Week 3. Transport layer

The subject of this week is the transport layer. Unlike issues discussed in the last two seminars, this week discusses the work of a hidden layer, which does not have direct contact with user applications. However, studying this layer enables us to get a clear picture about important issues that are taken care of in the process of communication between *processes* located on different computers. (A *process* is a running program, an operating systems term).

The transport-layer is said to provide **logical** communication between *processes* residing on different network nodes. Whenever the word **logical** is mentioned in the computer literature, it denotes an abstract of **physical.** In other words, making a complicated physical specification simpler. Thus we have *logical memory addresses* and *physical memory address, logical records* and *physical records* and so forth. In the same token we understand that the transport-layer simplifies the communication between processes, by providing an abstract form instead of the actual complex communication form. Accordingly, an application located on one computer, can exchange messages with another remote application, as if both are resident on the same computer.

The transport layer depends on the network layer to obtain a *logical* communication link to other remote *hosts.* (*at this point you should be able to see that the main difference between the transport and network layers is that the first establishes a communication between processes, while the second provides logical communication between hosts*). The network layer provides only best-effort service to pass data between hosts. In other words, no guaranteed communication channels. To offset the lack of reliability offered by the network layer, the transport layer applies data loss detection measures and retransmissions to provide applications with a reliable communication link. In addition, transport layers regulate data transmission rates to avoid network link congestion.

Within the transport layer, operate two protocols: TCP and UDP. These two protocols provide applications with the required quality of service: TCP guarantees error free service, while UDP provides constant data transmission. The choice between one protocol and the other depends on the application. In the following, we will study the work of both protocols in detail.

Multiplexing and Demultiplexing

A major job of the transport layer is to collect data units from the application layer, envelop them in headers and forward them to the *network layer* to be sent to a destination host. We will follow the textbook convention by calling enveloped data units *segments*. Collecting segments from

different applications is called *multiplexing.* At the receiving end, the transport layer receives segments from the network layer. In order to identify the recipient application, the receiving transport layer analyzes the header attached to a segment by the sending transport layer. Then, the receiving transport layer passes on received segments to corresponding applications. This act is called *demultiplexing.*

Transport layers identify the receiving applications with the aid of a three-number address. Applications are associated with *port* numbers, which identify the interface between application layer and transport layer. Since network applications follow the client/server model, port numbers for the sending and the receiving applications are the same. To allow more than one application of the same type to be active, a second number is added to distinguish between these applications. Furthermore, to allow applications of the same type to run on other hosts and send segments to applications at a server, a third number is used. The third number reflects the identity of the host on which a client application runs. The three numbers taken together (triplet) serve to identify a receiving application for the transport layer. The textbook explains the formation of the identification triplet through different scenarios. It would greatly increase your understanding if you look at these scenarios.

Connectionless Transport: UDP

We have already mentioned that there are two protocols operating within the transport layer (TCP and UDP). We also mentioned that the transport layer is responsible for the reliability of transmission (we will find later that other layers also do different degrees of reliability control). The UDP can be described as the less complex of the two transport-layer protocols, yet it is the more efficient one. Being a less complex protocol, also means UDP provides less services for applications. The main difference between UDP and TCP is the quality of service. TCP invests more time in assuring error free delivery than UDP. However, time critical applications are less willing to pay for this assurance, especially if there is tolerance for some glitches in the received data. Imagine if you lost several milliseconds of voice in a radio (or TV) broadcast! It is not that much different than a perfect, but with variable speed, transmission?

There are obviously advantages to using UDP over TCP. Most of these advantages stem from the smaller UDP segments and un-throttled delivery. For example, the overhead in UDP segments is only 8 bytes vs. 20 bytes for TCP. In addition, TCP senders and receivers require additional storage to store the connection state.

UDP is not totally free of error checking. In fact, UDP does provide indication of corrupted data to the receiver, and then, it is left to the receiver to decide what to do. Some applications take

advantage of this feature and include reaction to errors. Error detection is done in a simple way through a two-byte checksum attached to the segment header. The checksum contains the 1s complement of the sum of the data (plus header).

Principles of Reliable Data Transfer

As stated above, the transport layer provides reliable data transfer over unreliable channels. Reliability is achieved through protocols implemented within the transport layer. In the following we will move through several steps to build a reliable data-transfer protocol.
The textbook uses Finite State Machines (FSM) to explain algorithms to build a reliable data transfer protocol. FSM is a graphical depiction of protocol algorithms. As usual, graphical tools are used to simplify understanding of algorithms. For those who are seeing FSM for the first time, FSM is made of nodes that represent states, and arrows that show transitions between these states. Arrows are labeled with the actions that cause or take place during these transitions.
The simplest data-transfer protocol, rdt 1.0, is one that assumes reliable channels. Thus no reliability measures are needed. Figure 3.9 presents the FSM for this protocol. At the sending side, the protocol waits for a call from the application layer. The call comes from the upper layer in the form of a request to send data on a reliable channel, *rdt_send(data*). The protocol cuts the data down into packets that are appended with additional headers, and put in a special format. The transport layer requests the network layer to complete the transmission with the *udt_send (packet)* call. Having finished, this algorithm returns to the waiting state. A similar picture is seen on the receiving side.

The second simplest data-transfer protocol, rdt 2.0, relies on a simple error detection method, the checksum. Depending on matching received and calculated checksums, the receiver sends back a positive acknowledgment (ACK), or a negative acknowledgment (NAK). The sending side of this algorithm (Figure 3.10-a) moves between two states. While in the first state, the algorithm waits for a call from the application layer. When this call arrives, the protocol computes the checksum, forms packets, and sends them through the network layer. Having finished sending packets, the protocol moves to a state waiting for ACK/NAK from the other side. When ACK/NAK arrives from the receiver, the protocol either moves back to the waiting state on (ACK), or it resends the last packet and continues to wait for ACK/NAK. This protocol does not make assumptions about corrupted ACK or NACK messages, which makes it an unrealistic protocol.
The third protocol, rdt 2.1, resolves a corrupted, ACK/NAK message as follows: A sequence number (0 or 1) is added to each segment. Whenever the receiver receives an error-free segment it sends back an ACK with the sequence number, otherwise, it sends a NACK. If the sender receives a corrupted ACK/NACK message, then it simply resends the last segment. If the last

segment was indeed in error, then the sender must have done the right thing. Otherwise, the sender has sent a repetition of the last segment. The receiver finds out about repeated segments by keeping track of the sequence number: two consecutive segments with the same number are obviously the same segment (the later one is simply discarded). There is still a problem with this protocol: it does not consider lost ACK/NACK messages. If an ACK/NACK message is lost, then the sender will wait indefinitely. We will not go over the FSM for this and next scenarios; they are similar in concept to the earlier ones.

The fourth algorithm, rdt 3.0, adds a time-out counter to the protocol. Whenever a preset time expires without receiving an ACK/NACK message, the sender assumes that either the segment or the ACK/NACK was lost. In this case, it goes ahead and retransmits the last segment. Since segments have sequence numbers, as before, the receiver might choose to accept or discard the received segment. This protocol still has a problem: the count-down timer relies on an *estimated* round trip time of a segment and ACK/NACK message; a time which is hard to correctly estimate. Therefore, some repetitions are expected, however this repetition is taken care of by numbering segments.

Connection-Oriented Transport: TCP

Unlike UDP, with TCP there is three-way handshaking between client TCP and server TCP before any data transfer, or connection is established. The sending TCP sends a request for transmission to the receiver TCP. In response, the receiving TCP sends back its agreement to establish communication (acknowledgement). Finally, the sending TCP sends its acknowledgement to the receiver that it is ready to send. Thus, there are three messages exchanged (three-way handshaking) before data starts to flow from sender to receiver and vice versa.

To allow for the difference in data transfer rates between client and server on one hand, and the rate at which applications can consume data on the other, buffers are established at both ends. An application using TCP places data in the TCP send buffer. TCP reads data from the buffer and forms it into segments, and hands them down to the network IP protocol. At the receiving end, the TCP extracts data from received segments and places them into the buffer for the receiving application to read at its own convenience.

TCP segments are larger (for obvious reasons) than those of UDP. The structure of the TCP segment is in the textbook (section 3.5.2). We will explain the purpose of every part of this segment as needed.

TCP receives a stream of data from an application to be transferred to another application on a different host. The TCP protocol divides this data stream into segments. Each segment contains a part of the data stream starting at a certain position in the stream. This position in the data stream is taken as the segment sequence number (one of the TCP segment fields). For example, if a segment starts with the 100-th byte in the original data stream, then this segment is given the sequence number 100. The TCP also contains a field (the acknowledgement number) to tell the other end about the last segment correctly received.

In the following, we consider three scenarios to verify the ability of the sequence and the acknowledgment numbers to aid in achieving reliable data transfer:

1. A lost acknowledgment (Figure 3.34): TCP relies on the acknowledgment number to learn whether sent bytes have reached the other end safely. However, if this acknowledgment is not received within a given waiting time, TCP assumes that something went wrong and retransmits the missing segment. Since, in our case, the acknowledgment has been lost (but not the segment), the receiving TCP simply discarded the repeated segment (identifying it with the sequence number)
2. A delayed acknowledgment (Figure 3.35): In this scenario, TCP keeps sending segments while waiting for acknowledgments at the same time (we call this pipelined data transfer). Whenever a TCP times-out on a certain acknowledgment, it assumes the worst. TCP assumes that all segments starting from the lost one on are all lost. When a TCP starts resending old segments, the receiving TCP corrects the sending TCP by sending him an acknowledgment number indicating the actual last segment received.
3. A safe loss of acknowledgment (Figure 3.36): in this scenario, a later acknowledgement is received within the waiting period of an earlier one. In this case, the sending TCP does nothing because it understands that all segments up to an acknowledged one are all correctly received.

A key difference between TCP and UDP is regulation of data transfer rate. TCP regulates transfer rate to guard against any segment losses (UDP does not mind losing some segments). TCP makes sure that TCP buffers at either sending or receiving ends are not overfilled. Overfilling buffers obviously leads to loosing segments. TCP protocols inform each other about buffer status in the *rcvr window* size field in the TCP segment (Figure 3.29). Building on the status of the buffer on the other end, TCP protocol regulates its data transfer rate.

As we stated earlier, round trip time is not easy to estimate. Delays in links and routers depend on the overall network load, which is not predictable. To keep the estimated round trip time as

accurate as possible (and thus reducing the number of wasted retransmissions), the estimated time is changed on the run. The estimated round trip time is calculated depending on two factors: the current estimated round trip time, and the most recent actual round trip time. A 0.875 weight is given to the current estimated round trip time; while 0.125 weight is given to the most recent actual round trip time.

EstimatedRRT= 0.875 * EstimatedRTT + 0.125 * SampleRTT
Thus the estimated round trip time is kept from changing wildly see Figure 3.32 page 237. Timeout, on the other hand, is made slightly larger than round trip time to allow reasonable time for the acknowledgment. This slight increase is intelligently derived from the deviation between actual and estimated round trip time.

Timeout = EstimatedRTT + 4 * Deviation,
Where: Deviation = (1-x) * Deviation + x * |SampleRTT – EstimatedRTT|
And: $x<1$.

However a perfect guess cannot be expected, no matter how carefully the estimated time is calculated.

Principles of Congestion Control

TCP deals with network congestion through regulation of data transmissions. Network congestion results in loss of packets (segments), or delayed acknowledgement. Both of which cause TCP to retransmit some segments. In this section we will study different strategies that TCP uses to reduce network congestion.

The textbook explains drop in link performance due to loss of packets in a gradual and nice manner. We will briefly explore the three scenarios that appear in the book:

With two senders and a shared router, it is assumed that the router has an infinite buffer and a finite sending capacity. In other words, an overwhelmed router with a large number of incoming packets, buffers these packets to be forwarded later (no loss of packets).

At the output of the router, each route gets half of the router sending capacity, R/2. If the two senders keep sending at a collective rate less than the router capacity, the router can comfortably resend them and they reach the destination at the same speed at which they were sent. Data is received at the same rate at which they were sent. Nevertheless, there is a buffering delay at the router, which

gets increased with the increase of data rate. As the sending rate exceeds the router capacity, the rate at the output of the router stays the same but the buffer gets filled up quickly (and thus delay increases dramatically). *The exact relationship between the number of packets in the buffer and the incoming and outgoing packet rates is governed by queuing theory. The book did not give the full details of queuing theory, which is fortunate.*

Approaches toward Congestion Control

Whenever a network is congested, it is TCP's responsibility to regulate the application's data stream rate. The TCP has ways to infer about a congested network:

a) if segments are lost, or
b) b) if the acknowledgment round trip is long.

There are other direct approaches to inform the TCP about a congested network. Routers can either: a) send specialized packets (**choke packets**) to the sending TCP, or b) mark outgoing packets to indicate congestion. Option (b) requires the packet to reach the receiver, which then sends information back to the sender. In both cases the sender is notified to slow down.

ATM (a modern networking technology) adopts a totally different approach to control congestion. This approach is described as a network-assisted one. Special packets circulate at a constant (configurable) rate through the network. These special packets (which are called cells by ATM) have bits that indicate whether the network is congested (CI bit) or mildly congested (NI bit). If none of these bits is set, then the TCP can increase the sending rate to take advantage of the free network.

TCP Congestion Control

TCP uses interesting calculations to control congestion. In the following we will look at simplified models to understand the TCP congestion control mechanism. There are no mathematical proofs for much of this part, which we are not concerned about in our course any way.

Let us define some variables and formulas. The formulas below are easy to verify:

1. Window size *(w)* -, that is number of transmitted –yet-to-be-acknowledged segments.
2. Round trip time (RTT)- time taken by segments to reach destination plus time for acknowledgements to come back.
3. Segment size (MSS)- number of bytes in each segment.
4. Transmission rate = (w * MSS)/RTT bps. Since, each sender waits RTT time units before sending a whole window of bytes.

5. The end-to-end transmission rate of this connection $r = \min \{R_1/ K_1, ..., R_N/ K_N\}$. With the assumption that a segment needs to travel through N consecutive links. On the route, each link, n, has capacity R_n and is shared by K_n TCP connections (connections can be made by one or more hosts).

The well-known **Tahoe** TCP congestion control works as follows: Upon receiving data from an application, the TCP starts transmission cautiously by sending just one chunk of bytes of size MSS (i.e. w=1). If this chunk is received safely by the destination and acknowledged in due time, then the w value is increased to 2. If these in turn are acknowledged before their timeouts, then w is increased again by a factor of 2 (w becomes 4). TCP keeps increasing the value of w exponentially fast until some threshold value. Above this threshold value, w is increased linearly (i.e. increased by a constant 1 each time). If however the TCP receives an indication of congestion, then the window w is reduced abruptly back to 1; at the same time the threshold value is reduced to half of the maximum reached w value and the whole process is restarted. What does it mean to reduce w to 1? Essentially, this means that the speed at which TCP sends bytes is reduced to MSS/RTT. As we mentioned above, we are not going to prove the correctness of this algorithm in this course, there are references and alternative algorithms stated in the textbook for interested people.

Figure 3.51 in the text book gives a clear picture for Tahoe congestion control. TCP progressively assigns the values 1, 2, 4, and 8 to w, as no loss is assumed. Beyond threshold value of 8, TCP increases w linearly to 9, 10,11, and 12. As w reaches 12, a segment loss is assumed and the value w is reduced back to 1. At the same time, the threshold value is reduced to 6 (= . *12), were 12 is the w maximum value.

Fairness in link sharing is an important issue that is dealt with by the Tahoe algorithm. We will look at a simplified example to get a sense of TCP fairness through Figure 3.53 in the textbook. The solid line in this graph represents the full bandwidth utilization line. In other words, if the link is used to the maximum capacity then,

Connection 1 throughput + Connection 2 throughput = full link bandwidth R.

Note that the value of “Connection 1 throughput” is not necessarily equal to “Connection 2 throughput”. However, it is a goal to make both throughputs equal. Ideally, the total throughput should be in the middle of the solid line, or on the dotted line if the total application throughput is less than R. The example starts, with point A presenting the total throughput used by both connections at a certain moment of time. Note that the total throughput is less than R, which

means that no loss in segments would happen. Hence, both connections will increase the transmission rate linearly (the exponential part is dropped for simplicity). The increase in this example reaches a total throughput value greater than R; thus creating a possibility of segment loss. Having recognized network congestion, both TCPs drop throughputs sharply. A sharp drop in throughputs will take the total throughput back to C (rather than back to A. Why?). As the same cycle repeats, the total throughput point approaches the “equal bandwidth share” line. From the above, we observe a repeated cycle of linear increase in throughput followed by a sharp decrease by each TCP. We can view this behavior over time as a saw-tooth behavior of TCP throughput regulation. Mathematically we can express the average throughput of this behavior as:

(0.75 * W * MSS)/RTT

Where W is the maximum value reached by w.

Reading Assignment:

Chapter 3