EE122 Socket Programming Project (Version 5.0)
Spring 2005

1 Objective

You will implement the client side of a simple file-transfer application based on a receiever-oriented TCP-like protocol (c.f. Section 2) over UDP. The project is based on WebTP [1] which is a receiver-oriented request/response protocol for the web. The motivation for this is rooted in the inherent advantages in granting the receiver control and thus the user control of the transfer of data. Furthermore, the server side implementation is simplified which is a great advantage in a centralized system.

2 The Protocol Description

There are two aspects to the protocol: the sender side and the receiver side. The sender is passive and only responds to requests made by the receiver. To keep the sender design simple, it is the receiver's responsibility to ensure reliability of data transfer and also adapt to congestion. The following subsections describe the different aspects of the protocol.

2.1 Flow Initiation and Termination

The flow is initiated by the receiver (the client) asking the sender (the server) for the first ADU (Application Data Unit) (in fact, the zeroth ADU) for the file it wants to receive. This ADU contains metadata about the file that will be transferred. To keep things simple, this metadata is the file size in number of bytes. When the receiver requests data from the sender, it assumes that the file data has been split in to ADUs of size FIXED_ADU_SIZE (=2048 bytes) and the ADUs are numbered sequentially starting from ADU numbered 1 (ADU number 0 is metadata). The receiver then asks the sender for specific ADUs of the file from the sender by sending it a list of ADU numbers. From now on, we will use the term 'data packet' to mean a packetized ADU and the data packets will be numbered the same as the number of the ADU in the packet.

The flow terminates when the receiver sends an explicit 'close' request . In addition, if the sender does not get any more requests from the receiver it times out and terminates the flow. This happens at FIXED_SENDER_TIMEOUT (=60 seconds) after the last data packet the sender sent to the receiver.

2.2 Sender

The sender is modeled as a queue serving packets at an adjustable rate r. When the sender gets a request packet from the receiver, it processes it by packetizing the ADUs requested and putting them in a queue. The data packets to be retransmitted are given higher priority and placed at the head of the queue. Normal data packets are placed at the tail of the queue. Once a data packet has been sent, it is removed from the buffer – it is the responsibility of the receiver to specifically request retransmissions (in contrast, TCP requires the sender wait for an ACK before removing packets because it is the sender which retransmits in TCP!) . The sender inserts gaps between successive transmissions of data packets to conform to the rate imposed by the receiver. The rate information is explicitly given to the sender by the receiver. When the sender receives an updated rate information in a request packet, it adjusts its rate accordingly. However, the modified rate will not change the transmssion time of the packet at the head of the queue, which would already have been scheduled.

Requests to the sender could be of the following types:

- Send: The receiver explicitly gives the sender a list of the ADUs (identified by their numbers) it needs next. The sender prepares its queue and schedules the appropriate data packet(s) accordingly. To initiate the flow, the receiver requests ADU number 0 (as mentioned earlier, this ADU is the file metadata). In this special case, the flow is initiated and the sender responds by sending the size of the file. If the requested file is not accessible, the sender responds with error message in the reply packet.
- Retransmit: The receiver explicitly gives the sender a list of ADUs that need to be retransmitted. The sender prepares its queue accordingly.
- Close: The sender closes the connection immediately by freeing up the queue for sending packets to the receiver. It also frees up the memory for storing the state of the connection.

Note that if the request is invalid, i.e., not one of the above types, the sender does not send a response to the receiver. Also, the sender responds to send/retransmit requests for normal ADUs (i.e., ADUs numbered from 1 onwards; these ADUs contain the file data) only if the flow has already been initiated by requesting ADU number 0 (i.e., the file metadata). The sender does not respond to request packets which have errors and they are simply dropped.

2.3 Receiver

The receiver essentially sends requests to the sender. The first request packet (essentially requesting ADU number 0) in any connection initiates the flow for the transfer of a specific file. Depending on the sender's response to this, the receiver decides on which ADUs to ask for in the later 'send' requests. Also note that the request packets from the receiver cannot have a packet size of more than MAX_REQUEST_PKT_SIZE (=4096 bytes).

Since you will be using UDP, which (unfortunately) is unreliable, the receiver will need to implement a retransmission mechanism. Furthermore, the receiver also implements a congestion control mechanism. The details for these follow now.

2.3.1 Window Control

The receiver maintains a congestion window (CWND) which limits the maximum number of ADUs which have been requested but not arrived (the minimum window size is 1). At the start of the connection, the window undergoes a slow start phase. During this phase, the window is increased exponentially. The receiver does not convey any rate information to the sender in this phase (it asks the sender to send at a rate of 0 seconds per packet). The sender is expected to send data packets at the maximum possible rate it can, since the rate is unspecified in this phase. This phase continues until the receiver detects a packet loss. This is detected when there is a timeout (the receiver allows out-of-order packet delivery during slow start) or when the received packet has errors. After this, the receiver moves to the congestion avoidance phase, where it sends explicit rate information to the sender along with its requests. During congestion avoidance, the additive increase/multiplicative decrease strategy used in TCP is used to adjust the window size, i.e., every time a data packet is successfully received or a packet loss is detected, the window size is adjusted. The schemes for rate control (c.f. Section 2.3.2) and retransmission (c.f. Section 2.3.3) are used during the congestion avoidance phase. During the slow start phase, the receiver also computes the RTT for the connection (this RTT is used to compute the timeout for the slow start phase as $2 \cdot \text{RTT}$, and is later used in the congestion avoidance phase to compute the rate).

2.3.2 Rate Control

The window based scheme is used to control the flow. In addition, the rate-based scheme is used during the congestion avoidance phase. The rate control implicitly involves computing the RTT (round trip time). The RTT calculation is done by the receiver. Timestamps are explicitly inserted in the "request" packets sent to the sender. The sender also inserts the time lag between the time a request was received and the time when the packet was sent.

The EstimatedRTT is computed in the same way as it is done in TCP. Specifically, if the current packet implies a sampled RTT of SampledRTT, then the new value of EstimatedRTT is computed as $\text{EstimatedRTT}_{\text{new}} = \alpha \cdot \text{EstimatedRTT}_{\text{old}} + (1 - \alpha) \cdot \text{SampledRTT}$ (you may assume $\alpha = 0.5$). The desired rate is then calculated as $\frac{\text{EstimatedRTT}}{\text{CWND}}$, where CWND is the size of the receiver's window (in number of packets). The rate is assumed to be calculated in time/packet (TPP). The rate information is conveyed to the sender only when a request for a packet is sent to it. Therefore, the new rate information takes effect on the receiver side only when it is conveyed to the sender.

SampledRTT is computed by inserting timestamps explicitly in the packets. When the receiver sends a request packet to the sender, it also incorporates the the current time T_r in the packet. When the sender sends a data packet, it computes the held time T_h – the time elapsed since the last request, and writes both the times T_r and T_h in the packet. When the receiver gets the data packet at time T_c, EstimatedRTT is computed as $T_c - (T_r + T_h)$. Note that this approach works correctly even for retransmitted packets (why?). Figure 1 shows the timeline for RTT calculation.

2.3.3 Retransmission

For retransmissions, the receiver needs to figure out lost data packets and lost request packets. For this, the receiver emulates the sender's queue to find out the sequence in which data packets will arrive. The receiver maintains a queue of expected packets EPQ – data packets which have been requested but have not arrived. Once a data packet arrives, its entry is removed from the queue.

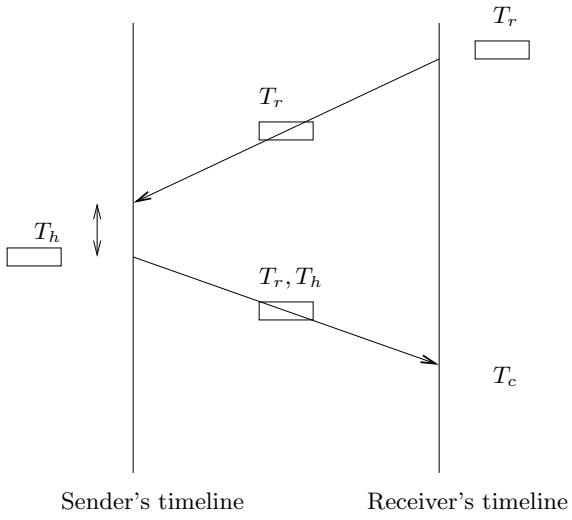

Figure 1: Calculating RTT at the receiver [1].

For figuring out a data packet loss, the receiver needs to compute the required timeout value. If the current rate is TPP seconds per packet, then the receiver expects a data packet arriving every TPP seconds. The timeout is, therefore, set to $2 \cdot \text{TPP}$ seconds. The receiver maintains only one timer – the timer for the next expected data packet. If the next data packet arrives before timeout, the timer is reset to $2 \cdot \text{TPP}$. If, however, there is a timeout, then the congestion window CWND is halved and the new rate TPP_{new} information is sent to the sender in the retransmission request. The new timeout is set to $2 \cdot \text{TPP}_{\text{new}}$. The expected data packets queue EPQ remains the same.

The receiver also employs a scheme similar to fast retransmit in TCP to detect lost requests. The idea is as follows. If an arriving packet is not the front data packet of the queue, the receiver increments an out-of-order counter before it removes the data packet from the expected data packets queue EPQ. Once the receiver gets 3 such out-of-order packets, it requests the sender for a retransmission of the data packets (identified by their ADU numbers) at the head of the queue EPQ. During these requests for retransmissions, the receiver does not cut down the congestion window or the rate. The requested data packets are moved back in the queue by $\left\lceil \frac{\text{EstimatedRTT}}{\text{TPP}} \right\rceil$ positions to take in to account the fact that the packets in the front will now arrive later in the order. Note that in case moving by $\left\lceil \frac{\text{EstimatedRTT}}{\text{TPP}} \right\rceil$ positions results in moving beyond the end of the queue, the packet is simply placed at the end of the queue.

Another case that could occur is that the receiver detects errors in the data packet (using the checksum). In this case, the erroneous data packet is discarded. The congestion window remains the same. The receiver requests the sender to retransmit at the updated rate TPP_{new} (note that the RTT is always updated). Furthermore, as in fast retransmit, the requested data packet is moved back in the queue using the same calculations.

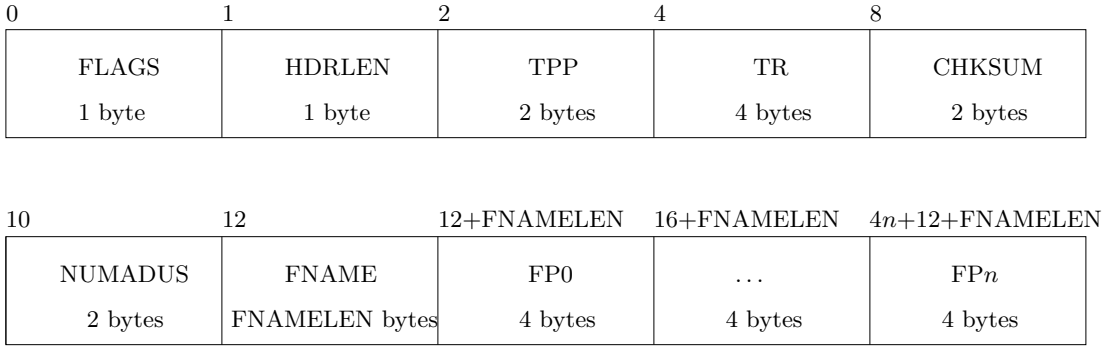

Figure 2: Format for packet from receiver to sender.

2.4 Packet Format

Figures 2 and 3 show the packet fomats for the requests and the response respectively.

For packets from receiver to sender (Figure 2), the FLAGS field indicates the type of the request and could be SEND (0x1), RETRANSMIT (0x2), or CLOSE (0x4). The field HDRLEN is the length of the header (which includes all fields up to the FNAME field) in the number of words (4 bytes). The time-per-packet TPP is in units of 0.1 milliseconds and is 2 bytes long. The field TR is the time (in 10 microseconds) at the receiver's side when the packet was transmitted (c.f. Section 2.3.2). The field CHKSUM is the 16-bit internet checksum (c.f. Appendix A) computed over the packet header and data (essentially all fields shown in the figure). This checksum is used to detect errors in the packets. The field NUMADUS is the number of ADUs ($= n+1$ in the figure) being requested by the sender. The list of these ADUs is given in the payload. The field FNAME is the name of the file. The variable FNAMELEN denotes the length of the filename. FNAMELEN is always adjusted to make the header end on a word boundary, i.e., FNAMELEN is always a multiple of 4. This is done by padding zeros at the end of the filename if it does not end on a word boundary. If the packet is a send/retransmit packet, then the requested ADU numbers FP0, FP1, ..., FPn follow the field FNAME. Each ADU number is 4 bytes long.

Figure 3 shows the format for the packets from sender to receiver. The field RSP is the response of the sender. RSP is set to ERROR (0x1) to indicate error (e.g., file does not exist, etc) or NOERROR (0x0) to indicate a successful availability of data. The field HDRLEN is the length of the header (the header includes all the fields up to FNAME) in the number of words. The field CHKSUM is the 16-bit checksum computed over the packet header and data (essentially all the fields shown in the figure). The field FSIZE/DATALEN is the length of the file in number of bytes if the packet is in response to an initiate request from the receiver. In case the packet contains normal file data, this field is the length of the ADU in the payload. The field TR is the receiver's time and TH is the held time (c.f. Section 2.3.2). Both these times are in 10 microseconds. The field FNAME is the name of the file. Similar to packets from receiver to sender, the filename length is always made a multiple of 4 by padding zeros at the end of the filename. The field PN indicates the ADU number of the data in the packet. Note that if the packet contains the metadata of a file, then the field PN is set to 0 (to indicate ADU number 0) and the payload is empty, i.e., the DATA field is empty, else the DATA field contains the ADU indicated in the field PN.

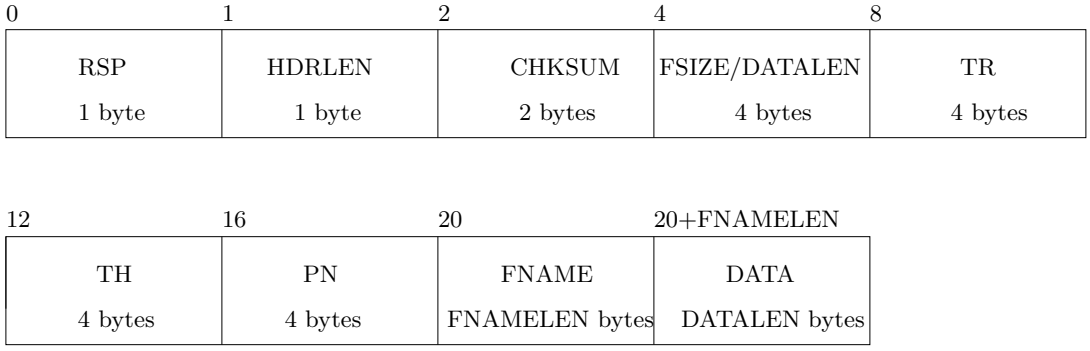

Figure 3: Format for packet from sender to receiver.

3 What you need to do

You will design a simple file-transfer application which uses the WebTP protocol described above. For your convenience, the sender side has already been implemented and will run on an instructional machine (more details to be posted). You will implement the receiver side of the protocol. **To get full credit, you need to implement the following parts of the protocol in the receiver:**

- **Flow initiation and termination**
- **Checksum based error checking**
- **Window based slow start**
- **RTT based rate control**
- **Timeout based retransmission**

In addition, you may implement out-of-order arrival based retransmission (fast retransmit) as extra credit.

You are expected to work in groups of 2-3 (groups of 2 are, however, preferred) for the project.

3.1 Programming

You will implement the project in C. You will need to use the socket programming interface and the time related functions. The UNIX man pages are an excellent resource to understanding these functions. Relevant examples are posted along with the sample test code on the course website.

A good programming practice is to think about the application at a high level before actually starting to write the code. This not only ensures a better code but it makes it easy to modify and debug. Therefore, you should think of defining the API of the file transfer that implements WebTP. The defined API should be a collection of appropriate functions which implement a portion of the file transfer application (the receiver side). The main code calls the functions of this API to initiate the file transfer. You will submit a preliminary report on your proposed API.

3.2 Testing

For your convenience, we provide you with a sample implementation to compare your results with. Instructions on how to use the sample program are given in the documentation for the file. Multiple versions of the sender program will run on an instructional machine. One version is a sender implementation which does not simulate congestion or packet errors, which can be used to verify a first version of your program. A second version simulates congestion by random dropping of packets. Thus, in this case, both the requests and the data packets may be 'lost' during transmission, hence packets might arrive out-of-order at the receiver. A third version simulates packet errors by modifiying some bits in the packets. The fourth version simulates both congestion and packet errors.

To compare your implementation with those of your classmates, you will test it by transferring a large file (approximately 100 Mbytes file to be provided). The best implementation will be the one with the least transfer time (you are welcome to make improvements to the receiver implementation). The five groups with the best transfer times will receive extra credit.

4 Design Report

Details on the final report will be posted later. Please keep checking the webpage for updates.

5 Submissions

The preliminary report describing your proposed API will be due on April 22, 2005. The final report will be due on May 5, 2005. In addition, you will also need to signup for group demos of your project around the due date (to be announced later).

You are strongly encouraged to start early (instead of a 'slow start') to 'avoid congestion' during the 'termination phase' of the semester.

6 Grading

The different modules will be graded using approximately the following weights:

- High level API description of the receiver – 10 %
- Flow initiation, termination and checksum based error checking– 15%
- Window based slow start – 15%
- RTT based Rate control – 20%
- Timeout based retransmission – 20%
- Final report and demo presentation – 20%
- Out-of-order arrival based retransmission – 20% (Extra credit)
- Best implementations – 10% (Extra credit)

A The Internet Checksum Algorithm

This is a brief description of the internet checksum algorithm [2]. For (much) more details you may refer RFC 1071 [2].

A.1 Computing the checksum

Suppose the checksum of the following sequence of bytes is to be computed: $a_1, a_2, \ldots, a_n$. Denote a 16-bit integer as an ordered pair (x, y) where x is the most significant byte and y is the least significant byte. Adjacent bytes are paired to form 16-bit integers as (a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n) if n is even and (a_1, a_2), (a_3, a_4), ..., (a_{n-2}, a_{n-1}), $(a_n, 0)$ if n is odd. The checksum field is cleared and the 1's complement sum of these 16-bit integers is computed. The 1's complement of this sum is computed and placed in the checksum field. The machines you will work on do not have 1's complement addition but instead have 2's complement addition. On 2's complement machines, the 1's complement sum is computed by adding the numbers and if there is a carry, then add the carry to the sum. Finally, 1's complement of a number is computed by simply complementing the bits of the number. For sample code, you may refer Section 2.4.2 of the textbook [3].

References

[1] Rajarshi Gupta, Mike Chen, Steven McCane, and Jean Walrand. A Receiver-Driven Transport Protocol for the Web.

[2] RFC 1071. Computing the Internet Checksum.

[3] Larry L. Peterson and Bruce S. Davie. Computer Networks – A Systems Approach. 3^{rd} Edition.