Transmission Control Protocol (TCP)

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

November 10, 2017

Outline

- Introduction to TCP
- Sequence numbers and acknowledgment numbers
- Timeouts and RTT estimation
- Reliable data transfer in TCP
- Connection management

Transmission Control Protocol

- The Internet’s primary transport protocol
  - defined in RFC 793, RFC 1122, RFC 1323, RFC 2018, and RFC 2581

Transmission Control Protocol

- The Internet’s primary transport protocol
  - defined in RFC 793, RFC 1122, RFC 1323, RFC 2018, and RFC 2581
- Connection-oriented service
  - endpoints “shake hands” to establish a connection
  - not a circuit-switched connection, nor a virtual circuit

Transmission Control Protocol

- The Internet’s primary transport protocol
  - defined in RFC 793, RFC 1122, RFC 1323, RFC 2018, and RFC 2581
- Connection-oriented service
  - endpoints “shake hands” to establish a connection
  - not a circuit-switched connection, nor a virtual circuit
- Full-duplex service
  - both endpoints can both send and receive, at the same time

Preliminary Definitions

Preliminary Definitions

- ***TCP segment:*** envelope for TCP data
  - TCP data are sent within TCP segments
  - TCP segments are usually sent within an IP packet

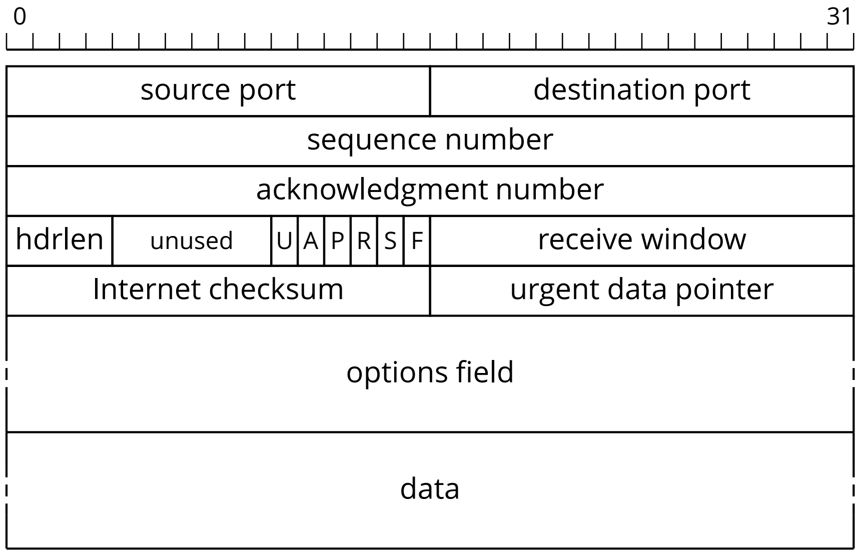
Preliminary Definitions

- ***TCP segment:*** envelope for TCP data
  - TCP data are sent within TCP segments
  - TCP segments are usually sent within an IP packet
- ***Maximum segment size (MSS):*** maximum amount of application data transmitted in a single segment
  - typically related to the MTU of the connection, to avoid network-level fragmentation (we'll talk about all of this later)

Preliminary Definitions

- ***TCP segment:*** envelope for TCP data
  - TCP data are sent within TCP segments
  - TCP segments are usually sent within an IP packet
- ***Maximum segment size (MSS):*** maximum amount of application data transmitted in a single segment
  - typically related to the MTU of the connection, to avoid network-level fragmentation (we'll talk about all of this later)
- ***Maximum transmission unit (MTU):*** largest link-layer frame available to the sender host
  - *path MTU:* largest link-layer frame that can be sent on all links from the sender host to the receiver host

TCP Segment Format



TCP Header Fields

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers
- *Sequence number:* (32-bit) used to implement reliable data transfer
- *Acknowledgment number:* (32-bit) used to implement reliable data transfer

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers
- *Sequence number:* (32-bit) used to implement reliable data transfer
- *Acknowledgment number:* (32-bit) used to implement reliable data transfer
- *Receive window:* (16-bit) size of the “window” on the receiver end

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers
- *Sequence number:* (32-bit) used to implement reliable data transfer
- *Acknowledgment number:* (32-bit) used to implement reliable data transfer
- *Receive window:* (16-bit) size of the “window” on the receiver end
- *Header length:* (4-bit) size of the TCP header in 32-bit words

TCP Header Fields

- *Source and destination ports:* (16-bit each) application identifiers
- *Sequence number:* (32-bit) used to implement reliable data transfer
- *Acknowledgment number:* (32-bit) used to implement reliable data transfer
- *Receive window:* (16-bit) size of the “window” on the receiver end
- *Header length:* (4-bit) size of the TCP header in 32-bit words
- *Optional and variable-length options field:* may be used to negotiate protocol parameters

TCP Header Fields

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown
- *FIN flag:* (1-bit) used during connection shutdown

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown
- *FIN flag:* (1-bit) used during connection shutdown
- *PSH flag:* (1-bit) “push” flag, used to solicit the receiver to pass the data to the application immediately

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown
- *FIN flag:* (1-bit) used during connection shutdown
- *PSH flag:* (1-bit) “push” flag, used to solicit the receiver to pass the data to the application immediately
- *URG flag:* (1-bit) “urgent” flag, used to inform the receiver that the sender has marked some data as “urgent”. The location of this urgent data is marked by the *urgent data pointer* field

TCP Header Fields

- *ACK flag:* (1-bit) signals that the value contained in the *acknowledgment number* represents a valid acknowledgment
- *SYN flag:* (1-bit) used during connection setup and shutdown
- *RST flag:* (1-bit) used during connection setup and shutdown
- *FIN flag:* (1-bit) used during connection shutdown
- *PSH flag:* (1-bit) “push” flag, used to solicit the receiver to pass the data to the application immediately
- *URG flag:* (1-bit) “urgent” flag, used to inform the receiver that the sender has marked some data as “urgent”. The location of this urgent data is marked by the *urgent data pointer* field
- *Checksum:* (16-bit) used to detect transmission errors

Sequence Numbers

Sequence Numbers

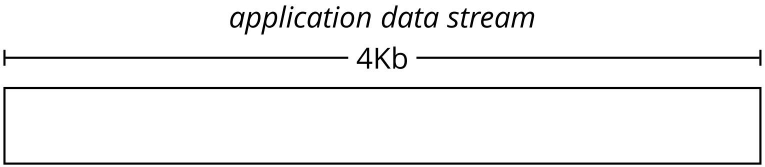
- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before

Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The ***sequence number*** in a TCP segment indicates ***the sequence number of the first byte carried by that segment***

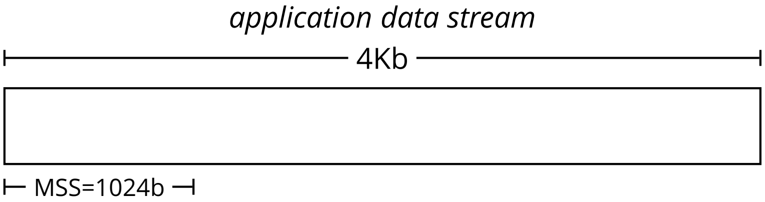
Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The ***sequence number*** in a TCP segment indicates ***the sequence number of the first byte carried by that segment***



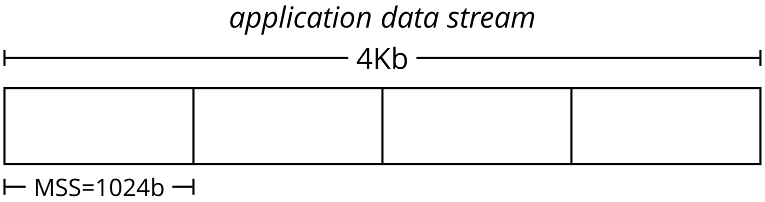
Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The ***sequence number*** in a TCP segment indicates ***the sequence number of the first byte carried by that segment***



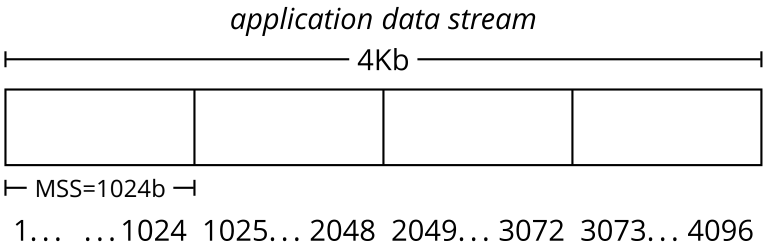
Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The ***sequence number*** in a TCP segment indicates ***the sequence number of the first byte carried by that segment***



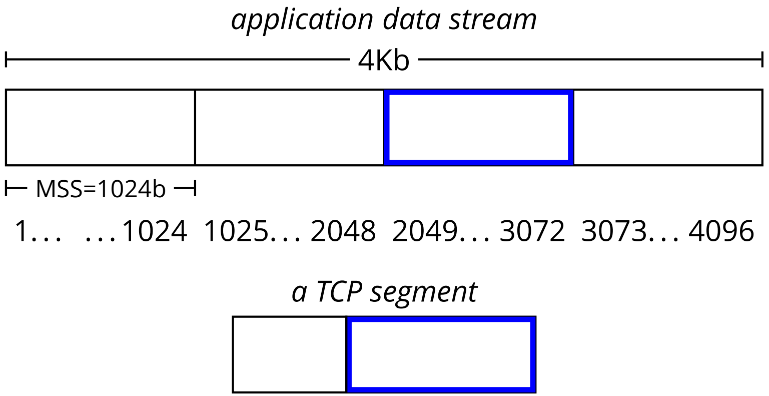
Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The ***sequence number*** in a TCP segment indicates ***the sequence number of the first byte carried by that segment***



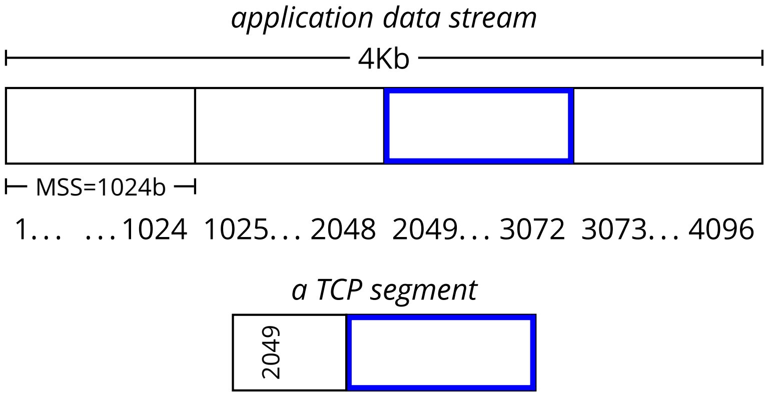
Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The ***sequence number*** in a TCP segment indicates ***the sequence number of the first byte carried by that segment***



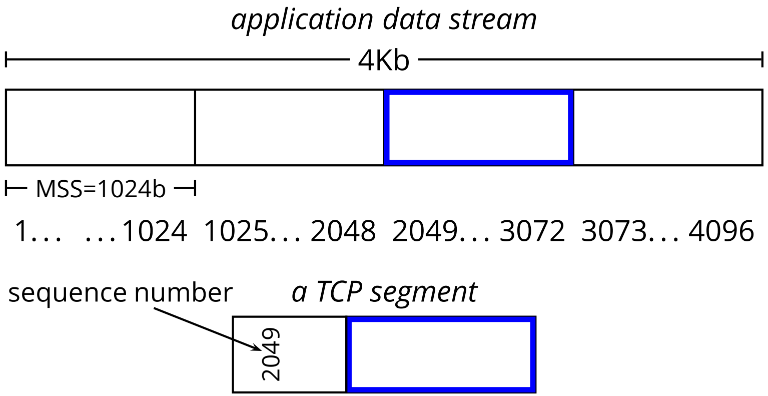
Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The ***sequence number*** in a TCP segment indicates ***the sequence number of the first byte carried by that segment***



Sequence Numbers

- Sequence numbers are associated with *bytes* in the data stream
  - not with segments, as we have used them before
- The ***sequence number*** in a TCP segment indicates ***the sequence number of the first byte carried by that segment***



Acknowledgment Numbers

Acknowledgment Numbers

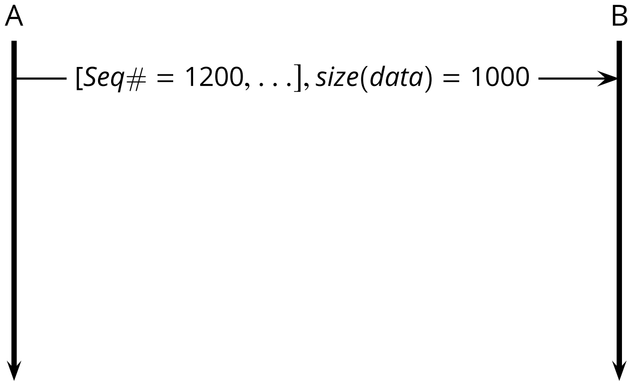
- An ***acknowledgment number*** represents the ***first sequence number not yet seen by the receiver***
  - TCP acknowledgments are *cumulative*

Acknowledgment Numbers

- An ***acknowledgment number*** represents the ***first sequence number not yet seen by the receiver***
  - TCP acknowledgments are *cumulative*

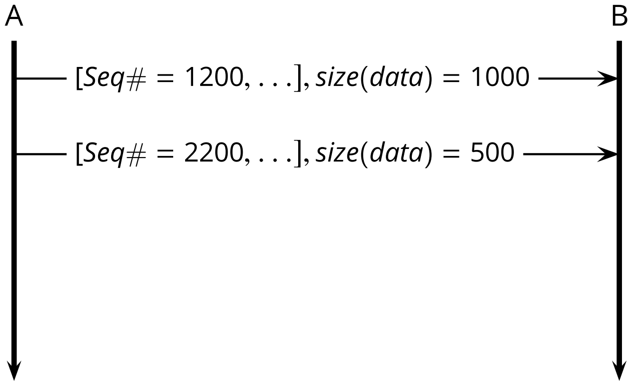
Acknowledgment Numbers

- An ***acknowledgment number*** represents the ***first sequence number not yet seen by the receiver***
  - TCP acknowledgments are *cumulative*



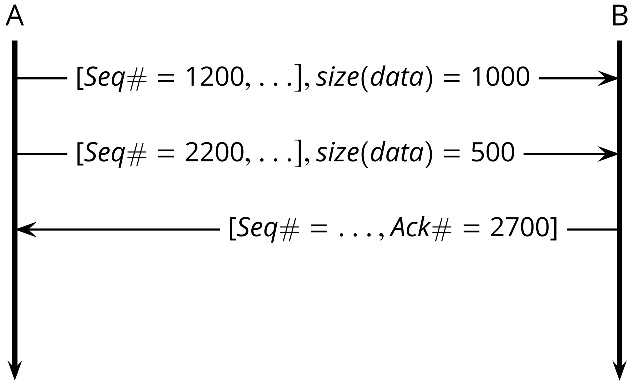
Acknowledgment Numbers

- An ***acknowledgment number*** represents the ***first sequence number not yet seen by the receiver***
  - TCP acknowledgments are *cumulative*


A
B
$[Seq\# = 1200, \ldots], size(data) = 1000$
$[Seq\# = 2200, \ldots], size(data) = 500$

Acknowledgment Numbers

- An ***acknowledgment number*** represents the ***first sequence number not yet seen by the receiver***
  - TCP acknowledgments are *cumulative*


A
B
$[Seq\# = 1200, \ldots], size(data) = 1000$
$[Seq\# = 2200, \ldots], size(data) = 500$
$[Seq\# = \ldots, Ack\# = 2700]$

Sequence Numbers and ACK Numbers

Sequence Numbers and ACK Numbers

- Notice that a TCP connection is a *full-duplex* link
  - therefore, there are ***two streams***
  - two different sequence numbers

Sequence Numbers and ACK Numbers

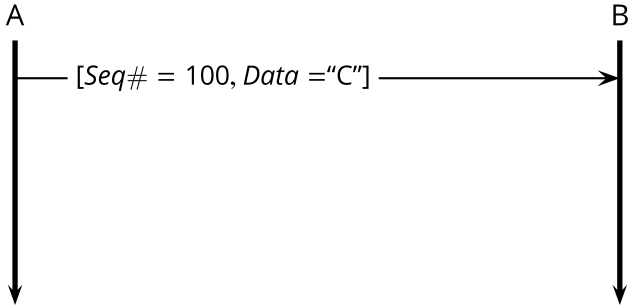
- Notice that a TCP connection is a *full-duplex* link
  - therefore, there are ***two streams***
  - two different sequence numbers

E.g., consider a simple “Echo” application:

Sequence Numbers and ACK Numbers

- Notice that a TCP connection is a *full-duplex* link
  - therefore, there are ***two streams***
  - two different sequence numbers

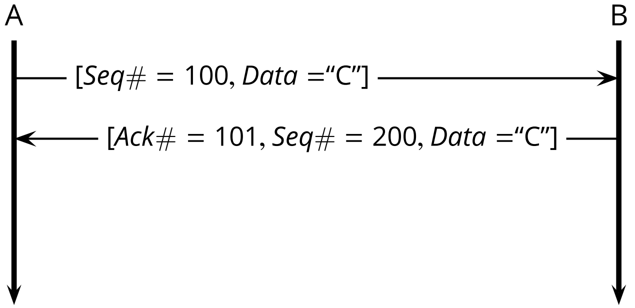
E.g., consider a simple “Echo” application:



Sequence Numbers and ACK Numbers

- Notice that a TCP connection is a *full-duplex* link
  - therefore, there are ***two streams***
  - two different sequence numbers

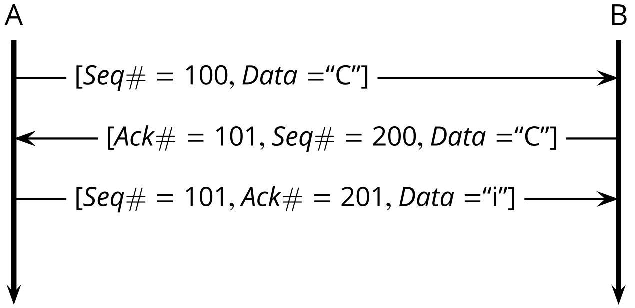
E.g., consider a simple “Echo” application:


A
B
[$Seq\# = 100$, $Data$ =“C”]
[$Ack\# = 101$, $Seq\# = 200$, $Data$ =“C”]

Sequence Numbers and ACK Numbers

- Notice that a TCP connection is a *full-duplex* link
  - therefore, there are ***two streams***
  - two different sequence numbers

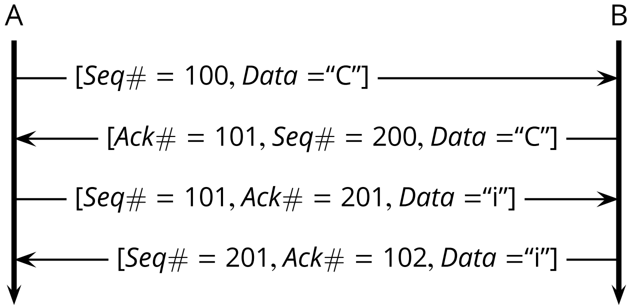
E.g., consider a simple “Echo” application:


A
B
[$Seq\# = 100$, $Data$ =“C”]
[$Ack\# = 101$, $Seq\# = 200$, $Data$ =“C”]
[$Seq\# = 101$, $Ack\# = 201$, $Data$ =“i”]

Sequence Numbers and ACK Numbers

- Notice that a TCP connection is a *full-duplex* link
  - therefore, there are ***two streams***
  - two different sequence numbers

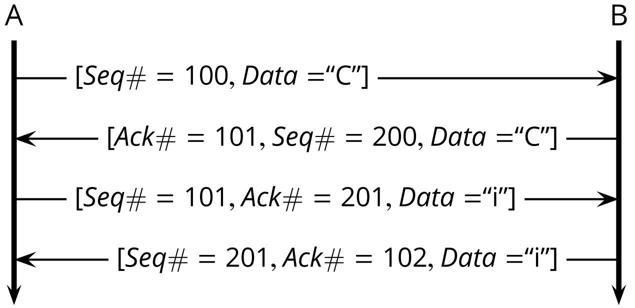
E.g., consider a simple "Echo" application:


A
B
[$Seq\#$ = 100, $Data$ ="C"]
[$Ack\#$ = 101, $Seq\#$ = 200, $Data$ ="C"]
[$Seq\#$ = 101, $Ack\#$ = 201, $Data$ ="i"]
[$Seq\#$ = 201, $Ack\#$ = 102, $Data$ ="i"]

Sequence Numbers and ACK Numbers

- Notice that a TCP connection is a *full-duplex* link
  - therefore, there are ***two streams***
  - two different sequence numbers

E.g., consider a simple "Echo" application:


A
B
[$Seq\# = 100, Data =$"C"]
[$Ack\# = 101, Seq\# = 200, Data =$"C"]
[$Seq\# = 101, Ack\# = 201, Data =$"i"]
[$Seq\# = 201, Ack\# = 102, Data =$"i"]

- Acknowledgments are "piggybacked" on data segments

Reliability and Timeout

- TCP provides reliable data transfer using a *timer* to detect lost segments
  - timeout without an ACK $\longrightarrow$ lost packet $\longrightarrow$ retransmission

Reliability and Timeout

- TCP provides reliable data transfer using a *timer* to detect lost segments
  - timeout without an ACK $\longrightarrow$ lost packet $\longrightarrow$ retransmission
- How long to wait for acknowledgments?

Reliability and Timeout

- TCP provides reliable data transfer using a *timer* to detect lost segments
  - timeout without an ACK $\longrightarrow$ lost packet $\longrightarrow$ retransmission

- How long to wait for acknowledgments?

- Retransmission timeouts should be larger than the round-trip time $RTT = 2L$
  - as close as possible to the RTT

Reliability and Timeout

- TCP provides reliable data transfer using a *timer* to detect lost segments
  - timeout without an ACK $\longrightarrow$ lost packet $\longrightarrow$ retransmission
- How long to wait for acknowledgments?
- Retransmission timeouts should be larger than the round-trip time $RTT = 2L$
  - as close as possible to the RTT
- TCP controls its timeout by continuously *estimating the current RTT*

Round-Trip Time Estimation

Round-Trip Time Estimation

- RTT is measured using ACKs
  - only for packets transmitted once
- Given a single sample S at any given time
- *Exponential weighted moving average (EWMA)*

$$\overline{RTT} = (1-\alpha)\overline{RTT}' + \alpha S$$

Round-Trip Time Estimation

- RTT is measured using ACKs
  - only for packets transmitted once
- Given a single sample *S* at any given time
- *Exponential weighted moving average (EWMA)*

$$\overline{RTT} = (1-\alpha)\overline{RTT}' + \alpha S$$

  - RFC 2988 recommends $\alpha = 0.125$

Round-Trip Time Estimation

- RTT is measured using ACKs
  - only for packets transmitted once
- Given a single sample S at any given time
- *Exponential weighted moving average (EWMA)*

$$\overline{RTT} = (1-\alpha)\overline{RTT}' + \alpha S$$

  - RFC 2988 recommends $\alpha = 0.125$
- TCP also measures the *variability of RTT*

$$\overline{DevRTT} = (1-\beta)\overline{DevRTT}' + \beta|\overline{RTT}' - S|$$

Round-Trip Time Estimation

- RTT is measured using ACKs
  - only for packets transmitted once

- Given a single sample *S* at any given time

- *Exponential weighted moving average (EWMA)*

$$\overline{RTT} = (1-\alpha)\overline{RTT}' + \alpha S$$

  - RFC 2988 recommends $\alpha = 0.125$

- TCP also measures the *variability of RTT*

$$\overline{DevRTT} = (1-\beta)\overline{DevRTT}' + \beta|\overline{RTT}' - S|$$

  - RFC 2988 recommends $\beta = 0.25$

Timeout Value

Timeout Value

- The timeout interval *T* must be larger than the RTT
  - so as to avoid unnecessary retransmission
- However, *T* should not be too far from RTT
  - so as to detect (and retransmit) lost segments as quickly as possible

Timeout Value

- The timeout interval T must be larger than the RTT
  - so as to avoid unnecessary retransmission
- However, T should not be too far from RTT
  - so as to detect (and retransmit) lost segments as quickly as possible
- TCP sets its timeouts using the estimated RTT ($\overline{RTT}$) and the variability estimate $\overline{DevRTT}$:

$$T = \overline{RTT} + 4\overline{DevRTT}$$

Reliable Data Transfer (Sender)

A simplified TCP sender

- r_send(*data*)

```
if (timer not running)
    start_timer()
u_send([$data$,$next\_seq\_num$])
$next\_seq\_num \leftarrow next\_seq\_num + length(data)$
```

Reliable Data Transfer (Sender)

A simplified TCP sender

- r_send(*data*)

```
if (timer not running)
    start_timer()
u_send([data,next_seq_num])
$next\_seq\_num \leftarrow next\_seq\_num + length(data)$
```

- timeout

```
u_send(pending segment with smallest sequence number)
start_timer()
```

Reliable Data Transfer (Sender)

A simplified TCP sender

- r_send(*data*)

```
if (timer not running)
    start_timer()
u_send([$data$,$next\_seq\_num$])
$next\_seq\_num \leftarrow next\_seq\_num + length(data)$
```

- timeout

```
u_send(pending segment with smallest sequence number)
start_timer()
```

- u_recv([ACK,*y*])

```
if ($y > base$)
    $base \leftarrow y$
    if (there are pending segments)
        start_timer()
else ...
```

Acknowledgment Generation (Receiver)

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
    - *Delayed ACK:* wait 500ms for another in-order segment; If that does not arrive, send ACK

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment; If that does not arrive, send ACK
- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment; If that does not arrive, send ACK
- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment; If that does not arrive, send ACK
- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)
- Arrival of out of order segment with higher-than-expected sequence number (gap detected)

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment; If that does not arrive, send ACK
- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)
- Arrival of out of order segment with higher-than-expected sequence number (gap detected)
  - *Duplicate ACK:* immediately send duplicate ACK

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment; If that does not arrive, send ACK
- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)
- Arrival of out of order segment with higher-than-expected sequence number (gap detected)
  - *Duplicate ACK:* immediately send duplicate ACK
- Arrival of segment that (partially or completely) fills a gap in the received data

Acknowledgment Generation (Receiver)

- Arrival of in-order segment with expected sequence number; all data up to expected sequence number already acknowledged
  - *Delayed ACK:* wait 500ms for another in-order segment; If that does not arrive, send ACK
- Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK (see above)
  - *Cumulative ACK:* immediately send cumulative ACK (for both segments)
- Arrival of out of order segment with higher-than-expected sequence number (gap detected)
  - *Duplicate ACK:* immediately send duplicate ACK
- Arrival of segment that (partially or completely) fills a gap in the received data
  - *Immediate ACK:* immediately send ACK if the packet start at the lower end of the gap

Reaction to ACKs (Sender)

Reaction to ACKs (Sender)

- u_recv([ACK,*y*])

```
if ($y > base$)
    $base \leftarrow y$
    if (there are pending segments)
        start_timer()
```

Reaction to ACKs (Sender)

- u_recv([ACK,y])

```
if ($y > base$)
    $base \leftarrow y$
    if (there are pending segments)
        start_timer()
else
    $ack\_counter[y] \leftarrow ack\_counter[y] + 1$
    if ($ack\_counter[y] = 3$)
        u_send(segment with sequence number $y$)
```

Connection Setup

Connection Setup

Three-way handshake

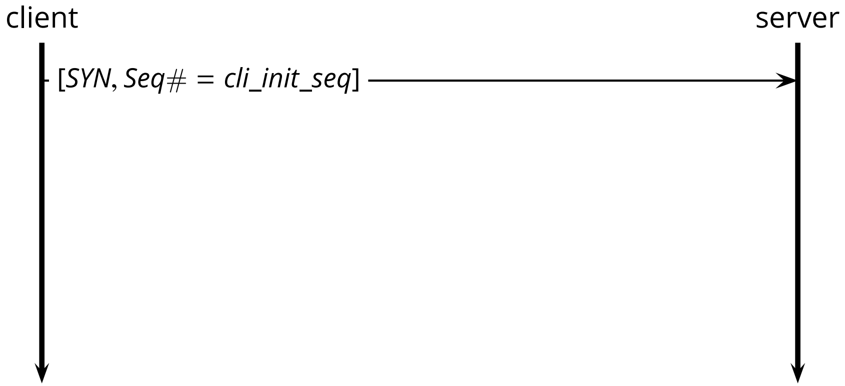
Connection Setup

Three-way handshake

client
server

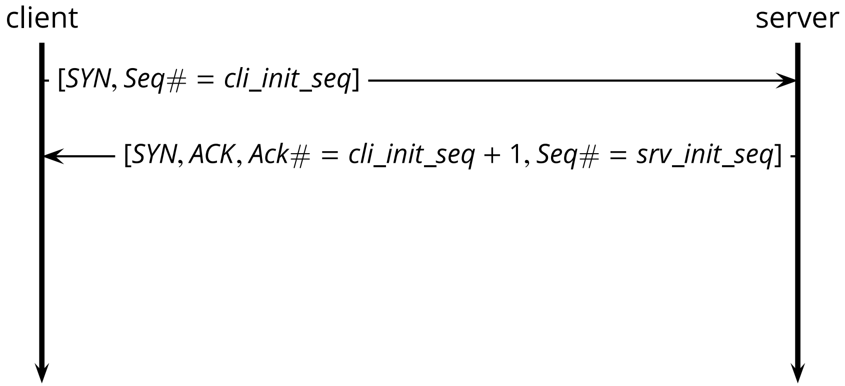
Connection Setup

Three-way handshake



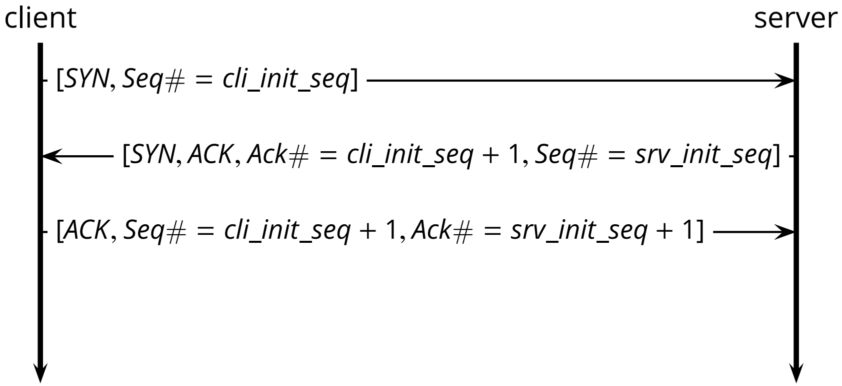
Connection Setup

Three-way handshake


client
server
$[SYN, Seq\# = cli\_init\_seq]$
$[SYN, ACK, Ack\# = cli\_init\_seq + 1, Seq\# = srv\_init\_seq]$

Connection Setup

Three-way handshake


client
server
[SYN, $Seq\#$ = cli_init_seq]
[SYN, ACK, $Ack\#$ = $cli\_init\_seq + 1$, $Seq\#$ = srv_init_seq]
[ACK, $Seq\#$ = $cli\_init\_seq + 1$, $Ack\#$ = $srv\_init\_seq + 1$]

Connection Shutdown

"This is it."
"Okay, Bye now."
"Bye."

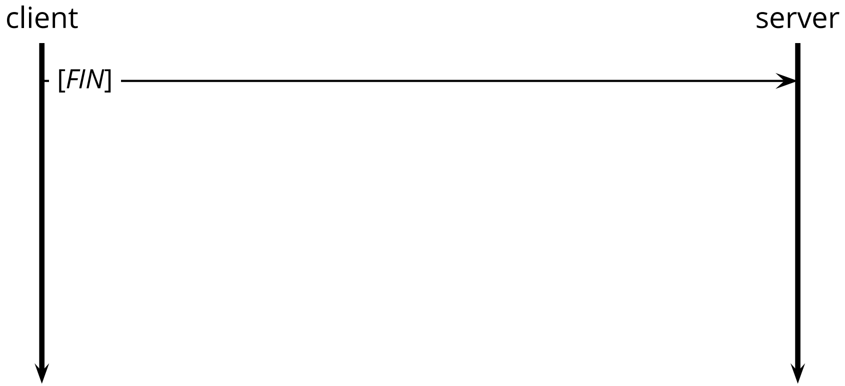
Connection Shutdown

“This is it.”
“Okay, Bye now.”
“Bye.”

client
server

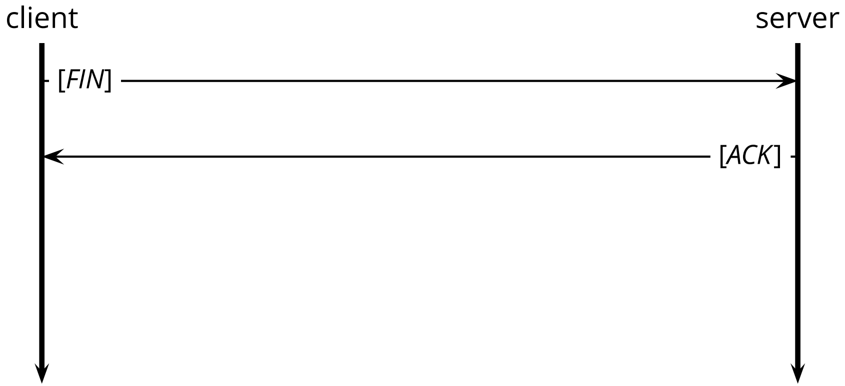
Connection Shutdown

“This is it.”
“Okay, Bye now.”
“Bye.”



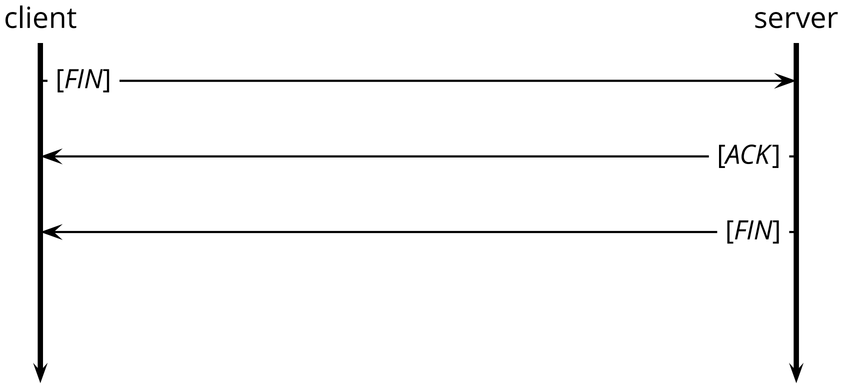
Connection Shutdown

"This is it."
"Okay, Bye now."
"Bye."



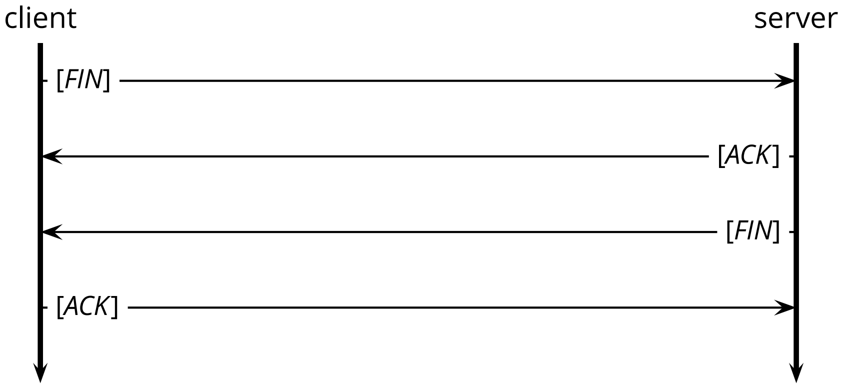
Connection Shutdown

“This is it.”
“Okay, Bye now.”
“Bye.”



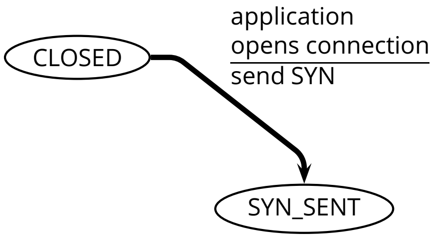
Connection Shutdown

“This is it.”
“Okay, Bye now.”
“Bye.”

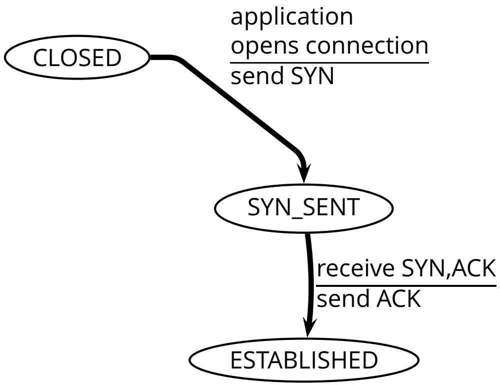


The TCP State Machine (Client)

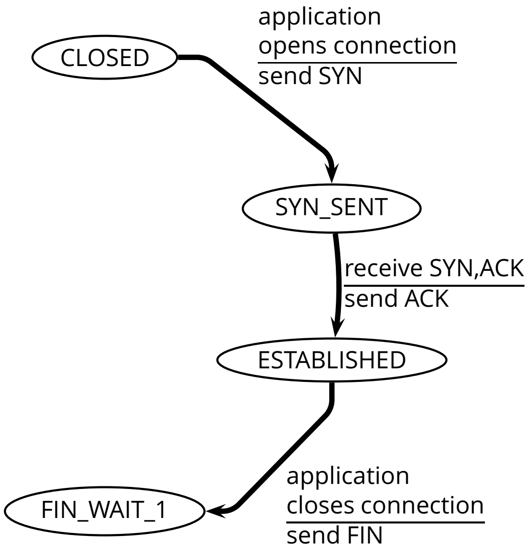
The TCP State Machine (Client)



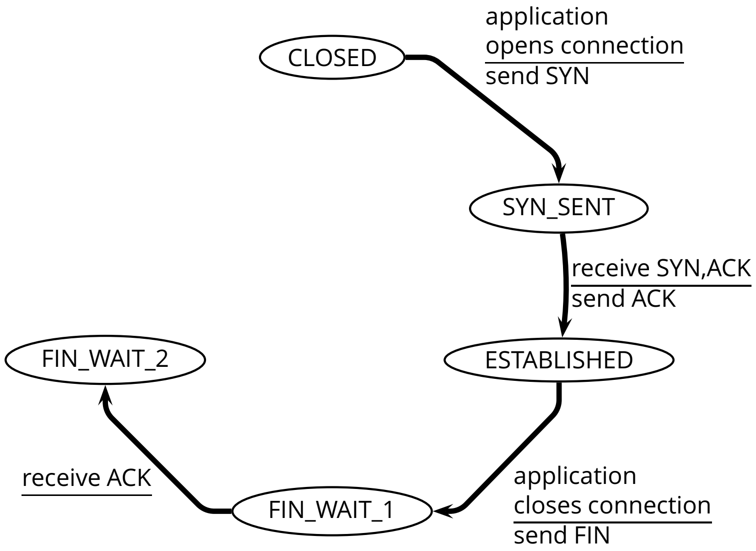
The TCP State Machine (Client)



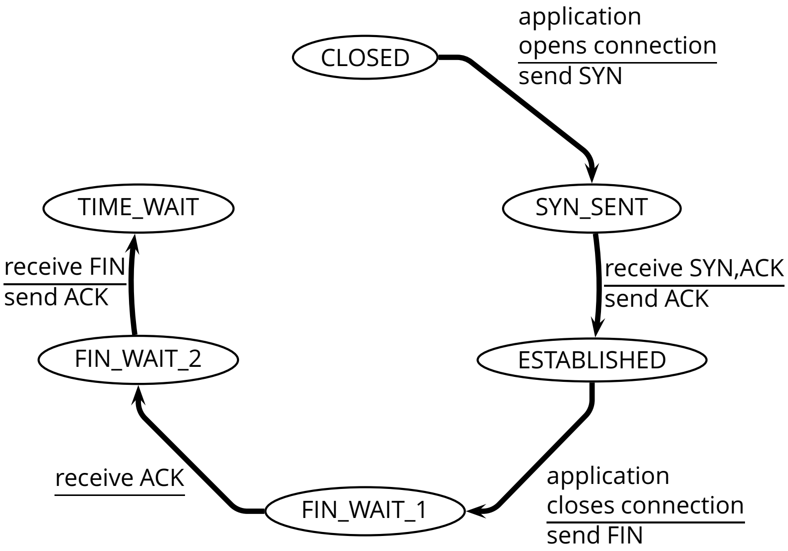
The TCP State Machine (Client)



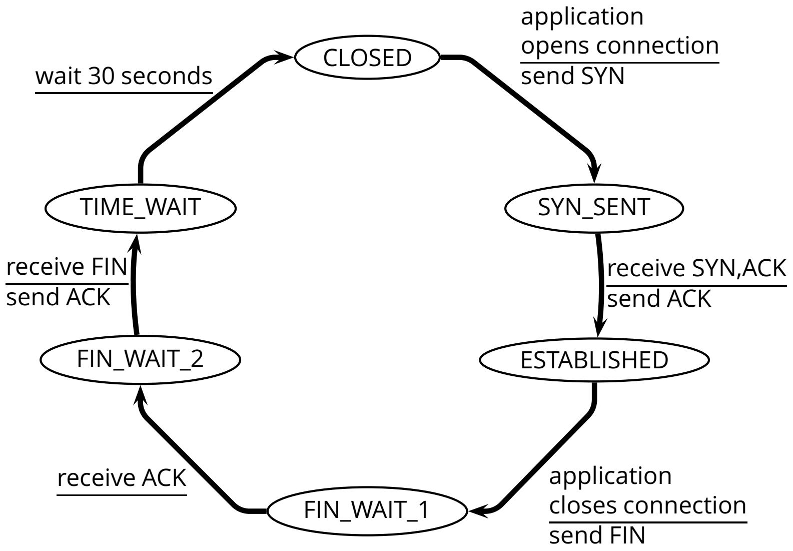
The TCP State Machine (Client)



The TCP State Machine (Client)

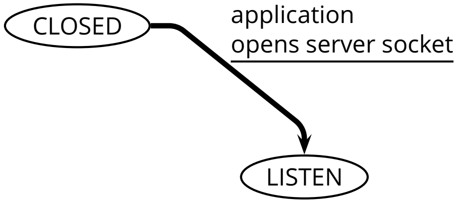


The TCP State Machine (Client)

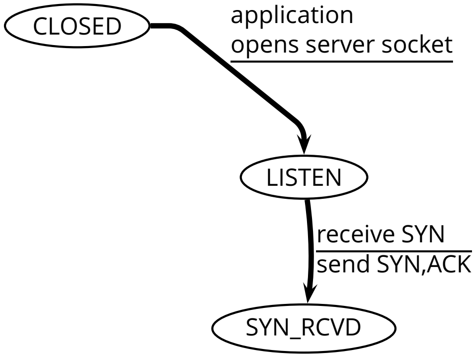


The TCP State Machine (Server)

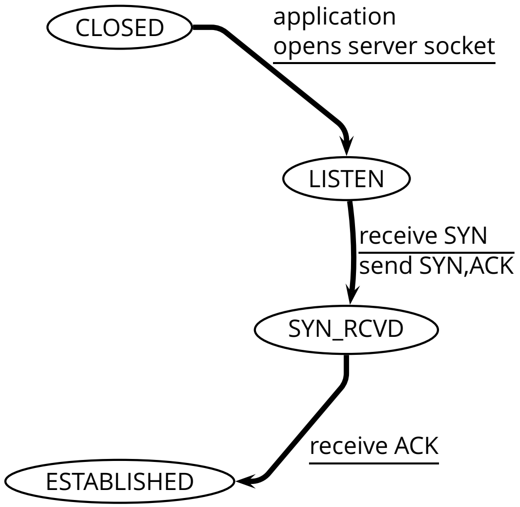
The TCP State Machine (Server)



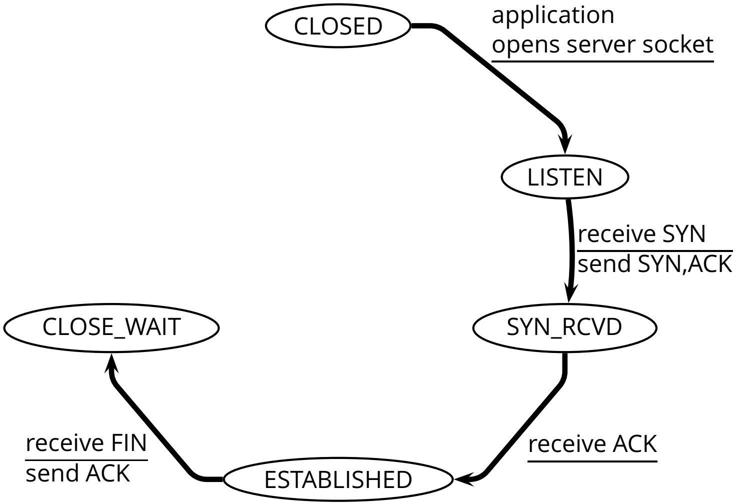
The TCP State Machine (Server)



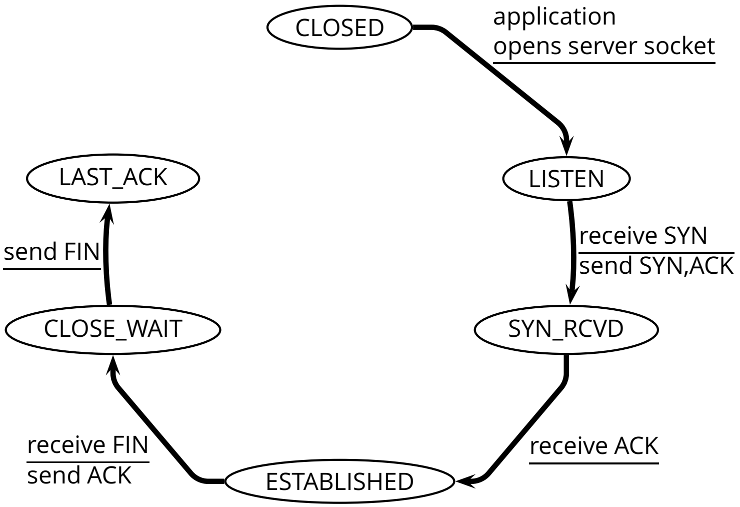
The TCP State Machine (Server)



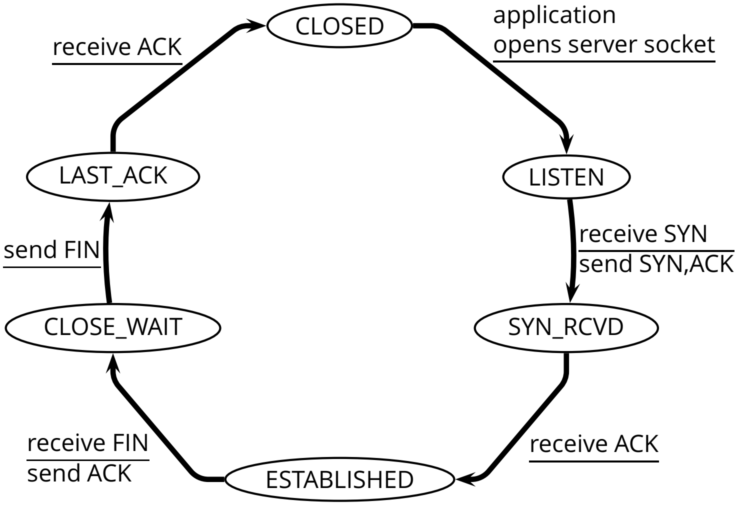
The TCP State Machine (Server)



The TCP State Machine (Server)



The TCP State Machine (Server)


CLOSED
application
opens server socket
LISTEN
receive SYN
send SYN,ACK
SYN_RCVD
receive ACK
ESTABLISHED
receive FIN
send ACK
CLOSE_WAIT
send FIN
LAST_ACK
receive ACK