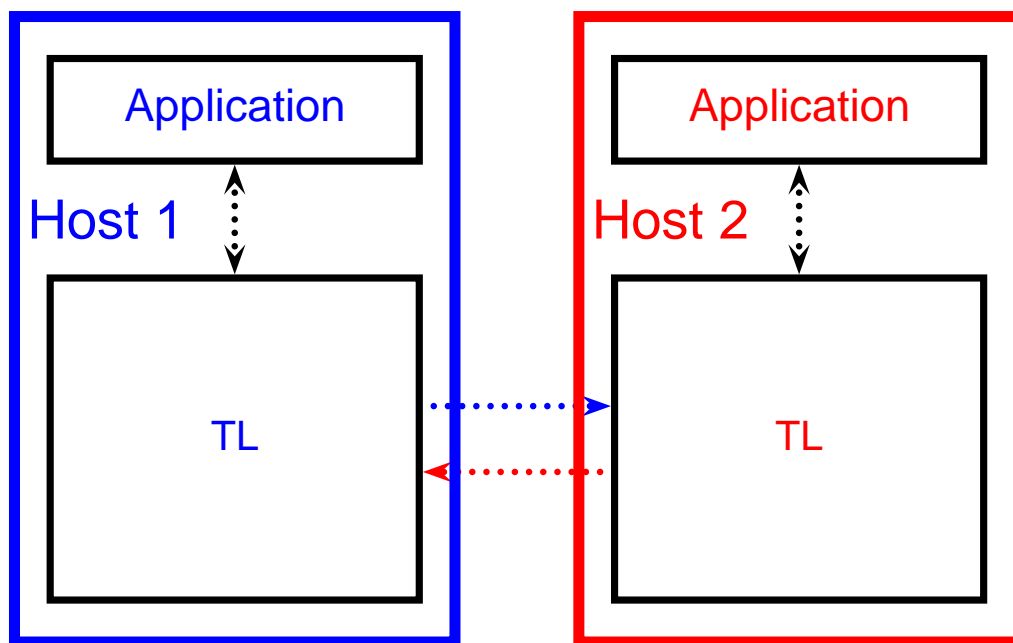


## Transport layer

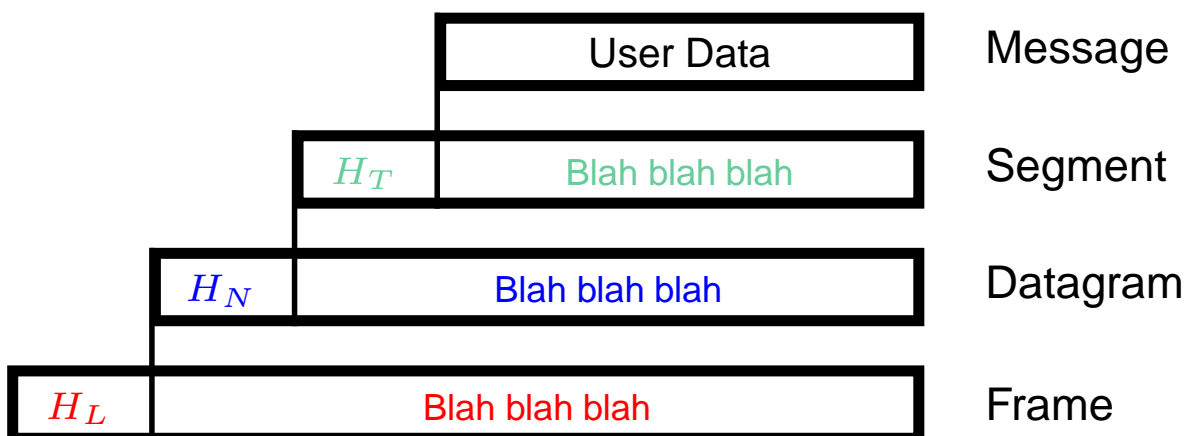
The TL deals with end-to-end transport only, so there is no packet loss due to congestion **in this layer**.



## Protocol Data Unit

Abbreviated as **PDU** this is the unit of data from the point of view of a given protocol and is used instead of “packet” in protocols. There is a synonym for the PDU used by each layer:

7	Application	A-PDU	Message (“Text”)
4	Transport	T-PDU	Segment
3	Network	N-PDU	Datagram
2	Link	DL-PDU	Frame
1	Physical	1-PDU	Bit/Symbol



## Transport layer

Two basic protocols are used: **TCP** and **UDP**.

TPDUs are called **segments** (**TCP**) or (somewhat incorrectly) **datagrams** (**UDP**).

**TCP** and **UDP** differ in a number of key properties:

- **Reliable** vs. unreliable.
- **Sequential** vs. random order arrivals.
- **Connection-oriented** vs. connectionless.
- **Congestion control** vs. swamping the net.

The logo for TCP (Transmission Control Protocol) features the letters 'TCP' in a bold, blue, sans-serif font. The text is contained within a white rectangular box with a blue border. This box is positioned on top of a larger, solid blue rectangular shape, creating a layered effect.

TCP (rfc793) offers:

- Sessions: it provides connection-oriented service.
- 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.

Source port		Dest. port	
Sequence number			
Acknowledged number			
HL	000000	Flags	Window size
Checksum		Urgent pointer	
Options			
Segment body			

**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.

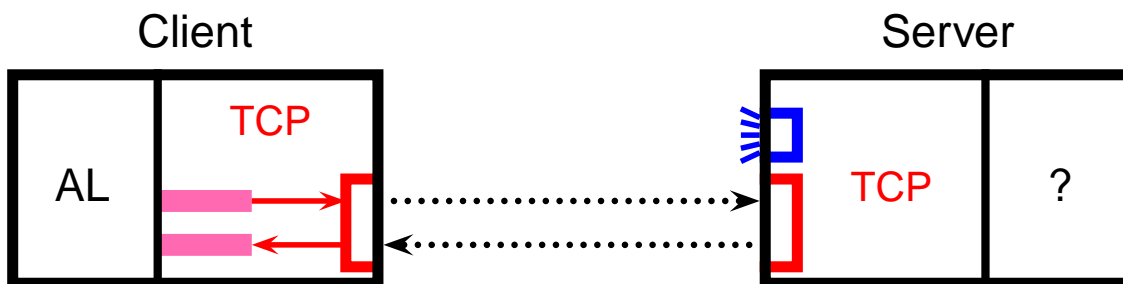
**Window size:** the size of free buffer space (imposes a maximum segment length).

**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 corresponds to the byte's position in the stream.

## Sequence numbers

TCP “thinks” in **bytes**. A stream starts its byte sequence with a random value **isn**. 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  $1 + \text{isn}$  (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 that acknowledgments may also 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 (they may still be discarded at a later point).

## Control flags

URG	ACK	PSH	RST	SYN	FIN
-----	-----	-----	-----	-----	-----

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

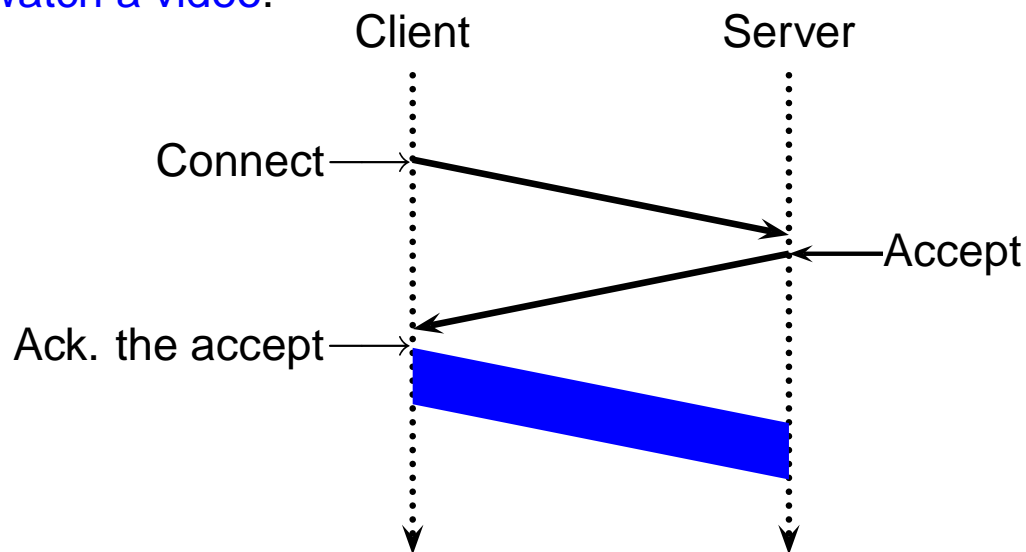
**TCP** requires a 3–way handshake which establishes a trusted connection:

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

Watch a video.



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

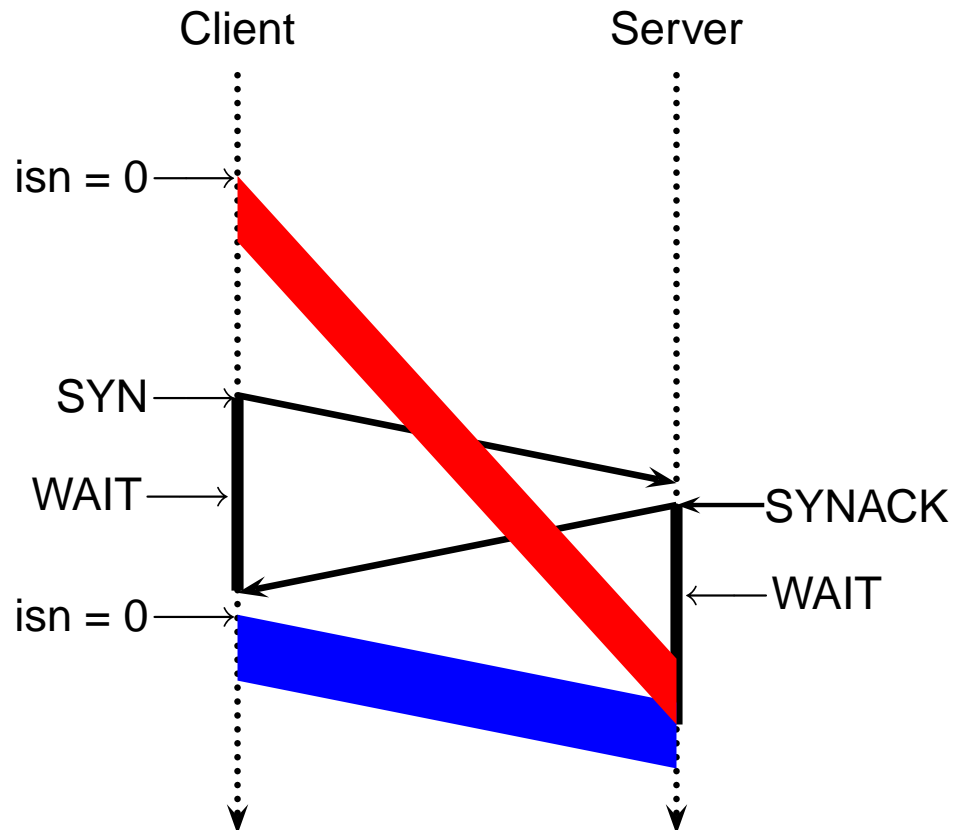
It is natural to count the bytes of a sequential stream starting with 1 or 0. This leads to two potential problems:

**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.

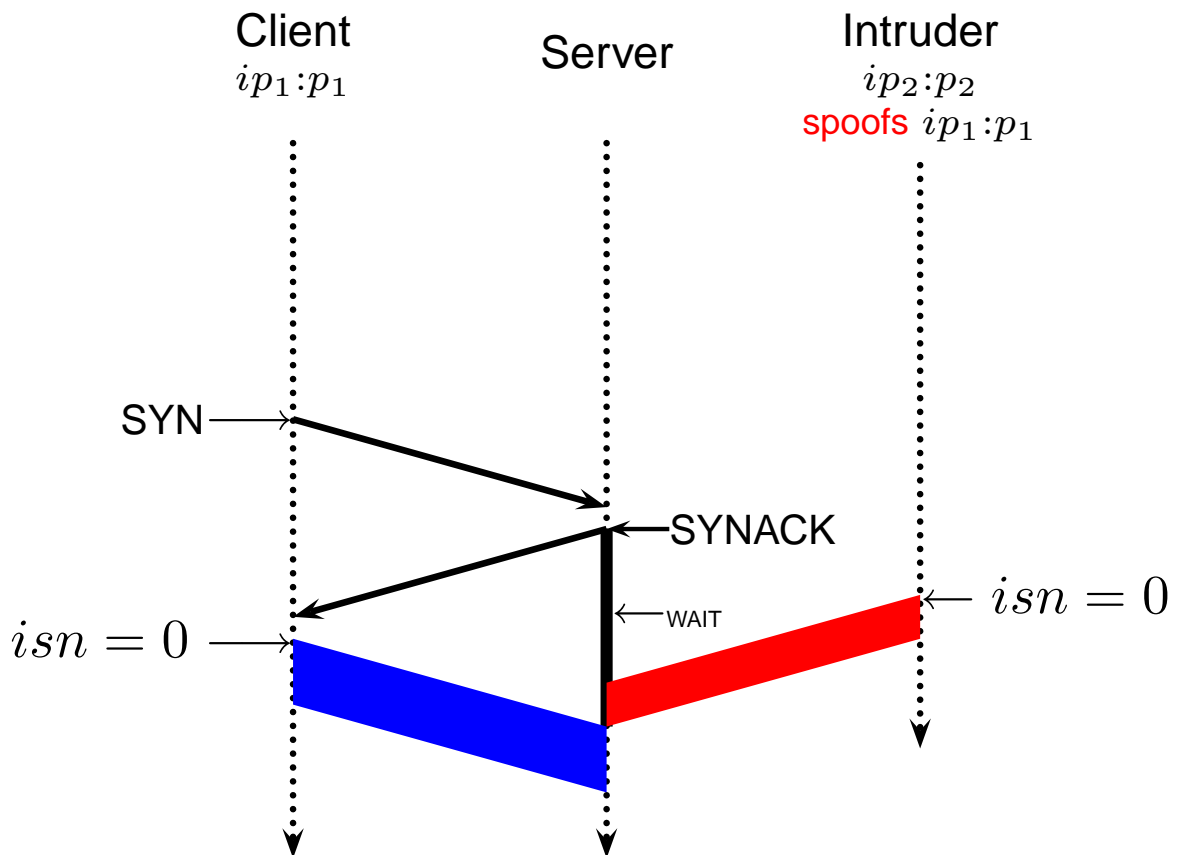


## Segment arriving late



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

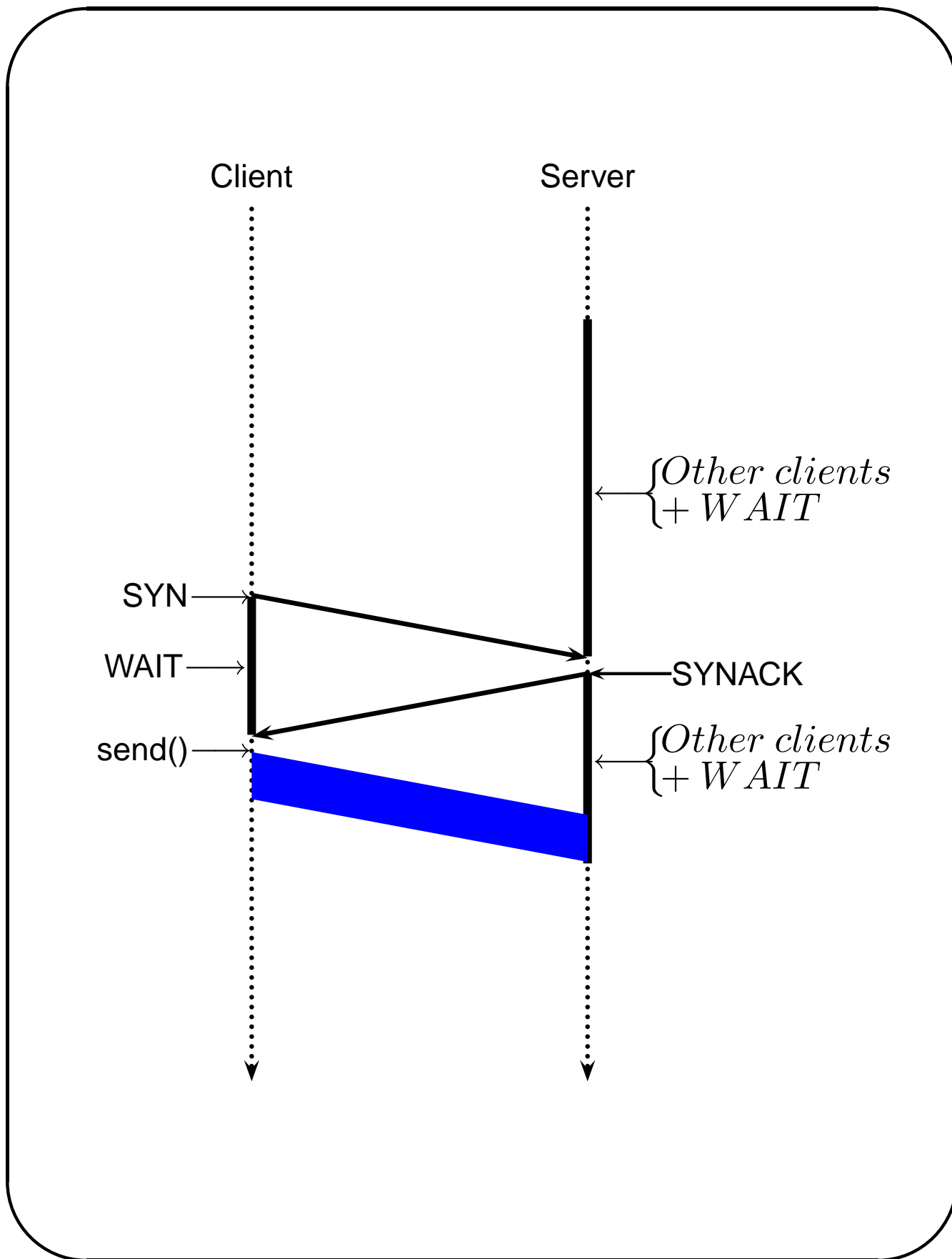
RFC793 tried to prevent **isn** attacks by requiring that the initial sequence numbers be chosen using a function of the current time and the number of opened **TCP** connections. Serious hackers had no difficulty guessing **isns** derived in this manner.

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.



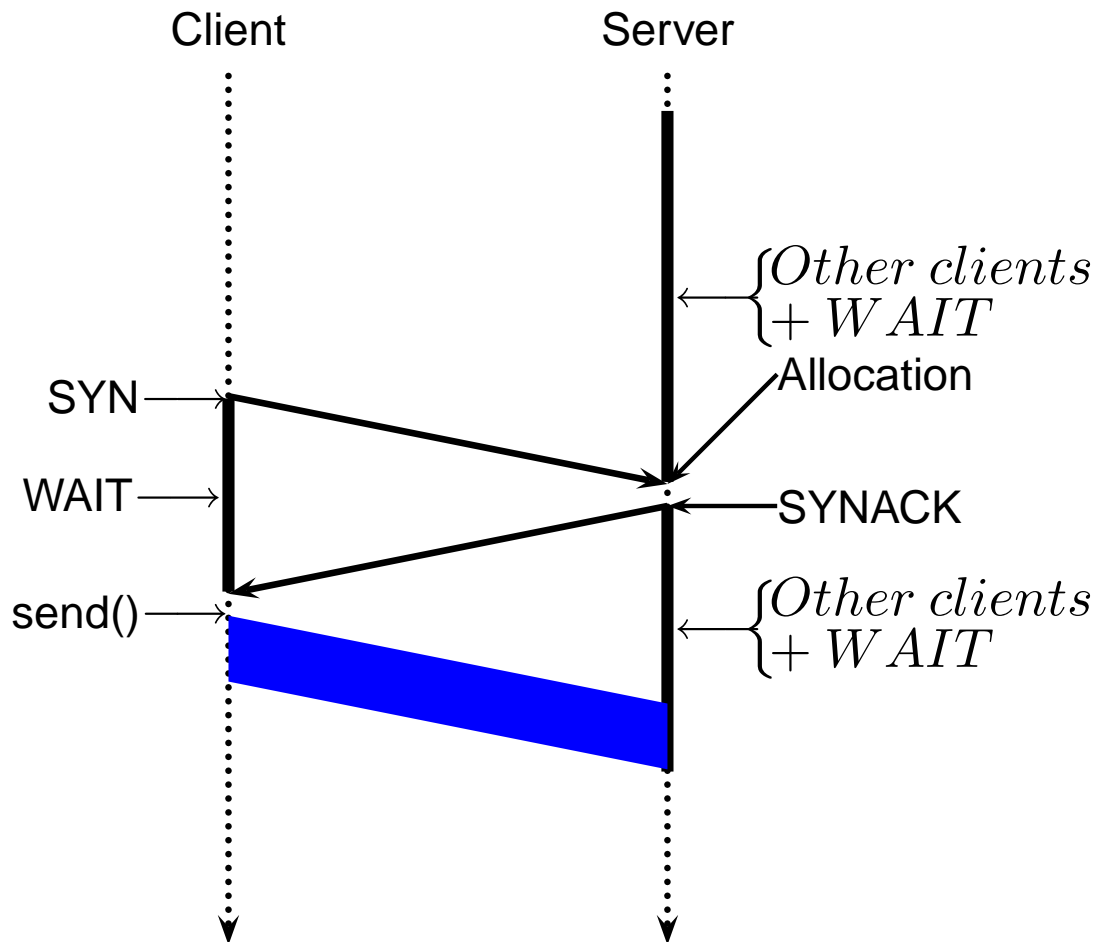
**SYN:** SYN = 1 , ACK = 0 ,  
SeqNum = client\_isn.

**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.

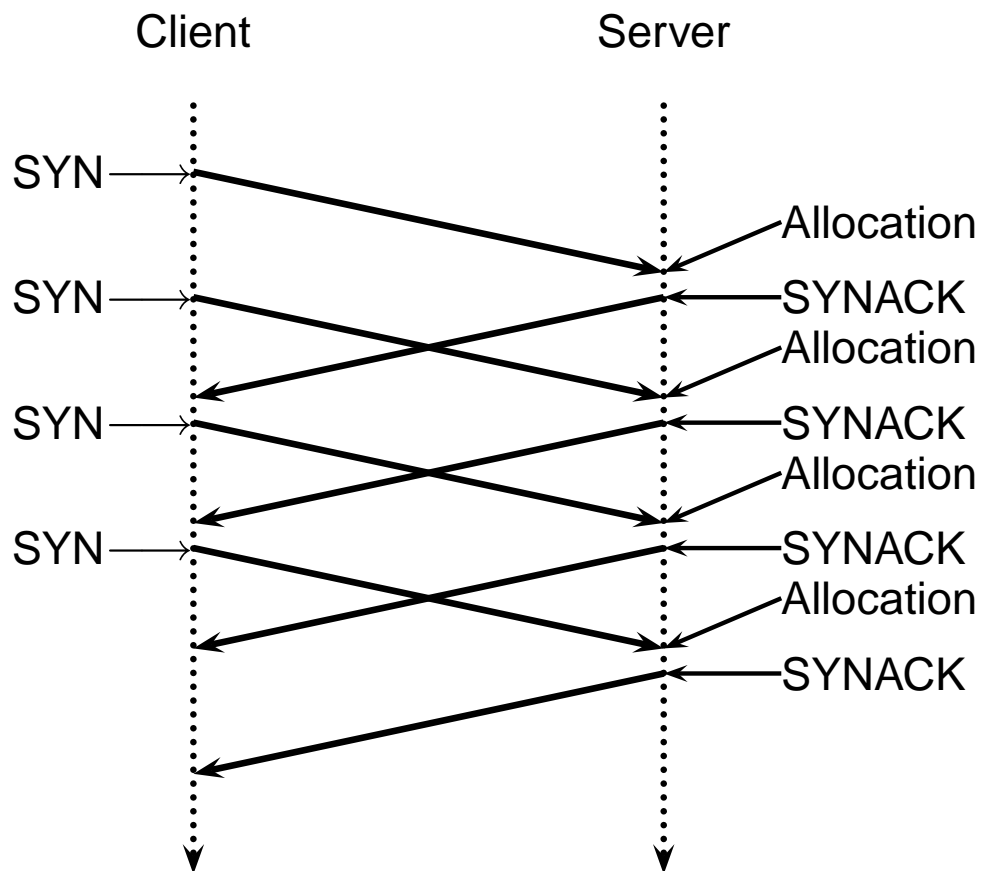
## The SYN flooding attack

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



This implementation works if all the clients are legitimate.

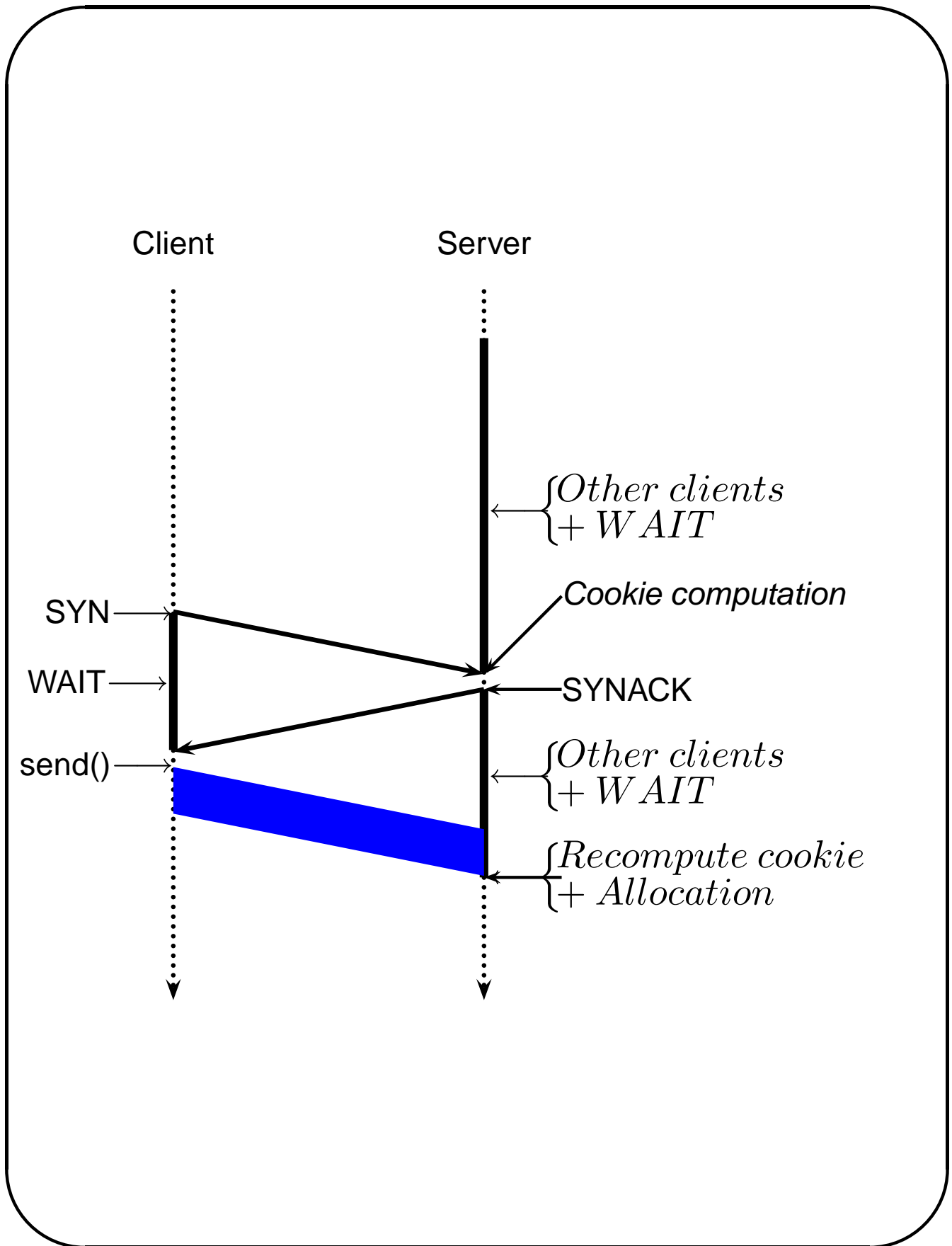
If a client chooses not to complete the 3-way handshake, the data structure kept for connections-in-the-making can be filled, preventing the server from accepting connections.





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.



## 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.