Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.829 Fall 2005 **Problem Set 1** September 8, 2005

This problem set has 4 questions, most with multiple parts. Answer them as clearly and concisely as possible. You may discuss ideas with others in the class, but your solutions and presentation must be your own (for each such discussion, please mention whom you collaborated with). Do not look at anyone else's solutions or copy them from anywhere (in particular, "bibles" are not allowed).

Turn in your solutions by **5:00 pm, Thursday, September 22, 2005** to 32-G940.

This problem set includes a programming exercise. More detailed instructions on that are provided within the question.

1 Multiplexing

In this problem, we will compare statistical multiplexing to time-division multiplexing (TDM).[1]

In our statistical multiplexing scheme, packets of all sessions are merged into a single queue and transmitted on a first-come first-served (FCFS) basis. In the TDM scheme, packets from each session are queued separately. Time is divided into slots and each queue is assigned a time slot in a round robin manner. We assume the size of a slot is much smaller than the time required for transmitting a frame/packet on the link.

A switch is said to be *work conserving* if the only time it is idle is when there are no frames waiting for service.

1. Is our TDM scheme work conserving? What about our statistical multiplexing scheme?
2. Let's study the impact of statistical multiplexing on queuing delays. Suppose there are N concurrent sessions each with a Poisson traffic stream with rate λ frames/second. Also suppose that frame lengths are exponentially distributed, such that the average rate at which frames are serviced at the switch is μ frames per second ($\mu > N\lambda$). What is the average delay seen by a frame in TDM and in statistical multiplexing? What is the physical interpretation of your result?
3. Assume that all the sessions send frames at a simple constant bit rate and that there are A active sessions at a given point in time out of a possible N, sharing an output link. What is the utilization of the output link when the aggregate input rate for the A active sessions is μ frames/second. Sketch this as a function of A for both statistical multiplexing and TDM.
4. Explain why TDM has smaller variation in the delay of a frame through a switch, compared to statistical multiplexing. (This delay variation is sometimes called the *jitter*.)

[1] If you have no background in queueing, check *Data Networks* by Bertsekas and Gallager.

2 To delay ACK or not?

A TCP receiver *must* acknowledge every alternate segment under normal operation. Sometimes, segments may be lost or delayed, requiring a *delayed ACK timer* to acknowledge the last in-sequence byte so that the connection may continue. Most BSD-derived TCP implementations implement the delayed ACK timer using a 200 ms *heartbeat timer*. Whenever the heartbeat timer fires, the receiver looks through all its established TCP connections and sends and ACK on each connection that has an ACK pending. In contrast, some versions of Solaris TCP implement this using a 50 ms *interval timer*, which schedules an ACK for a connection to be sent exactly 50 ms after a segment arrives on the connection. Of course, if the next segment were to arrive in the interim, this timer is canceled and the appropriate delayed ACK sent.

1. Suppose the delayed ACK timer is set to δ seconds, the maximum available bandwidth for the path is r bits/s, the connection's round-trip time is ρ seconds, and data packets are of size s bits. Suppose the connection has a window size of larger than one segment. Under what condition does the delayed ACK timer prevent normal delayed ACKs from ever being sent? That is, under what condition do all the ACKs get sent triggered by the interval timer, in terms of the parameters above?

2. If the average TCP segment size s is 512 bytes, δ is 50 ms, and ρ is 120 ms, what values of r cause an ACK for every incoming segment triggered by the interval timer? Is this a realistic bandwidth range in today's Internet? Therefore, is this 50 msinterval timer a reasonable implementation of delayed ACKs?

3 TCP checksums

TCP has an end-to-end checksum that covers part of the IP header, in addition to the TCP header and data. When the receiver receives a data segment whose checksum doesn't match, it can do one of two things:

1. Discard the segment and send an ACK to the data sender with the cumulative ACK field set to the next in-sequence byte it expects to receive, or

2. Discard the segment and do nothing else.

Is one action preferable to the other (or are they both equivalent)? Why? (You should look up the TCP and headers in any standard networking textbook; note that the header formats in Cerf & Kahn's paper aren't used any more.)

4 Reliable Message Transfer Protocol

The goal of this assignment is to implement a reliable message transport protocol (RMTP), similar to TCP. The assignment will involve building a simple server and client that realize this protocol. The simple client requests a filename from the server and the server transmits the file reliably to the client. Unlike a TCP client, the RMTP client does not need to handle reordered packets very carefully. The full specification of the protocol is as follows:

- All messages exchanged between the client and the server have a message type (*e.g.*, SYN, SYNACK, SYNRETRY, SYNACKACK, DATA, ACK, FIN, FINACK), a sequence number, a timestamp and the payload (*i.e.*, the actual data).
- The client initially contacts the server (which is listening on a specific host and port) with a SYN message. The data field of the SYN message specifies the filename that the client wants to download. The sequence number of the SYN message is set to 0 by convention.
- On receiving the SYN message, the server tries to open the specified file for reading. If the file open succeeds, the server a) picks a random sequence number, and b) sends a SYNACK message with the sequence number picked in (a). The server specifies the size of the chunks in which the file will be transmitted in the payload field of the SYNACK message.

  If the file open fails, the server sends a SYNRETRY message to the client.
- The client has the responsibility of reliably transmitting the SYN message. If the SYN, SYNACK or SYNRETRY messages are lost, the client times out and retransmits the SYN until it is successfully received and acknowledged by the server.
- If the client gets a SYNRETRY message instead of a SYNACK, indicating that the requested file was not found, the client retransmits the SYN again with a (possibly different) filename.
- On receiving a SYNACK message from the server, the client sends back a SYNACKACK message to the server, echoing the sequence number of the SYNACK message. The client also sets its chunk size from the data field of the SYNACK. If the server sets an illegal chunk size (*e.g.*, a zero value), the client uses a default chunk size.
- It is the client's responsibility to ensure that the SYNACKACK is reliably delivered to the server and the client should timeout and retransmit if it does not hear back from the server after it sends the SYNACKACK. Since the server does not start sending data until it receives the SYNACKACK, receving data messages with sequence numbers greater than the SYNACK indicates to the client that the server has received the SYNACK.
- On receiving the SYNACKACK message, the server begins transmitting the data. The server uses a sliding window protocol. The window size W is a fixed, user-defined constant. The server can have a maximum of W unacknowledged messages outstanding at any point of time. It is the server's responsibility to ensure that the data messages are reliably delivered to the client. For every data message sent, the server maintains a timer. If the timer fires before an ACK for that message is received, the server retransmits the message. On receiving an ACK for a message, the server advances its window.
- The client does not maintain any timers after reliably transmitting the SYNACKACK; it simply waits for data from the server. On receiving a data message, the client sends an ACK

message, echoing the sequence number and timestamp of the data message. The client does not buffer out of order messages; it simply writes every message it receives into the output file at the correct position.

- The client should not accept messages with sequence numbers less than the sequence number of the SYNACK message. The client need not worry about duplicate messages - duplicate messages are simply written multiple times into the file. The client also need not worry about transmitting the ACKs reliably. If the ACK is lost, the server will timeout and retransmit the data message again.
- The server should ignore any SYN or SYNACKACK messages from the client once the data transmission begins - they would be duplicate messages arriving out of order.
- The client should ignore duplicate SYNACK messages or SYNACK messages received after data transmission has begun.
- A client should ignore any data messages before it receives the SYNACK.
- The server should ignore duplicate ACKs.
- In addition to using the ACKs to infer reliable message delivery, the server uses the ACKs to set the timer for timeouts. The server obtains the estimate of the round trip time (RTT) from every ACK (since the timestamp at which the data message was sent is echoed by the client in the ACK). The server uses a moving average with a fixed weight parameter α to update the RTT. The retransmission timeout is set to β*RTT estimate for some fixed β. The initial RTT estimate for any connection is 1.5 seconds.
- After transmitting the last chunk of the file and receiving all ACKs, the server transmits a FIN message to the client. The server needs to retransmit the FIN message until it is reliably received at the client.
- On receiving the FIN message, the client closes the downloaded file, sends a FINACK and exits. The client transmits the FIN multiple times to ensure that it will reach the server, but does not wait for any acknowledgements from the server.
- On receiving the FINACK, the server resets the connecion state (*e.g.*, sequence numbers) and waits for the next connection. The server ignores any unexpected FINACKs.

Assignment

Your job in this assignment is to complete a Python implementation of an RMTP server and client. We will provide you with code skeletons for the server and client, along with helper classes for communication. You must fill in the methods in the server and client that we have left empty, such that the server and client conform to the above protocol and communicate correctly.

We chose to implement RMTP in Python for several reasons. For those not familiar with programming, the learning curve is fairly easy. There's no need to worry about memory management or cross-platform compilation quirks, with all the benefit of a normal object-oriented language. Plus, it's one of those things that's great to learn, and could be very useful in your lives as researchers. If you are unfamiliar with Python, please visit the "Useful Links" section of the 6.829 website for links to Python tutorials and examples.

The code is available in tar'd, gzip'd format at `http://nms.csail.mit.edu/6.829/rmtp/rmtp.tar.gz`. Download that file onto your local machine, and run the following to get to the code:

```
tar -xzf rmtp.tar.gz; cd rmtp
```

You will find four Python files in this directory:

- `msg.py`: Defines a class for the messages sent between the client and server. Also defines the messages types.
- `network.py`: A networking layer used by both the client and the server to transmit messages. This file handles all the messy details of scheduling timeouts and writing to sockets. The client and server interact with the network mostly through `Network.sendMessage` and `Network.cancelTimeout`.
- `client.py`: Defines the `ReliableClient` class, which implements the client side of the RMTP protocol. You must fill in the following methods: `sendSyn`, `handleSynAck`, `handleSynRetry`, `handleData`, `handleFin`, and `handleTimeout`.
- `server.py`: Defines the `ReliableServer` class, which implements the server side of the RMTP protocol. You must fill in the following methods: `handleSyn`, `handleAck`, `handleSynAckAck`, `sendData`, `handleFinAck`, and `handleTimeout`.

Code documentation for these four files is available at `http://nms.csail.mit.edu/6.829/rmtp`. The code is *event-based*, and *asynchronous*, which means you don't have to worry about threads or network blocking. You can run `python server.py --help` or `python client.py --help` for a list of available options.

You will only be handing in `client.py` and `server.py`, so do not alter `msg.py` or `network.py`, or create any new files. We left many state variables in `ReliableClient` and `ReliableServer`, which you can use, not use, or alter as you see fit. Be aware, however, that we will not be using the `main` methods in those files during our tests.

Testing and Handing in Your Code

You can manually test your code as follows. Choose one machine to run the server, and one machine to run the client (they can be the same machine). On the server machine, run:

```
python server.py --port <server_port> --loss 0
```

On the client machine, run:

```
python client.py --server <hostname_of_server> --port <server_port> --file server.py
--loss 0 --output server-copy.py
```

If your server and client are working correctly, this will create a file called "server-copy.py" on the client machine, which should be identical to the "server.py" file on the server machine. In addition to this simple case, we will be testing your code under high loss rates (*e.g.*, at least .25 loss), and under specific corner cases covered by the RMTP specification.

We have tested the code on FreeBSD, Linux, and Athena machines, using Python 2.3. In theory, Python code *should* run on any platform, but we have not tested it on Windows or Mac, and can't guarantee anything about behavior on those operating systems.

To turn in your code, run `make handin` in your rmtp/ directory. This will create a file called `rmtp-<username>.tar.gz`. Email this file as an attachment to `6.829-tas@mit.edu` by **5:00 pm, Thursday, September 22, 2005**.