Foundations of TCP

A six-module workshop on the Transmission Control Protocol

Network Engineering Workshop

August 1, 2026

INSTRUCTOR EDITION

Contents

Welcome 3
A. Overview 3
B. Schedule 3
C. Expectations for completion 3
D. Affirmations for completion 4
Why Reliability Matters 5
A. **Every** packet on a real network can be **lost, reordered, duplicated, or corrupted** 5
B. A definition of reliability 5
C. Reliability begins with the realities of **the IP layer** 5
D. TCP defines the nature of **the byte stream abstraction** 6
E. Reliability culminates in the **end-to-end principle** 8
The Connection Story 9
A. What is a connection again? 9
B. The connection is meant to **frame** every byte we send 9
C. Below the abstraction: the **TCP segment header** 11
D. We expect every connection to **transition** through a defined set of states 12
E. Endpoints become participants in TCP's **larger story** through their sockets 13
The Connection State Machine 15
A. Definition of a connection 15
B. A connection is a **four-tuple,** unique on the wire 15
C. A connection is a **pair of sequence-number spaces** governed by the windows – RFC 9293 §3.4 16
D. A connection is a **bidirectional contract** until both sides FIN – RFC 9293 §3.6 16
E. A connection is the **smallest unit** of TCP behavior, holding the contract between endpoints 16
F. The **mission** of a connection is to deliver bytes reliably end-to-end – RFC 9293 §1.5 17
G. Reference reading on connection lifecycle 17
Flow & Congestion Control in Practice 18
A. How do we treat **the network** in our designs? 18
B. What are the **rhythms** of a healthy connection? 18
Reliability Made Visible in Hardware 20
A. A pragmatic example - **the line driver** 20
B. A second example - **the supply rail decoupling** 20
Specification Statement 21
A. The Purpose of a specification statement 21
B. The Service 21
C. The Specification 21
D. The doctrine of segments 22
E. Sequence numbers 22
F. Acknowledgments 22
G. Flow control 23
H. Congestion control 23
I. Reset and abort 23
J. Things to come 23
Compatibility with Real Networks 24
A. The goal 24
B. Middleboxes and NAT 24
C. Wireless and lossy links 24
D. Operational visibility 24
Sample Capture & Code Repository 25
A. Explanation 25
B. Practices 25

Welcome

A. Overview

We are excited that you are diving deeply into one of the most consequential pieces of software ever written. Vint Cerf and Bob Kahn's 1974 paper *A Protocol for Packet Network Intercommunication* sketched the kernel of what would become TCP, and nearly every byte that crosses the modern Internet still passes through a direct descendant of that design. The fact that you are here, learning the protocol from the inside out, is one more sign that the engineering tradition behind it is alive and well.

The purpose of this workshop is to help you build a thorough mental model of TCP — its goals, its mechanisms, and the trade-offs the designers made. We are not trying to sell you on TCP as a silver bullet. Rather, our aim is to explain what the specification actually says, why it says it, and how to reason about TCP behavior when you are debugging a stalled connection at 2 AM.

B. Schedule

MONDAY	Why Reliability Matters
TUESDAY	The Connection Story
WEDNESDAY	The Connection State Machine
THURSDAY	Flow & Congestion Control
FRIDAY	Specification Statement
SATURDAY	Capstone Lab & Wireshark Walk-Through

Facilitator note: Wednesday runs long. Start the state-machine lab immediately after lunch and hold the last twenty minutes for the packet-trace warm-up.

All who complete the modules, work through the lab notebooks, and pass the capstone packet-trace exercise will be presented with a workshop completion certificate on **Demo Day**. After the demo, all attendees and their teams are invited to a networking lunch.

C. Expectations for completion

- Mark on the attendance sheet that you intend to complete the certificate so we can pair you with a lab partner.
- If you haven't set up a Linux VM with `tcpdump` and Wireshark yet, please do so before module 2.
- Participate in all six Foundations of TCP modules.
- Read RFC 9293 (Transmission Control Protocol) — at minimum sections 3.1–3.10.
- Submit a packet-capture exercise summary.
- Write a one-page reflection on a TCP-related production incident you have seen or read about.
- Meet with a workshop mentor before Demo Day.

> If you have not been writing networked code for at least six months, you are still welcome to attend, but we kindly ask you to wait to take the certificate exam until the next cohort. We hold a new cohort every four months. We want time to get to know you as engineers before signing off on your work, and we want you to have time to absorb the material without feeling rushed. That said, completion is not a one-way door — engineers who pass the exam frequently return as mentors. We don't hold tightly to alumni; if your career leads you elsewhere, we trust your judgment. But we believe knowing one another well is key to a healthy engineering community.

D. Affirmations for completion

We expect every workshop participant to affirm the following statements, which we hold dear as core engineering convictions that unite us as practitioners.

- Do you acknowledge that networks are fundamentally unreliable, that packets are routinely lost, reordered, duplicated, and corrupted, and that no amount of hope substitutes for explicit acknowledgments?
- Do you believe that protocol specifications, being precise and testable, are the ultimate authority on correct behavior — above intuition, above tribal knowledge, and above what a particular implementation happens to do?
- Do you commit to reading RFCs in their original form rather than relying solely on secondary explanations, and to citing section numbers when you make claims about protocol behavior?
- Do you pledge to always grow in your knowledge of the wire-level reality of the systems you build, depending on packet captures and reproducible experiments rather than guesswork?
- Do you pledge to take up the discipline of writing code that is robust to the worst the network can throw at it, treating every recv() as potentially partial and every send() as potentially blocked?
- Do you commit to collaborating openly with other engineers, sharing captures and reproductions, and watching out for one another when systems misbehave under load?
- For the cohort: Do you commit to welcoming new members, encouraging their curiosity, and working together to represent good engineering practice in the broader community?

Why Reliability Matters

Reliable networking demands clarity about the underlying medium and clarity about the contracts a transport protocol offers, because everything that runs on TCP — HTTP, TLS, SSH, SMTP — leans on those contracts.

A. **Every** packet on a real network can be **lost, reordered, duplicated, or corrupted**

An unreliable substrate is the key reality that makes a transport protocol necessary in the first place.

In RFC 1122 and again in RFC 9293, the IETF emphasizes a **robust transport layer**: the network underneath TCP is permitted to do almost anything short of inventing data out of thin air. Therefore, it is of utmost importance that we grasp the failure modes with engineering clarity, and that each protocol mechanism we study has been justified against a specific failure it was designed to address.

B. A definition of reliability

TCP provides a **reliable, ordered, byte-stream** service over an unreliable packet-switched network. Concretely, this means that bytes written by one endpoint will, barring a connection abort, be delivered to the peer exactly once, in order, with corrupted segments retransmitted, duplicate segments discarded, and reordered segments reassembled — without the application having to manage any of that.

C. Reliability begins with the realities of **the IP layer**

There are many ways to break down what a transport must do. The contract does not change, however: the IP layer makes only a best-effort delivery promise, and TCP must compensate.

1. IP must be **honestly understood**

- IP is a *best-effort* protocol; it does not retransmit, reorder, or detect duplicates – RFC 791 §1.4
- If we do not respect what IP guarantees and what it doesn't, we will design protocols that work in the lab and fail in the wild – RFC 1122 §1.1.3

2. The IP layer must be **thoroughly understood**

- IP is **packet-switched**, not circuit-switched – RFC 791 §1.1

  There is no reserved path between sender and receiver. Each datagram is routed independently and may take a different path than its neighbors.

- IP datagrams may be **fragmented** in transit – RFC 791 §3.2

  A router whose outbound MTU is smaller than the datagram may fragment it. Modern stacks rely on Path MTU Discovery to avoid this where possible.

- IP routes can **change** mid-connection – RFC 1122 §3.3.4

  Routes are not fixed. A long-lived TCP connection may traverse a different path each minute as routing tables update.

- IP datagrams have a **TTL** that decrements at each hop – RFC 791 §3.2

  The TTL prevents stale packets from circulating forever; this is the foundation that justifies TCP's 2×MSL `TIME-WAIT`.

- IP delivery is **unordered** – RFC 791 §1.4

  Two datagrams sent back-to-back may arrive in either order or not at all.

- IP delivery is **non-exclusive** – RFC 1122 §3.2.2

  The same datagram may be duplicated by a misbehaving link or replicated by routing pathologies; receivers must tolerate seeing the same bytes twice.

- IP can **silently corrupt** payloads – RFC 1122 §4.1.3.4

  Although IP has a header checksum, the payload checksum is the responsibility of the transport. End-to-end checksums are not optional in practice.

- IP says **nothing** about congestion – RFC 2914 §1

  The network can drop packets when overloaded, but offers no mechanism to throttle senders. That responsibility falls on TCP.

- IP is **stateless** at routers – RFC 791 §1.4

  Routers do not remember individual flows. Any per-connection state lives at the endpoints.

- IP supports **multiple address families** – RFC 8200 §1

  IPv4 and IPv6 differ in addressing, fragmentation, and option handling, but TCP runs over both unchanged.

  IP is exactly what the specification says it is — no more, no less. Its best-effort nature does not negate its usefulness, nor does its statelessness imply unreliability of the network as a whole — it simply pushes reliability to the endpoints, where TCP picks it up.

  Unfortunately, if the underlying medium is not honestly understood, no transport protocol layered above it will behave well under stress.

D. TCP defines the nature of **the byte stream abstraction**

COMMON BELIEF TCP delivers messages, framed by send() and recv() boundaries.

REALITY TCP delivers an undifferentiated byte stream; framing is the application's problem.

1. TCP clearly defines the **service** it provides
   - TCP presents a stream, not a message – RFC 9293 §3.1

     A single send() may be split across many segments and a single recv() may return data from several sends. Applications that need message boundaries must add length prefixes or delimiters.
   - TCP is **full-duplex** – RFC 9293 §3.1

     Each endpoint can send and receive simultaneously; the two directions are largely independent.
   - TCP is **connection-oriented** – RFC 9293 §3.5

     Before any data can flow, a three-way handshake establishes shared state on both endpoints.
   - TCP is **point-to-point** – RFC 9293 §3.1

     Each connection has exactly two endpoints; there is no broadcast or multicast variant of TCP.
2. TCP rightly defines the **guarantees** of the stream
   - Bytes will arrive **in order** – RFC 9293 §3.4

     The sequence-number machinery ensures the receiver presents bytes to the application in the order the sender placed them in the stream.
   - Bytes will arrive **without duplication** – RFC 9293 §3.4

     The receiver discards segments whose sequence numbers fall before the next expected byte.
   - Lost bytes will be **retransmitted** – RFC 9293 §3.8.1

     Until the sender sees an acknowledgment, it will retransmit data on a timer. Acknowledgments are cumulative.
   - Connections are **closed cleanly** – RFC 9293 §3.6

     Both directions are torn down explicitly; an unexpected close indicates a real fault.

     A protocol that does not provide all of these guarantees is a different protocol. UDP, QUIC, SCTP, and DCCP each make different trade-offs and are useful in their place — but TCP's contract is what it is, and applications written against TCP should rely on exactly what TCP promises and no more.

E. Reliability culminates in the **end-to-end principle**

1. The Internet is built on **endpoint responsibility**

- Reliability is properly an end-to-end property – Saltzer, Reed & Clark (1984)
- The network can help with performance, but cannot guarantee correctness on behalf of endpoints – RFC 1958 §2.1

2. Reliability is achieved by **acknowledgment and retransmission**

- Every byte sent is sequenced; every received byte is acknowledged – RFC 9293 §3.4
- Without an acknowledgment within a calculated timeout, the sender retransmits – RFC 6298 §2

  TCP's retransmission timeout (RTO) is calculated from a smoothed RTT estimate plus a multiple of its variance.

- Lost retransmissions are themselves retransmitted, with the timer doubled – RFC 6298 §5

♦

Reliability demands clarity about what the lower layer guarantees and what it does not, because every higher-level protocol assumes TCP got the basics right.

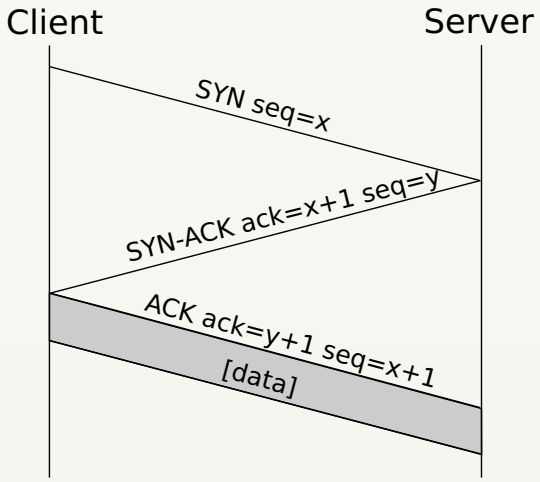
The Connection Story

TCP is more than a checksum and a sequence number; it is the story of how two endpoints come to agreement, exchange data, and part ways without losing track of what has and hasn't been delivered.

A. What is a connection again?

We can never remind ourselves of the basic mechanics too often!

A TCP connection is identified by the four-tuple `(source IP, source port, destination IP, destination port)` and is bound to a pair of *sequence-number spaces* — one for each direction. Once established, the connection persists until both directions are closed or until the connection is aborted.



TCP three-way handshake (Wikimedia Commons)

B. The connection is meant to **frame** every byte we send

1. A handshake establishes shared **initial sequence numbers**

- The handshake is the SYN, SYN-ACK, ACK exchange – RFC 9293 §3.5
- Initial sequence numbers should be unpredictable to mitigate spoofing – RFC 6528 §3
- Three-way is the minimum that defends against delayed duplicates – RFC 9293 §3.4.1

  A two-way handshake cannot reject an old SYN that arrives late. The third ACK is what closes that loophole.

2. How does the handshake **protect us**?
   - From accepting spurious segments from a previous incarnation of the same four-tuple – RFC 9293 §3.4.1

     The 2×MSL `TIME-WAIT` and the unpredictability of ISN together ensure that segments from a previous connection are not mistaken for the new one.
   - From off-path attackers injecting data – RFC 5961 §1

     By guessing only one of the two sequence-number spaces, an attacker cannot inject acceptable data; they would have to guess both.
   - From establishing connections to peers that aren't actually listening – RFC 9293 §3.5.2

     A peer that has not opened the port responds with RST, which the initiator treats as an immediate failure.
3. As we **send** bytes, the receiver **acknowledges**
   - Cumulative ACKs say "I have received everything up through byte N" – RFC 9293 §3.4
   - Selective ACKs (SACK) communicate that out-of-order ranges have been received – RFC 2018 §3

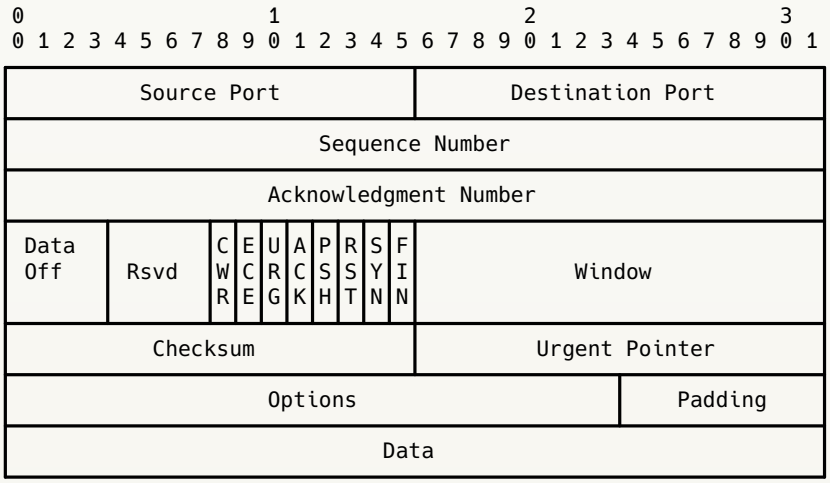
     SACK lets the sender retransmit only the missing range, rather than everything from the loss point onward.
   - The receive window tells the sender how much more it may send without waiting – RFC 9293 §3.4

     A zero window halts the sender until the receiver advertises space.

     Each segment carries one or more of these signals: SYN, ACK, FIN, RST, PSH, URG. The flags are the protocol's vocabulary; the sequence and acknowledgment numbers are the grammar.

C. Below the abstraction: the **TCP segment header**

1. A TCP segment carries a fixed twenty-byte header plus options



TCP Segment Header (RFC 9293 §3.1)

The header is small because the design intent is to make TCP cheap on the wire and rich in semantics — every field earns its place.

2. Famous quotes worth memorizing

> *Be conservative in what you do, be liberal in what you accept from others.* — RFC 793 §2.10

Postel's Robustness Principle

> *At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability: be conservative in what you send, be liberal in what you receive.* — RFC 1122 §1.2.2

RFC 1122 reminds us

> *The Internet is held together by software, not by central coordination, and the only reason it scales is that software is willing to drop, retransmit, and try again.* — paraphrased

David Clark, MIT

D. We expect every connection to **transition** through a defined set of states

1. Why are state machines so powerful?

- State machines are precise

  A connection is in exactly one state at a time, and each state defines exactly which segments are valid and what the response should be. Ambiguity is the enemy of debugging.

- State machines are testable

  Because the rules are explicit, you can write conformance tests that drive an implementation through every transition and assert the correct output.

- An example

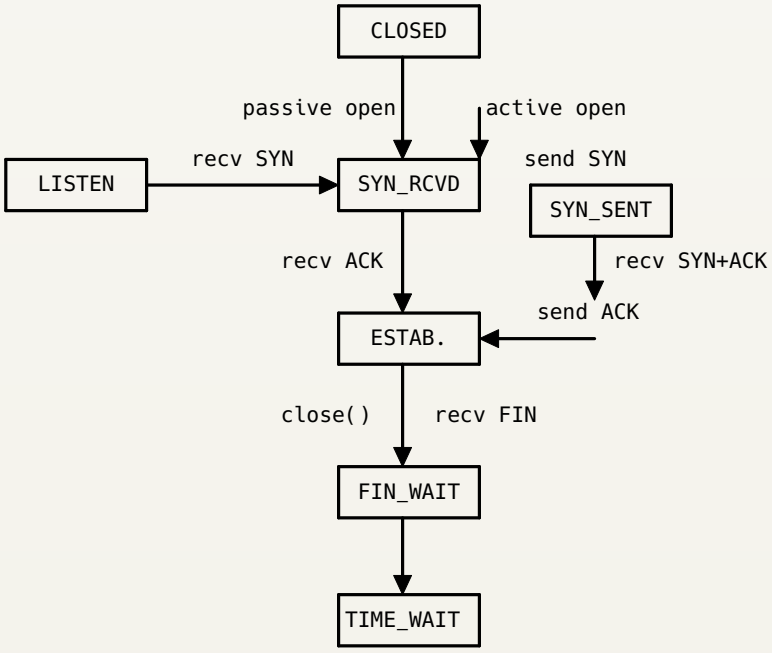
  Two engineers can argue all afternoon about whether a server is "stuck"; but if you can show that the connection is in `CLOSE-WAIT` and the server has not yet called close(2), the argument ends.

2. Where the state machine is defined

- TCP state transitions are normative – RFC 9293 §3.6

  The diagram in section 3.6 of RFC 9293 is not illustrative; it is the definition.

- A reference state diagram in ASCII



TCP States (simplified)

E. Endpoints become participants in TCP's **larger story** through their sockets

1. We code against TCP through the **sockets API** – RFC 3493 §3

We are made participants in this protocol when our applications call socket(2), bind(2), listen(2), accept(2), connect(2), send(2), and recv(2). The kernel manages the state machine for us.

2. A minimal echo server, in C

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>

int main(void) {
    int s = socket(AF_INET, SOCK_STREAM, 0);
    int yes = 1;
    setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof yes);

    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(7);
    bind(s, (struct sockaddr *)&addr, sizeof addr);
    listen(s, 16);

    for (;;) {
        int c = accept(s, NULL, NULL);
        char buf[1024];
        ssize_t n;
        while ((n = recv(c, buf, sizeof buf, 0)) > 0)
            send(c, buf, n, 0); // echo back
        close(c);
    }
}
```

Echo Server (BSD sockets)

3. Active and passive opens **share** the same state machine – RFC 9293 §3.10.1

The server reaches `ESTABLISHED` via `LISTEN → SYN_RCVD`; the client via `SYN_SENT`. Both arrive in the same state, ready to exchange data.

4. **All implementations** must implement the same state transitions

   A conforming TCP, whether on Linux, BSD, Windows, or a microcontroller running lwIP, must obey the same state machine. Compatibility comes from the specification, not from the implementation.

♦

We want the connection lifecycle to be front and center, because every production incident in TCP eventually traces to a state the operator did not realize the connection was in.

The Connection State Machine

The TCP state machine is the ground truth at the center of every connection — RFC 9293 §3.6

A. Definition of a connection

1. What a connection **is**

DEFINITION A TCP connection is the shared state held by two endpoints, indexed by the four-tuple, sequenced by the initial sequence numbers, and governed by the state machine in RFC 9293 §3.6.

EXPLANATION The connection is thoroughly bilateral — established by mutual handshake, maintained by mutual acknowledgment, and torn down by mutual close. Neither endpoint owns the connection; both do.

2. What a connection **does**

A TCP connection moves bytes reliably and in order. It detects loss, drives retransmission, manages flow and congestion, signals out-of-band conditions via the URG flag, and either closes cleanly or aborts loudly. We will look at flow and congestion control in the next module; in this module we focus on what the connection *is*.

Because a connection is unlike any other object in your program, the specification primarily describes it through four overlapping abstractions: the four-tuple, the sequence-number spaces, the windows, and the state machine.

B. A connection is a **four-tuple,** unique on the wire

1. Each connection is identified by `(srcIP, srcPort, dstIP, dstPort)` – RFC 9293 §3.4

Two connections with the same four-tuple cannot coexist; this is why `TIME-WAIT` exists, to prevent the next incarnation of a four-tuple from receiving stale segments from the previous one.

2. As listening sockets we accept **many** connections sharing one local endpoint – RFC 3493 §6.1

A server listening on `(*, 80)` may have thousands of connections active, each distinguished by its remote endpoint.

3. The kernel uses the four-tuple as a **hash key** – Linux TCP source

On Linux and BSD, the established connection table is a hash keyed on the four-tuple. Lookup is O(1) amortized.

4. Misuse of the four-tuple **is** a real bug – RFC 6056 §3

Failing to randomize source ports on a NAT or load balancer can cause four-tuple collisions and connection failures.

C. A connection is a **pair of sequence-number spaces** governed by the windows – RFC 9293 §3.4

Each direction has an independent 32-bit sequence-number space. Wrapping is real on fast links, which is why timestamps (RFC 7323) extend the effective sequence space.

D. A connection is a **bidirectional contract** until both sides FIN – RFC 9293 §3.6

A half-closed connection is legal: one side has sent FIN, the other has not. The receiver application can keep reading until it has consumed the last byte.

E. A connection is the **smallest unit** of TCP behavior, holding the contract between endpoints

1. The ultimate purpose of a connection is to **deliver bytes reliably** – RFC 9293 §1.5

   Everything else — windows, options, congestion control — exists to support that purpose.

2. TCP **fills** the connection with state – RFC 9293 §3.4

   SND.UNA, SND.NXT, SND.WND, RCV.NXT, RCV.WND, and the rest are all variables maintained per-connection.

3. The connection **transitions** through eleven states – RFC 9293 §3.6

CLOSED	LISTEN	SYN-SENT	SYN-RECEIVED
ESTABLISHED	FIN-WAIT-1	FIN-WAIT-2	CLOSE-WAIT
CLOSING	LAST-ACK	TIME-WAIT	

4. The **rules** govern transitions

   A SYN in the wrong state is rejected with RST; an ACK in `LISTEN` is rejected with RST; a FIN in `ESTABLISHED` moves the receiver to `CLOSE-WAIT`.

5. We are **bound** to peers

   Each connection ties two endpoints together; an action by either is observable by the other within roughly one RTT.

6. We expose the connection **visible** to the application via the socket

   The socket file descriptor is the application's handle on the underlying state. Closing the descriptor triggers the close half of the state machine.

7. We represent the connection's **liveness** through keepalives – RFC 9293 §3.8.4

   A connection without traffic is not detectably dead. Keepalive probes, sent at long intervals, surface dead peers.

F. The **mission** of a connection is to deliver bytes reliably end-to-end – RFC 9293 §1.5

The ultimate goal is for the receiver application to read exactly what the sender application wrote, in order, without duplication.

1. A connection **fragments** the byte stream into segments – RFC 9293 §3.7.1

Segmentation is constrained by the MSS option negotiated at handshake time and by the path MTU.

2. A connection **reassembles** segments at the receiver – RFC 9293 §3.4

Out-of-order segments are buffered until the gap fills; then they are passed to the application as a contiguous run of bytes.

But reassembly without flow control is dangerous, since a fast sender can exhaust the receiver's buffer (see RFC 9293 §3.8.6).

The receive window is the receiver's mechanism for telling the sender to slow down; the congestion window is the sender's self-imposed restraint, based on its inference of network conditions.

G. Reference reading on connection lifecycle

1. RFC 9293 is **canonical** – RFC 9293 §1

RFC 9293 supersedes the venerable RFC 793 and incorporates corrections accumulated over four decades. Keep it within arm's reach; we will help each other follow it.

2. The goal is a **clean** state machine – Stevens, *TCP/IP Illustrated, Vol. 1*

Implementations should not invent states; the specification's eleven states are sufficient. Any "extra" state in an implementation tends to indicate a misunderstanding.

3. We want every transition to be **defensible from the spec** – RFC 9293 §3.6

Each transition should be justifiable from the spec. If you cannot point to the rule, the implementation is probably wrong.

4. We need each other because TCP bugs are **subtle** – Cerf & Kahn (1974)

Every implementation has been refined over decades by review and bug fixing. We fight subtle bugs by sharing packet captures and reading the spec together.

5. The spec is the only **standard** – RFC 9293 §1.3

Implementations may add extensions, but the wire-level contract is fixed by the spec.

♦

The TCP state machine is the centerpiece of every TCP connection — RFC 9293 §3.6

Flow & Congestion Control in Practice

A well-written network service prioritizes correct backpressure as we partner with the kernel to keep the network healthy.

A. How do we treat **the network** in our designs?

1. Since the network is shared, we **rate-limit** ourselves – RFC 5681 §2

Senders must reduce their offered load when they detect loss, because loss is the network telling us it is overloaded.

2. Since buffers are finite, we **respect** the receive window – RFC 9293 §3.8.6

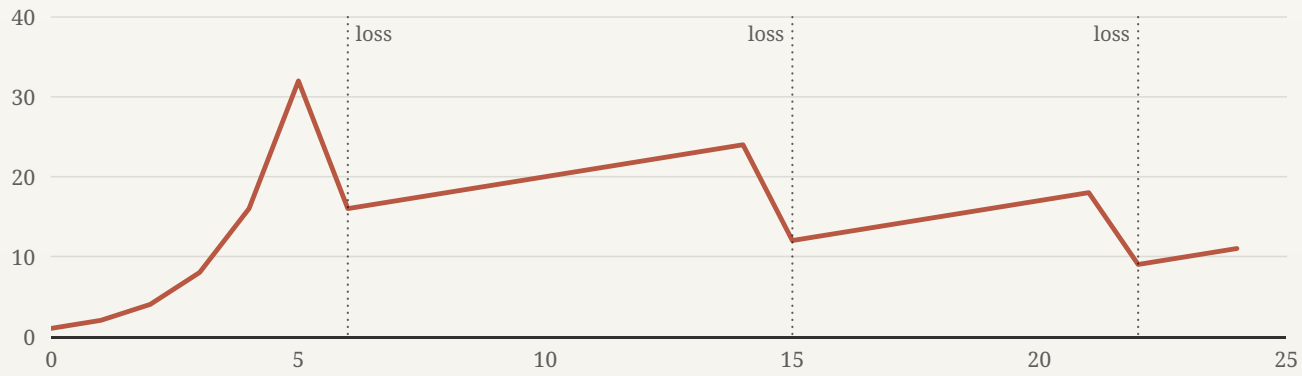
A sender must never send more than the receiver advertised, no matter how fast its own NIC is.

3. Since congestion is real, we **probe gently** – RFC 5681 §3.1

Slow Start grows the congestion window exponentially until loss is observed; this is intentionally cautious.

4. Since fairness matters, we **back off multiplicatively** – RFC 5681 §3.1

On loss, the congestion window is halved (or set to 1 in some recovery paths). The combination of additive increase and multiplicative decrease (AIMD) yields fairness across competing flows.



The AIMD sawtooth — slow start, then additive increase, multiplicative decrease (idealized)

B. What are the **rhythms** of a healthy connection?

A rhythm is a behavior the implementation performs continuously. These behaviors aren't optional in production code.

1. We **ack** what we receive – RFC 9293 §3.8.6

Acknowledgments are how the sender learns its data arrived. Delayed ACKs (up to 500ms) reduce ACK traffic but must not be applied to every segment.

2. We listen for **window updates** – RFC 9293 §3.8.6

A receiver that has freed buffer space sends a window update so the sender can resume.

3. We **retransmit** when ACKs go missing – RFC 6298 §5

   The retransmission timeout is exponentially backed off on repeated loss.

4. We **partner** with the OS – Linux `tcp(7)`

   Tunables like `tcp_rmem`, `tcp_wmem`, `tcp_congestion_control`, and `tcp_no_metrics_save` let operators adjust behavior to their environment.

5. We **stay in step** with the receiver – RFC 9293 §3.4

   The sender's decisions are informed by what the receiver tells it. There is no point in pushing data faster than the peer can absorb.

6. We **estimate** the round-trip time – RFC 6298 §2

   SRTT and RTTVAR are continuously updated based on observed ACK timings; RTO is computed from them.

7. We **react** to congestion signals – RFC 3168 §5

   ECN marks (when negotiated) let the network signal congestion without dropping packets.

8. We do **not** congestion-collapse – RFC 896 §1

   Pre-TCP networks suffered congestion collapse because senders kept retransmitting into already-overloaded networks. AIMD prevents this.

9. We **make room** for others – RFC 5681 §3.1

   A polite sender backs off generously on loss; this is what allows competing flows to coexist.

10. We **pay** for correctness – RFC 9293 §3.5.2

    A clean close costs an RTT and 2×MSL of `TIME-WAIT`. Aborting with RST is faster but loses data; we do the right thing even when it's expensive.

♦

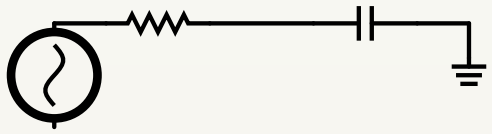
Healthy networking prioritizes correct backpressure as we partner with the kernel and the network to keep traffic flowing for everyone.

Reliability Made Visible in Hardware

Hardware-level diagrams help us understand things that are otherwise invisible at the software level.

A. A pragmatic example - **the line driver**

Even at the lowest level, reliable communication is enabled by analog circuitry that conditions the signal. Below is a basic AC-coupling and termination network used at the entrance of an Ethernet line interface.

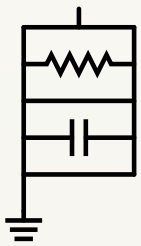


Ethernet AC-Coupled Line Termination

The capacitor blocks DC bias; the resistor terminates the line at its characteristic impedance. Without these, the receiver sees reflections and the link silently corrupts data.

B. A second example - **the supply rail decoupling**

Every digital pin transition draws current. Without local decoupling, supply rail noise corrupts logic levels and packet checksums fail intermittently.



Power Supply Decoupling

The voltage divider sets a stable bias; the bypass capacitor sinks transient currents. Reliable bytes on the wire start with a clean power rail at the chip.

Software reliability rests on hardware reliability. The transport layer cannot recover from corruption it cannot detect, and corruption is much easier to detect than to debug.

Specification Statement

A. The Purpose of a specification statement

The primary purpose of our specification statement is to foster shared understanding so that, as practitioners, we all agree on the behavior our implementations must exhibit. Since every workshop participant affirms this statement, we can be confident that we are unified in our reading of the specification.[1] The statement also serves as a guard against quiet drift, protecting our implementations from subtle deviations that creep in over time.[2]

The specification statement should not be so narrow that it excludes implementations we want to interoperate with. But it should also not be so wide that it permits behavior that breaks the contract. Thus this statement centers on the wire-level guarantees that every TCP must provide. Good engineering — and therefore a good specification statement — always holds the various trade-offs in tension and is careful to qualify in what sense each guarantee holds. For example, "TCP delivers data reliably" is one of the most foundational claims in networking, yet *reliable* doesn't mean *guaranteed*: a network partition can still cause an abort. To not hold reliability in proper relationship with timeouts and aborts is to misunderstand the true nature of TCP's guarantee.

B. The Service

TCP is a **reliable, in-order, byte-stream, connection-oriented, full-duplex transport** layered on a best-effort packet network.[3] It establishes connections via a three-way handshake,[4] delivers bytes via cumulative and selective acknowledgment,[5] terminates connections via a four-way close,[6] and provides flow control and congestion control mechanisms that allow many concurrent flows to share a network without collapse.[7]

C. The Specification

The Transmission Control Protocol is normatively defined by **RFC 9293**, which obsoletes RFC 793, RFC 879, RFC 2873, RFC 6093, RFC 6429, RFC 6528, and RFC 6691.[8] We rely on RFC 9293 as our primary reference and treat it, together with its cited dependencies, as the authoritative description of TCP behavior.[9] Without this document, interoperability would be a coincidence rather than an engineering property.[10]

We believe that an implementation conforms to TCP only when it implements every MUST-level requirement in RFC 9293 §3 and §4 and does not violate any MUST-NOT.[11] SHOULD-level requirements may be relaxed only with documented justification, and in interoperability testing the resulting implementation must still cooperate correctly with conforming peers.[12]

[1] RFC 1958 §1; RFC 9293 §1
[2] RFC 7414 §1
[3] RFC 9293 §1.5; Cerf & Kahn (1974)
[4] RFC 9293 §3.5
[5] RFC 9293 §3.4; RFC 2018 §3
[6] RFC 9293 §3.6
[7] RFC 5681; RFC 9293 §3.8
[8] RFC 9293 §1
[9] RFC 9293 §1.3
[10] RFC 1958 §3
[11] RFC 2119 (key word definitions); RFC 9293 §3
[12] RFC 8174 (BCP 14 update)

We believe that the IETF process — open mailing lists, public drafts, working group review, and rough consensus — is what produces specifications worth implementing.[13] An RFC that has not been through that process should be treated as a single party's proposal, not a community standard.[14]

We believe that since the specification is a contract among independent implementers, behavior outside the spec is undefined and must not be relied upon.[15] Code that depends on the quirks of one particular stack will eventually meet a different stack.

We believe that errata, when published, are part of the specification, and that implementers are responsible for tracking them.[16]

D. The doctrine of segments

We believe that a TCP segment is the unit of communication on the wire,[17] consisting of the TCP header followed by zero or more payload bytes.[18] The header has a fixed twenty-byte portion plus optional extensions[19] and is checksummed over a pseudo-header that includes the IP source and destination addresses.[20] The checksum is mandatory and must be validated by the receiver.[21]

E. Sequence numbers

We believe that every byte transmitted on a TCP connection is assigned a 32-bit sequence number that increases monotonically (modulo 2^{32}) as data flows.[22] The connection's initial sequence number is chosen unpredictably[23] and the SYN and FIN flags each consume one sequence-number slot.[24] PAWS (Protection Against Wrapped Sequences) using TCP timestamps protects long-lived high-bandwidth connections against wrap-around ambiguity.[25]

F. Acknowledgments

We believe that the receiver acknowledges contiguous bytes cumulatively via the ACK field of the header,[26] and may additionally report non-contiguous out-of-order ranges via the SACK option.[27] An acknowledgment is a positive statement: it asserts that the named bytes have been received and stored.[28] An unacknowledged byte must be assumed lost only after the retransmission timer expires or after sufficient duplicate ACKs trigger fast retransmit.[29]

[13] RFC 2026 §1
[14] RFC 2026 §3
[15] RFC 1958 §2.4
[16] RFC 2026 §6.1
[17] RFC 9293 §3.1
[18] RFC 9293 §3.1
[19] RFC 9293 §3.1; RFC 7323 §2
[20] RFC 9293 §3.1; RFC 1071
[21] RFC 9293 §3.1; RFC 1122 §4.2.2.7
[22] RFC 9293 §3.4
[23] RFC 6528 §3
[24] RFC 9293 §3.4 — note that SYN and FIN consume sequence space even though they carry no data.
[25] RFC 7323 §5
[26] RFC 9293 §3.4
[27] RFC 2018 §3; RFC 2883
[28] RFC 9293 §3.4
[29] RFC 5681 §3.2; RFC 6298 §5

G. Flow control

We believe that the receiver advertises its remaining buffer space in the Window field of every segment it sends,[30] scaled by the WSCALE option negotiated at handshake time.[31] The sender must not transmit more unacknowledged bytes than the advertised window permits.[32] When the window reaches zero, the sender enters persist mode and probes periodically until the receiver opens the window again.[33]

H. Congestion control

We believe that on top of flow control, the sender maintains a congestion window driven by an algorithm such as Reno, CUBIC, or BBR.[34] The effective send window is the minimum of the receive window and the congestion window.[35] Loss, detected via timeout or duplicate ACKs, is treated as a signal of congestion and triggers reduction of the congestion window.[36] ECN, when negotiated, lets routers signal congestion explicitly without dropping packets.[37]

I. Reset and abort

We believe that RST aborts a connection unilaterally and immediately,[38] is appropriate when no graceful close is possible,[39] and must be validated against the current sequence-number window before being accepted.[40] Off-path RST injection is a known attack and is mitigated by RFC 5961 challenge ACKs.[41]

J. Things to come

We believe that QUIC and other UDP-based transports complement rather than replace TCP.[42] The principles in this specification — reliable delivery, ordered streams, flow control, congestion avoidance — recur in all serious transport designs and will outlive any particular wire format.[43]

[30] RFC 9293 §3.4
[31] RFC 7323 §2
[32] RFC 9293 §3.8.6
[33] RFC 9293 §3.8.6.1
[34] RFC 5681 (Reno); RFC 9438 (CUBIC); Cardwell et al. 2017 (BBR)
[35] RFC 5681 §2
[36] RFC 5681 §3.1; RFC 6298 §5
[37] RFC 3168
[38] RFC 9293 §3.5.3
[39] RFC 9293 §3.10.4
[40] RFC 5961 §3
[41] RFC 5961 §4
[42] RFC 9000 §1
[43] RFC 1958 §2.1

Compatibility with Real Networks

A. The goal

The goal of this document is to walk through how the specification interacts with the real-world conditions our deployments will face — NAT, middleboxes, asymmetric paths, lossy wireless — so that we may build software that works in production and not only on the loopback interface.

We will address the topic under three categories: 1) middleboxes and NAT, 2) wireless and lossy links, 3) operational visibility.

B. Middleboxes and NAT

Middleboxes — NATs, firewalls, load balancers, deep-packet-inspection appliances — see TCP segments and often modify them. NATs rewrite source IP and source port; some load balancers terminate and re-establish connections; some firewalls strip TCP options they don't recognize. RFC 8546 articulates the principle that the wire image of a protocol must be treated as the protocol; whatever the middlebox sees becomes part of what implementations must tolerate.

We acknowledge that some otherwise-compliant TCP implementations behave poorly in the presence of options the spec adds later (TFO, MPTCP). This is one of the practical reasons new transport features have moved to QUIC, where ossification is harder.

C. Wireless and lossy links

On wired Ethernet, packet loss is overwhelmingly the result of buffer overflow at routers; AIMD interprets it correctly as a congestion signal. On wireless links, loss is often the result of interference, and interpreting it as congestion is wrong — but the kernel has no general way to tell the two apart. ECN, where deployed, helps. So does running a congestion control algorithm such as BBR that infers bandwidth and RTT directly rather than treating loss as the sole signal.

D. Operational visibility

Most production TCP problems are diagnosed with `ss`, `tcpdump`, Wireshark, eBPF, and `netstat`. Without those tools, the connection state machine is invisible. Operators should be able to read packet captures fluently and correlate them to the state machine in RFC 9293 §3.6.

We believe an engineer who cannot read a SYN, SYN-ACK, ACK in a `tcpdump` output is not yet ready to debug TCP issues in production. This is not snobbery — it is the same standard we apply to ourselves.

Sample Capture & Code Repository

A. Explanation

As practitioners, we believe that the best way to internalize the specification is to read packet captures alongside the RFC. We provide a sample capture (`tcp-handshake.pcap`) that walks through SYN, SYN-ACK, ACK, three data segments with cumulative ACKs, and a graceful close. The corresponding annotation document maps each segment to a section of RFC 9293.

However, even sample captures cannot replace running your own. We strongly encourage every workshop participant to capture their own production traffic on a non-sensitive endpoint (e.g., a localhost echo server) and step through it byte-by-byte.

Facilitator note: open tcp-handshake.pcap in Wireshark before the session starts — the SYN, SYN-ACK, ACK walkthrough lands better live than in screenshots.

At Foundations of TCP, we eagerly welcome questions, captures, and reproductions from any engineer. We welcome all who are willing to read the spec carefully. Yet we also recognize that some kinds of broken behavior require careful attention. Therefore, we have established a triage policy for issues raised against our reference implementation.

B. Practices

- An issue against the reference implementation must include a packet capture or a minimal repro program.
- An issue claiming a specification violation must cite the specific RFC and section.
- All proposed fixes must be accompanied by a regression test that fails before the fix and passes after.
- The maintainers and reviewers will be made aware of all open issues affecting protocol correctness.
- The community needs to be informed when correctness issues are found in widely deployed code; however, named affected vendors are disclosed only when the working group chairs believe it to be necessary.

The goal of this policy is not to embarrass any particular implementation, but to ensure that the specification remains the shared ground we all stand on. Good engineering takes shared mistakes seriously, and so we celebrate good repros.

♦

If you have any questions about this policy, please contact the workshop organizers.