Teaching Network Performance Measurement Using Java When The Students Don't Already Know Java

Phillip T. Conrad   Ben Greenstein
Department of Computer and Information Sciences, Temple University
1805 N. Broad Street, Philadelphia PA, 19122
conrad@cis.temple.edu, bengreen@nimbus.ocis.temple.edu

Abstract

In this paper we describe a series of Java-based exercises to explore three measures of computer network performance: throughput, delay and packet error probability. These exercises were designed for the senior level undergraduate or graduate survey course in computer networking under the constraint that students might not have had any prior exposure to Java before entering the course. The exercises involve developing simple client and server programs along with simple protocols for communication, designing and carrying out performance experiments, and presenting results. Optionally, as a way of distinguishing service, protocol and implementation, the exercises can *also* be done simultaneously in C or C++; students can then be required to demonstrate the interoperability of the various implementations, and asked to compare and contrast the ease of development in the two languages. The overall goal is to go beyond presenting students with networking *facts* by providing an opportunity to learn some *skills* needed to conduct network research and experimentation.

1. Introduction

A survey course on computer networking and communications should include a discussion of metrics of network performance, such as throughput, delay and packet error probability.[1] In this paper we describe a series of exercises designed to explore these concepts in the senior level undergraduate or graduate survey course in computer networking. These exercises were designed under the constraint that students might not have had any prior exposure to Java before entering the course, since the primary language of instruction in the CIS program at Temple University is C++. In this paper, we describe our exercises and our approach to addressing the problem of bringing students up to speed in Java.

Most of the exercises we present involve experiments with client/server pairs running on two different host computers connected by a TCP/IP internet. Students are given specifications for both the client and server programs, which they must implement in Java, and a description of various experiments to perform. In later exercises, only the goals of the experiment are given, and the students must devise the experimental design.

Performance measurements are taken throughout the experiment. Typically, while the experiment is in progress, the experimental data is stored in arrays rather than written out to files or updated dynamically on the display. This is because care must be taken to ensure that the processing and scheduling overhead of I/O operations do not interfere with the outcomes of the experiments. Once the data has been collected, it can either be processed directly in Java,[2] or they can be output to a file for processing in a spreadsheet program. Because our students are Java beginners, we opted for the spreadsheet route; this also provides students an opportunity to practice using standard off-the-shelf tools to present experimental results.

Another key concept in networking is the distinction between service, protocol, and implementation.[3] Because the client/server programs are in pairs that communicate over the network, the message formats and rules that govern communication between the client and server comprise a simple *protocol.* (In fact this is the definition of the term protocol: rules that govern communication between peer entities.) Thus, the students are provided with an opportunity to face protocol design issues, such as network byte ordering. If the software is structured in a way that the communicating entities are encapsulated in a Java class, then the protocol also is providing a *service* to some user, i.e. the main method that instantiates the class, thus illustrating the distinction between protocol and service.

[1] See, for example: (Comer 1999, §13.8), (Tannenbaum 1996, §6.6).
[2] If the students are more knowledgeable in Java, and doing a large Java project is one of the course objectives this presents an opportunity to assign the development of classes for statistics and presentation of graphs, charts, etc..
[3] See, for example: (Comer 1999, §13.4, §14.2), (Tannenbaum, 1996, §1.3).

Finally, all of the exercises described here could also be implemented in C or C++. Interoperability between the Java client and server and C/C++ client and server can then be demonstrated, provided that a well-defined protocol is followed by each; this re-enforces the distinction between protocol and *implementation*. Implementing the programs in both C/C++ and Java also provides an opportunity to have students compare and contrast the two languages. The manipulation of TCP and UDP sockets in Java is quite a bit easier than in C/C++, owing to the fact that Java was designed from the beginning for network communication. On the other hand, because Java is interpreted, one might expect throughput and delay to suffer on the Java implementation as compared to a C/C++ implementation. Students can be required to write a few paragraphs about their development experiences, present some statistics comparing performance, and make some interpretations based on their knowledge of the differences between the two languages used.

The remainder of the paper is structured as follows: Section 2 describes the problem of using Java in a networking course where students may have never seen it before. Section 3 describes the exercises we used to help students bridge the gap. Section 4 describes our network performance exercises. Finally, Section 5 presents some preliminary conclusions based on our experiences so far[4] with using Java in two networking courses at Temple:

- CIS320, an undergraduate course (two 80-minute lectures, 1 two-hour structured lab per week)
- CIS662, a graduate course (one 150 min-lecture per week.)

2. Using Java in a Networking Course When Students Don't Know Java

The fact that Java has built-in classes for doing simple network operations makes it a good choice for teaching simple socket operations using stream communication (TCP) and datagram communication (UDP). However, students in CIS320 and CIS662 do not necessarily enter the course with any Java experience whatsoever, or any network programming experience. The CIS320 students were asked to fill out a survey on the first day of class indicating their level of Java experience, with the following results:

No experience	Some experience	A lot of experience	No response
13 (68%)	6 (32%)	0 (0%)	0 (0%)

On the other hand, students are not starting completely from scratch. They are required (via pre-requisites) to have extensive C/C++ programming experience, and have taken an operating systems course in which they are required to do some form of single-host inter-process communication (e.g. one of the following: pipes, message queues or shared memory.) Thus, we chose to supplement the basic networking texts used in course, which are (Comer 1999) for CIS320, and (Tannenbaum 1996) for CIS662, with two additional required textbooks:

(1) A tutorial on Java Network Programming. We wanted a text that would cover the basics of using sockets for TCP and UDP in Java, for both clients and servers. We also wanted the book to provide a summary of relevant TCP/IP concepts from a programmer's point-of-view, since the main textbooks we use (Tannenbaum 1996 and Comer 1999) cover these concepts fairly late in the text. We chose (Harold 1997) but also considered (Courtois 1998), which meets the same criteria. As it turns out, both of these books assume that the reader is already fairly proficient in basic Java. Thus we also required:

(2) A textbook providing a tutorial approach to learning Java. We had several criteria for such a book. We wanted a fairly complete tutorial approach to Java, focusing primarily on applications not applets. We wanted the book to rely solely on code in the standard Java libraries, rather than on any "extra" libraries provided by the textbook authors. We wanted to avoid an introductory programming text (i.e., a CS1 type text) that might seem overly simplistic to seniors and grad students, preferring an approach that assumes basic programming skills are already known. We wanted a book that would provide a good summary of OOP concepts, which we regard as a nice re-enforcer for the OOP concepts the students should already have encountered in their C++ sequence. Finally, we wanted a physically small book, so it would be reasonable to ask the CIS320 students to bring all three required textbooks with them to the lab. The book we chose, (Budd 1998), meets all of these criteria.

3. Exercises To Help Network Students Come Up To Speed On Java

We present three warm-up exercises to introduce the students to Java.

[4] At the time of writing this draft of the paper, both courses were still in progress. The last lab meets on April 28th, so if the paper is accepted, we will update this section before the final camera ready copy is due on May 1.

Warm-up Exercise 1: We first require the students to read chapters 1-4 in (Budd 1998). We then require them to simply compile and run the two example programs from chapter 4. These are "Hello, World" style programs to introduce Java program structure, basic concepts of syntax and semantics in Java, and procedures for compiling and executing applications. In CIS320 and CIS662, we had students do this on both Windows NT and Linux as an individual assignment.

Warm-up Exercise 2: The second exercise is a slightly more elaborate group project, requiring considerable background reading and understanding, but relatively little code. The project involves taking an example from (Budd 1998 chapter 5) that implements a class `BallWorld` (a simple frame containing a red ball in motion, which "bounces" off the edges of the Frame), and combining this with a simple TCP client and server from (Harold 1997, chapter 7,8). The result is a new class `BallWorldServer,` which brings up the `BallWorld`, but also listens on a TCP port for connections from clients. When a client connects, the client can send messages (terminated by newline characters) containing colors (e.g. red, blue, green, etc.) The server then changes the color of the ball to the color requested by the client.

When we assigned this exercise in CIS320 and CIS662, we required the students to demonstrate their programs. Most used the example Java TCP client to connect to the server. We then asked them to connect to their server from another machine where they had a Unix account (an account where we had previously done no Java coding) using a regular telnet client; they did not know in advance that this would be required. This allowed the students to see the platform independence of a correctly designed protocol.

Although seemingly not earth-shattering in its practical application, this simple exercise also accomplishes several goals related to introducing Java and Java network programming. (1) It requires students to study Chapter 5 of Budd in some detail, introducing basic concepts of Java inheritance, and basic operations of the AWT (including the creation of Frames, and basic graphics, including simple animation). (2) In order to provide the capability of receiving network packets while simultaneously keeping the ball in motion, it is necessary to have multiple threads of execution. Conceptually, this is an important concept to understand, since most modern network clients have a GUI that must be serviced simultaneously while processing network events. Therefore we also had the students review section 20.1 of Budd on threads, which thankfully does not require any additional Java knowledge beyond that in chapters 1-5. (3) It provides a motivation for students to study chapters 7 and 8 in the Harold text, which cover basic TCP socket operations for clients and servers. We had previously required chapters 1-4 of reading, which provided all the necessary background for understanding chapters 7 and 8. (4) It requires an understanding of streams, thus motivating students to study sections 14.1 and 14.2 of Budd. (5) Because the server has to compare an incoming string such as "blue" to a string constant "blue", it provides an opportunity to cover the difference between the "`==`" operator, and the `equals` method of class `String`.

Despite the fact that the assignment required quite a few Java concepts, we made it clear that beyond introducing students to the procedures for compiling and executing Java applets and applications, we would not cover this material in lecture. We explained that we expected the students to rely on the textbooks and each other (in their groups) to learn the necessary Java; and that while we would answer specific questions and help with debugging specific problems, we would not do their reading for them. We limited ourselves to providing help only when it was clear that the students could not proceed without it. We were pleased that all eight groups (in both the undergraduate and graduate courses) were successful in delivering a working application within two or three weeks.

Warm-up exercise 3:: In the second warm-up exercise, students implemented a simple ASCII protocol for setting the color of a ball. In this exercise, students explore the issue of packing and unpacking binary fields in network byte order. Working again in groups, students must implement a class `ByteArrayConv,` with six static public methods. The first two are:

```
public static void packInt(byte[] dest, int src, int offset);
  // place a 32-bit integer src in network-byte order
  // into the bytes dest[offset], dest[offset+1], dest[offset+2], dest[offset+3]
public static int unpackInt(byte[] src, int offset);
  // return the 32-bit integer which is encoded in network byte order
  // in dest[offset] through dest[offset+3]
```

The class also contains similarly defined methods packShort, unpackShort, packLong and unpackLong for 16 and 64 bit integers. The students are also required to design three driver applications to test the class. First, the students show that via a standalone application their implementation can correctly encode and decode byte arrays on a single host. Then they must build a client/server pair demonstrating that Application Protocol Data Units (APDUs) containing integers of various sizes can be sent across the network using either TCP or UDP. The simple TCP and UDP client and server programs in (Harold 1996) or (Courtois 1998) can be used as a model.

To practice the art of software testing, the students can be told that the driver program will be graded on the basis of a reasonable tradeoff between completeness vs. ease of testing. That is, there are tradeoffs among testing as many cases as feasible, completing the test in a reasonable amount of time, and giving a clear indication to the tester as to whether the implementation under test is correct. After students submit their programs electronically, the instructor can post them on a web site; students can then be required to submit evidence that their implementation of `ByteArrayConv` works with all the other groups' driver programs. Some incentive can be offered to any group showing that their driver finds a bug in any of the other groups' implementations.

This exercise also serves as a good departure point for a general discussion of software testing, as well as presentation layer issues such as byte-ordering, and protocols like ASN.1 (used in Network Management, and OSI protocols) and XDR (used in Sun Remote Procedure Call.)

4. Exercises for Investigating Network Performance

We have devised three exercises related to measuring network performance. In each case, the first step is the development and testing of some software. Once the software is written and tested, then a variety of experiments can be performed. In each case, we suggest some initial hypotheses, and some experiments to test these. We then require the students to interpret their data, propose further hypotheses and experiments, conduct the experiments, and then write up the results. Some of the exercises below require methods from Java 1.1.

The three experiments are each designed to investigate a different aspect of performance measurement:
Exercise 1: The difficulty of measuring time on a single host
Exercise 2: The difficulty of measuring time between two hosts, and estimating round trip time
Exercise 3: Measuring throughput, and packet loss probability

4.1 Measuring Time on a Single Host

In this experiment, students develop a `StopWatch` class, with methods `reset(), stop(), start() and read()`. The `read()` method returns a `long`, which is the number of milliseconds on the `StopWatch`. Basically, the class is a wrapper for the `System.currentTimeMillis()` method. Once the `StopWatch` class is developed, we require students to test it by timing a long CPU bound loop (e.g. incrementing a counter millions of times) or a `sleep()` call statement in a thread. Groups can also be required to do code reviews of one another's code, after turning in their own code. Once students are satisfied that the class operates correctly, we propose two conflicting hypotheses about the sequence of operations, which starts and stops the `StopWatch` as fast as possible *n* times, storing each sample in an array:

```
    for (int i=1; i<n; i++) {clear(); start(); stop(); sample[x] = read();}
```

- Hypothesis A: The sample values will almost always be zero, because our target architectures are fast enough to process all the related calls in less than one millisecond. The samples will occasionally be 1, but less than 0.01% of the time, so this event is negligible, and no sample will ever exceed 1 millisecond.
- Hypothesis B: If we do the sequence above, a significant number of the samples will be greater than zero; sometimes much greater than zero, perhaps because of the effect of context switching, page faults, or other implementation details.

The experiment is then to measure the samples for various values of n, determining the min, max, mean, and standard deviation. Students may also be required to present a histogram of how many values fall in certain ranges, or a graph of a probability distribution. Our experience suggests that students can find significant evidence of hypothesis B if they run the program on a heavily loaded multi-user machine.[5] This can be the starting point for a discussion of how experiments involving measurements of throughput and delay should be designed to deal with such anomalies in time measurement.

[5] For example, in one experiment with *n*=1,000,000 we found on the machine "grumpy.cis.temple.edu" (a DEC Alpha 3000/300L running Digital Unix 4.0D, an NFS client of a heavily loaded server), that while the average sample was .024 ms, there were two samples that were above 100ms (specifically, 144ms and 398ms).

Several variations are possible from this point. A second stop watch can be used to measure the total time taken by the loop; if this is divided by n, this can be compared with the mean of the samples. Students can be required to form a hypothesis about the relationship between these two numbers, and then investigate their hypothesis.

4.2 Measuring Time Between Two Hosts, And Estimating Round Trip Time

This exercise explores the difficulty in measuring time when we are not sure whether clocks are synchronized between two hosts. The students are required to build a client and server that exchange UDP packets via a simple protocol. In addition, four timestamps are taken by assigning long variables a, b, c, and d the value returned by `System.currentTimeMillis()` at various points in the exchange, as described below.

The client sends a packet containing only a 4-byte sequence number (the `ByteArrayConv` class developed in warm-up exercise 3 is used to encode the sequence number.) Immediately before the `send` method of `DatagramSocket` is invoked, the timestamp `a` is taken. The server invokes the `receive` method of `DatagramSocket` to read the datagram from the client; it then immediately takes the timestamp `b`. The server then forms an outgoing datagram of 20 bytes, containing the sequence number, and space for two 64-bit timestamps; the value of b is placed in the first of these. Then, the timestamp c is taken by the server, and placed in the packet immediately before invoking the send method of `DatagramSocket` to send this reply back to the client. When the client receives the reply, it immediately takes timestamp d, and verifies that the sequence numbers match. If they do, then the values a, b, c and d are stored for later interpretation.

Because UDP is a datagram protocol, there is the possibility of packet loss. In Java 1.1, this can be handled by using the method `setSoTimeout` method of class `DatagramSocket` (Harold 1996, p. 228). This method sets a maximum number of milliseconds that the client should wait for a response after invoking `receive`; the client can catch the exception thrown if the response does not come within that period.

Once the students understand the basic operation of the protocol, they should be asked to make some predictions about the values of a, b, c and d *before* they write the code. In particular, some questions they might consider are:

1. Depending on the nature of the network connecting the client and server programs, the duration (d-a) may include many components: propagation delay, packet transmission time, processing time, and queuing delays, among other things. Based on your knowledge of the network that connects your server and client, describe all the components that make up d-a for your network. Then, make an educated guess as to the min, max, mean and standard deviation that you expect to observe if you take a large number of samples (e.g. n=1000, or n=1,000,000 or more)
    2. The value (c-b) is an estimate of the processing time at the server. What will cause this value to vary? Is there processing that this quantity is not taking into account? What value do you predict for the min, max, mean and std. dev. of this value. (Hint: look back at the `StopWatch` class assignment.)
    3. Suppose you know for sure that packets travelling between the server and client take the exact same route, thus the propagation delay from client-to-server and server-to-client are equal. However, you suspect that the clocks on the two computers are not synchronized. In that case, devise a formula in terms of (a,b,c,d) to measure the difference between the two clocks.
    4. What might cause the formula you derived in the previous question (#3) to give you an inaccurate measurement of the difference between the two clocks?
    5. What quantity should we measure in terms of (a,b,c,d) to get the best estimate of the true round-trip propagation delay?

After submitting answers to these questions, the students should write the programs, test them, and design experiments to test their hypotheses, for example, by repeatedly taking 1,000 or 1,000,000 samples of (a,b,c,d) and computing the mean, max, mean and standard deviation of various quantities. They can then be required to write a brief report of their results, how they compare with their predictions, and what conclusions they can draw from these results.

They can also consider the following questions, based on considering the min, max, average and standard deviation of the values (b-a) and (d-c):

6. Do these values seem to present accurate indicators of the one-way propagation delay for the systems you measured? Why or why not? If you knew for sure that the clocks were synchronized, would that change your answer?

7. If you don't know whether the clocks are synchronized, and you don't know whether the one-way propagation times are equal in both directions, can you conclude anything about either of these issues from the (a,b,c,d) values? Why or why not?

Finally, the exercise can be a starting point for discussing the time in general; see for example RFC956 (Mills, 1985), and the various RFCs on the Network Time Protocol.

4.3 Measuring Throughput, and Packet Loss Probability

Our third exercise involves measuring throughput and packet loss probability, but it also touches on several other themes. It provides students an opportunity to demonstrate to themselves and to the instructor whether they have really understood concepts like IP fragmentation, the overhead of header sizes, and the necessity of flow control to prevent queues from overflowing. After making various predictions and designing experiments, the instructor can make suggestions regarding the experiments, and then have students carry them out and present their results.

In this exercise, the students implement classes for a `UDPExperimentClient` and a `UDPExperimentServer`. The `UDPExperimentServer` has a `main()` method which implements a standalone server application. The server's main loop services requests for experiments until the server is terminated. The `UDPExperimentClient` class implements an object that can communicate with the server to perform repeated experiments, and report the results of these experiments. This class provides the following public methods:

```
public UDPExperimentClient // constructor
      (String host, short port) // host and port where server is running
      throws Exception; // if host unknown

public void doExperiment // perform one experiment
      (int count, short size, // number of packets(>=1), size of packets
       long gap, // interval between packets in milliseconds (>=0)
       long timeout) // how long to wait for last packet in milliseconds(>0)
       throws Exception;

// The following methods return performance statistics for the most recent
// successful experiment.  They throw an exception if (1) they are called
// before the first call to doExperiment, (2) when the most recent experiment
// was not successful (i.e. threw an exception) or (3) if the statistic is
// undefined for the particular experiment (see ”throughput()” below.)

public boolean lastExperimentValid (); // true if last experiment succeeded
public long packetsReceived() throws Exception; // ignoring duplicates
public long totalBytesReceived() throws Exception; // includes duplicates
public long duplicatesReceived() throws Exception;
public long totalTime() throws Exception; // total time on StopWatch
public float throughput() throws Exception; //total bits received/totalTime;
      // undefined if packetsReceived != count
public float packetLossProb() throws Exception; // 1-(packetsReceived/count)
```

The following protocol is used: The client sends a 14-byte PDU containing `count` (4 bytes), the `size` (2 bytes) and `gap` (8 bytes). The server then responds by sending `count` PDUs to the client in reply; each of these contains a 4 byte sequence number, and then zero padding of length (`size`-4). If `gap` > 0, between each send, the server pauses for `gap` milliseconds. To measure throughput, the client uses the `StopWatch` class developed in exercise 4.1. Before sending the request packet, the `setSoTimeout(timeout)` is invoked on the socket, and the stopwatch is started. It is stopped when either: (a) count packets have been received or (b) the `setSoTimeout` expires. (No retransmission is done at the server, and there are no acknowledgments from the client.)

As with the previous exercise, students are asked to write up answers to several questions before running experiments with their software:

1. Ideally, we would like to record the time the server begins sending packets and the time the client receives its last packet. The throughput is then the number of bits received divided by the difference between these two times. However, on this example, as with the others, we run into the problem that the client and server's clocks may not be synchronized. Thus, as an alternative, we have suggested using the time the client first makes the request as our start time, even though this may introduce some error. How can you construct an experiment and interpret the results in such a way as to get the most meaningful results for throughput? Can you estimate the size of the error, or adjust for it in some way?
2. What do you predict will happen to throughput if we hold `count` fixed at some value (whether that be 1,000, 1,000,000 or whatever) and let size take on various values? (e.g. 10, 50, 100, 500, 1000, 5000, and 10000?) Take into account the effects of IP fragmentation, and the overhead of packet headers.

3. Suppose we were to run an experiment on a single Ethernet with count = 10000, size = 1000, and gap = 0ms, and we found that packetsReceived==count, that is there was no loss. Form a hypothesis about which values you would need to change (count, size or gap) in order to measure some loss on your network; that is, to find an outcome where packetsReceived < count. Keep in mind that since UDP lacks flow control, packet loss may be as result of buffer overflows in the sending or receiving protocol software, rather than data transmission problems such as collisions or noise.

   Taking this one step further, assume for the moment that it *is* possible to find some value of count, size and gap that result in packet loss on your system. Design an experiment that will efficiently find some specific value of (count, size, gap) for which at least one packet is lost, at least 50% of the time on average.

   As an additional constraint, ensure that the experiment you design is one that (after the software development and testing is finished) you (or your group) could actually carry out over the course of one week of the semester. You will need to estimate how long each experiment will take to run, and how long it will take you to interpret the results. For example, running four billion experiments is probably not feasible unless you automate the process, and each experiment runs remarkably quickly. Also, suggest a reasonable value for the timeout parameter, and justify your argument.

   Finally, suggest a stopping point for declaring your search unsuccessful: that is, given your time constraints, when can you conclude that you have tested enough cases, and made enough repetitions of your experiment, that you could argue that it isn't reasonable to continue looking for a packet loss?

   Note that you also need to consider the fact that other students and/or groups may be using the network, so you need to assess the effect of your experiments on regular work in the lab, as well as interference between your experiments and those of your colleagues. (Before running *any* experiments involving sending large numbers of packets, you should get your instructor or TA's approval; she/he will coordinate approval from the network administrators.)

4. Now suppose you carried out the experiment from question 3 and found some value of (count, size and gap) for which the packet loss probability is, on average, *p*. Now predict what would happen to the packet loss probability if we fixed `count` and `size`, and varied the `gap` parameter. Consider the effect of varying `gap`
   (a) in a linear fashion, e.g. starting with 50, then trying 60, 70, 80, 90... and 40, 30, 20 10), and
   (b) exponentially via repeated doubling, or repeated halving e.g, if `gap` is 50, try the values 100, 200, 400, etc., and the values 25, 12.5, 6.25, 3.125, etc, rounding to the nearest millisecond: 25, 13, 6, 3, etc..

What relationship do you predict between gap and packet loss probability: linear, quadratic, exponential or some other relationship? Make similar predictions for what would happen if we fixed `gap` and `size` and varied `count`, or fixed `gap` and `count`, and varied `size`.

5. Observations and Future Work

The purpose of this paper is merely to share some ideas about how to incorporate Java into a survey course on networking, and was undertaken after the course was already in progress. Therefore, we did not lay the groundwork for any formal study of the effectiveness of this approach over other approaches; that would be a good subject for future study. What we can offer is some anecdotal observations based on our experience so far. On the positive side, we have observed that the students seem to really enjoy the Java Programming exercises. Much of this may be due to the fact that both networking and Java are "hot" topics. However, there is a certain degree of fun in exploring communication between two computers; until this course, most of their programming assignments had involved a single host only. On the negative side, one concern we have is that in spite of the fact that the textbook we used very much emphasizes careful object-oriented design principles, some students still have a tendency to "hack" their solutions together. In the early set of "warm-up" assignments, we observed two particular maladies:

(1) throwing in sections of code from any one of the hundreds of Java books on the market without really understanding what that code does or how it is structured, in an attempt to cobble something together, and

(2) focussing too much on fancy user interfaces rather than the core functionality required in the assignment.

Our strategy for ameliorating this situation is to provide very specific class specifications and avoid graphics altogether in the second half of the course.

For the future, we plan several additional exercises. One exercise will repeat the throughput client/server exercise using TCP, to compare the throughput of the TCP vs. UDP. As a Java programming exercise, the students could be asked to submit a design for restructuring both the TCP and UDP client and server software by designing a generic `interface ExperimentClient` and `interface ExperimentServer;` the specific UDP and TCP client and server entities would be implementations of this interface. We also plan to design an exercise to build a UDP packet reflector that can simulate packet loss, packet transmission time (equivalently: bit-rate or bandwidth) and propagation delay. Finally, we plan a set of exercises to implement simple protocols for positive acknowledgement with retransmission over UDP, along with sliding window flow control techniques such as stop-and-wait, go-back-N, and selective repeat (cf. Tanenbaum 1996, Chapter 3). The students could then use the techniques for performance measurement and experimental design they have learned to design experiments comparing the performance of their protocol implementation with the analytic results presented in various networking textbooks.

Bibliography

(Budd 1998)	Budd, Timothy. *Understanding Object-Oriented Programming with Java*. Addison-Wesley.
(Courtois 1998)	Courtois, Todd. *Java Networking and Communications*. Prentice-Hall.
(Comer 1999)	Comer, Douglas E. *Computer Networks and Internets*, 2nd Edition. Prentice-Hall.
(Harold 1997)	Harold, Elliotte Rusty. Java Network Programming (First Edition), O'Reilly.
(Mills 1985)	Mills, David, *Algorithms for Synchronizing Network Clocks*, Internet RFC956.
(Tanenbaum,1996)	Tanenbaum, Andrew. *Computer Networks*, 3rd Edition. Prentice-Hall.

Acknowledgements: The `StopWatch` class presented here was inspired by a C++ version written by Chris Brown at the University of Delaware.