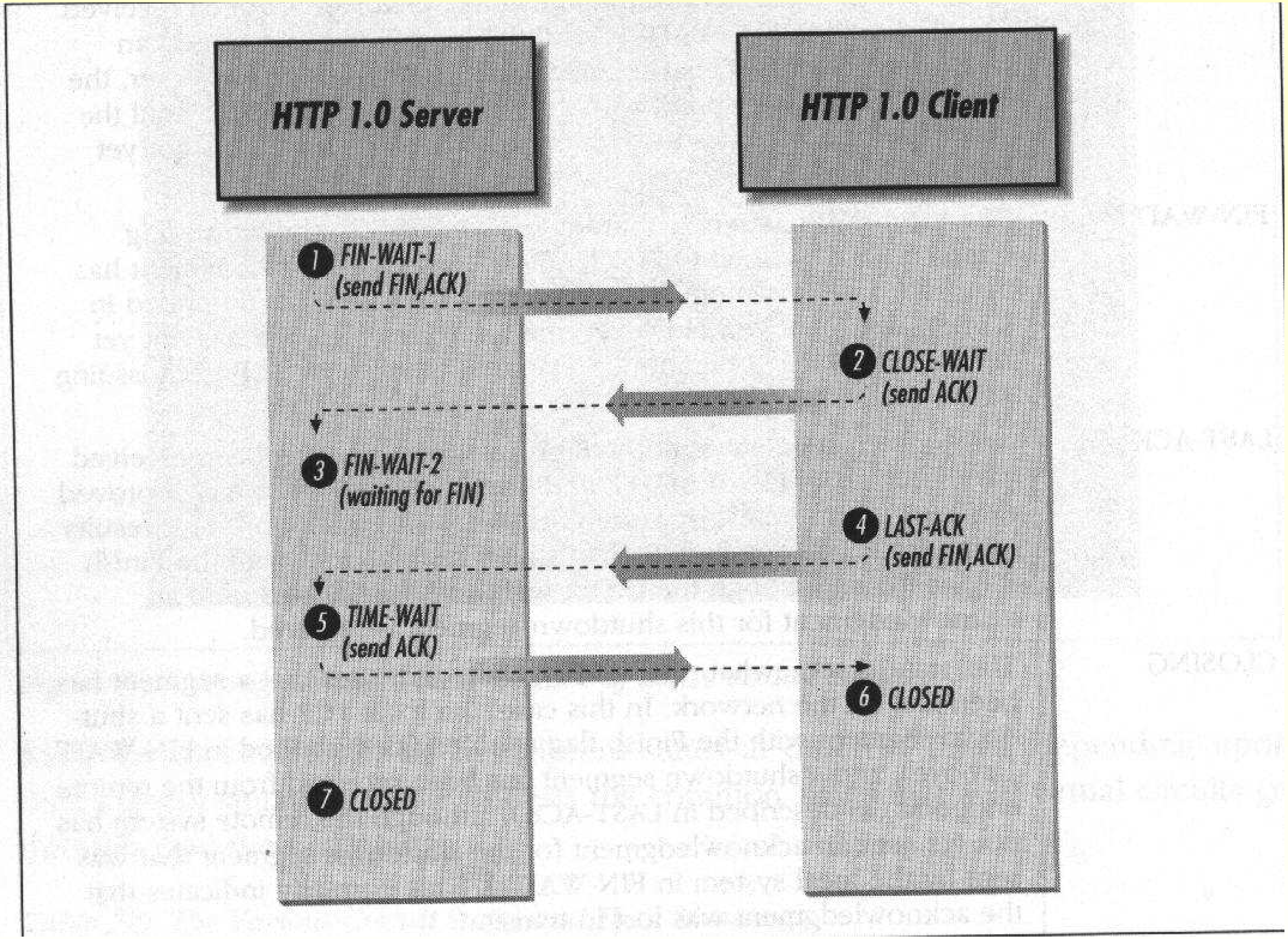
TCP States

LISTEN	passive open: server waits for conn
SYN_SENT	active open: client has sent a SYN
SYN_RECVD	server has received client's SYN
ESTABLISHED	conn operational
FIN_WAIT1	process has sent a FIN
FIN_WAIT2	FIN has been ACKed
CLOSE_WAIT	process has received a FIN
LAST_ACK	upon receiving FIN, process has sent a FIN
CLOSING	upon sending FIN, process has received a FIN
TIME_WAIT	shutdown completed, wait 2MSL
CLOSED	shutdown completed, (host, port) pair disappe

TCP Shutdown Example



netstat – network monitoring

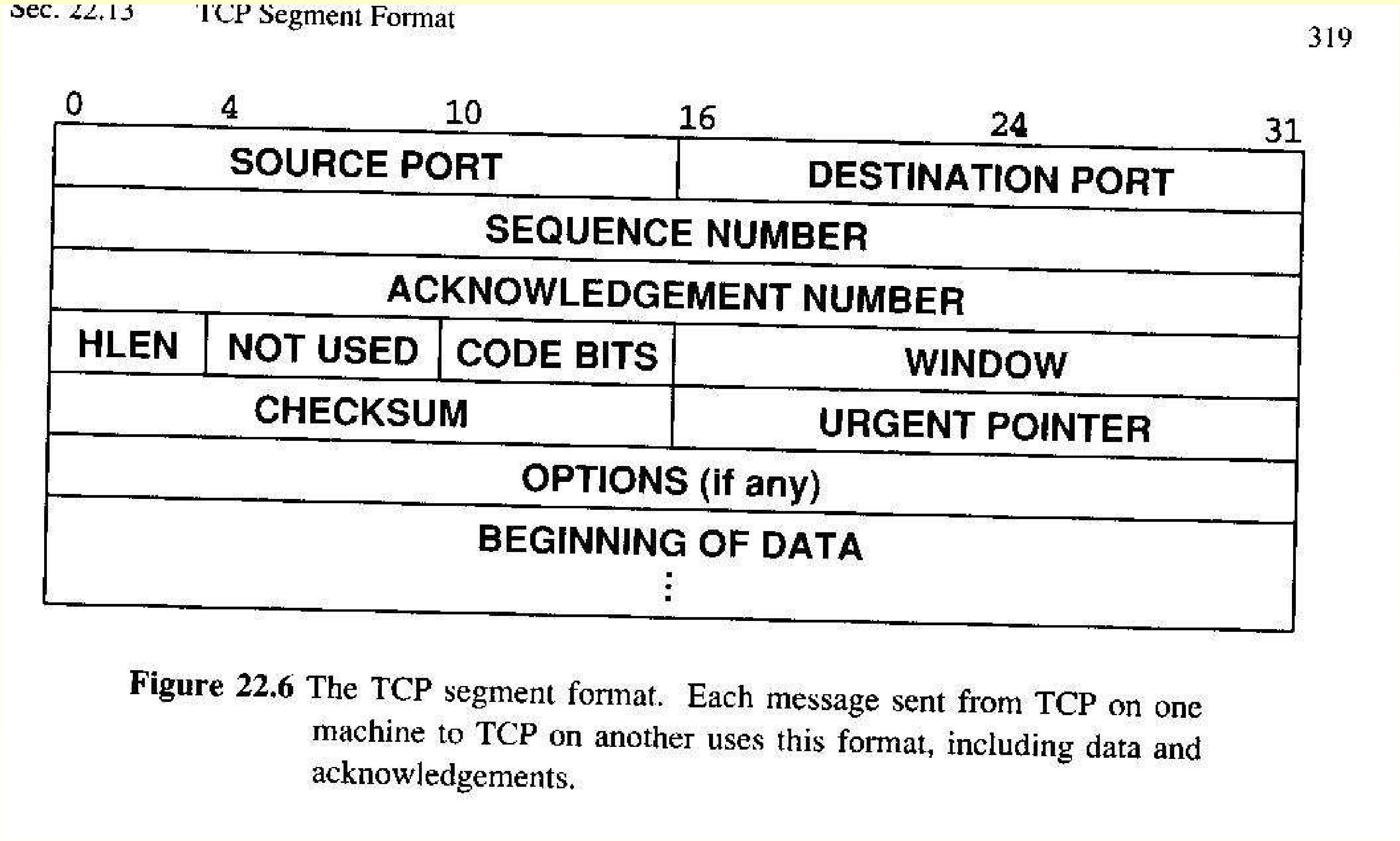
```
TCP: IPv4
Local Address     Remote Address      State
----------------- -----------------  -------
 *.rpcbind        *.*                LISTEN
 *.ftp            *.*                LISTEN
 *.telnet         *.*                LISTEN
 *.shell          *.*                LISTEN
 *.login          *.*                LISTEN
 *.echo           *.*                LISTEN
stl-s-stud.1022   stl-s-ad.683       ESTABLISHED
stl-s-stud.35755  stl-s-ad.902       CLOSE_WAIT
stl-s-stud.43470  stl-s-stud.1000    CLOSE_WAIT
stl-s-stud.40562  stl-s-stud.1000    CLOSE_WAIT
stl-s-stud.41396  stl-s-stud.1000    CLOSE_WAIT
stl-s-stud.1000   stl-s-stud.41396   FIN_WAIT_2
stl-s-stud.1000   stl-s-stud.41469   TIME_WAIT
```

```
stl-s-stud.42912  stl-s-stud.1000    CLOSE_WAIT
stl-s-stud.1000   stl-s-stud.42912   FIN_WAIT_2
stl-s-stud.1000   stl-s-stud.43727   FIN_WAIT_2
```

Control Flags

Flag	meaning	remarks
SYN	synchronize	sync seq numbers
FIN	finish	sending is terminated
RST	reset	conn aborted or reject conn request
ACK	acknowledgement	acknowledge data
URG	urgent	urgent data contained in segment
PSH	push	flush buffer immediately

TCP Segment Format



0 4 10 16 24 31

<table>
<tr><td colspan="3">SOURCE PORT</td><td>DESTINATION PORT</td></tr>
<tr><td colspan="4">SEQUENCE NUMBER</td></tr>
<tr><td colspan="4">ACKNOWLEDGEMENT NUMBER</td></tr>
<tr><td>HLEN</td><td>NOT USED</td><td>CODE BITS</td><td>WINDOW</td></tr>
<tr><td colspan="3">CHECKSUM</td><td>URGENT POINTER</td></tr>
<tr><td colspan="4">OPTIONS (if any)</td></tr>
<tr><td colspan="4">BEGINNING OF DATA
⋮</td></tr>
</table>

Figure 22.6 The TCP segment format. Each message sent from TCP on one machine to TCP on another uses this format, including data and acknowledgements.

Windowing

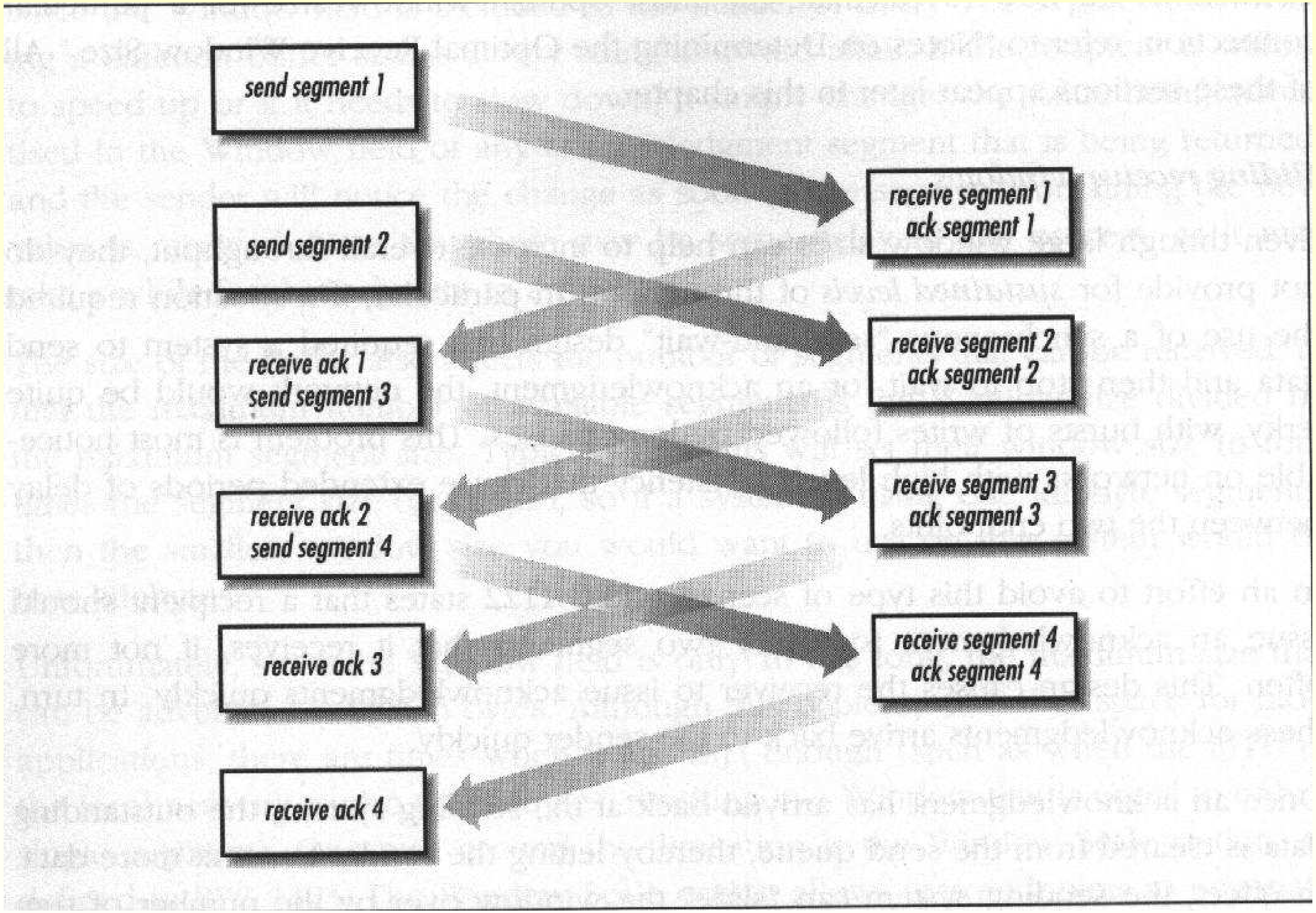
improve performance by allowing a window of non–ACKed data

bounded by 16 bit ⇝max 64K non–ACKed data

sliding window:

```
RECEIVING SITE         offered window
                       ------------------------
              1  2  3  4  5  6  7  8  9  10  11  12  13
              -------  ----------  ------------  ------
SENDING SITE  sent+ACK sent/noACK  can be sent   wait
```

Windowing Illustrated



Silly Window Syndrome

scenario

- window of 64K
- sender sends at full throttle

$\leadsto$ receiving buffer fills quickly

- receiver clears 1K from buffer due to `read(...,...,1024)`

$\leadsto$ window 1K

- sender sends 1K immediately
- receiver clears 1K from buffer due to `read(...,...,1024)`

$\leadsto$ window 1K

- ...

⇝window size never bigger than 1K

⇝lots of traffic for small data size

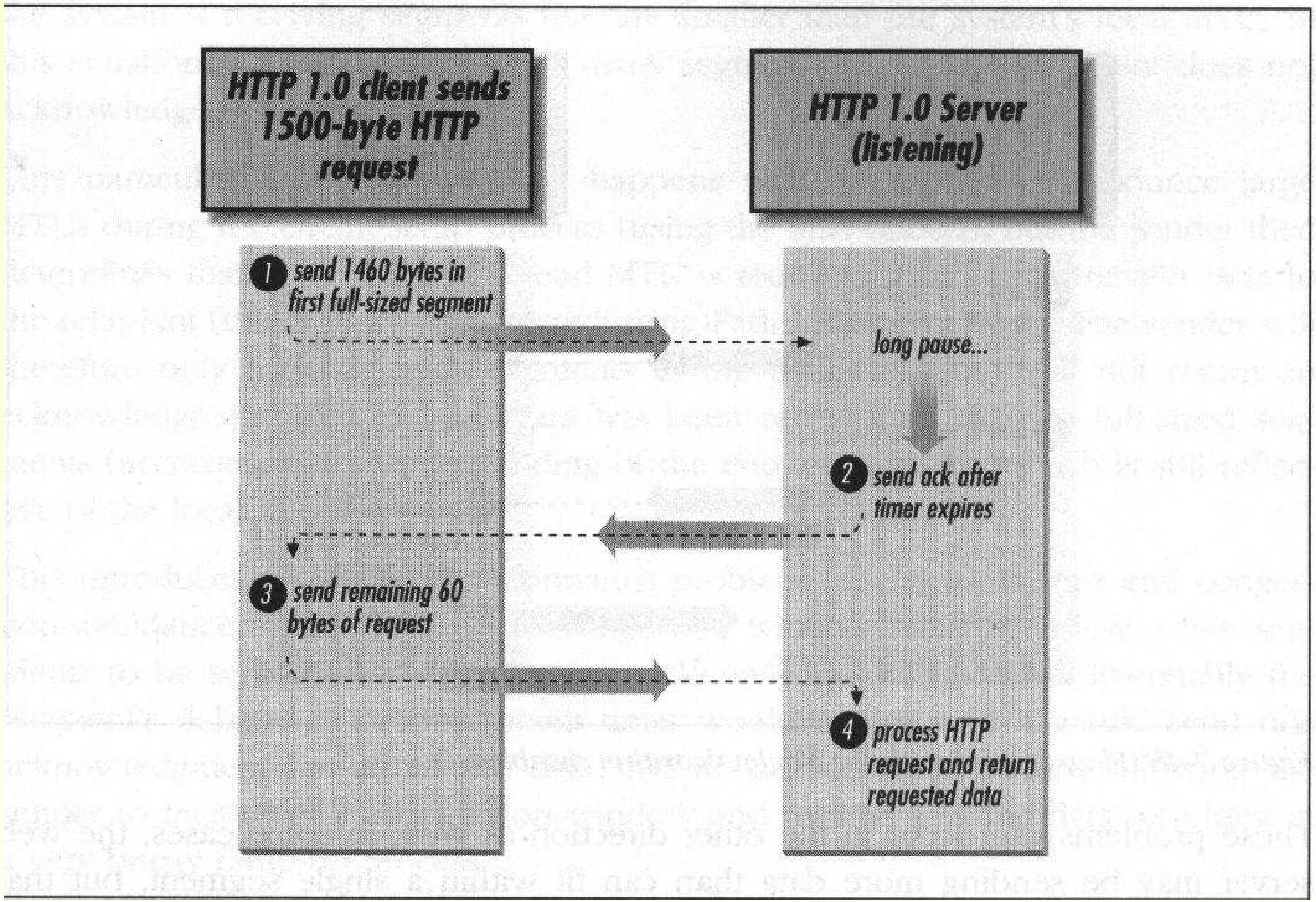
Nagle Algorithm

avoids silly windows

attention: may be intentional (see TELNET)

send small segment only if all prior segments are acknowledged

problem: interacts with delayed ACK


HTTP 1.0 client sends 1500-byte HTTP request
HTTP 1.0 Server (listening)
1 send 1460 bytes in first full-sized segment
long pause...
2 send ack after timer expires
3 send remaining 60 bytes of request
4 process HTTP request and return requested data

Congestion

route with capacity R bytes/sec

two hosts sending each at R bytes/sec

throughput is $R/2$ for each, but . . .

delay for indiviual packet tends to ∞

typical scenario : burst – stalled – burst – stalled . . .

$\leadsto$congestion must be avoided

Congestion Countermeasures

1. **Slow Start** (1 Segment)
2. exponential growth until threshold reached
3. **Congestion Avoidance** (linear growth)

on each timeout begin with Slow Start again and threshold/2

auxiliary variable: congestion window

send **min(cong-win,recv-win)** bytes

research for best congestion algorithms still going on

this algorithm valuable for SMTP, FTP, TELNET, but . . .

how about voice over IP, video–on–demand

TCP Errors

- rejected connections
  - no process listening $\leadsto$ICMP
  - firewall rejects request $\leadsto$ICMP, silent dropping
  - application closes connection immediately after `ESTABLISHED`
- lost connections
  - network failure during `ESTABLISHED`
  - host power outage

  $\leadsto$ timeout, retransmit

TCP Application Quirks

- no graceful shutdown
  - following a performance recommendation to avoid shutdown sequence
  - requests for retransmission are answered by RST segments
  - stalled client . . . World Wide Wait
- weird flag combinations (hacker probe tools)

TCP–Client

1. fill `struct sockaddr_in`
2. create socket
3. connect to server
4. send / write
5. recv / read
6. shutdown the connection
7. close socket

TCP–Client Functions

```
int  connect(int sockfd, const struct sockaddr *serv_addr,
               socklen_t addrlen);
/* 0=ok, -1=error */

EBADF (no filedes), ENOTSOCK (filedes but not socket),
EFAULT (socket not allocated by process), EISCONN (already connecte
ECONNREFUSED (refused), ETIMEDOUT (server busy),
ENETUNREACH (network unreachable), EAGAIN (local ports exhausted),

int send(int s, const void *msg, size_t len, int flags);
/* return # characters or -1 on error */

int write(int s, const void *msg, size_t len);
/* return # characters or -1 on error */
```

```
int recv(int s, void *msg, size_t len, int flags);
/* return # characters available, blocks if none */

int read(int s, void *msg, size_t len);
/* return # characters or -1 on error */

int shutdown(int s, int how);
```

If how is 0, further receives will be disallowed. If how is 1, further sends will be disallowed. If how is 2, further sends and receives will be disallowed.

return 0 on success, -1 on error

```
int close(int fd);
```

TCP–Server

1. fill `struct sockaddr_in`
2. create server socket
3. bind server socket to address
4. listen on server socket
5. accept ⇝new socket
6. read from new socket
7. write to new socket
8. shutdown new socket
9. close new socket ⇝accept next connection
10. on process termination close server socket

TCP–Functions Server Only

```
int listen(int s, int backlog);
```

backlog=queue length for waiting connect()’s

return 0 on success, error=-1

```
int accept(int s, struct sockaddr *addr,
           socklen_t *addrlen);
```

blockiert, uebernehme Verbindung,

return new socket or -1 (error)

IPv6

IPv4 address space too small (exhausted between 2008 and 2018)

now 128 Bit addresses $\approx 10^{38}$ hosts

40–byte fixed length header

flow descriptor (audio/video)

no fragmentation anymore $\rightsquigarrow$ICMP packet too big

no header checksum anymore (TCP/UDP do it anyway)

new ICMP