SFWR 4C03: Computer Networks and Computer Security Feb 2-5 2004

Lecturer: Kartik Krishnan Lectures 13-15

User Datagram Protocol (UDP):

UDP is a connectionless transport layer protocol: each output operation by an application produces exactly one UDP datagram, which in turn causes one IP datagram to be sent. This is different from a stream oriented protocol such as TCP (see below), where the amount of data written by an application has little to do with what actually gets sent in a single IP datagram.

The UDP layer is responsible for communicating between two applications within two host computers; each application has a 16 bit port number assigned to it. There are reserved port number for various applications (see Section 12.9 of Comer). On the other hand, the IP layer only provides communication between the two host computers, and there can be multiple applications running on each computer.

UDP provides no reliability: it sends the datagrams that the application writes to the IP layer, but there is no guarantee that they ever reach their destination. This is again unlike the TCP (see below).

Format of UDP messages:

Each UDP message is called a UDP datagram (just like an IP datagram, since UDP is after all an extension of IP to provide communication between two applications). Each UDP datagram has two parts: a UDP header and a UDP data area. The header is divided into the following four 16 bit fields (see Figure 12.1 of Comer for more details):

1. **Source port number (16 bits):** This contains the 16 bit UDP protocol source port number. This port number is optional; if used it specifies the port to which replies should be sent; if not used, it should be zero.

2. **Destination port number (16 bits):** This contains the destination port number, and it is used to demultiplex datagrams among the various processes waiting to receive them in the destination computer.

3. **UDP message length (16 bits):** This is a count of bytes in the UDP datagram, and includes the length of the UDP header and data (unlike IP which includes only the header length). The minimum value is 8, the length of the header alone.

4. **UDP checksum (16 bits):** This is optional; a value of zero indicates that the checksum has not been computed. The UDP checksum covers more information than is present in the UDP datagram alone. To compute the checksum, UDP prepends a *pseudo-header* to the UDP datagram, appends an octet of zeros to pad the datagram to an exact multiple of 16 bits, and computes the checksum over the entire object. The pseudo header is shown in Figure 12.2 of Comer, and includes the 32 bit source and destination IP addresses, a 8 bit protocol field which is 17 for UDP, and a 16 bit UDP length (which is also present in the UDP header). To compute a checksum, the UDP software computes the 16 bit one's complement sum of the pseudo header, UDP header, and user data, and takes its 1s complement. If the calculated checksum is 0, it is stored as all one bits (65535), which is equivalent in one's complement arithmetic. This is to distinguish it from 0 which implies that the checksum was not computed. The purpose of prepending a pseudo header to the UDP header before computing the checksum is to ensure that the UDP datagram has reached its correct destination, since this requires the correct destination IP address in addition to the correct destination port number.

Since the receiving UDP software requires the pseudo header while computing its checksum, it needs to communicate with the receiving IP layer, which provides it with these details. An alternative is that the sending UDP layer actually obtains these details from the sending IP layer while it is filling out the UDP datagram, and the sending IP layer has only to fill out the remaining fields in the IP datagram. Thus, we see that there is a great degree of interaction between the UDP and IP layers.

One area where UDP is especially useful is in client-server situations. Often the client sends a short request to the server and expects a short reply back. If either the request or the reply is lost, the client just times out and tries again. Chapter 21 of Comer has a discussion on the client-server model. One such application of UDP is DNS (Domain Name System). DNS is described in detail in Chapter 24 of Comer. This is a program that returns the IP address of some host name, for example, *optlab.cas.mcmaster.ca*. The client requests the IP address of the host in a UDP datagram, and the server responds with the IP address in another UDP datagram. No connection setup or release is required as is the case with TCP (see below).

Transmission Control Protocol (TCP):

The Transmission Control Protocol (TCP) coordinates the transmission of data between a pair of applications. Applications communicate by reading from and writing to a socket that presents data as an ordered, reliable stream of bytes. TCP provides a logical connection between two end points by building on top of IPs packet-delivery service, i.e., we have a connection oriented protocol on top on a connectionless protocol. The two end points are identified to be a pair of integers (host, port), where host is the IP address for a host, and port is a 16 bit TCP port number on the host. As in the case of UDP, there are assigned TCP port numbers for various applications (see Figure 13.16 of Comer). The TCP header appears in the data portion of the IP header. The routers inside the network need not inspect the bits in the TCP header. Once the IP datagram reaches the destination machine, the operating system inspects the TCP header to direct the data to the appropriate application via a socket. The socket is identified via the pair of integers described above. TCP must deal with the fact that IP packets may be lost, corrupted, or delivered out of order. These challenges are addressed by cooperation between the TCP sender and receiver. Before transmitting data, the two end points must coordinate to establish the TCP connection. During the data transfer, the end points cooperate to control the flow of data and retransmit lost IP packets. In addition, each end point adapts its transmission rate in response to congestion to avoid overloading the network. The TCP header includes the necessary information to coordinate the ordered, reliable delivery of segments.

TCP is a connection oriented protocol (analogous to the telephone network) and unlike UDP and IP which are connectionless protocols (analogous to the postal network). Also, unlike IP, TCP is an end to end protocol, i.e., it is implemented only the sending and receiving machines. Intermediate routers do not have any TCP layers.

TCP provides a *full-duplex* communication between two applications, i.e., data can flow in either direction with no apparent interaction. An application process can terminate flow in one direction making the connection *half duplex*. One advantage of a full duplex communication is that the TCP software can combine control information for one stream back to the source (acknowledgements) in datagrams that also carry data in the opposite direction. This phenomenon is called *piggybacking*, and it helps reduce network traffic.

Consider the transmission of a message from one application to another. The operating system on the sending machine divides the message into segments; each segment is a contiguous set of bytes that fits into a single IP packet. The TCP header identifies the connection associated with the segment. Each segment has a sequence number that distinguishes it from other segments. The recipient uses the sequence number for reordering packets that appear out of order. If packet 2 arrives before packet 1, the operating system on the receiving machine knows to wait for packet 1 before delivering the data to the receiving application. The TCP sender also includes a checksum computed over the contents of the packet; the receiver recomputes the checksum and discards the packet if the results do not match.

TCP achieves reliable transport over the unreliable IP layer by having the receiver send acknowledgements to the sender, indicating that the packets have been received. If the receiver has data awaiting transmission, the acknowledgement and the outgoing segment could be included in a single packet, a phenomenon known as *piggybacking*. The sender starts a timer once a packet has been sent. If no acknowledgement is forthcoming before the timer is reset, the sender assumes that the packet was lost, and transmits yet another copy. Upon receiving at least one uncorrupted copy the packet, the receiver sends an acknowledgement to the sender. If more than one copy arrives, the recipient simply discards the extra copies. The acknowledgement scheme is illustrated in Figures 13.1 and 13.2 of Comer.

TCP header details:

The TCP sender transmits each segment in a single IP packet, along with a TCP header. The TCP header includes the following:

1. **Source port number (16 bits):** The 16 bit port number associated with the TCP sender. The sender's IP address is available in the IP header.

2. **Destination port number (16 bits):** The 16 bit port number associated with the TCP receiver. The receiver's IP address is available in the IP header too.

3. **Sequence number (32 bits):** The 32 bit sequence number identifies the position of the first byte of the data contained in the packet. The receiver uses the sequence number to identify where the segment fits in the data byte stream and to reorder segments that arrive out of order. Before transmitting the first segment, the sender selects an *initial sequence number* that represents the beginning of the ordered byte stream. The sequence number for all other segments exchanged during the connection are relative to this initial sequence number.

4. **Acknowledgement number (32 bits):** To acknowledge the receipt of data, the TCP header includes a 32 bit acknowledgement number. This field indicates the next byte of data that the receiver expects to receive, and is valid only if the ACK bit is set to 1.

5. **Header length (4 bits):** The 4 bit header length indicates the number of 32 bit words in the TCP header. The header is usually 20 bytes long (corresponding to five 32 bit words). Longer headers occur when the sender uses *TCP options* that contain additional control information.

6. **Reserved (6 bits):** The 6 bit field is reserved for future use.

7. **TCP flags (6 bits):** The TCP header also includes a 6 bit field with six 1 bit flags. These flags are the following:

   (a) **URG:** The URG (Urgent) flag instructs the TCP receiver to inspect the portion of the segment identified by the 16 bit urgent pointer field in the TCP header.

   (b) **ACK:** The ACK (Acknowledge) flag is set when the TCP segment contains an acknowledgement. When the ACK bit is set, the 32 bit acknowledgement field indicates how much data has been received by the sender.

   (c) **PSH:** The PSH (Push) flag indicates that the TCP receiver should deliver the incoming data to the application's socket.

   (d) **SYN:** The SYN (synchronize) flag is set when establishing a TCP connection (see below). When the SYN bit is set, the value of the sequence number field identifies the initial sequence number.

   (e) **FIN:** The FIN (finish) flag is set when the sender has finished transmitting data.

8. **Receiver window (16 bits):** The 16 bit receiver window indicates the number of additional bytes the receiver can handle beyond the data that has been acknowledged so far. The TCP sender respects this limit to avoid overflowing the receiver buffer.

9. **TCP checksum (16 bits):** The 16 bit checksum aids the TCP receiver in detecting corrupted packets. In contrast to the IP header (see Lecture

3), the TCP checksum covers both the header and the data. The sender computes the checksum over the header, data and some portions of the IP header (see the discussion in computing the UDP checksum above). The receiver recomputes the checksum and compares it with the value in the TCP checksum header. If the answers differ, the receiver discards the corrupted packet. The receiver does not acknowledge the receipt of the corrupted packet. Hence the sender would eventually retransmit the missing data.

10. **Urgent pointer (16 bits):** When the URG flag is set, the 16 bit urgent pointer directs the receiver's attention to a particular portion of the incoming data. The 16 bit urgent pointer identifies the last byte of urgent data in terms of an integer offset from the sequence number in the TCP header.

Opening and closing a TCP connection:

The SYN, ACK, FIN, and RST flags in the TCP header are used in opening and closing a TCP connection. TCP segments with these flags set are sent in response to system calls that open or close the corresponding socket. When an application on host A creates a socket, the operating system on A coordinates the establishment of the TCP connection with the application on host B. Establishing a TCP connection involves the following *three-way* handshake. This handshake is illustrated in Figure 13.13 of Comer.

1. **SYN from A to B:** Host A initiates the connection by sending a packet with the SYN bit set to 1. The SYN packet includes the *initial sequence number* for the stream in the 32 bit sequence number field in the TCP header. The SYN packet synchronizes the sequence number. The SYN segment itself counts as part of the stream and consumes the first sequence number. For example if the SYN packet has the sequence number of 4500 (say), then the first data segment from A would have a sequence number of 4501 to identify the first byte of data.

2. **SYN-ACK from B to A:** The arrival of the SYN packet from A triggers both the creation of socket and the transmission of an acknowledgement packet from B to A. B sends a segment to A that has both the SYN and ACK flags set to 1. The SYN flag initiates the reverse direction of the connection (from B to A), and the ACK flag acknowledges receipt of A's SYN packet. The acknowledgement number in the header of B's TCP segment is set to the value one larger than the initial sequence number in A's SYN packet. As with A's SYN packet, B's SYN-ACK packet includes an initial sequence number (which is unrelated to the sequence number in A's SYN packet). This sequence number marks the beginning of the stream of bytes travelling from B to A.

3. **ACK from A to B:** Once the SYN-ACK packet has arrived, the connection from A to B is complete, and the operating system in host A can inform the application that the connection has been established, and the application can now begin writing to and reading data from the socket. However, at this time, the application on host B does not know if A has

received the SYN-ACK packet. So the third part of the three way handshake involves sending an ACK packet from A to B to acknowledge the creation of the connection from B to A. The ACK packet from A has the acknowledgement number set to the value one larger than the initial sequence number in B's SYN-ACK packet. Upon receiving this packet, the application in host B is ready to transmit data over its socket to the application in A.

The application program on host A, initiating the connection, is said to perform a *passive open*, while the application program on host B is said to perform an *active open*. The operating system selects different sequence numbers for TCP connections. Consider what could happen if every TCP connection used an initial sequence number of 0. Suppose that a TCP connection between A and B had an outstanding duplicate packet inside the unreliable network between A and B that experienced a long transmission delay, but this packet eventually makes it to the receiving host. Suppose that A and B close the TCP connection once the original data transfer has completed. Later, A and B might establish a new TCP connection with the same port numbers as the original connection. Suppose that, after the new connection has been established, the duplicate packet from the old connection finally reaches the destination. In this situation, the recipient might mistakenly associate the old packet with the new connection and deliver the data in this packet to the application. To avoid this problem each connection begins with a different initial sequence number which is chosen randomly.

Either end point can terminate the TCP connection. In our earlier example, let this be the application on host B. This termination typically involves a *four-way* handshake (illustrated in Figure 13.14 of Comer) which involves the following:

1. **FIN from B to A:** The application on host B closes its socket and transmits a TCP segment with the FIN flag set to 1. At this point, the TCP layer on host B does not transmit any new data. The TCP layer is however responsible for completing the ordered, reliable delivery of the previous data sent to A. In addition, the TCP layer continues to receive and acknowledge TCP segments from A. Like the SYN segment, the FIN packet from B consumes one byte in the sequence number space.

2. **ACK from A to B:** Upon receipt of this segment, A transmits an ACK packet with an acknowledgement number that is one higher than the sequence number in B's FIN packet.

3. **FIN-ACK from A to B:** As soon as the application on host A is done, the TCP layer on host A closes its socket and sends a FIN-ACK segment to B. In this segment, A also acknowledges the receipt of B's previous FIN packet.

4. **ACK from B to A:** Upon receiving the ACK-FIN segment from A, B transmits an ACK packet to acknowledge the closure of the connection from A to B. Upon receiving this ACK segment, A knows that B has received its FIN packet. At this point the connection between A and B is closed. Transmitting any new data between A and B would require opening a new TCP connection.

Sliding-window flow control:

The TCP sender limits the transmission of data to avoid overflowing the buffer space at the receiving end. In theory, TCP could transmit data whenever the application writes data into the socket. However, TCP limits the transmission of data for two important reasons:

1. The sender should send more data than the receiver can store in its buffers - transmitting excess data would overflow this buffer and result in lost packets.
2. The sender should also not transmit data more quickly than the network can handle - sending too aggressively overloads the network, creating congestion that increases communication latency and the likelihood of lost packets.

Each TCP sender thus limits the number of outstanding (unacknowledged) segments in the network using a *sliding-window* control. On the other hand to avoid overflow of the buffer at the receiving end, packets from B to A include the *receiver window* in the TCP header.

Maximum segment size:

Both the sender and the receiver need to agree on a maximum segment they will transfer. TCP software uses the OPTIONS field in the TCP header to negotiate the maximum segment size with the TCP software at the other end of the connection. The following points require consideration when choosing a maximum segment size for transmission:

1. The segment size should not be too small. For instance, if the segment were carrying only one byte of data, then only $\frac{1}{41}$ of the network bandwidth is utilized for user data. This is because the TCP/IP headers themselves take 20 bytes each.
2. On the other hand, extremely large segment sizes can also produce poor performance. This is because large segments result in large IP datagrams. When such datagrams travel across a network with small MTU, the routers in the IP layer must fragment them. There is always a chance that a fragment is lost, or transmitted out of order.

In theory, the optimum segment size occurs when the IP datagrams are as large as possible without requiring fragmentation anywhere along the path from source to destination. This is however a difficult task because intermediate routers dynamically change paths in response to network congestion.

Retransmission of lost packets:

The retransmission of lost packets plays a crucial role in how TCP provides a reliable delivery of a stream of bytes. IP does not inform the TCP sender when the packet is lost; instead the sender must infer that a packet is lost based on the lack of response from the receiver. The receiver acknowledges receipt of data from the sender by transmitting acknowledgments - packets with the ACK bit set, with the acknowledgement number indicating the next byte expected in the stream of bytes from the sender; the acknowledgement information can be piggybacked with the data the receiver sends. The sender can infer that the packet is lost in two ways; we discuss both in detail below:

1. **Retransmission timeout:** The sender sets a retransmission timer after transmitting data to the receiver. If the timer expires before the acknowledgement arrives, the sender assumes that the packet was lost en route to the receiver. Selecting the appropriate value for the retransmission timeout (RTO) is a delicate process. Setting RTO too low results in a false alarm, and the sender unnecessarily sends a segment that was not actually lost. On the other hand setting RTO too high postpones the detection of a lost segment, resulting in unnecessary delay in retransmitting the segment. The right value of the retransmission timeout depends on the distance between the sender and the receiver, as well as network congestion. The appropriate RTO varies from one situation to another, and the TCP sender learns the appropriate RTO value by estimating the round trip time (RTT) to the receiver - the time between the transmission of a packet and the receipt of an acknowledgement. Based on these measurements, the sender can estimate the average RTT, as well as its variance. The RTO is set to the average RTT plus an additive factor that depends on the variance of the RTT's computed for different samples. See Sections 13.16 and 13.19 of Comer for more details. In theory, although measuring the round trip travel time (RTT) is trivial, complications arise whenever segments are retransmitted. This is because acknowledgements are cumulative (see point (2) below) in that the acknowledgement refers to data received, and not to the instance of the specific datagram that carried the data. To see this, consider the retransmission of a segment that was believed to be lost. This TCP segment which is retransmitted is identical to the original segment except that it travels in another IP datagram. When an acknowledgement is received for this segment from the receiver, the sender has no way of knowing whether this acknowledgement corresponds to the original or retransmitted TCP segment. Thus TCP acknowledgements are *ambiguous*. Thus when computing the RTT, one adopts *Karn's algorithm* where one ignores samples corresponding to retransmitted segments, but uses a backoff strategy where one multiplies the previous RTT value by a multiplicative factor (say 2), utilize this value as the new RTT until a valid sample (corresponding to an acknowledged packet) is obtained. Once a valid sample is obtained TCP recomputes the new RTT using the round trip estimate for the new sample. See Section 13.17 and 13.18 of Comer for more details.

2. **Duplicate acknowledgements:** In some cases, the sender can infer that a packet is lost based on duplicate acknowledgements. Consider that sender A starts with the initial sequence number of 4500 and transmits data in 100-byte segments, and the length of the sliding window is 4. The sequence number 4500 represents the opening of the TCP connection. The first data segment has sequence number 4501 and length 100 and spans bytes 4501 to 4600, the 2nd segment has sequence number 4601 and spans bytes 4601-4700 and so on. Suppose that B receives the 1st, 2nd and 4th segments, while the 3rd data a segment has been lost or delayed. Because B has received only 200 contiguous bytes of data, the 2nd and the 3rd

acknowledgements on receipt of 2nd and 4th segments would both have an acknowledgement number of 4701 (4501 + 200), constituting duplicate acknowledgements. Here byte number 4701 is the sequence number of the next data byte that B expects to receive. Thus the TCP acknowledgement scheme is *cumulative* because it reports how much of the data stream it has accumulated. A cumulative acknowledgement is less efficient in the sense that the sender has no way of knowing that the 4th segment (in this case) was actually received. Receiving duplicate acknowledgements allows the sender to infer that the 3rd packet may be lost. In some cases, the sender could perform a fast retransmission without having to wait for the timer to expire resulting in a better recovery from packet loss.

Forced Data Delivery:

Consider a user logged in a remote machine and typing on his keyboard. The user expects that the characters he types be echoed on the computer screen of his local machine. If the sending and receiving TCPs buffer the data rather than sending it or passing it to the receiving application, the remote machine response will be delayed. Thus, to accomodate such interactive applications, TCP provides a *push* operation (identified by the PSH flag set to 1). The sending application can use the operation to ensure that the sending TCP delivers the data immediately; the receiving TCP too delivers this data to the receiving application without delay. Thus, in the above interactive session the sending application uses the push function after every keystroke by the user.

Urgent Mode:

TCP provides an *urgent mode*, allowing one end to tell the other end that urgent data of some form has been placed in the normal stream of data. This notification is provided by setting the URG flag to 1; the 16 bit urgent pointer field is then set to a positive offset that must be added to the sequence number field in the TCP header to obtain the sequence number of the last byte of urgent data. The URG flag is used to alert interactive applications, such as *Telnet*, to the presence of control characters (e.g. CTRL C) in the data stream. A user might type such a control character if the program executing on the remote machine is misbehaving, and the user wants the remote server to terminate the program.

TCP congestion control:

The TCP layer adapts to network congestion by decreasing the transmission rate. Typically congestion occurs when a collection of aggressive TCP connections overload the network. This results in a number of lost packets, which then have to be retransmitted. Retransmission of these packets would only exacerbate the congestion. The connectionless nature of the IP protocol makes it difficult for the routers inside the network to control congestion. Thus the responsibility for congestion control is relegated to the end hosts.

The TCP sender adjusts the data transmissions based on a sliding window that depends on the available buffer space at the receiver known as the *receiver* *window*, and the available bandwidth in the network represented by the *congestion window*. The sender transmits data based on the minimum of the two values. Upon detecting that a packet has been lost, the sender decreases the size of the congestion window to lower the transmission rate. On the other hand in the absence of packet loss, the TCP sender gradually increases the congestion window to transmit data more aggressively.

TCP implements an *additive increase* and a *multiplicative decrease* algorithm to change the size of the congestion window. In the absence of packet loss, the sender linearly increases the congestion window, one segment at a time until the size of segments in the window is the maximum segment size (MSS) for the TCP connection. For an Ethernet network with an MTU of 1500 bytes, the MSS would be 1460 bytes (after excluding 40 bytes for the TCP/IP headers). In response to packet loss, the sender multiplicatively (quickly) decreases the size of the congestion window. In fact after receiving the 3rd duplicate ACK, the congestion window is set to one half the current window size. The TCP sender also backoffs the RTTs of the segments in the current congestion window. The process continues until the congestion window has size 1. This process of incrementing and decrementing the size of the congestion window amounts to experimenting with the network to determine the appropriate transmission rate.

Once congestion ends, TCP initiates a *slow start* recovery phase, where the congestion window starts at the size of a single segment, and this window size is increased by one each time an acknowledgement comes through. Slow start avoids swamping the internet with additional traffic immediately after congestion clears or when a new TCP connection is started. The term *slow start* is actually a misnomer because the congestion window grows *multiplicatively* rather than linearly. This is because TCP initializes the connection window to 1, sends an initial segment, and once an acknowledgement comes through increases this size to 2. It now sends 2 segments and waits for acknowledgements. When the two ACKs come through, it increases the window size to 4 and so on. Thus in each iteration the window size is growing by a factor of 2. The slow start phase terminates when the congestion window reaches the *slow start threshold*, which is usually set to one half of its original size before congestion. Once this threshold is reached, the TCP sender now increases the congestion window linearly in response to ACKs received. More details are available in Section 13.20 of Comer.

Silly Window Syndrome:

Consider a TCP connection that has been established between two applications. When this connection is first established, the receiving TCP allocates a buffer of K bytes (say), and advertises this in the ACKs it sends. If the sending application generates data quickly, the sending TCP will transmit segments which quickly fill up the receiver's buffer. Eventually, the sender will receive an acknowledgement from the receiver, that advertises an available buffer space of 0 bytes. Once the receiving application has read one byte of data from this full buffer, the TCP on the receiving machine can use an ACK that specifies a buffer size of 1 byte. If this is the case, the sending TCP responds to this window advertisement by sending a TCP segment with 1 byte of data. This results in the two TCPs exchanging TCP segments containing only 1 byte of data resulting in a phenomenon known as *silly window syndrome.*

Transferring such small segments consumes unnecessary network bandwidth, and also introduces unnecessary computational overhead. Both the receiving and sending TCPs work to avoid this problem in the following manner:

1. **Receive-side silly window avoidance:** Before sending an updated window advertisement after advertising a zero window, the receiving TCP waits for space to become available that is at least 50% of the total buffer size or equal to a maximum sized segment. The receiving TCP also delays acknowledgements when silly window avoidance specifies that the window is not sufficiently large to advertise. The ACK is in fact piggybacked with data sent, or with future window advertisements.

2. **Send-side silly window avoidance:** When a sending application generates additional data to be sent over a connection for which previous data has been transmitted but not acknowledged, the sending TCP places the new data in its output buffer, but does not send additional TCP segments until there is sufficient data to fill a maximum sized TCP segment. If, however, an acknowledgement arrives while this buffer is being filled, the sending TCP sends all the data that has accumulated in the buffer. This is also commonly known as *Nagle's algorithm.*

Recommended Reading

1. Chapter 12 of Comer [1], and Chapter 11 of Stevens [2] for a discussion of UDP.

2. Chapter 13 of Comer for a discussion of TCP. Chapters 19-25 of Stevens on the other hand provide a detailed description of TCP interspaced with several examples. Chapter 6 of Tanenbaum [3] also has a good discussion of transport protocols including TCP and UDP.

References

[1] D.E. Comer, *Internetworking with TCP/IP: Principles, Protocols, and Architectures*, 4th edition, Prentice Hall, NJ, 2000.

[2] W. Richard Stevens, *TCP/IP Illustrated, Volume I: The Protocols*, Addison Wesley Professional Computing Series, 1994.

[3] Andrew S. Tanenbaum, *Computer Networks*, 4th edition, Prentice Hall, NJ, 2003.