System Support to Socket Programming

Chapter 2 of Stevens textbook, Edition 2, Volume 1.
Section 6.4 of Tanenbaum textbook, Edition 4.

1. Aims:
   - Better understanding of Socket Programming.
   - Unix Kernel Support of TCP/IP.
   - Difference of TCP and UDP.
   - To study some research problems of TCP/IP.

2. Hierarchy of Implementation

- Socket interface, a set of transport service primitives, are not always in the kernel. Note the difference between Berkeley sockets and System V XTI.
- We are lucky that the socket interface is as simple as a set of system calls. But its system implementation is actually complicated and tricky. We are now ready to get into details.
3. Message Format

![Message Format Diagram]

<table>
<thead>
<tr>
<th>Features</th>
<th>TCP</th>
<th>UDP</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Connection</td>
<td>✓</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>2  Message boundary</td>
<td>×</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3  Data check sum</td>
<td>✓</td>
<td>optional</td>
<td>Data corrupt</td>
</tr>
<tr>
<td>4  ack</td>
<td>✓</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>5  Timeout and retransmit</td>
<td>✓</td>
<td>×</td>
<td>TTL, RTT, packet lost</td>
</tr>
<tr>
<td>6  Sequencing</td>
<td>✓</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>7  Flow control</td>
<td>✓</td>
<td>×</td>
<td>Buffer, receive window, congestion window</td>
</tr>
<tr>
<td>8  Error control</td>
<td>✓</td>
<td>×</td>
<td>= 3 + 5 + 6</td>
</tr>
<tr>
<td>9  flow</td>
<td></td>
<td>Byte stream</td>
<td>datagram</td>
</tr>
<tr>
<td>10 buffer</td>
<td>✓</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>11 OOB</td>
<td>✓</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

Segmentation can take place both at TCP layer (MSS) and IP layer (MTU).

4. TCP and UDP
Analogy of TCP and Telephone:
Socket---------------------- register
Bind ---------------------- Tell phone number
Listen --------------------- set ringer on
Connect------------------- dial
Accept--------------------- pick up
Gethostbyname------------- lookup phone book by name
Gethostby addr------------ lookup phone book by phone number
See RFC 793 for first formally defined TCP, RFC 1122 for bugs fixed, and RFC 1323 for extensions.

5. Connection establishment and termination
5.1 3-way handshaking (Tomlinson, 1975)

- **SYN J**: to synchronize/negotiate between Client and Server. J is the starting sequence number.
- **J→J+1, K→K+1**: SYN or ACK occupies 1 byte of sequence number space, so acknowledgement number is increased by 1.
- **Options**: **MSS**: Maximum amount of data for each segment, TCP_MAXSEG is used to change MSS using setsockopt( ) and getsockopt( ).
  - **Window Size and Scale**: for flow control. Maximum amount of data I can or I like to receive. SO_RCVBUF is for this option.
  - **Timestamp**: check the repeated lost packets.
- **Example:** MSS=1460, window size=4k, window size is dynamically changing.
- **Bi-directional or Full-Duplex:** 2 buffers.
- **Point-to-point:** Doesn’t support multicasting or broadcasting.
- **Why is it 3 ways, not 2 ways or 4 ways?**
- **Why can 3-way handshaking work in the presence of delayed duplicate control segment?**
  See Tanenbaum:Fig 6-11.

### 5.2 Data transmission
- **Acknowledgement:** sending needs to be acknowledged.
- **One write() ≠ One Segment. ≠ One Packet.** When PUSH flag is set, write data will not be buffered.
- **Piggybacking** (ACK=1): The receiver delays the acknowledgement, and combines this acknowledgement together with the next sending packet in order to reduce the network traffic. Condition \( \leq 200 \text{ ms} \)

### 5.3 Connection termination
- **Symmetric termination:** each direction is closed separately independent of the other one.
- **Close( ):** I have no data to send but I am still willing to accept your data.
- **Why 4-way not 3-way? 3-way is possible if ② and ③ are so close.**
- **Half close between ② and ③:** server can continue to read if there are some data in buffer.
- **Full close:** both ends want to terminate the connection. Can use shutdown
- **2 FINs and 2 ACKs because 1) socket is full duplex and 2) asymmetric termination (like phone hand up) leads to data loss;** See Tanenbaum: Fig. 6-12.
- **It is not always the client who performs the active close.** For Telnet or echo, client closes first. For HTTP, or daytime, server closes first.
- **Two army problem [Tanenbaum: Fig 6-13]:** Can we find a protocol to close both end simultaneously?

---

![Connection Termination Diagram](image-url)
5.4 State transition diagram for connection establishment and termination.
### 6. TCP Header and Segment (Tanenbaum: section 6.4.3)

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acknowledgement number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TCP Header Length</th>
<th>6 bits</th>
<th>U</th>
<th>A</th>
<th>P</th>
<th>R</th>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>unused</td>
<td>R</td>
<td>C</td>
<td>S</td>
<td>S</td>
<td>Y</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>G</td>
<td>K</td>
<td>H</td>
<td>T</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Window Size</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Check sum</th>
<th>Urgent pointer</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Options( 0 or more 32-bit word)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Data(optional)</th>
</tr>
</thead>
</table>

- **Sequence number**: If “write” is small, system will package many “write” into one TCP segment. If “write” is large, one “write” is divided into many segments. Sequence number ensures the segments are repackaged in the original order at the receiving end.
- **TCP Header Length**: Due to options, TCP header length is variable. Maximum value of this 4-bit field is 15 in word, giving a maximum TCP header length of 60 bytes.
- **URG=1**: used together with urgent pointer to send OOB data, such CTRL-Z. Urgent accident causes TCP to stop accumulating in buffer and transmit everything in buffer immediately. SIGURG is generated to inform the receiving process. Question: Will OOB data be put into receiver buf or another special buf?
- **ACK=1**: this is an acknowledgement message and acknowledgement number is used.
- **PSH=1**: Send the data without buffering, e.g., telnet command line.
- **RST=1**: reject a segment or refuse a connection or other problems on the receiver end.
- **SYN=1**: to establish a connection. **SYN=1 and ACK=0**: SYN message. **SYN=1 and ACK=1**: ACK message and piggyback acknowledgement is used.
- **FIN=1**: FIN message to finish/close a connection.
- **Error control**: Each segment needs to be ACKed. The lost segment will be resent.
- **Window Size for Flow Control**: the maximum number of bytes I like to receive. E.g. When I need a rest, Set Window size=0. Later permission is granted by sending ACK again with same Acknowledgement number and nonzero Window Size.
- **Checksum**: add up all 16-bit words in one’s complement and then take one’s complement of sum.
- **Options**: extra facilities not covered by TCP header.
  - **MSS**: refer to the data part, up bounded by 16 bits or 65535 and also bounded by MTU. Often MSS=MTU-IP header–TCP header to avoid fragmentation
  - **Window Scale**: window size scales from $2^{16}$ up to $2^{30}$ by shifting 0-14 bits to the left. See RFC 1323.
  - **Timestamp**: check the data corruption caused by lost packets that then reappear. See RFC 1323.
  - **Selective Repeat**: NAK reply: to allow the receiver to ask only for a specific bad segment, not the good
segments. See RFC 1106.

- **Data**
  - **Segments without any data are legal** for acknowledgements and control message.
  - **Do you know the length of data?** 16-bit TCP segment length is in IP header, see Steven V:1:Fig6-25 or below.
  - **Maximum Segment:** Up to 65535-20-20=65495 bytes of data may follow.
  - **Actual segment size** is bounded by MSS and also by MTU which is much less than MSS. The minimum MTU is 68 for IPv4, and 576 for IPv6.

### 7. IP Header and Datagram (Steven V.2: Appendix A)

<table>
<thead>
<tr>
<th>Version (4)</th>
<th>Header length</th>
<th>Type of service (**???0)</th>
<th>Total length in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification</td>
<td>0</td>
<td>D M F F</td>
<td>Fragment offset</td>
</tr>
<tr>
<td>TTL</td>
<td>Protocol</td>
<td>Header checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit source IPv4 address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit destination IPv4 address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Options( if any)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Fields in italics** are excluded in IPv6 header. They could be included in IPv6 optional headers.
- **Version:** It has been 4 since early 1980’s.
- **IP Header length:** same as TCP header length. 4bits-->15words-->60 bytes-->at most 40 bytes for options.
- **Type of Service (TOS):** with pattern **????. Use IP_TOS to set 4-bit ????.. See RFC 1349.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPTOS_LOWDELAY</td>
<td>Minimize delay</td>
<td>telnet, rlogin</td>
</tr>
<tr>
<td>IPTOS_THROUGHPUT</td>
<td>Maximize throughput</td>
<td>FTP</td>
</tr>
<tr>
<td>IPTOS_RELIABILITY</td>
<td>Maximize reliability</td>
<td></td>
</tr>
<tr>
<td>IPTOS_LOWCOST</td>
<td>Minimize cost</td>
<td></td>
</tr>
</tbody>
</table>

- **16-bit total length:** length of the whole IP datagram including the IPv4 header. But IPv6 payload length doesn’t include the header part. So the maximum IPv6 datagram length is 65575=65535+40.
- **Identification +DF(don’t fragment)+MF(more fragment)+fragment offset:** used for fragmentation. But there are no such fields in IPv6 header since fragmentation is the exception rather than the rule and exceptions should not slow down normal processing.
- **TTL:** decremented by 1 by each router, discarded when the value is 0. It limits the lifetime of any IP
datagram to 255 hops. Changeable with IP_TTL and IP_MULTICAST socket options.

- **Protocol**: type of data contained in the IP datagram. 1: ICMPv4, 2: IGMPv4, 6: TCP and 17: UDP. See RFC 1700.
- **Header checksum**: only for header part including options. Omitted in IPv6 since TCP header’s checksum includes the IP header.

8. **TCP transmission control**

   One write() ≠ One Segment. ≠ One Packet

   ![TCP Transmission Control Diagram](image)

   - **Send buffer**: use SO_SNDBUF to change it. Default size is 4k.
   - If remaining space of send buffer ≤ write data, write is blocked until all data is put into send buffer.
   - TCP send data to IP in MSS size or smaller.
   - Only after the data in send buffer is acknowledged from the peer TCP, can it be removed from the buffer.
   - One goal of MSS is to avoid fragmentation. But if MSS+40 (or 60) > MTU, IP fragmentation takes place.
   - IP searches the routing table to determine the outgoing interface and pass the datagram to the appropriate datalink.
   - If output queue is full, the packet is discarded and an error is returned up the protocol stack to TCP. TCP resends it and doesn’t tell the application process.

- **Question**: What is the purpose of sender bugger? For reliability and for performance.
- **Question**: Please figure out the buffer situation at the receiver end.
- For UDP, 1) no send buffer, 2) higher probability of fragmentation without MSS, 3) ENOBUFS(no room on the queue for datagram or one of its fragments) is returned to the application which is responsible for resending.
The receiver piggybacks both acknowledgement and buffer size onto the reverse traffic.

- **If WIN=0** sender blocks but may use persistence timer to avoid deadlock.
- **Delaying/Buffering for performance on sender side:** TCP knows WIN=4k, and so buffer the 1st 2k data until another 2k data come in to send a segment with a 4k payload.
- Similarly on receiver side by combining many ACK into one. If receiver gets segments 0,1,2,4,5,6,7, it can acknowledge everything up to and including the last byte in segment 2. When retransmitted segment 3 after timeout is received, it can acknowledge all bytes up to the end of segment 7.

**Problem 1:** TELNET connection to an interactive editor that reacts on every keystroke.
- **send a byte at a time.**
  - **If not buffer:** refer one keystroke $\rightarrow$ 41 bytes to send it $\rightarrow$ 40 bytes to ACK $\rightarrow$ 40 bytes for window update $\rightarrow$ 41 bytes to echo it. See 1 byte triggers 4 segments and 162 bytes transmitted.
  - **If buffer:** if send data $<$ WIN $\land$ data $<$ MSS, wait/buffer. No interactive response.
  - **Time limit:** 500msec. Can be used on both ends.
  - **Nagle’s algorithm** (Nagle 1984): Send 1st byte, then buffer the rest while waiting for ACK. Again send the buffered data while waiting.
  - **Question:** Is Nagle’s Algorithm always useful?

**Problem 2:** Silly Window Syndrome (Clark 1982) shown in Fig. 6-30 of Tanenbaum textbook.
- **Read a byte at a time** $\rightarrow$ window update $\rightarrow$ send one byte $\rightarrow$ wrap around again and again.
- Clark algorithm: delay the window update until there is a MSS space or half space, whichever is smaller.
9. TCP timers

- **Retransmission timer:**
  - Neither too small nor too large. Why?
  - Timer value should be dynamically changing to reflect the current network traffic situation.
  - RTT (round trip time): determined at different level. dynamically changing, \( \text{RTT} = \alpha \text{RTT} + (1-\alpha) M \) where \( M \) is ACK time.
  - Retransmission timer = \( \beta \) RTT. How to choose \( \beta \)? \( \beta = 2 \)?
  - Jacobson Algorithm(1988): \( D = \alpha D + (1-\alpha) |\text{RTT} - M| \), Timer = RTT + 4D.

- **Persistence timer:** to avoid the deadlock during transmission when \( \text{WIN}=0 \) and ACK(\text{WIN}>0) is lost.

- **Keepalive timer:** Changeable with TCP\_KEEPALIVE. 2 problems: 1) adds overhead, 2) terminate a possibly healthy connection.

```
probe(WIN=?)
wait
if WIN=0 continue wait
if WIN>0 Send
ACK(WIN=?)
update WIN
ACK(WIN>0)
timeout
alive?
yes
No
terminate
```
10. TCP Congestion control

- **Purpose**: want a transmission rate that is best; not too fast, not too slow.
- **2 windows**: Both are dynamically changing.

  - **Receiver window (RW)**: receiving capacity determined by receiver using “ACK(WIN=?)”.
  - **Congestion window (CW)**: determined by bandwidth/congestion/traffic density.

- **transmission rate** ≤ minimum(RW, CW), actually |CW| ≤ |RW|.
- **How to determine CW**? 1) Periodically monitor the congestion, 2) Acknowledged or not?
- **Algorithm** (see Tanenbaum: Section 6.5.9)

  1) Initially CW = 1MSS.
  2) Increase CW = 2CW, *i.e.*, 1MSS→2MSS→4MSS→… → until timeout or RW.
  3) If RW is reached, fix CW unless RW is further changed.
  4) If timeout occurs, threshold = 1/2 current window, and CW=1 MSS.
  5) Increase CW = 2CW until threshold, then CW=CW+1 until timeout or RW.
  6) Go up to repeat step 3).

**Example:**

- **Question**: Why does CW suddenly jump down to 1 MSS when timeout?
- **Any other algorithms**? The Sender transmits at the highest rate, counts the number of messages/bytes acknowledged during some time period and divides the number by the time period. See Tanenbaum:P 510.