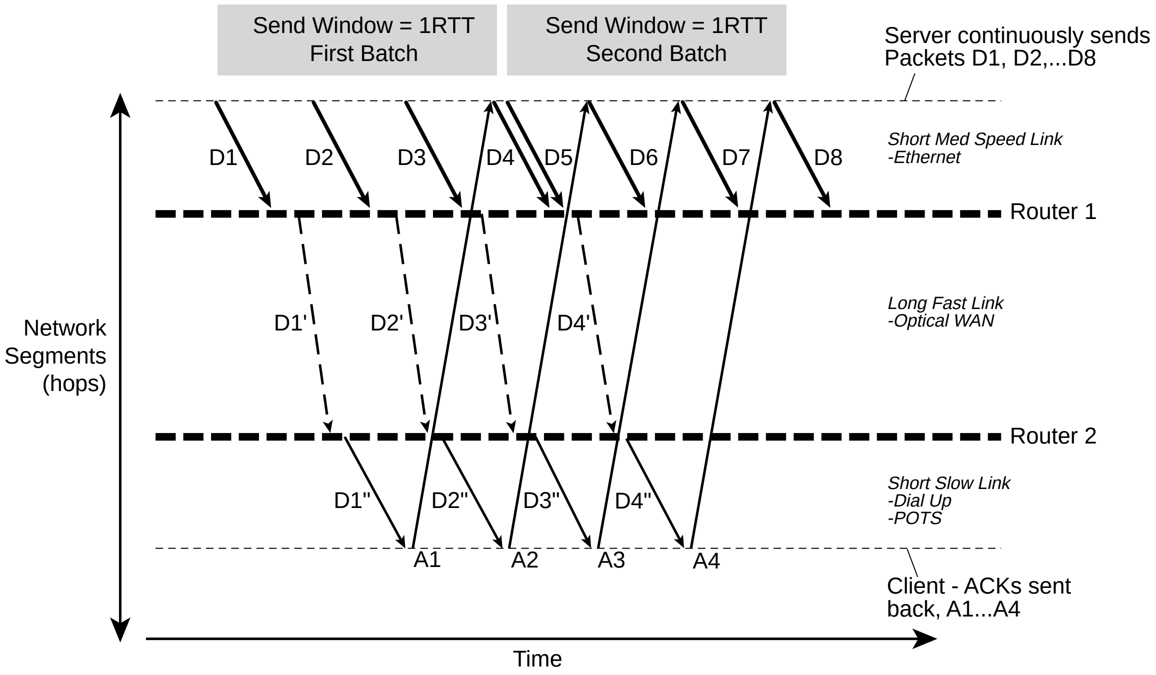


FIGURE 3-3 Perfectly Tuned TCP/IP System

In a perfectly tuned TCP system spanning several network links of varying distances and bandwidths, the clients send back ACKs to sender in perfect synchronization with the start of sending the next window.

The objective of an optimal system is to maximize the throughput of the system. In the real world, asymmetric capacities require tuning on both the server and client side to achieve optimal throughput. For example, if the network latency is excessive, the amount of traffic injected into the network will be reduced to more closely maintain a flow that matches the capacity of the network. If the network is fast enough, but the client is slow, the feedback loop will be able to alert the sender TCP process to reduce the amount of traffic injected into the network. Later sections will build on these concepts to describe how to tune for wireless, high-speed wide area networks (WANs), and other types of networks that vary in bandwidth and distance.

FIGURE 3-4 shows the impact of the links increasing in bandwidth; therefore, tuning is needed to improve performance. The opposite case is shown in FIGURE 3-5, where the links are slower. Similarly, if the distances increase or decrease, delays attributed to propagation delays require tuning for optimal performance.

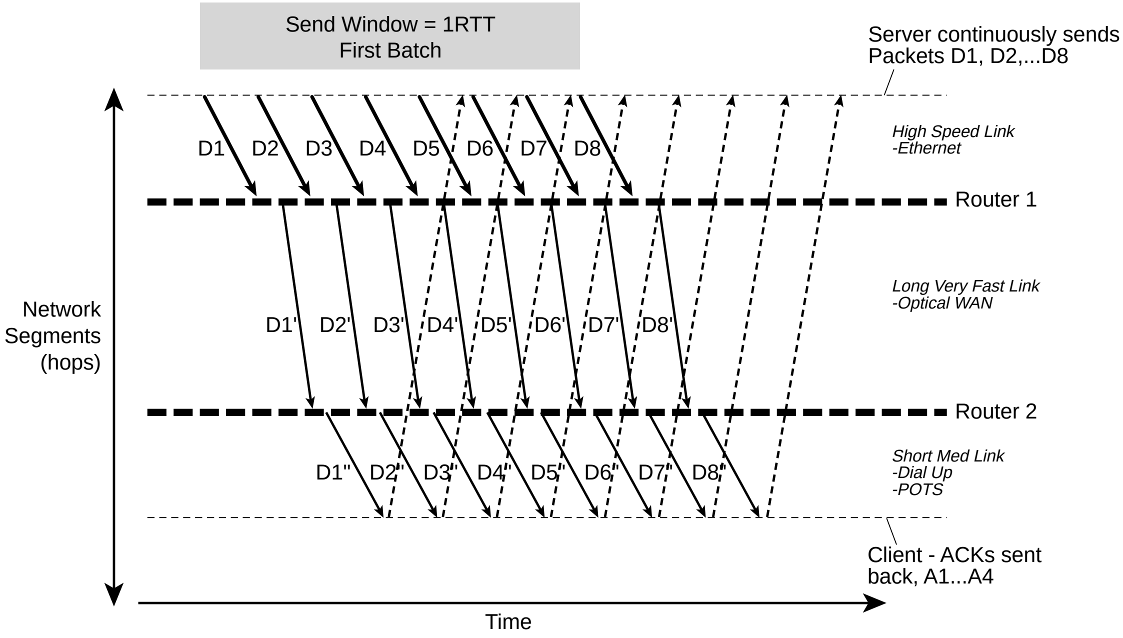

Send Window = 1RTT
First Batch
Server continuously sends Packets D1, D2,...D8
D1
D2
D3
D4
D5
D6
D7
D8
High Speed Link -Ethernet
Router 1
Long Very Fast Link -Optical WAN
Network Segments (hops)
D1'
D2'
D3'
D4'
D5'
D6'
D7'
D8'
Router 2
Short Med Link -Dial Up -POTS
D1''
D2''
D3''
D4''
D5''
D6''
D7''
D8''
Client - ACKs sent back, A1...A4
Time

FIGURE 3-4 Tuning Required to Compensate for Faster Links

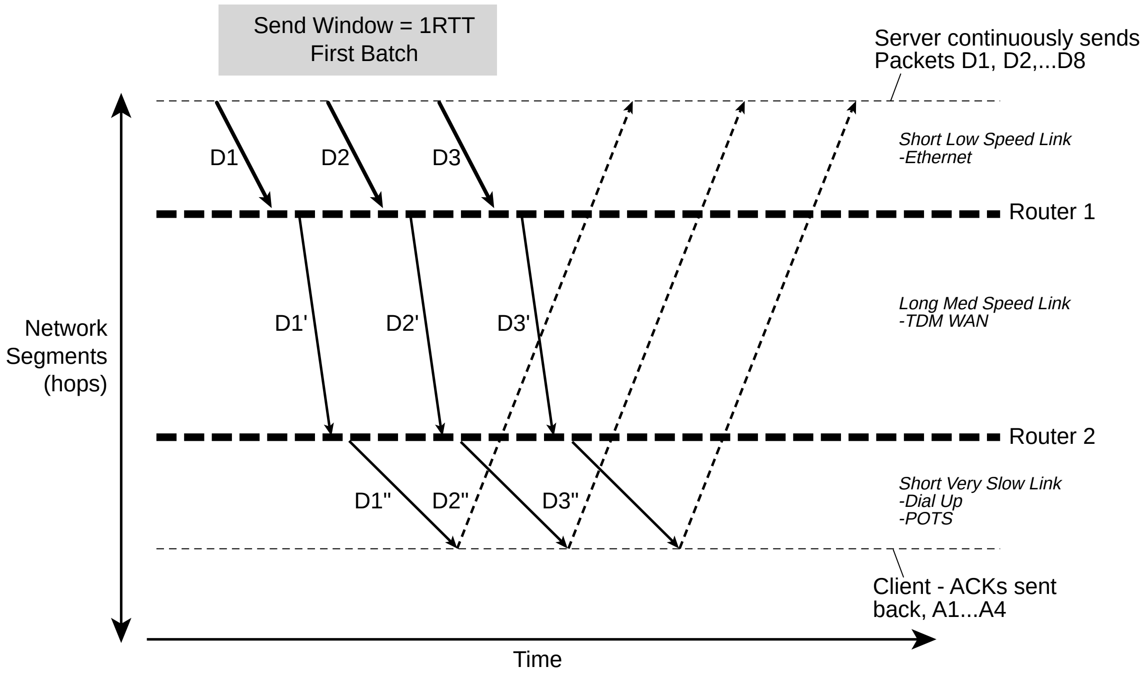


FIGURE 3-5 Tuning Required to Compensate for Slower Links

TCP Packet Processing Overview

Now let's take a look at the internals of the TCP stack inside the computing node. We will limit the scope to the server on the data center side for TCP tuning purposes. Since the clients are symmetrical, we can tune them using the exact same concepts. In a large enterprise data center, there could be thousands of clients, each with a diverse set of characteristics that impact network performance. Each characteristic has a direct impact on TCP tuning and hence on overall network performance. By focusing on the server, and considering different network deployment technologies, we essentially cover the most common cases.

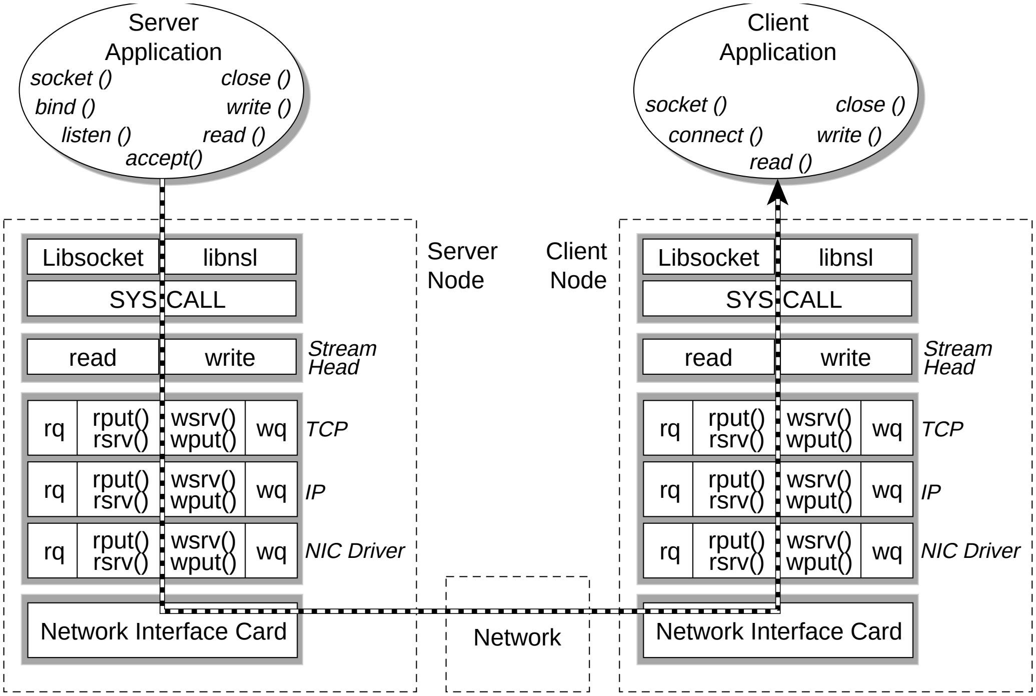


FIGURE 3-6 Complete TCP/IP Stack on Computing Nodes

FIGURE 3-6 shows the internals of the server and client nodes in more detail.

To gain a better understanding of TCP protocol processing, we will describe how a packet is sent up and down a typical STREAMS-based TCP implementation. Consider the server application on the left side of FIGURE 3-6 as a starting point. The following describes how data is moved from the server to the client on the right.

1. The server application opens a socket. (This triggers the operating system to set up the STREAMS stack, as shown.) The server then binds to a transport layer port, executes listen, and waits for a client to connect. Once the client connects, the server completes the TCP three-way handshake, establishes the socket, and both server and client can communicate.

2. Server sends a message by filling a buffer, then writing to the socket.

3. The message is broken up and packets are created, sent down the streamhead (down the read side of each STREAMS module) by invoking the `rput` routine. If the module is congested, the packets are placed on the service routine for deferred processing. Each network module will prepend the packet with an appropriate header.

4. Once the packet reaches the NIC, the packet is copied from system memory to the NIC memory, transmitted out of the physical interface, and sent into the network.
5. The client reads the packet into the NIC memory and an interrupt is generated that copies the packet into system memory and goes up the protocol stack as shown on right in the Client Node.
6. The STREAMS modules read the corresponding header to determine the processing instructions and where to forward the packet. Headers are stripped off as the packet is moved upwards on the write side of each module.
7. The client application reads in the message as the packet is processed and translated into a message, filling the client read buffer.

The Solaris™ operating system (Solaris OS) offers many tunable parameters in the TCP, User Datagram Protocol (UDP), and IP STREAMS module implementation of these protocols. It is important to understand the goals you want to achieve so that you can tune accordingly. In the following sections, we provide a high-level model of the various protocols and provide deployment scenarios to better understand which parameters are important to tune and how to go about tuning them.

We start off with TCP, which is, by far, the most complicated module to tune and has the greatest impact on performance. We then describe how to modify these tunable parameters for different types of deployments. Finally, we describe IP and UDP tuning.

TCP STREAMS Module Tunable Parameters

The TCP stack is implemented using existing operating system application programming interfaces (APIs). The Solaris OS offers a STREAMS framework, originating from AT&T, which was originally designed to allow a flexible modular software framework for network protocols. The STREAMS framework has its own tunable parameters, for example `sq_max_size`, which controls the depth of a STREAMS syncq. This impacts how raw data messages are processed for TCP. FIGURE 3-7 provides a more detailed view of the facilities provided by the Solaris STREAMS framework.

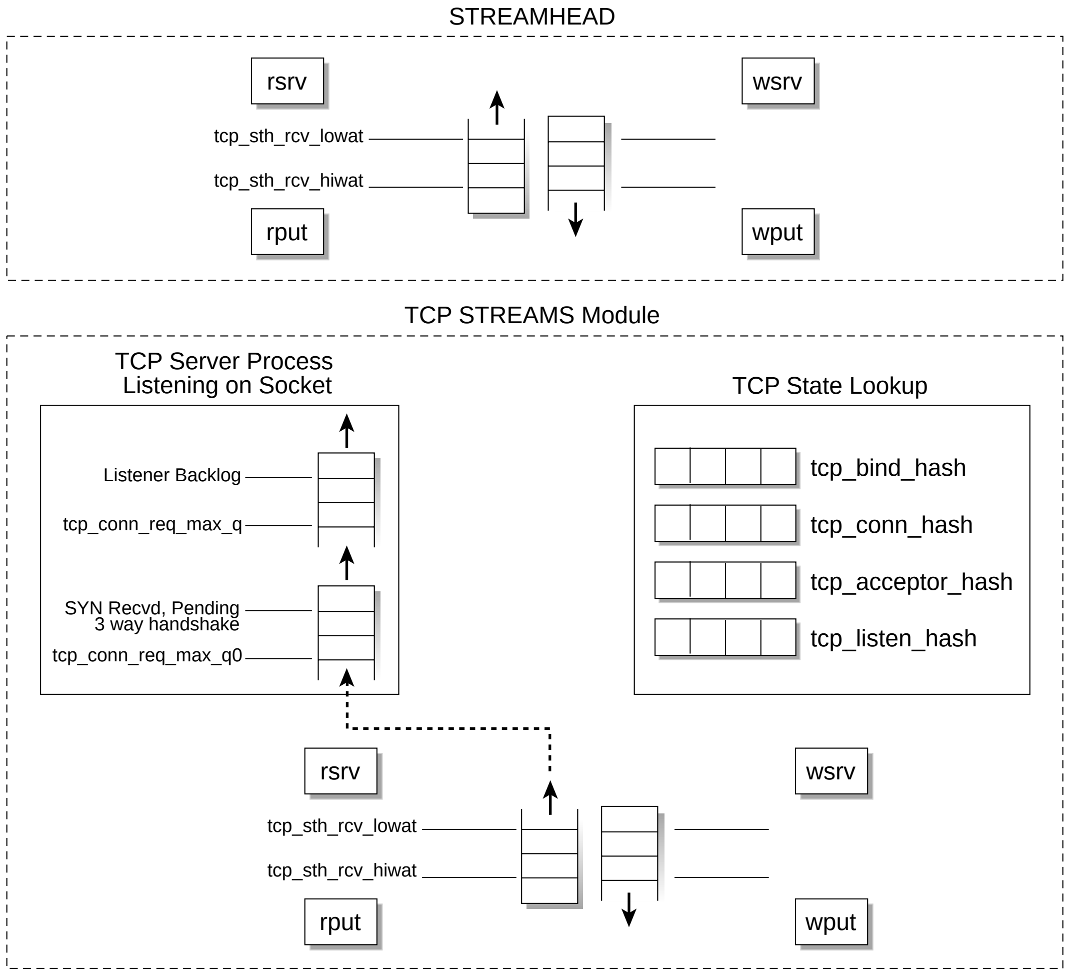


FIGURE 3-7 TCP and STREAM Head Data Structures Tunable Parameters

FIGURE 3-7 shows some key tunable parameters for the TCP-related data path. At the top is the streamhead, which has a separate queue for TCP traffic, where an application reads data. STREAMS flow control starts here. If the operating system is sending up the stack to the application and the application cannot read data as fast as the sender is sending it, the stream read queue starts to fill. Once the number of packets in the queue exceeds the high-water mark, `tcp_sth_recv_hiwat`, streams-based flow control triggers and prevents the TCP module from sending any more packets up to the streamhead. There is some space available for critical control messages (`M_PROTO`, `M_PCPROTO`). The TCP module will be flow controlled as long as the number of packets is above `tcp_sth_recv_lowat`. In other words, the streamhead queue must drain below the low-water mark to reactivate TCP to forward data messages destined for the application. Note that the write side of the streamhead does not require any high-water or low-water marks because it is injecting packets into the downstream, and TCP will flow control the streamhead write side by its high-water and low-water marks tcp_xmit_hiwat and tcp_xmit_lowat. Refer to the Solaris AnswerBook2™ at docs.sun.com for the default values of your version of the Solaris OS.

TCP has a set of hash tables. These tables are used to search for the associated TCP socket state information on each incoming TCP packet to maintain state engine for each socket and perform other TCP tasks to maintain that connection, such as update sequence numbers, update windows, round trip time (RTT), timers, and so on.

The TCP module has two new queues for server processes. The first queue, shown on the left in FIGURE 3-7, is the set of packets belonging to sockets that have not yet established a connection. The server side has not yet received and processed a client-side ACK. If the client does not send an ACK within a certain window of time, then the packet will be dropped. This was designed to prevent synchronization (SYN) flood attacks, where a bunch of unacknowledged client SYN requests caused servers to be overwhelmed and prevented valid client connections from being processed. The next queue is the listen backlog queue, where the client has sent back the final ACK, thus completing the three-way handshake. The server socket for this client will move the connection from LISTEN to ACCEPT. But the server has not yet processed this packet. If the server is slow, then this queue will fill up. The server can override this queue size with the listen backlog parameter. TCP will flow control on IP on the read side with its parameters tcp_recv_lowat and tcp_recv_hiwat similar to the streamhead read side.

TCP State Model

TCP is a reliable transport layer protocol that offers a full duplex connection byte stream service. The bandwidth of TCP makes it appropriate for wide area IP networks where there is a higher chance of packet loss or reordering. What really complicates TCP are the flow control and congestion control mechanisms. These mechanisms often interfere with each other, so proper tuning is critical for high-performance networks. We start by explaining the TCP state machine, then describe in detail how to tune TCP, depending on the actual deployment. We also describe how to scale the TCP connection-handling capacity of servers by increasing the size of TCP connection state data structures.

FIGURE 3-8 presents an alternative view of the TCP state engine.

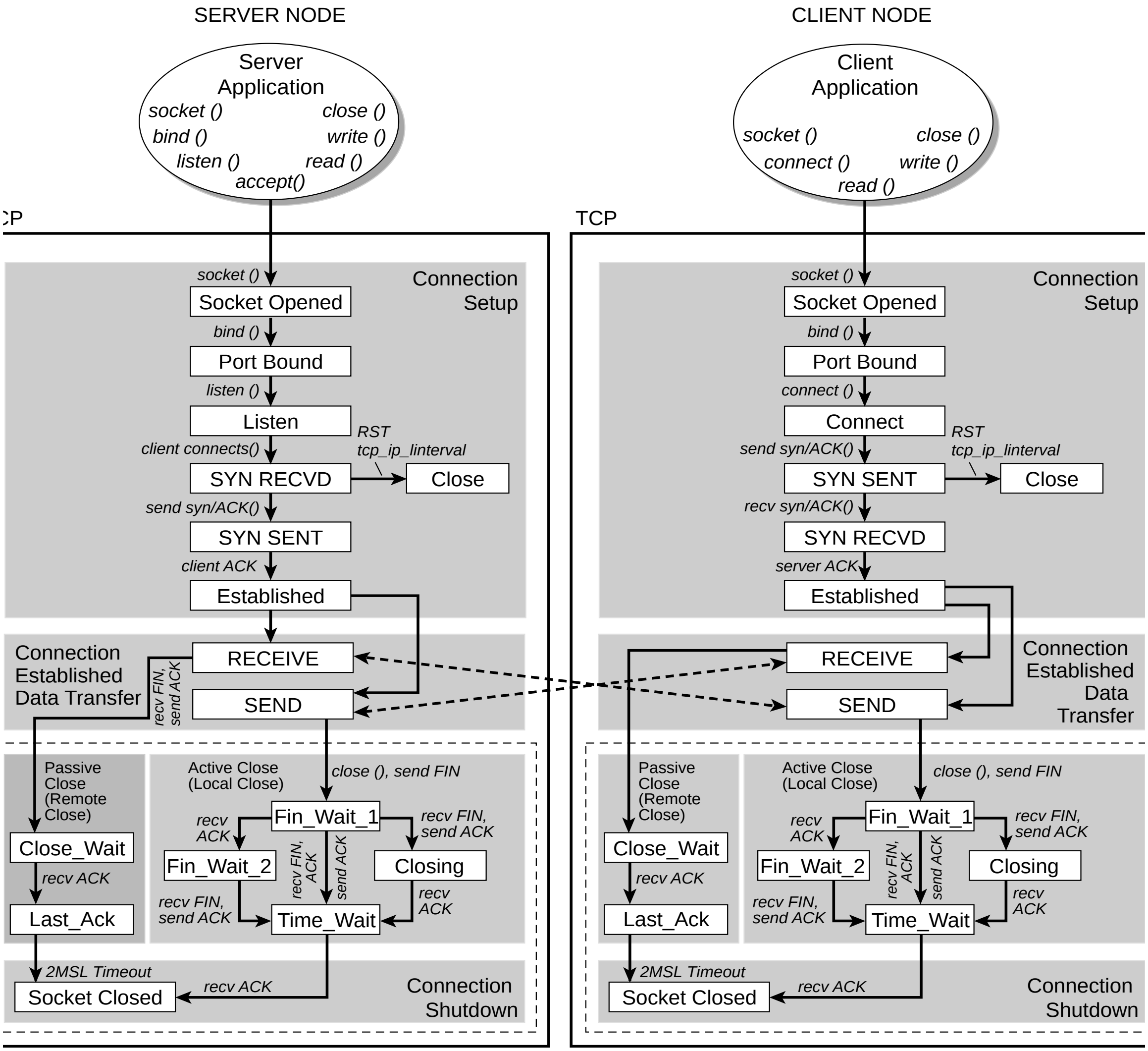


FIGURE 3-8 TCP State Engine Server and Client Node

This figure shows the server and client socket API at the top and the TCP module with the following three main states:

Connection Setup

This includes the collection of substates that collectively set up the socket connection between the two peer nodes. In this phase, the set of tunable parameters includes:

- `tcp_ip_abort_cinterval`: the time a connection can remain in half-open state during the initial three-way handshake, just prior to entering an established state. This is used on the client connect side.
- `tcp_ip_abort_linterval`: the time a connection can remain in half-open state during the initial three-way handshake, just prior to entering an established state. This is used on the server passive listen side.

For a server, there are two trade-offs to consider:

- **Long Abort Intervals** – The longer the abort interval, the longer the server will wait for the client to send information pertaining to the socket connection. This might result in increased kernel consumption and possibly kernel memory exhaustion. The reason is that each client socket connection requires state information, using approximately 1–2 kilobytes of kernel memory. Remember that kernel memory is not swappable, and as the number of connections increases, the amount of consumed memory and time delays for lookups for connections increases. Hackers exploit this fact to initiate Denial of Service (DoS) attacks, where attacking clients constantly send only SYN packets to a server, eventually tying up all kernel memory, not allowing real clients to connect.
- **Short Abort Intervals** – If the interval is too short, valid clients that have a slow connection or go through slow proxies and firewalls could get aborted prematurely. This might help reduce the chances of DoS attacks, but slow clients might also be mistakenly terminated.

Connection Established

This includes the main data transfer state (the focus of our tuning explanations in this chapter). The tuning parameters for congestion control, latency, and flow control will be described in more detail. FIGURE 3-8 shows two concurrent processes that read and write to the bidirectional full-duplex socket connection.

Connection Shutdown

This includes the set of substates that work together to shut down the connection in an orderly fashion. We will see important tuning parameters related to memory. Tunable parameters include:

- until this time has expired. However, if this value is too short and there have been many routing changes, lingering packets in the network might be lost.
- `tcp_fin_wait_2_flush_interval`: how long this side will wait for the remote side to close its side of the connection and send a FIN packet to close the connection. There are cases where the remote side crashes and never sends

a FIN. So to free up resources, this value puts a limit on the time the remote side has to close the socket. This means that half-open sockets cannot remain open indefinitely.

Note – `tcp_close_wait` is no longer a tunable parameter. Instead, use `tcp_time_wait_interval`.

TCP Tuning on the Sender Side

TCP tuning on the sender side controls how much data is injected into the network and the remote client end. There are several concurrent schemes that complicate tuning. So to better understand, we will separate the various components and then describe how these mechanisms work together. We will describe two phases: Startup and Steady State. *Startup Phase* TCP tuning is concerned with how fast we can ramp up sending packets into the network. *Steady State Phase* tuning is concerned about other facets of TCP communication such as tuning timers, maximum window sizes, and so on.

Startup Phase

In Startup Phase tuning, we describe how the TCP sender starts to initially send data on a particular connection. One of the issues with a new connection is that there is no information about the capabilities of the network pipe. So we start by blindly injecting packets at a faster and faster rate until we understand the capabilities and adjust accordingly. Manual TCP tuning is required to change macro behavior, such as when we have very slow pipes as in wireless or very fast pipes such as 10 Gbit/sec. Sending an initial maximum burst has proven disastrous. It is better to slowly increase the rate at which traffic is injected based on how well the traffic is absorbed. This is similar to starting from a standstill on ice. If we initially floor the gas pedal, we will skid, and then it is hard to move at all. If, on the other hand, we start slowly and gradually increase speed, we can eventually reach a very fast speed. In networking, the key concept is that we do not want to fill buffers. We want to inject traffic as close as possible to the rate at which the network and target receiver can service the incoming traffic.

During this phase, the *congestion window* is much smaller than the receive window. This means the sender controls the traffic injected into the receiver by computing the congestion window and capping the injected traffic amount by the size of the congestion window. Any minor bursts can be absorbed by queues. FIGURE 3-9 shows what happens during a typical TCP session starting from idle.

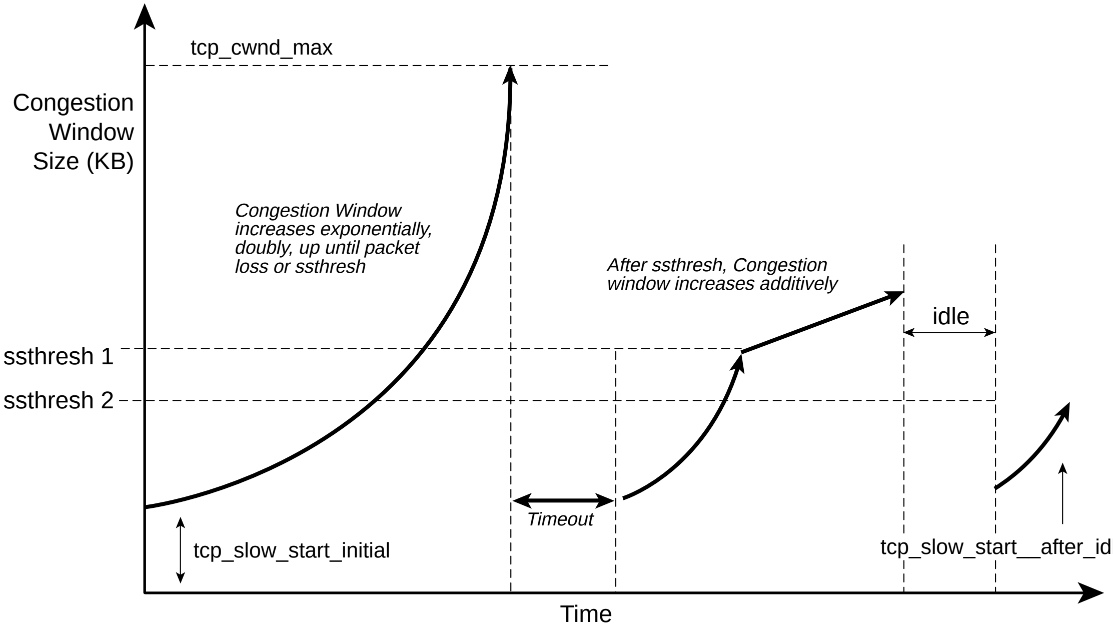

tcp_cwnd_max
Congestion Window Size (KB)
Congestion Window increases exponentially, doubly, up until packet loss or ssthresh
After ssthresh, Congestion window increases additively
idle
ssthresh 1
ssthresh 2
Timeout
tcp_slow_start_initial
tcp_slow_start__after_idle
Time

FIGURE 3-9 TCP Startup Phase

The sender does not know the capacity of the network, so it starts to slowly send more and more packets into the network trying to estimate the state of the network by measuring the arrival time of the ACK and computed RTT times. This results in a self-clocking effect. In FIGURE 3-9, we see the congestion window initially starts with a minimum size of the maximum segment size (MSS), as negotiated in the three-way handshake during the socket connection phase. The congestion window is doubled every time an ACK is returned within the timeout. The congestion window is capped by the TCP tunable variable `tcp_cwnd_max`, or until a timeout occurs. At that point, the `ssthresh` internal variable is set to half of `tcp_cwnd_max`. `ssthresh` is the point where upon a retransmit, the congestion window grows exponentially. After this point it grows additively, as shown in FIGURE 3-9. Once a timeout occurs, the packet is retransmitted and the cycle repeats.

FIGURE 3-9 shows that there are three important TCP tunable parameters:

- `tcp_slow_start_initial`: sets up the initial congestion window just after the socket connection is established.
- `tcp_slow_start_after_idle`: initializes the congestion window after a period of inactivity. Since there is some knowledge now about the capabilities of the network, we can take a shortcut to grow the congestion window and not start from zero, which takes an unnecessarily conservative approach.

- `tcp_cwnd_max`: places a cap on the running maximum congestion window. If the receive window grows, then `tcp_cwnd_max` grows to the receive window size.

In different types of networks, you can tune these values slightly to impact the rate at which you can ramp up. If you have a small network pipe, you want to reduce the packet flow, whereas if you have a large pipe, you can fill it up faster and inject packets more aggressively.

Steady State Phase

In Steady State Phase, after the connection has stabilized and completed the initial startup phase, the socket connection reaches a phase that is fairly steady and tuning is limited to reducing delays due to network and client congestion. An average condition must be used because there are always some fluctuations in the network and client data that can be absorbed. Tuning TCP in this phase, we look at the following network properties:

- **Propagation Delay** – This is primarily influenced by distance. This is the time it takes one packet to traverse the network. In WANs, tuning is required to keep the pipe as full as possible, increasing the allowable outstanding packets.
- **Link Speed** – This is the bandwidth of the network pipe. Tuning guidelines for link speeds from 56kbit/sec dial-up connections differ from 10Gbit/sec optical local area networks (LANs).

In short, tuning will be adjusted according to the type of network and associated key properties: propagation delay, link speed, and error rate. These properties actually self-adjust in some instances by measuring the return of acknowledgments. We will look at various emerging network technologies: optical WAN, LAN, wireless, and so on—and describe how to tune TCP accordingly.

TCP Congestion Control and Flow Control – Sliding Windows

One of the main principles for congestion control is avoidance. TCP tries to detect signs of congestion before it happens and to reduce or increase the load into the network accordingly. The alternative of waiting for congestion and then reacting is much worse because once a network saturates, it does so at an exponential growth rate and reduces overall throughput enormously. It takes a long time for the queues to drain, and then all senders again repeat this cycle. By taking a proactive congestion avoidance approach, the pipe is kept as full as possible without the danger of network saturation. The key is for the sender to understand the state of the network and client and to control the amount of traffic injected into the system.

Flow control is accomplished by the receiver sending back a window to the sender. The size of this window, called the receive window, tells the sender how much data to send. Often, when the client is saturated, it might not be able to send back a receive window to the sender to signal it to slow down transmission. However, the *sliding windows* protocol is designed to let the sender know, before reaching a meltdown, to start slowing down transmission by a steadily decreasing window size. At the same time these flow control windows are going back and forth, the speed at which ACKs come back from the receiver to the sender provides additional information to the sender that caps the amount of data to send to the client. This is computed indirectly.

The amount of data that is to be sent to the remote peer on a specific connection is controlled by two concurrent mechanisms:

- The congestion in the network - The degree of network congestion is inferred by the calculation of changes in Round Trip Time (RTT): that is the amount of delay attributed to the network. This is measured by computing how long it takes a packet to go from sender to receiver and back to the client. This figure is actually calculated using a running smoothing algorithm due to the large variances in time. The RTT value is an important value to determine the *congestion window*, which is used to control the amount of data sent out to the remote client. This provides information to the sender on how much traffic should be sent to this particular connection based on network congestion.
- Client load - The rate at which the client can receive and process incoming traffic. The client sends a *receive window* that provides information to the sender on how much traffic should be sent to this connection based on client load.

TCP Tuning for ACK Control

FIGURE 3-10 shows how senders and receivers control ACK waiting and generation. The general strategy is that clients want to reduce receiving many small packets. Receivers try to buffer up a bunch of received packets before sending back an acknowledgment (ACK) to the sender, which will trigger the sender to send more packets. The hope is that the sender will also buffer up more packets to send in one large chunk rather than many small chunks. The problem with small chunks is that the efficiency ratio or useful link ratio utilization is reduced. For example, a one-byte data packet requires 40 bytes of IP and TCP header information and 48 bytes of Ethernet header information. The ratio works out to be 1/(88+1) = 1.1 percent utilization. When a 1500-byte packet is sent, however, the utilization can be 1500/(88+1500) = 94.6 percent. Now, consider many flows on the same Ethernet segment. If all flows are small packets, the overall throughput is low. Hence, any effort to bias the transmissions towards larger chunks without incurring excessive delays is a good thing, especially interactive traffic such as Telnet.

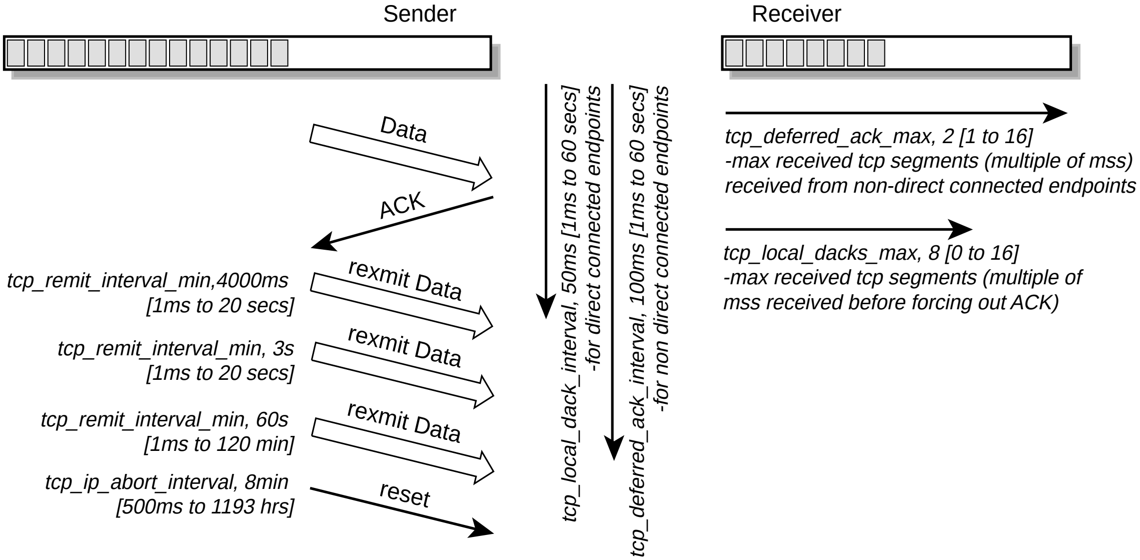

Sender
Receiver
Data
ACK
tcp_remit_interval_min,4000ms [1ms to 20 secs]
rexmit Data
tcp_remit_interval_min, 3s [1ms to 20 secs]
rexmit Data
tcp_remit_interval_min, 60s [1ms to 120 min]
rexmit Data
tcp_ip_abort_interval, 8min [500ms to 1193 hrs]
reset
tcp_local_dack_interval, 50ms [1ms to 60 secs] -for direct connected endpoints
tcp_deferred_ack_interval, 100ms [1ms to 60 secs] -for non direct connected endpoints
tcp_deferred_ack_max, 2 [1 to 16] -max received tcp segments (multiple of mss) received from non-direct connected endpoints
tcp_local_dacks_max, 8 [0 to 16] -max received tcp segments (multiple of mss received before forcing out ACK)

FIGURE 3-10 TCP Tuning for ACK Control

FIGURE 3-10 provides an overview of the various TCP parameters. For a complete detailed description of the tunable parameters and recommended sizes, refer to your product documentation or the Solaris AnswerBooks at `docs.sun.com`.

There are two mechanisms that are used by senders and receivers to control performance:

- **Senders—*timeouts waiting for ACK***. This class of tunable parameters controls various aspects of how long to wait for the receiver to send back an ACK of the data that was sent. If tuned too short, then excessive retransmissions occur. If tuned too long, then excess wasted idle time elapses before the sender realizes the packet was lost and retransmits.
- **Receivers—*timeouts and number of bytes received before sending an ACK to sender***. This class of tunable parameters allows the receiver to control the rate at which the sender sends data. The receiver does not want to send an ACK for every packet received because the sender will send many small packets, increasing the ratio of overhead to actual useful data ratio and reducing the efficiency of the transmission. However, if the receiver waits too long, there is excess latency that increases the burstiness of the communication. The receiver side can control ACKs with two overlapping mechanisms based on timers and the number of bytes received.

TCP Example Tuning Scenarios

The following sections describe example scenarios where TCP require tuning, depending on the characteristics of the underlying physical media.

Tuning TCP for Optical Networks – WANS

Typically, WANS are high-speed, long-haul network segments. These segments introduce some interesting challenges because of their properties. FIGURE 3-11 shows how the traffic changes as a result of a longer, yet faster, link, comparing a normal LAN and an Optical WAN. The line rate has increased, resulting in more packets per unit time, but the delays have also increased from the time a packet leaves the sender to the time it reaches the receiver. This has the strange effect that more packets are now in flight.

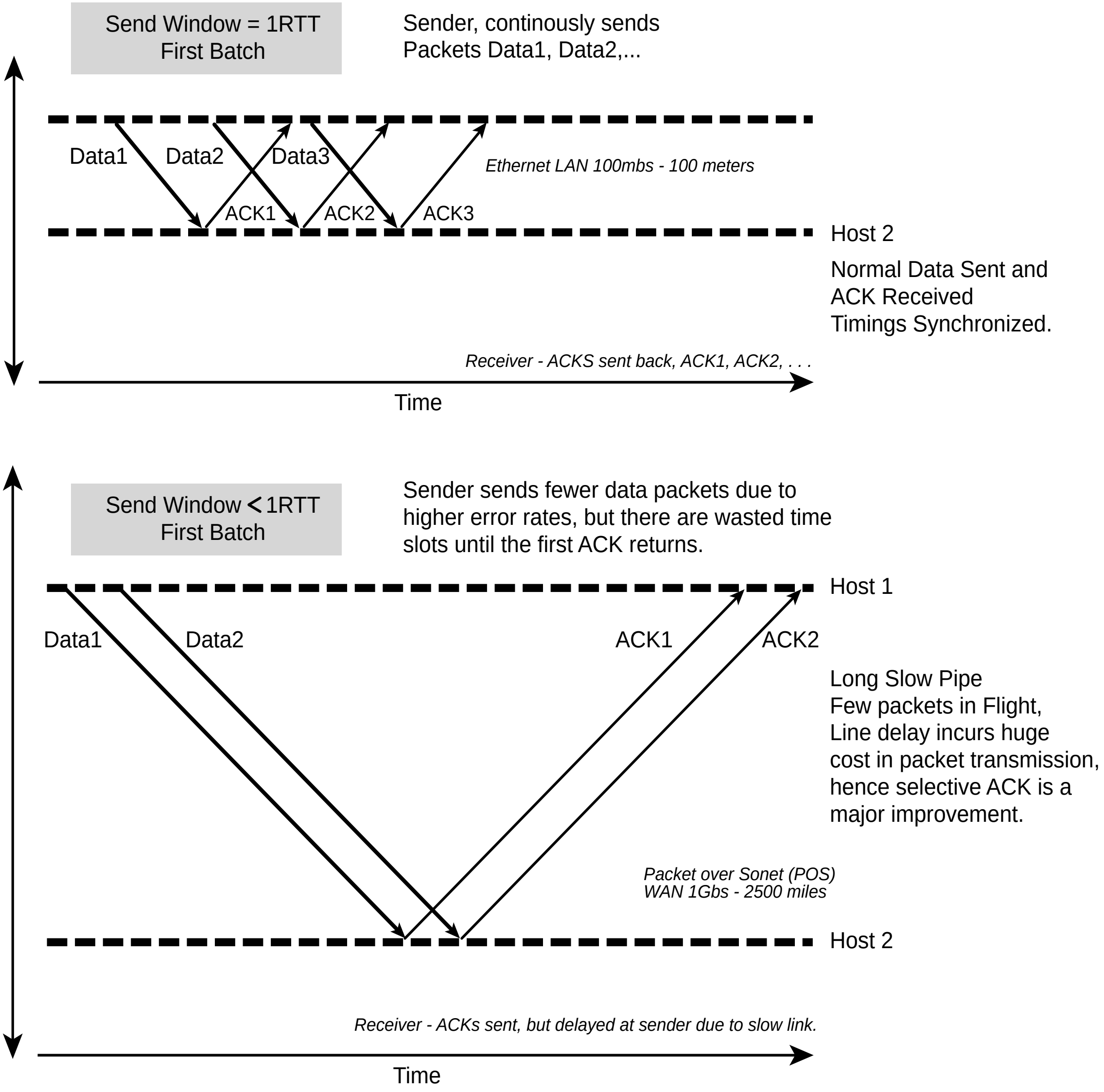


FIGURE 3-11 Comparison between Normal LAN and WAN Packet Traffic

FIGURE 3-11 shows a comparison of the number of packets that are in the pipe between a typical LAN of 10 mbps/100 meters with RTT of 71 microseconds, which is what TCP was originally designed for, and an optical WAN, which spans New York to San Francisco at the rate of 1 Gbps with RTT of 100 milliseconds. The bandwidth delay product represents the number of packets that is actually in the network and implies the amount of buffering the network must provide. This also gives some insight into the minimum window size, which we discussed earlier. The fact that the optical WAN has a very large bandwidth delay product as compared to a normal network requires tuning as follows:

- The window size must be much larger. The current window size allows for 2^{16} bytes. To achieve larger windows, RFC 1323 was introduced to allow the window size to scale to larger sizes while maintaining backwards compatibility. This is achieved during the initial socket connection, where during the SYN-ACK three-way handshake, window scaling capabilities are exchanged by both sides, and they try to agree on the largest common capabilities. The scaling parameter is an exponent base 2. The maximum scaling factor is 14, hence allowing a maximum window size of 2^{30} bytes. The window scale value is used to shift the window size field value up to a maximum of 1 gigabyte. Like the MSS option, the window scale option should only appear in SYN and SYN-ACK packets during the initial three-way handshake. Tunable parameters include:
  - `tcp_wscale_always:` controls who should ask for scaling. If set to zero, the remote side needs to request; otherwise, the receiver should request.
  - `tcp_tstamp_if_wscale:` controls adding timestamps to the window scale. This parameter is defined in RFC 1323 and used to track the round-trip delivery time for data in order to detect variations in latency, which impact timeout values. Both ends of the connection must support this option.
- During the slow start and retransmissions, the minimum initial window size, which can be as small as one MSS, is too conservative. The send window size grows exponentially, but starting at the minimum is too small for such a large pipe. Tuning in this case requires that the following tunable parameters be adjusted to increase the minimum start window size:
  - `tcp_slow_start_initial:` controls the starting window just after the connection is established.
  - `tcp_slow_after_idle:` controls the starting window after a lengthy period of inactivity on the sender side.

Both of these parameters must be manually increased according to the actual WAN characteristics. Delayed ACKs on the receiver side should also be minimized because this will slow the increasing of the window size when the sender is trying to ramp up.

RTT measurements require adjustment less frequently due to the long RTT times, hence interim additional RTT values should be computed. The tunable `tcp_rtt_updates` parameter is somewhat related. The TCP implementation knows when enough RTT values have been sampled, and then this value is cached. `tcp_rtt_updates` is on by default, but a value of 0 forces it to never be cached, which is the same as the case of not having enough for an accurate estimate of RTT for this particular connection.

- `tcp_recv_hiwat` and `tcp_xmit_hiwat:` control the size of the STREAMS queues before STREAMS-based flow control is activated. With more packets in flight, the size of the queues must be increased to handle the larger number of outstanding packets in the system.

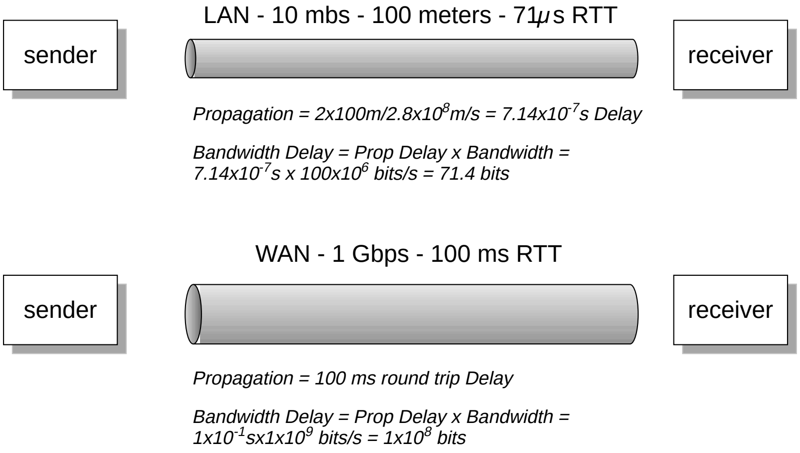
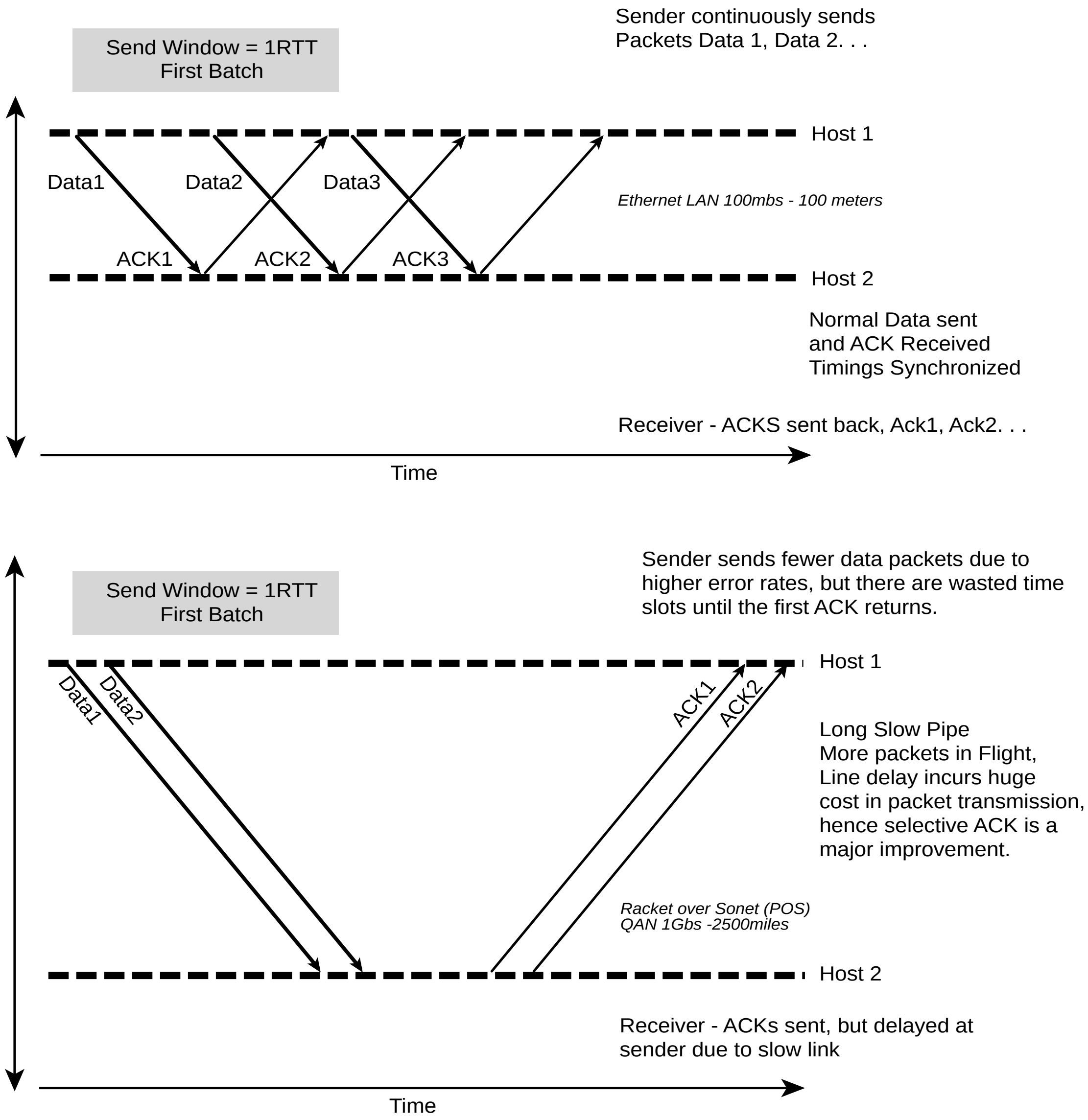

LAN - 10 mbs - 100 meters - 71μs RTT
sender
receiver
Propagation = 2x100m/2.8x10⁸m/s = 7.14x10⁻⁷s Delay
Bandwidth Delay = Prop Delay x Bandwidth =
7.14x10⁻⁷s x 100x10⁶ bits/s = 71.4 bits
WAN - 1 Gbps - 100 ms RTT
sender
receiver
Propagation = 100 ms round trip Delay
Bandwidth Delay = Prop Delay x Bandwidth =
1x10⁻¹sx1x10⁹ bits/s = 1x10⁸ bits

FIGURE 3-12 Tuning Required to Compensate for Optical WAN

Tuning TCP for Slow Links

Wireless and satellite networks have a common problem of a higher bit error rate. One tuning strategy to compensate for the lengthy delays is to increase the send window, sending as much data as possible until the first ACK arrives. This way, the link is utilized as much as possible. FIGURE 3-13 shows how slow links and normal links differ. If the send window is small, then there will be significant dead time between the time the send window sends packets over the link and the time an ACK arrives and the sender can either retransmit or send the next window of packets in the send buffer. But due to the increased error probability, if one byte is not acknowledged by the receiver, the entire buffer must be re-sent. Hence, there is a trade-off to increase the buffer to increase throughput. But you don't want to increase it so much that if there is an error the performance is degraded by more than was gained due to retransmissions. This is where manual tuning comes in. You'll need to try various settings based on an estimation of the link characteristics. One major improvement in TCP is the selective acknowledgement (SACK), where only the one byte that was not received can be retransmitted, not the entire buffer.



Sender continuously sends
Packets Data 1, Data 2. . .

Normal Data sent
and ACK Received
Timings Synchronized

Receiver - ACKS sent back, Ack1, Ack2. . .

Sender sends fewer data packets due to
higher error rates, but there are wasted time
slots until the first ACK returns.

Long Slow Pipe
More packets in Flight,
Line delay incurs huge
cost in packet transmission,
hence selective ACK is a
major improvement.

Receiver - ACKs sent, but delayed at
sender due to slow link

FIGURE 3-13 Comparison between Normal LAN and WAN Packet Traffic—Long Low Bandwidth Pipe

Another problem introduced in these slow links is that the ACKs play a major role. If ACKs are not received by the sender in a timely manner, the growth of windows is impacted. During initial slow start, and even slow start after an idle, the send window needs to grow exponentially, adjusting to the link speed as quickly as possible for coarser tuning. It then grows linearly after reaching `ssthresh` for finer-grained tuning. However, if the ACK is lost, which has a higher probability in these types of links, then the performance throughput is again degraded.

Tuning TCP for slow links includes the following parameters:

- `tcp_sack_permitted:` activates and controls how SACK will be negotiated during the initial three-way handshake:
  - 0 = no sack – disabled.
  - 1 = TCP will not initiate a connection with SACK information, but if an incoming connection has the SACK-permitted option, TCP will respond with SACK information.
  - 2 = TCP will both initiate and accept connections with SACK information.

  TCP SACK is specified in RFC 2018 TCP selective acknowledgement. TCP need not retransmit the entire send buffer, only the missing bytes. Due to the higher cost of retransmission, it is far more efficient to only re-send the missing bytes to the receiver.

  Like optical WANs, satellite links also require the window scale option to increase the number of packets in flight to achieve higher overall throughput. However, satellite links are more susceptible to bit errors, so too large a window is not a good idea because one bad byte will force a retransmission of one enormous window. TCP SACK is particularly useful in satellite transmissions to avoid this problem because it allows the sender to select which packets to retransmit without requiring an entire window (which contained that one bad byte) for retransmission.
- `tcp_dupack_fast_retransmit`: controls the number of duplicate ACKs received before triggering the fast recovery algorithm. Instead of waiting for lengthy timeouts, fast recovery allows the sender to retransmit certain packets, depending on the number of duplicate ACKs received by the sender from the receiver. Duplicate ACKs are an indication that possibly later packets have been received, but the packet immediately after the ACK might have been corrupted or lost.

Adjust all timeouts to compensate for long-delay satellite transmissions and possibly longer-distance WANs; the timeout values must be compensated.

TCP and RDMA Future Data Center Transport Protocols

TCP is ideally suited for reliable end-to-end communications over disparate distances. However, it is less than ideal for intra-data center networking primarily because over-conservative reliability processing drains CPU and memory resources, thus impacting performance. During the last few years, networks have grown faster in terms of speed and reduced cost. This implies that the computing systems are now the bottleneck—not the network—which was not the case prior to the mid-1990s. Two issues have resulted due to multi-gigabit network speeds:

- Interrupts generated to the CPU – The CPU must be fast enough to service all incoming interrupts to prevent losing any packets. Multi-CPU machines can be used to scale. However, the PCI bus then introduces some limitations. It turns out that the real bottleneck is memory.
- Memory Speed – An incoming packet must be written and read from the NIC to the operating system kernel address space to the user address. You can reduce the number of memory-to-memory copies to achieve zero copy TCP by using workarounds such as page flipping, direct data placement, and scatter-gather I/O. However, as we approach 10-gigabit Ethernet interfaces, memory speed continues to be a source of performance issues. The main problem is that over the last few years, memory densities have increased, but not speed. Dynamic random access memory (DRAM) is cheap but slow. Static random access memory (SRAM) is fast but expensive. New technologies such as reduced latency DRAM (RLDRAM) show promise, but these seem to be dwarfed by the increases in network speeds.

To address this concern, there have been some innovative approaches to increase the speed and reduce the network protocol processing latencies in the area of remote direct memory access (RDMA) and infiniband. New startup companies such as Topspin are developing high-speed server interconnect switches based on infiniband and network cards with drivers and libraries that support RDMA, Direct Access Programming Library (DAPL), and Sockets Direct Protocol (SDP). TCP was originally designed for systems where the networks were relatively slow as compared to the CPU processing power. As networks grew at a faster rate than CPUs, TCP processing became a bottleneck. RDMA fixes some of the latency.

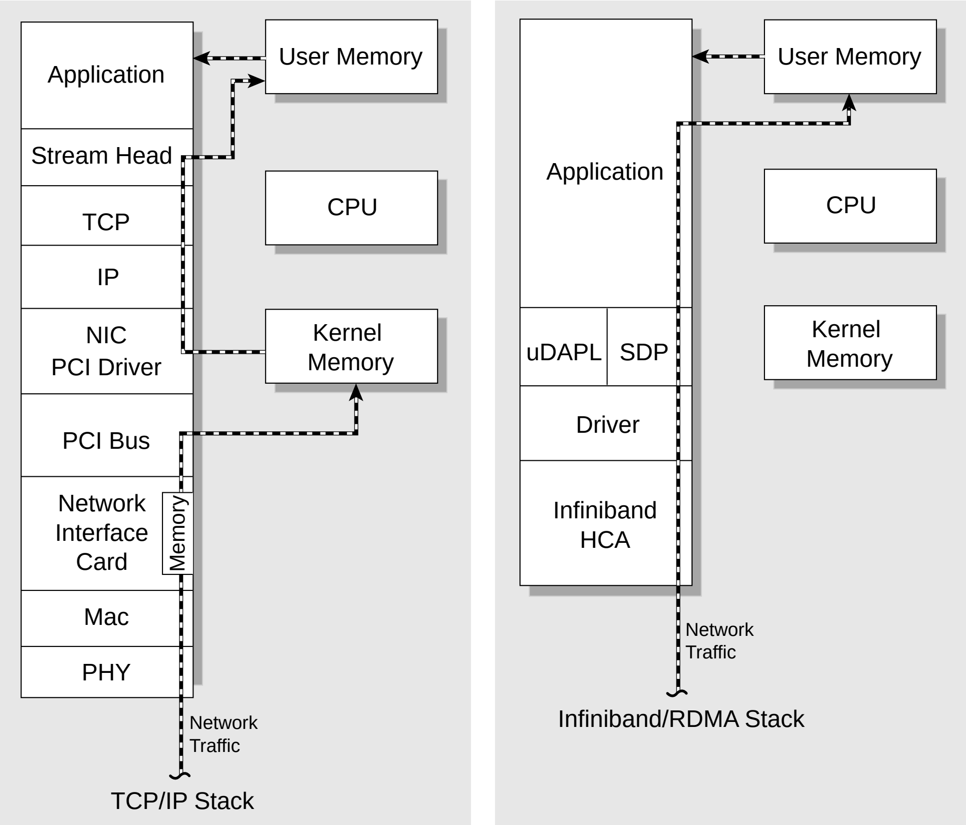


FIGURE 3-14 Increased Performance of InfiniBand/RDMA Stack

FIGURE 3-14 shows the difference between the current network stack and the new-generation stack. The main bottleneck in the traditional TCP stack is the number of memory copies. Memory access for DRAM is approximately 50 ns for setup and then 9 ns for each subsequent write or read cycle. This is orders of magnitude longer than the CPU processing cycle time, so we can neglect the TCP processing time. Saving one memory access on every 64 bits results in huge savings in message transfers. Infiniband is well suited for data center local networking architectures, as both sides must support the same RDMA technology.