**Massachusetts Institute of Technology**
**Department of Electrical Engineering and Computer Science**

6.829 Fall 2005 | **Problem Set 3** | October 4, 2005

---

This problem set has four questions. The first two questions involve programming and experimental analysis. Turn in your solutions on **Friday, October 14, 2005** in recitation. Do a `make handin` in your code directory and email the tarball to `6.829-tas@mit.edu` before 1:30pm on October 14, 2005. Your code will be tested and graded in a manner similar to Problem Set 1.

## Congestion Control in RMTP

The first two questions in the problem set involve implementing and understanding congestion control in RMTP. To answer these questions, you must extend the RMTP framework from Problem Set 1 to include TCP-style congestion control. We have re-factored the RMTP code to be more generally extensible, and so your earlier `ReliableServer` implementation cannot be used for this problem set.

The framework 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 nine 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` (please note that `cancelTimeout` now returns the canceled timeout).
- `queue.py`: Used by the network layer to emulate a bottleneck link. Schedules packets at a specified rate, and drops new packets whenever a queue of finite size is full.
- `client.py`: Defines the `ReliableClient` class, which implements the client side of the RMTP protocol. This file now contains a full implementation.
- `vserver.py`: Defines the `VirtualServer`, the parent class to all server types. It's a bit of a Python hack – you don't need to understand how this class works.
- `server.py`: Defines the `ReliableServer` class. This file now contains a full implementation.
- `ccserver.py`: Defines the `AIMDReliableServer` class, a sub-class of `ReliableServer`. You must complete this class by extending and overriding `ReliableServer` methods to implement TCP-style congestion control. This file also contains the `AIMDData` class, which you must populate with any data that is used directly by your `AIMDReliableServer` class (*e.g.*, `cwnd`, `ssthresh`, `rtt`, `outstanding`, etc.).

- `manyservers.py`: Creates multiple `AIMDReliableServer`s that all share the same network bottleneck.
- `mhserver.py`: Creates a single logical, multi-homed server that allows data to be striped across multiple network paths.

Code documentation for these files is available at `http://nms.csail.mit.edu/6.829/rmtp`.

Your implementation of `AIMDReliableServer` should implement three main aspects of TCP-style congestion control: slow start, AIMD congestion avoidance, and fast retransmit.[1] To implement slow-start, the `AIMDReliableServer` starts transmitting data with a `cwnd` of 1 packet. On every ACK, `cwnd` is incremented by 1 packet until `cwnd` reaches `ssthresh` (initially, set `ssthresh` to be `AIMDData.SSTHRESH_MAX`). Thereafter, the server enters the congestion avoidance phase and implements AIMD style congestion control, *i.e.,* `cwnd` is incremented by `1/cwnd` on every ACK. Out-of-order ACKs are treated as indicative of lost packets. Upon receiving three out-of-order ACKs, it retransmits the lost packet (even before it times out) and halves its `cwnd`. This multiplicative decrease is done at most once per RTT. Note that the server needs to maintain additional state about which packets have been transmitted and which ACKs have been received, in order to identify out-of-order ACKs. Note also that the RTT estimate should now be stored as part of `AIMDData` since it is used in AIMD-related computations. On a timeout, the server sets `ssthresh` to half the current value of `cwnd`, resets `cwnd` to 1 and begins slow start.

The `Network` class now has several new parameters: size, rate, and delay. The rate specifies the rate at which packets are drained from some queue in the network, and can be used to emulate bottleneck links with a specified bandwidth (in packets per second). The queue size specifies the maximum number of untransmitted packets the queue can hold; the network layer will drop new packets when the queue is full. In addition, if a non-zero delay value is specified, all packets through the network layer are transmitted after the specified delay. This can be used to emulate a connection with a specific RTT. Also, as in Problem Set 1, when a non-zero loss rate is specified, all packets being transmitted over the network are dropped with the specified probability. By convention, any queuing and delay occurs on the server side of the connection.

Note that the questions in this problem set are intended to be research-oriented, and in many cases are under-specified on purpose. Use your good judgment to fill in any holes in the problem with reasonable choices, and explain in your solutions what you did.

**Security advisory: Running the server exposes any file on your computer to anyone who knows the protocol and the port number. Do not leave the server running for long periods of time.**

# 1 Multiple TCP connections sharing a bottleneck

In this problem we will study the behavior of multiple TCP connections sharing a bottleneck link. You will first need to complete the `AIMDReliableServer` and `AIMDData` classes. The file `manyservers.py` creates multiple AIMD servers all of which share the same bottleneck, emulating the real-life situation of multiple TCP connections sharing the same bottleneck link. For example, to start two servers sharing a bottleneck link of 10 packets/second with a queue size of 5 packets, you can do the following:

[1]Note that the `fast_rxmit` field in `AIMDReliableServer` should turn fast retransmits on and off.

```
python manyservers.py --port 6829 --delay .5 --loss 0 --port 6830 --delay .1 --loss
.05 --rate 10 --queue-size 5
```

In this case, the first server listens on port 6829, has a delay of 500 ms, and no loss. The second server listens on port 6830, has a delay of 100 ms, and a loss rate of 5%. To download files from the two servers simultaneously, run a command like this on the client machine (all on one line):

```
python client.py --port 6829 --file <file1> --loss 0 --output <outfile1> &
python client.py --port 6830 --file <file2> --loss .05 --output <outfile2>
```

All the TCP connections for the following questions should share a bottleneck link with a queue size of 10 packets and a rate of 20 packets per second.

### 1.1 Throughput vs RTT

We will now observe how the throughput of the connection depends on its RTT. For connection $i$, set the delay in the network layer to be $0.5i$ seconds, and the loss rate to 0. Consider 2, 3, and 4 concurrent connections. Download files large enough so that all connections are concurrent for at least 30 seconds. Answer all questions for this period of time the connections are concurrent.

1. In each case, in what ratio do the individual connections share the bottleneck bandwidth?
2. In each case, plot the variation of `cwnd` of each connection over time. You should use the provided function `dumpCwndLog` to write the time and congestion window to a log file (named `cwnd<port>`) each time the window changes.
3. Plot a graph of the throughput (in packets per second) observed by the client and the RTT of the connection. Run the experiment 5 times and plot the mean and standard deviation in the graph. What can you say about the relationship between throughput and RTT?

### 1.2 Throughput vs Loss rate

We will now observe how the throughput of the connection depends on the loss rate on the network. For connection $i$, set the loss along the path to be $5i$ %, and the delay to be 0.1 seconds. Consider 2, 3 and 4 concurrent connections.

1. In each case, in what ratio do the individual connections share the bottleneck bandwidth?
2. In each case, plot the variation of `cwnd` of each connection with time.
3. Plot a graph between throughput observed by the client and loss rate. Run the experiment 5 times and plot the mean and standard deviation in the graph. What can you say about the relationship between throughput and loss rate?

## 2 TCP and Multi-homing

A conventional TCP connection is insensitive to the presence of multiple paths between the server and the client (which might be the case when the server and/or the client are multi-homed, or when ISPs load balance traffic across multiple paths). A TCP server maintains only one `cwnd` per connection irrespective of the presence of multiple paths. When packets from one window are "striped" across multiple paths in this manner, packets sent through the paths of lower quality suffer losses, resulting in the reduction of the `cwnd`, thus preventing more packets from being sent over the better paths. We hypothesize that a multi-homed TCP server can benefit from maintaining separate congestion control variables (like `cwnd` and RTT estimates) for the different paths between server and client. In such a case, the losses and delays on one path do not effect the `cwnd`s of the other paths. Thus the "better" paths can ramp up their `cwnd`s and increase the throughput of the connection. In this setup, whenever the TCP connection has packets to send, it sends it along the path that has space in its `cwnd`. In this problem you will investigate this hypothesis.

We provide a class `MultiHomedReliableServer` in `mhserver.py`, which is a TCP server that is aware of the multiple paths between the server and client. We emulate multiple paths between the server and client by opening multiple sockets and multiple `Network` objects on the server side with different loss and delay characteristics. The `MultiHomedReliableServer` instantiates multiple servers, one for each of the paths between the server and client. In this problem, each server is only a "virtual server", *i.e.*, all the servers share state like sequence numbers and file descriptors between them using the methods of the `VirtualServer` class.

The `MultiHomedReliableServer` can be run in one of two modes - corresponding to the multi-homing aware and multi-homing unaware modes. When `mhserver.py` is run with the option `--vserver-type=aimd`, multiple `AIMDReliableServer` objects are created each with its own `AIMDData` object. Thus the `cwnd`s of these multiple paths can evolve independently of each other. On the other hand, when the multi-homed server is run with the option `--vserver-type=shared`, all the `AIMDReliableServer` objects are passed the same `AIMDData` object and hence share the same congestion control variables like `cwnd`. In this problem we will investigate if one of these options is better than the other.

You can start a multi-homed server using the `mhserver.py` file. Paths are specified using the `--network` option with a 4-tuple describing loss rate, packet rate, queue size, and delay. For example, to start a server that starts independent AIMD virtual servers on two different paths, one with a 5% loss rate and a 100 ms delay but no bottleneck, and the other with no loss rate or delay but a 10 packets/second, 5 packet bottleneck, run the following command:

```
python mhserver.py --port 6829 --network .05,0,0,.1 --network 0,10,5,0 --vserver-type
aimd
```

Consider the following network configurations, each with four underlying network paths:

- **a)** All four paths have a bandwidth of 20 packets/second and a queue size of 10 packets. Path $i$ has a delay of $.5i$ seconds. Over four experiments, vary loss rate between 0 and 15%, where all connections have the same loss rate.
- **b)** All four paths have a bandwidth of 20 packets/second and a queue size of 10 packets. Path $i$ has a loss rate of $5i$%. Over four experiments, vary delay between 0 and 1.5 seconds, where all connections have the same delay.
- **c)** All four paths have a delay of .5 seconds. Path $i$ has a bandwidth of $10i$ packets/second and a queue size of $5i$ packets. Over four experiments, vary loss rate between 0 and 15%, where all connections have the same loss rate.

Perform all experiments using for both the `aimd` and `shared` cases. Download a file large enough so that each experiment lasts for at least 30 seconds. Run experiments 5 times and use the mean and standard deviations in the questions below.

1. For each of these cases, find the throughput (in packets per second) observed by the client.
2. For each case, plot the evolution of the `cwnd`(s) of the connection over time.
3. From these answers, can you say anything about how equipped the current design of TCP is to handle multi-homing?
4. By default, `mhserver.py` turns off fast retransmissions. Using the `--fast-rxmit` flag to toggle this behavior, compare the behavior of both the `aimd` and `shared` cases. What can you say about the effect of fast retransmissions in these cases, and how do you explain any difference in behavior?

## 3 Buffering in a fast router

Louis Reasoner has been recruited by a hot new startup to design the packet queueing component of a high-speed router. The link speed of the router is 40 Gigabits/s, and he expects the average Internet round-trip time of connections through the router to be 100 ms. Louis remembers from his 6.829 days that he needs to put in some amount of buffering in the router to ensure high link utilization in the presence of the TCP sources in the network growing and shrinking their congestion windows. Louis hires you as a consultant to help design his router's buffers.

Assume for the purposes of these questions that you're dealing with exactly one TCP connection with RTT 100 ms (it turns out that this assumption does not change the results too much, for a drop-tail gateway, *if* the connections all end up getting synchronized). Also assume that the source is a long-running TCP connection implementing additive-increase (increase window size by 1/W on each acknowledgment, with every packet being acknowledged by the receiver, such that the window increases by about 1 segment size every RTT) and multiplicative-decrease (factor-of-two window reduction on congestion). Assume that no TCP timeouts occur. Don't worry about the effects during slow start; this problem is about the effects on router buffering during TCP's AIMD congestion-avoidance phase.

You should (be able to) answer this question without running any simulation.

1. Show that if the amount of buffering in the router is equal to the product of bandwidth and round-trip delay (the *bandwidth-delay product*), the TCP connection can achieve a 100% link utilization (or very close to it). How much packet buffer memory does this correspond to for the router under consideration?

2. Louis is shocked at how much memory is needed and thinks he may not be able to provide this much buffering in his router. He asks you what the average link utilization is likely to be when the amount of buffering in the router is very small compared to the bandwidth-delay product. Explain your answer.

3. Louis asks you how the average link utilization, $U$, varies as a function of $r$, the ratio of the amount of router buffering, $B$, to the "pipe-size," $P$ (the bandwidth-delay product). Calculate $U(r)$ for $0 \leq r \leq 1$. (We know, from part 1, that $U(1) = 1$, and the answer to part 2 is $U(0)$.)

4. Sketch $U$ versus $r$, for $0 \leq r \leq 1$.

*Hint:* A good way to think about this problem is in terms of the TCP congestion window vs. time plots and look at how $P$, $B$, and the (time-varying) TCP congestion window size relate to each other. Think about the "steady-state" of a TCP connection and about how much data a TCP can send in one round-trip time. Don't worry about the TCP receiver flow control window limitation (i.e., assume that the receiver has a very large buffer). You may also make the simplifying assumption that the RTT of the connection does not vary.

# 4 Cheating with XCP

After reading the XCP paper in 6.829, Alyssa P. Hacker and Cy D. Fect get into an argument about just how fragile XCP is in the presence of senders who lie about the information they are given. Consider a version of XCP where the sender reports to the network its RTT and its throughput. In this protocol, the fields in the congestion header are: (i) RTT (ii) throughput (instead of cwnd), and (iii) feedback. (This change from the scheme in the XCP Sigcomm '02 paper will make it easier for you to reason about this problem.)

We will assume that the sender can lie about the RTT and throughput to the network in an attempt to persuade the network to give it more (or less, but that's not too interesting) than its correct share, or in an attempt to mess up the resulting link utilization (causing the network to run under-utilized or to be in persistent congestion). However, the sender *does not send at an arbitrary rate*; it always updates its congestion window according to the feedback the network sends.[2] More precisely,

$$cwnd \leftarrow cwnd + feedback \cdot \frac{true\_RTT}{declared\_RTT}.$$

This adjustment ensures that the increase in the sender's throughput is the one intended by the network.

Alyssa and Cy get into an argument about what happens when users misbehave and lie to the routers. Help them resolve their argument. Be precise and concise in your answers to these questions. These questions are intentionally somewhat under-specified, to encourage you to think about all the things that can go wrong.

1. Suppose the sender(s) lie only about RTT. What would XCP's performance in terms of link utilization and fairness be? (A sender might lie in either direction, of course.)
2. Suppose the sender(s) lie only about throughput. What would XCP's performance in terms of link utilization and fairness be? (A sender might lie in either direction, of course.)

[2]This constraint on the sender is reasonable; after all, in XCP, a sender that simply wished to ignore the network's feedback could do so anyway, and such behavior would require other network-level mechanisms to detect and control.