TCP (rfc793) provides the following functionality:

- Session layer functionality: it controls the maintenance of sessions (virtual circuits).
- A reliable transport–if TCP delivers a segment to an application, it is “guaranteed” to be correct and in order.
- A certain degree of protection against spoofing.
- Possibly some protection against certain DoS attacks.
- Congestion control.

The main drawback is the overhead added by these features.

Segment format

A TCP segment has a header and a variable–length segment body. The header is made of two parts: the fixed 20–byte part possibly followed by a variable–length options part; typically, the options part is empty.

<table>
<tr><td colspan="3">Source port</td><td>Dest. port</td></tr>
<tr><td colspan="4">Sequence number</td></tr>
<tr><td colspan="4">Acknowledged number</td></tr>
<tr><td>HL</td><td>000000</td><td>Flags</td><td>Window size</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td colspan="4">Options</td></tr>
<tr><td colspan="4">Segment
body</td></tr>
</table>

Ports: 16–bit port numbers.

Sequence number, Acknowledged number are explained below.

HL: is the length of the header in 32–bit words. The minimum is 5 (fixed header only); the maximum is 15 (40 bytes of options).

Flags: 6 control flags.

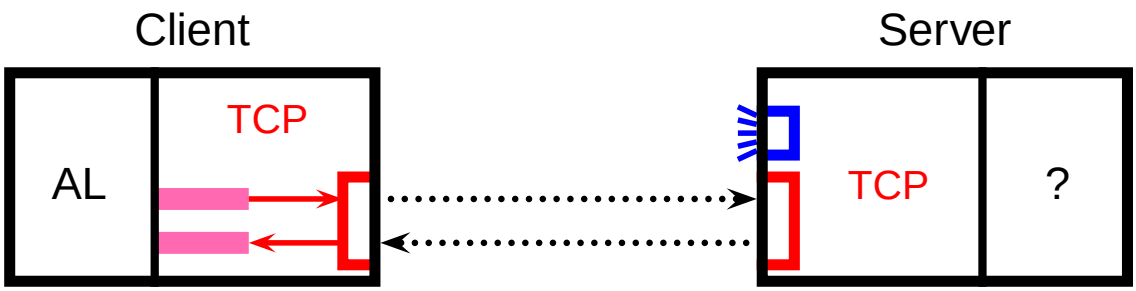
Window size: the size of free buffer space (imposes a maximum segment length).

Checksum: computed for the whole segment **and** for an imaginary IP header that will prefix the segment.

Urgent pointer: points to the first byte of non–urgent data in the segment.

Data flow control

A TCP connection provides two unidirectional sequential streams of bytes flowing in opposite directions.



The streams are sequential: bytes must be handed to the receiver's application in the same order in which they were sent. This is implemented by assigning each byte a byte number which correspond to the byte's position in the stream.

Sequence numbers

TCP “thinks” in **bytes**. A stream starts with byte 1. Every segment is given an identifier that is equal to the byte number (in the stream) of its first byte. Hence, the first segment is 1 + isn (to be explained).

An out–of–band stream is piggybacked on each byte stream (it is called “urgent data”). The “urgent” bytes are not part of the byte stream (but they, too, count for the purpose of byte numbering).

Acknowledgments

Reliability is guaranteed by forcing the receiving end of each stream to acknowledge the bytes it received.

TCP acknowledges by putting a number in the segment's header. This number is the number (in the stream) of the byte it expects next. This method is called **cumulative acknowledgments** (a form of **selective acknowledgments**).

When a TCP sends a segment, it starts a **timer** which ticks until an acknowledgment arrives or the time expires, a situation called **timeout**.

A timeout makes TCP resend all the segments that were not acknowledged. This in turn creates the problem of reacting to a late acknowledgment (one that comes when some segments are being retransmitted): the retransmission should be aborted.

The amount of time before the timer expires is a carefully computed value (it is a variable). Choosing a value too small reduces effective throughput in the network, a value too large increases delays seen by the application. When an arrives or a new message is submitted by the AL, the timeout interval is derived from the most recent values of two variables (EstimatedRTT and SampleRTT); in the case of a timeout, the timeout interval is doubled instead of being recomputed.

Acknowledgment loss

The network layer (IP) is unreliable, so segments may be lost. That implies also that acknowledgments may be lost. Hence the need to handle duplicate acknowledgments.

Acknowledgments may arrive late (after a timeout, but before all the unacknowledged segments were retransmitted). Then, the sender must stop retransmitting.

Finally, when an out–of–order segment arrives, the receiver must send a duplicate acknowledgment (RFC2581), so the sender must be prepared to receive duplicate acknowledgments.

The importance of the window

The TCP header allows each host to declare (dynamically) a window size $\mathcal{W}$–interpreted as the amount of space left in the host's receiver buffer.

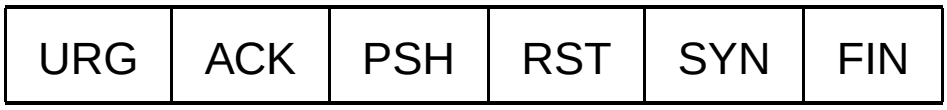
No segment may be larger than the receiver's window size. When a receiver is backlogged, it may declare its window to be of size 0, barring the sender from sending anything until further notice.

RFC793 requires a receiver to drop segments with a sequence number that falls outside the receiver's window, i.e. only segments with a sequence number $\mathcal{N}$ that satisfies:

$$ack.last \leq \mathcal{N} < ack.last + \mathcal{W}$$

will be accepted for further scrutiny (it may still be discarded at a later point).

Control flags



The flags are set (to 1) if:

URG urgent data pointer is valid (implies that there is some out–of–band data at the start of the payload).

ACK the value of the acknowledgment field is valid.

PSH asks the receiver to push the contents of its input buffer to the application layer.

RST the sender reset the connection.

SYN the packet attempts to start the handshaking process. Its main purpose is to exchange the initial sequence numbers.

FIN the sender closes its end of the connection.

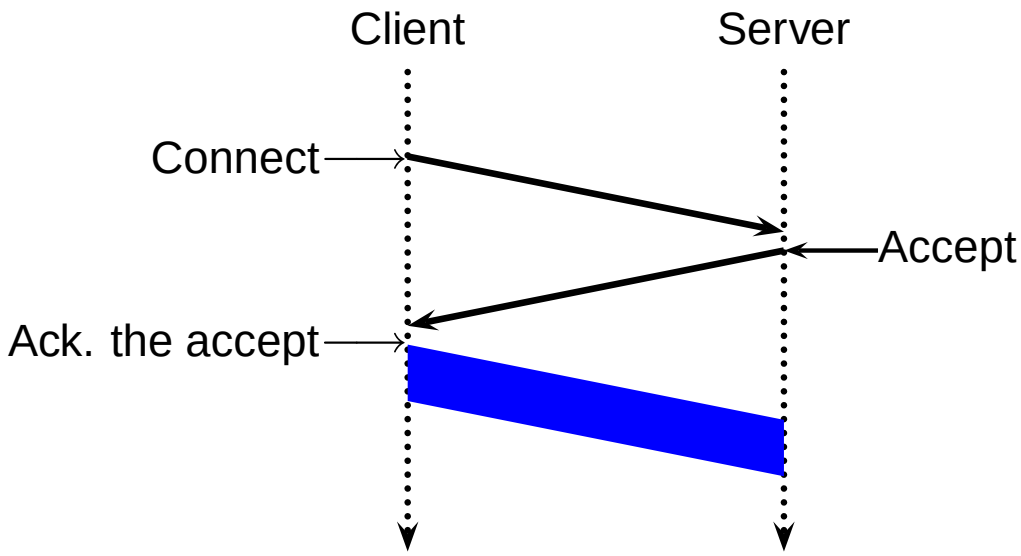
Connection

TCP requires a 3–way handshake which establishes a trusted connection:

1. The **client** sends a packet to the **server**. In this packet the client identifies itself and requests the connection.
2. The **server** replies with a packet acknowledging the request and accepting the connection. Some connection parameters may be sent at this point.
3. The **client** concludes the setup by sending a packet which acknowledges the acceptance (so that the **server** knows that the connection was truly established).

The last step is necessary to prevent dangling connections, a common memory leak as well as a serious DoS threat.

Simplistic 3–way handshake



The third segment serves as acknowledgment for the "Accept" segment **and** as the first segment of the $client \rightarrow server$ byte stream.

The first two segments are called:

- SYN ("Connect")
- SYNACK ("Accept")

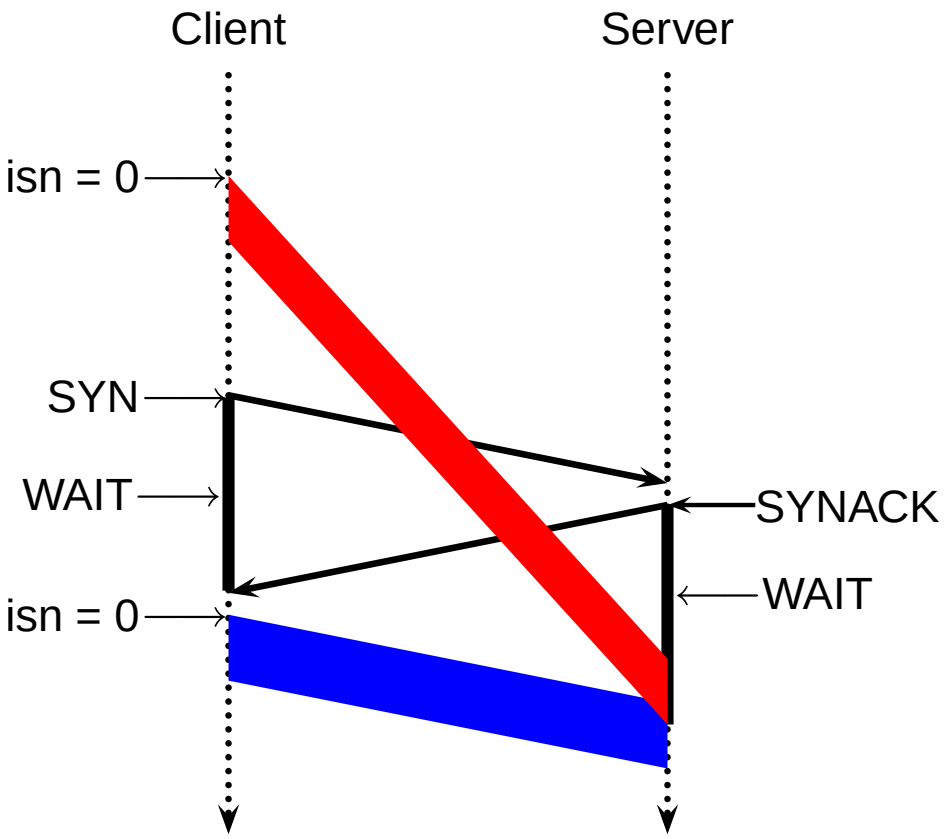
Initial sequence numbers

It is natural to count the bytes of a sequential stream starting with 1 or 0. This leads to two potential problems:

Error: After a network turbulence, segments from a terminated connection may reach the receiver and be accepted as part of the current connection.

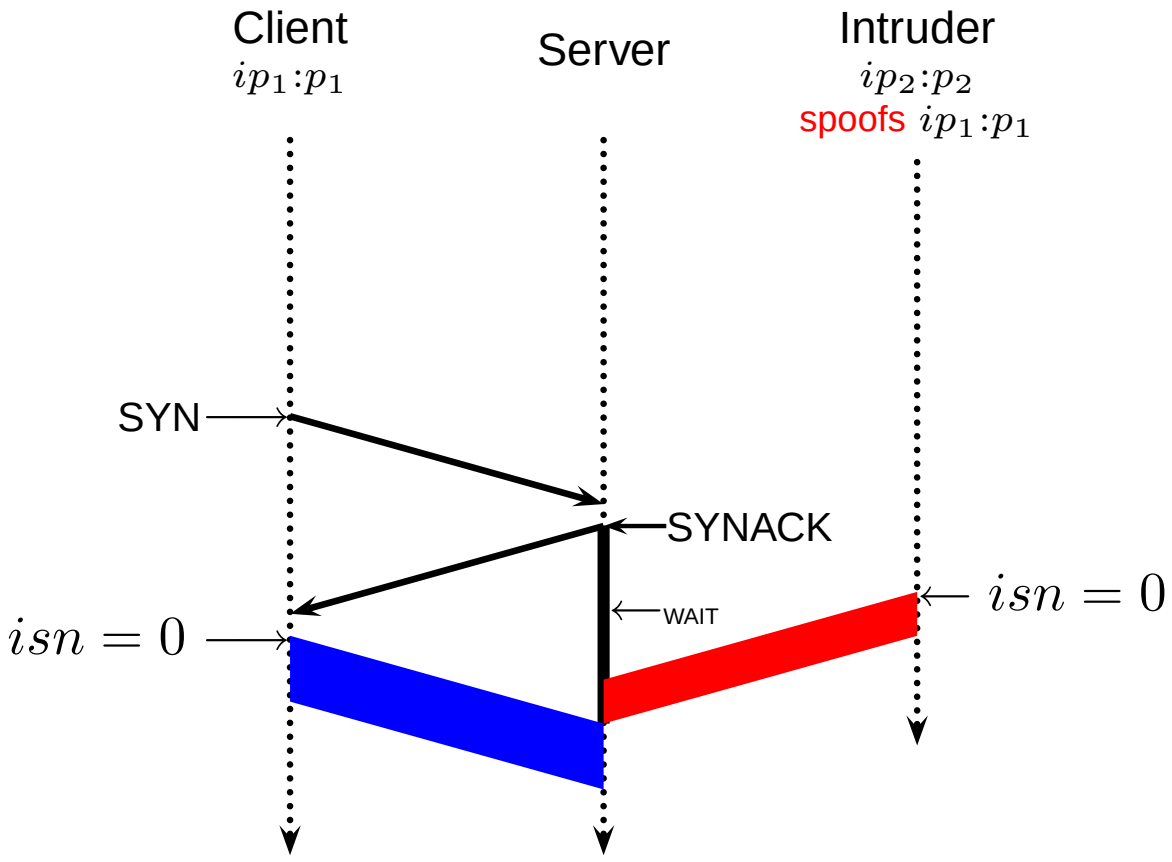
Attack: An intruder may hijack a connection being established by providing the third part of the 3–way handshake.

Segment arriving late



The older segment becomes the third part of the handshake and the correct segment is discarded as duplicate. If the two segments have the same length, the only problem is that different bytes are inserted into the stream. If the lengths differ, the resulting confusion cannot be fixed easily.

ISN attack



The intruder succeeds in placing a block of bytes in the stream and disrupts the connection, if desired.

Defeating the ISN attack

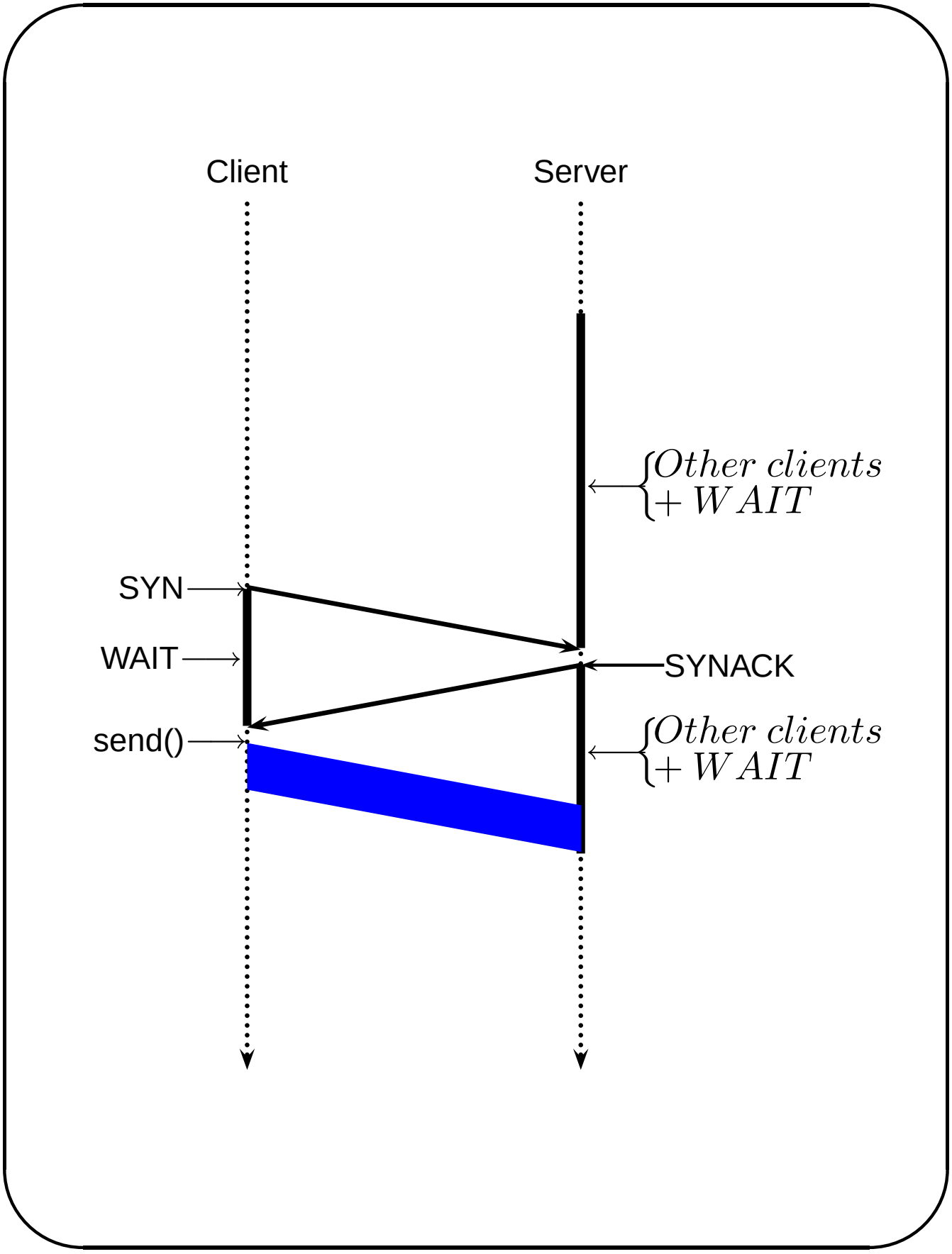
RFC793 tried to prevent isn attacks by requiring that the initial sequence numbers be chosen using a function of the current time and the number of opened TCP connections. Serious hackers had no difficulty guessing isns derived in this manner.

The current defense is to generate initial sequence numbers at random. This reduces greatly the chance of a successful isn attack unless the standard pseudo–random number generator is used.

Connection revisited

The simplistic 3–way handshake has several weaknesses and the protocol requires a more elaborate handshake which does the following:

- Sets up a virtual circuit (the hosts' identities and access rights are checked as part of the setup).
- Makes the two sides exchange initial sequence numbers which serve as passwords and make breaking into a circuit difficult. To reduce vulnerability, the two sides pick their isn values in some semi–random fashion.


Client
Server
$\left\{\begin{matrix}Other\ clients\\+\ WAIT\end{matrix}\right.$
SYN
WAIT
SYNACK
send()
$\left\{\begin{matrix}Other\ clients\\+\ WAIT\end{matrix}\right.$

SYN: SYN = 1 , ACK = 0 ,

SeqNum = client_isn.

SYNACK: SYN = 1 , ACK = 1 ,

AckNum = client_isn+1 ,

SeqNum = server_isn.

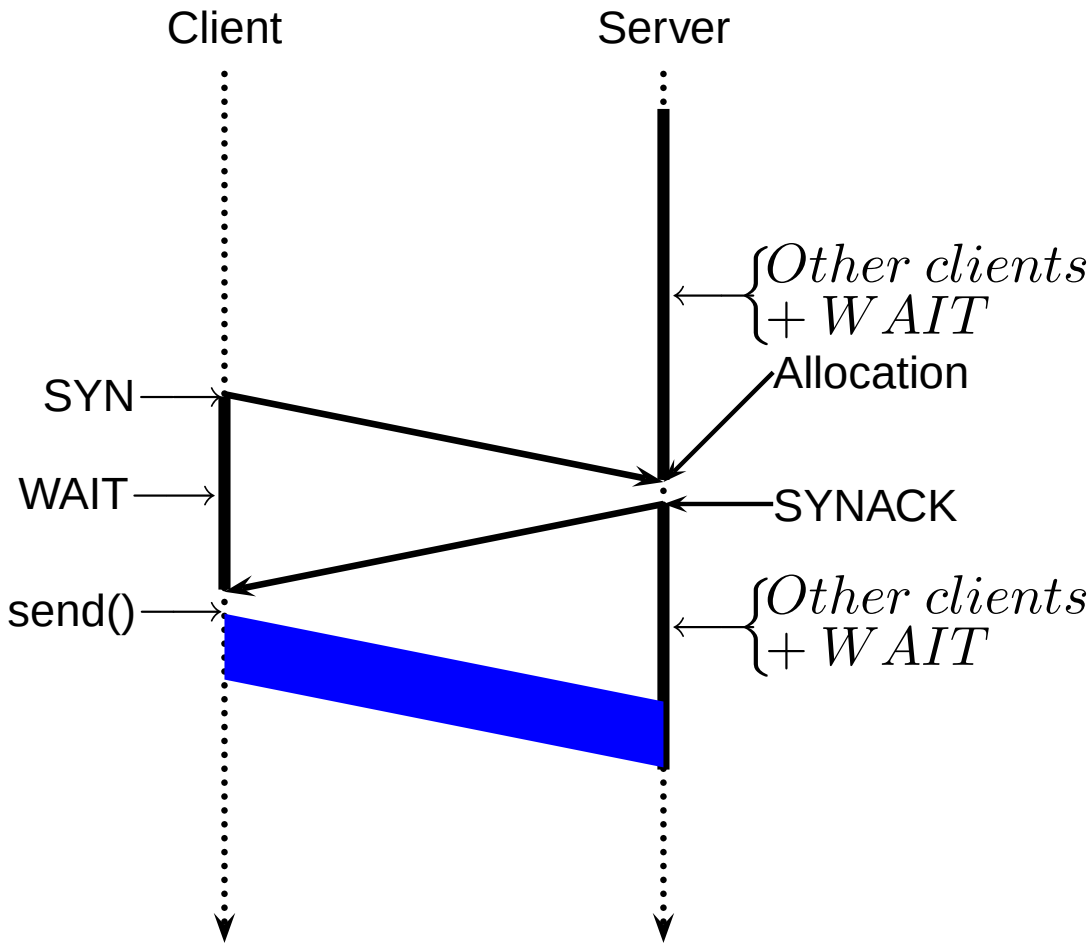
send(): SYN = 0 , ACK = 1 ,

SeqNum = client_isn+1 ,

AckNum = server_isn+1.

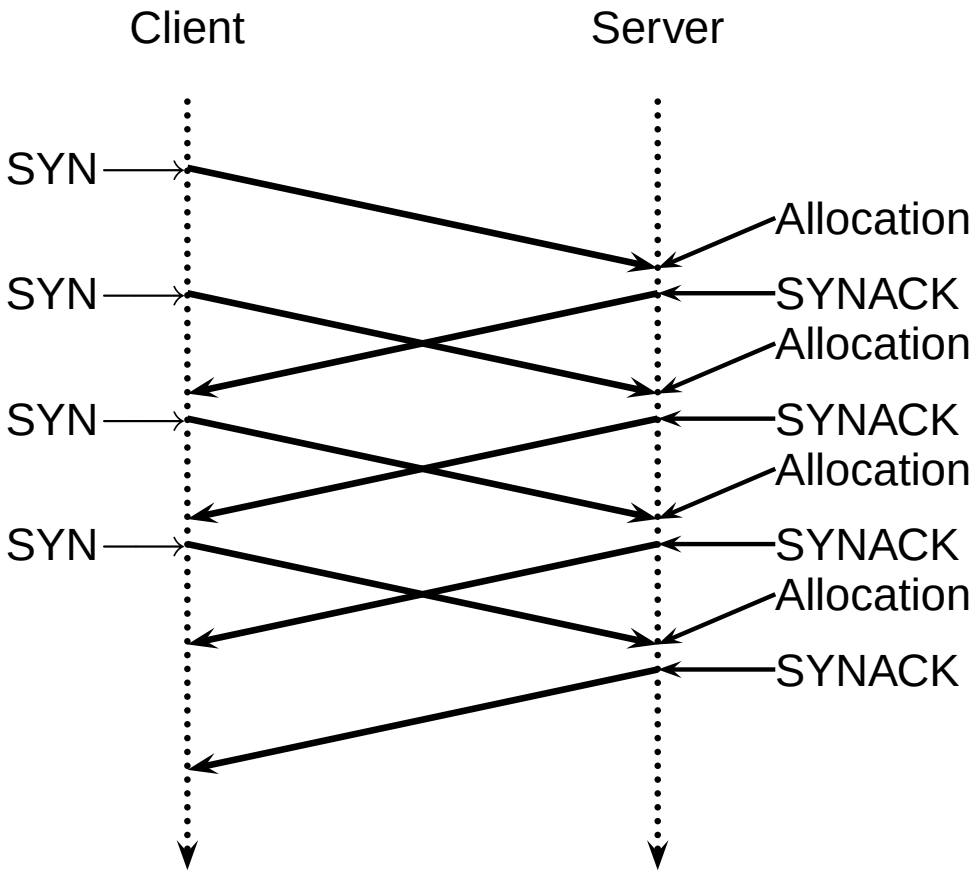
The SYN flooding attack

A naive implementation of TCP reserves memory for a connection when a SYN segment arrives:



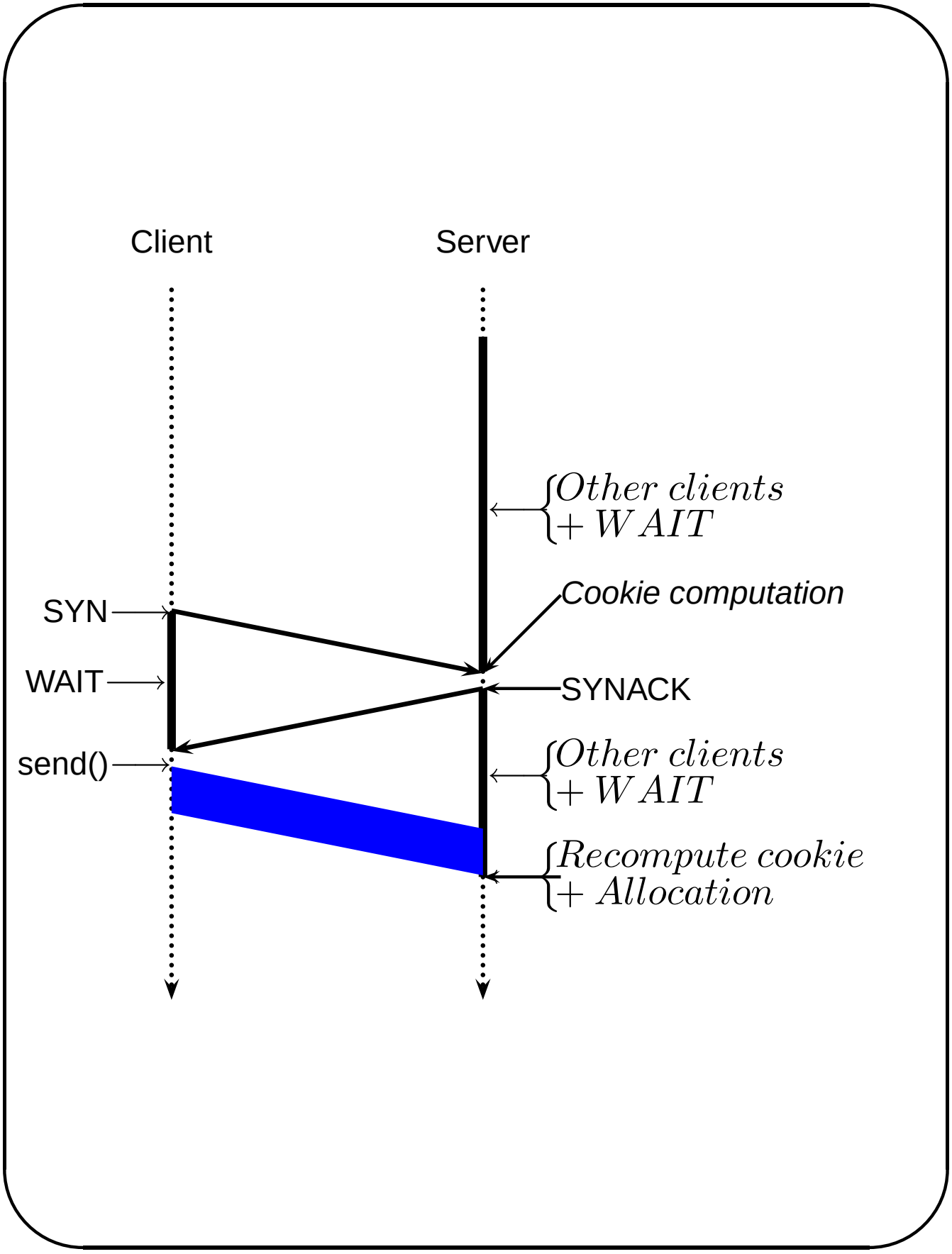
This implementation works if all the clients are legitimate.

If a client chooses not to complete the 3–way handshake, the data structure kept for connections–in–the–making can be filled, preventing the server from accepting connections.



One method to reduce the damage of a flooding SYN attack is to delay allocating resources until the end of the third phase of the handshake. The server computes a cookie using information coming on the SYN segment and a secret function and sends it as its isn back to the (potential) client. The value of the cookie is not stored anywhere.

When the client's first send() arrives, it must contain the server's isn+1 in its acknowledgment field. The server recomputes the cookie and checks it against the acknowledged value. If they match, the connection is legitimate (probabilistically) and the data structures can be allocated.


Client
Server
$\{Other\ clients + WAIT$
Cookie computation
SYN
WAIT
SYNACK
send()
$\{Other\ clients + WAIT$
$\{Recompute\ cookie + Allocation$

Control flag attacks

The flooding SYN attack is one case of attacks based on the abuse of control flags. Another SYN attack, the spoofing SYN attack is a weaker variant of the Reset Attack which uses the RST bit to destroy other clients' connections.

The idea is simple. A bad guy spoofs a segment with this header:

- Destination address: the address of the server. This is easy to obtain.
- Source address: the address of the victim client (the IP part is relatively easy; the port number has to be guessed or sniffed).
- A legitimate client sequence number. This is the tough part.
- The RST bit set.

When a packet with this header reaches the server, it will immediately reset (i.e. close) the connection, which causes DoS and worse.

Reset attack helpers

It appears difficult to guess the source port number and the current sequence number. However, some tricks make it easier:

- Guessing the client's port number may be simple.
- Faking an acceptable sequence number may require just a few seconds of persistent attempts.

Guess the port number

Many Operating Systems assign port numbers sequentially (both Windows and Linux used to do so).

Launching a passive FTP connection to the client's host should then give a good starting point for guessing the port number of the client's end of the targeted connection (if the FTP server returns port $\mathcal{P}$, one should try ports $\mathcal{P} - 1$ and further down).

Guessing a sequence number

RFC793 defines as valid a segment that has the correct source and destination addresses **and** has a sequence number that is no more than one window size away from the last acknowledged sequence number.

If the window size is 2^{16} the maximum number of attempts needed to produce a fake sequence number that falls into the window is $2^{32}/2^{16} = 65,536$, which can be generated in a matter of seconds (each attempt requires sending a 20–byte segment).

RFC793 asks to subject acknowledgment numbers to the same window test, but real–life protocols tend not to do so.

Flow control in TCP

The basic protocol of TCP is augmented by an algorithm for flow control. This algorithm, called Congestion Avoidance is invoked in two different situations; it has more than one description, each of them having more than one interpretation.

Congestion avoidance is invoked when one of two events is observed by a TCP receiver:

- A timer expires when a host is waiting for an acknowledgment. In this case it must be assumed that a segment was lost and must be retransmitted. This situation results in a scenario called Slow Start.
- The host keeps receiving "negative" acknowledgments[a] hinting that the segments are not getting through. This case is called Fast Retransmit.

[a] In TCP negative acknowledgments take the form of duplicate acknowledgments, a slightly different concept.

Both cases imply the presence of congestion (the main reason for packet loss).

The obvious first step is to slow down the transmission rate (this is an egoistic act: maintaining the sending rate does not serve the sender). This is achieved by reducing the flow to a minimum (resending the lost segment—which is a must) followed by a cautious gradual increase in the transmission rate.

The purpose of Congestion Avoidance is to restart the flow of data while attempting to avoid recreating congestion.

Slow Start

When no feedback comes from the other end of a connection, the recovery timer will eventually expire. Its expiry suggests that one of two several events happened:

1. The other end is dead.
2. The connection is broken.
3. A segment was lost and a deadlock occurred (one side waiting for a segment, the other for an acknowledgment).

The first two cases have no cure; the deadlock in the third can be broken by retransmitting segments.

Fast retransmit

TCP does not have negative acknowledgments, so a duplicate ACK serves as one. When a duplicate acknowledgment arrives, the sender must guess whether it is a result of packets delivered out of order or the result of a lost packet.

The suggested approach is to assume that 3 consecutive duplicate acknowledgments imply a lost packet. Thus, the sender's TCP ignores the first two duplicate ACKs assuming they resulted from an out–of–order delivery; when a third arrives, the sender must act as if its timer timed out and immediately resend the presumably lost segment.

There are various interpretations of what needs to be sent: just the oldest segment or more segments not acknowledged so far. Note the receiver may (does not have to) keep the out–of–order segments in its buffer.

(Maximum) Segment Size

TCP maintains a very important constant called Maximum Segment Size (MSS) for each virtual circuit. The value of MSS represents the maximum size of the payload of a datagram handled by the underlying NL.

For example, in IP a datagram has a default maximum size of 576B, hence an IP–oriented **SS** would be 536 (20B TCP header and 20B IP header). Systems expecting to work over an Ethernet connection often adopt **SS** = 1460 (1460+20+20 = 1500B, the maximum Ethernet frame size).

TCP has the right to send segments no larger than **MSS**; most **TCP** implementations will not send segments smaller than **MSS** unless forced to do so. There are several reasons for that, including the Silly Window Syndrome.

Congestion Avoidance uses two standard TCP–specific variables that are measured in multiples of MSS.

Congestion Window (CongWin or cwnd) is the actual maximum segment size (unless it exceeds the window size advertised by the other side). CongWin is initialised to 1 or 2 MSS.

Slow Start Threshold Size ssthresh. It is initialised to 64KB (2^{16} bytes).

Congestion Avoidance

The trick is to sense what is the bandwidth currently available for the circuit used. The basic assumption is that the rate at which new segments should be transmitted should match the rate at which acknowledgments are sent back by the other end[a].

The algorithm limits the number of outstanding segments (i.e. sent but not acknowledged) changing the limit based on observed traffic.

By definition, whenever a TCP output routine is called to send data, it sends $\min(data\text{-}available, CongWin, \mathcal{W})$ bytes of data.

The amount of available data is an external factor, and so is $\mathcal{W}$, the receiver's window size (a bound coming from the header of the latest receiver's segment); CongWin is a variable changing every time the output routine is called.

[a] This assumes that the load is symmetric, an assumption wrong in many applications, e.g.. video–on–demand.

Congestion Avoidance operates in two phases:

- Exponential growth when the limit doubles in every round.
- Linear growth when the limit increases by 1 MSS in every round.

When a virtual circuit is created, CongWin is initialised to MSS (seldom 2 MSS) and Congestion Avoidance enters its exponential phase.

Exponential growth

While in the exponential phase, TCP increases CongWin by MSS for every segment acknowledged.[a]

Thus, if an acknowledgment for k segments arrives in time, TCP may:

- Increase CongWin by k (RFC3390) if CongWin is below a threshold value ssthresh.
- Switch to linear growth phase if CongWin reached the threshold.

[a] If one assumes that segments are sent in rounds, every round in the exponential growth phase will be twice as large as the previous round.

Linear growth

In this phase, CongWin is increased by $MSS \times \frac{MSS}{CongWin}$ each time a segment is acknowledged[a] (RFC 2581).

CongWin is increased only if it is less than 64 KB, the maximum possible segment size.

[a] If one assumes that segments are sent in rounds, then a round of length k (k = CongWin) segments will be followed by a round of length $k + 1$ segments.

Slow Start

When a timeout occurs, ssthresh is set to CongWin/2 and CongWin is set to MSS.

the Congestion Avoidance algorithm starts in its exponential growth phase.

Fast recovery

When 3 duplicate acknowledgments arrive, two conclusions can be reached:

1. A segment was probably lost: the receiver keeps reporting that it did not receive it.
2. The network must be at least partially functional; otherwise the 3 ACKs would not get through.

The second conclusion implies that one should not resort to a full slow start. Hence, both CongWin and ssthresh are set to ssthresh/2.

The Congestion Avoidance algorithm starts in its linear growth phase.