



## I. Introduction – (GROUP SHEET)

With the ever increasing dependence on the Internet and computer networks in general, the importance of well designed client/server architecture is most certainly clear. Client Serverlutions comprises of a group of individuals who strive to produce systems that apply the fundamentals of such architecture. This paper outlines how Client Serverlutions was created and the process the group followed for the completion of Assignment #1 for CS-604, Fall 2006.

All six students in this group registered for the Distance Learning section of this course instead of the in-class version for a primary reason, the convenience afforded by remote learning through the Internet was either difficult or impossible to pass up. It was actually for this reason that Professor George Blank recognized their compatibility and officially grouped them together by September 19, 2006. Immediately following the ratification of our group, the first step was to determine a methodology for working together.

## II. Group Plan

As all six members of the group had already introduced themselves upon the start of the semester, we already had an idea about what skills and experience each member possessed. For this reason the first task was to define a platform for communication. At this point all group members agreed to having remote meetings rather than the impossible option of meeting in person. A complicating factor was also that between the six of us, having live meetings would not always be possible as there were ever present time zone conflicts as well as work hours for those of us who were employed. Email was the preliminary method of communication for determining this information and it was also leveraged for allowing everyone to join a “Google Group” for subsequent communication. The Google group allowed members to post to a forum similar to the NJIT WebBoard, however with more administrative power than what a student would ordinarily have. This board facilitated the concept of a “Continuous Meeting”, where at any time a member could alert the others with a question or information regarding their thoughts on the task at hand. Signing up for an account with Google also affords the luxury of live chatting when necessary, and this suite of capability served well for our project. The first topic on the Google Group was “Assignment #1”

## III. Completing the Assignment

Assignment #1 consisted of two similar, sub-assignments dealing with sockets. The first was to create a TCP Java Client/Server system, and the second was a UDP C/C++ Client/Server system. All of the member skills were placed into a spreadsheet dealing with member proficiencies and this helped us determine that creating two sub-groups would be a natural way to divide up the required work. It worked out that having three members focus on the actual JAVA coding and the other three write in C++ would be best. In parallel, both sub-groups followed the following process to complete the sub-assignment: A single individual created a rough working version of the code. Then the entire group discussed missing features for each program (a total of four, two clients, two servers). After these suggestions had been unanimously approved the other members implemented these features. Examples of these additional features were: Additional Platform Testing, Code Readability and Commenting, and even multithreading support for the server applications. It should also be mentioned that this dynamic level of concurrent was made possible through the use of a code versioning server that was implemented from the start of the project. If at any time, more than one member made a change to an application that was already in progress, the system would alert the conflicting author and provide for them a way to handle the conflict before committing the latest code. One final note is that all members customized the applications to their liking for the final step.

## Table of Contents

<b>I. Time Sheet</b> .....	2
<b>II. Modifications Made</b> .....	2
<b>a. C++ Version</b> .....	2
<b>b. JAVA Version</b> .....	4
<b>III. Problems Encountered and Solutions</b> .....	4
<b>a. Missing Library</b> .....	5
<b>b. Incorrect Library</b> .....	5
<b>c. JAVA SDK Upgrade</b> .....	5
<b>IV. Lessons Learned</b> .....	5
<b>V. Screen Shots</b> .....	6
<b>a. UDP C++ Client/Server</b> .....	7
Figure 1: C++ UDP Server in Execution .....	7
Figure 2: Client execution in bottom terminal window as well as packet capture .....	8
<b>b. TCP JAVA Client/Server</b> .....	8
Figure 3: JAVA TCP Server in Execution.....	9
Figure 4: Execution and Packet Capture of JAVA TCP client.....	10

## I. Time Sheet

Steven Regula		Time Sheet	
Date	Time	Task	Time Spent (hrs)
9/19/2006	5:45 PM	Summarize Group Member Skills and Contact Information in spreadsheet	0.5
9/19/2006	6:15 PM	Compose initial team welcome email (initiates many important topics as per group sheet)	0.5
9/20/2006	3:30 PM	Created a Google Group and Proposed Team Subgrouping Method	1
9/21/2006	7:30 AM	Replied to (cc'd all) Robert Campbell about Google Group and my Assign1 plans for weekend	0.25
9/21/2006	12:00 PM	Replied to (cc'd all) Robert Campbell about myself hosting the CVS server and using AFS for testing	0.5
9/22/2006	7:30 PM	Created Subversion (like CVS) repository on my home machine, created accounts for other 5 members	1.5
9/22/2006	9:00 PM	Replied to (cc'd all) Ross Romano about using OS X, provided brief tutorial on Subversion (like CVS)	0.25
9/23/2006	8:00 AM	Created and committed clean and object oriented UDP C++ Client and Server Program	4
9/23/2006	12:00 PM	Alerted team (most notably subteam) about remaining tasks (documentation, output formatting, threading)	0.5
9/24/2006	8:30 AM	Reviewed and customized JAVA TCP Client/Server application	1
9/24/2006	9:30 AM	Determined compiler flags for enabling successful compilation of the C++ version on SunOS (-lxnet)	0.5
9/24/2006	10:00 AM	Advised team of remaining tasks, (commenting, group sheets)	0.5
9/25/2006	7:30 AM	Modified Ross's udpCommon.h to use <stdarg.h> instead of varargs to enable compilation on my platform	0.5
9/28/2006	6:15 AM	Devised and proposed team name "Client Serverlutions"	0.5
10/1/2006	9:00 AM	Group Sheet	1
10/1/2006	10:00 AM	Individual Report	3
10/1/2006	3:00 PM	Individual Report (Cont'd)	1.5
		Total:	17.5

## II. Modifications Made

### a. C++ Version

As the lead C++ composer in my subgroup it was my job to present the initial working framework of both Client and Server C++ UDP socket applications. In the past I had not created a UDP C++ application so I began looking for an appropriate tutorial and/or code sample. It was not before long that I found an appropriate, however C, tutorial at:

<http://gnosis.cx/publish/programming/sockets2.html>. This tutorial was only a

starting point and by the end of this part of the assignment, my code had evolved into something very different, only retaining the socket creation and related structures. Changes that we made by me include the utilization of object oriented design. This allows the code to be extended very easily by my teammates and also reuse at a later date. The following class was created for the client:

```
class udpClient {
public:
    udpClient() {}
    void connectSocket(char *, char *, char *);
    void gracefullyDie(char *, ...);
private:
    int testVar;
};
```

I implemented a similar class for the server:

```
class udpServer {
public:
    udpServer();
    void createSocket(char *);
    void gracefullyDie(char *, ...);
private:
    string ourName;
};
```

Both of the C++ applications were modified to utilize command-line parameters so that hosts, ports, and/or client name could be provided at the time of execution rather than design time. Below is a sample of that code from the client main function:

```
if (argc != 4) {
    cerr << "USAGE " << argv[0] << " <host> <udp port> <word>"
         << endl;
} else {
    udpClient p;
    p.connectSocket(argv[1], argv[2], argv[3]);
}
```

Another important modification was to change the way the client verified the buffer length received from the server. Originally, the server passed back the exact string originally sent by the client. To verify that the transmission was successful, the client would compare the newly received buffer size and abort if it did not match the originally passed string. Since the server was also passing back its name to the client, the client buffer size was re-written to simply terminate the string and not perform the check as follows:

```
received = recvfrom(mySock, buffer, BUFSIZE, 0,
                   (struct sockaddr *) &echoclient,
                   &clientlen);
...
/* Assure null terminated string */
buffer[received] = '\0';
```

Finally, after a team member had subsequently modified my code, I ran into a compilation error due to a missing library. The problem is described in the next section, however the modification to `udpCommon.h` is below.

```
Old:
#include <sys/varargs.h>
New:
#include <stdarg.h>
```

## b. JAVA Version

While I was busy creating the C++ routines, my counterpart was doing similar work on the JAVA TCP client/server system. It was not before long that I was finished customizing the C++ version and was ready to perform my own modifications to the JAVA as well. The first thing I wanted to do with the JAVA version was to again allow all server and client parameters to be specified at execution time, rather than being hard coded. This was not exceedingly difficult, but indeed very useful as I was testing the system on multiple platforms for my own learning experience. My modified client JAVA source is below:

```
if (args.length != 3) {
    System.out.println ("Usage: java NameEchoClient <host>
    <port> <name>");
} else {
    String host = args[0];
    String name = args[2];
    int port = Integer.parseInt(args[1]);
    ...
```

This was also done for the JAVA server:

```
if (args.length != 1) {
    System.out.println ("Usage: java NameEchoClient <port>");
} else {
    int port = Integer.parseInt(args[0]);
    ...
```

Another important change to the JAVA code I performed was to format the output in a way similar to what I had created in the C++ version. By design the server is what sends the messages so it is that program that I needed to modify:

```
if (str != null) {
    out.println("Hello: " + str);
    out.flush();
    out.println("From: " +
        InetAddress.getLocalHost().getHostName());
    out.flush();
}
```

## III. Problems Encountered and Solutions

Throughout the development process I encountered a few problems. The C++ version seemed to be the most prone to problems, however, I did need to do

additional research when upgrading my Linux development machine to the latest JAVA sdk.

**a. Missing Library**

C++ Compilation of my program on AFS. I had done my C++ development on my Linux machine, and had not been able to test compilation on the AFS cluster. Soon, Jigar mentioned errors, which was initially a surprise to me until I remembered I needed to test on the NJIT system. It was not long before I realized he was talking about compilation so I promptly got to work on that problem. Fortunately, the compiler was very verbose and it was clear that I was missing a library that needed to be linked. Posting the error to the gcc.gnu.org yielded several postings that did not specifically address my problem, however, they all incorporated a compiler flag “-lxnet”. By revising my compiler statement of:

```
g++ udpServer.cpp -o udpServer
```

Instead to:

```
g++ udpServer.cpp -lxnet -o udpServer
```

Compilation was successful. At that point I promptly replied to Jigar with my information.

**b. Incorrect Library**

The next problem with the C++ portion of the assignment came when Ross added the multi-threading support our team had discussed. He utilized an include file I had no previous experience with that was called: <varargs.h>. This library, albeit legacy, is used to handle variable argument lists. This library also was not found on my Linux development machine. It was not before long that I found numerous posts on the web from people compiling various applications with gcc who encountered similar problems. The usual advise was to replace this library with stdarg.h. After trying this recommendation and successfully compiling on my development machine I tested this on AFS successfully as well and committed the code. Also, I notified the team members about this change and my reasoning for doing it.

**c. JAVA SDK Upgrade**

The only problem I had was while upgrading my development JAVA environment on my Linux development machine. In the hopes of a smooth upgrade, I downloaded the latest RPM (Redhat Package Manager) from the Sun website. After installing the RPM, the compiler “javac” could not be found. My solution was to uninstall the RPM and simply download and execute the Sun BIN file instead. The difference from using an RPM is that you manually create a directory and execute the BIN file from the directory where you would like to install JAVA. This is the process I usually use and should have utilized that technique first instead, as it installed flawlessly.

**IV. Lessons Learned**

Completing this assignment involved lessons on several different levels. The first level is purely academic learning. Items that can be grouped into this category are itemized below

- 1) Getting my first C++ UDP client/server system running. I had not previously created a C++ UDP program and overall it was a rewarding experience.
- 2) The purpose of `sys/varargs.h` and eventually `stdargs.h`. As mentioned above these libraries allow for variable argument lists, I can see myself utilizing this functionality in the future to support extensibility.
- 3) The “dox” documentation project. I have not mentioned it previously, but in addition to source file remarks, Ross leveraged the “dox” project to allow for Javadoc format comments in both our C++ and Java files. Though this documentation went above and beyond what was required for the assignment, it facilitated an understanding of our source files for a person who cannot read code. The comments that are created are made available in HTML format as well as LaTeX. In the future, should documentation become more of a priority in our assignments, we have a sound framework already in place from as early as our first assignment.

In addition to this learning, I also refined my interpersonal skills

- 4) I was a de-facto manager of the project, always keeping a history of the work that was being done, who was doing it, and what needs to be done next.
- 5) I learned that using the well known technique of “divide and conquer” can work very well in group projects. (As with our sub-group technique)
- 6) A group does not necessarily need to meet in person to complete a professional application.
- 7) Exploit the talents of each individual member to keep them interested.

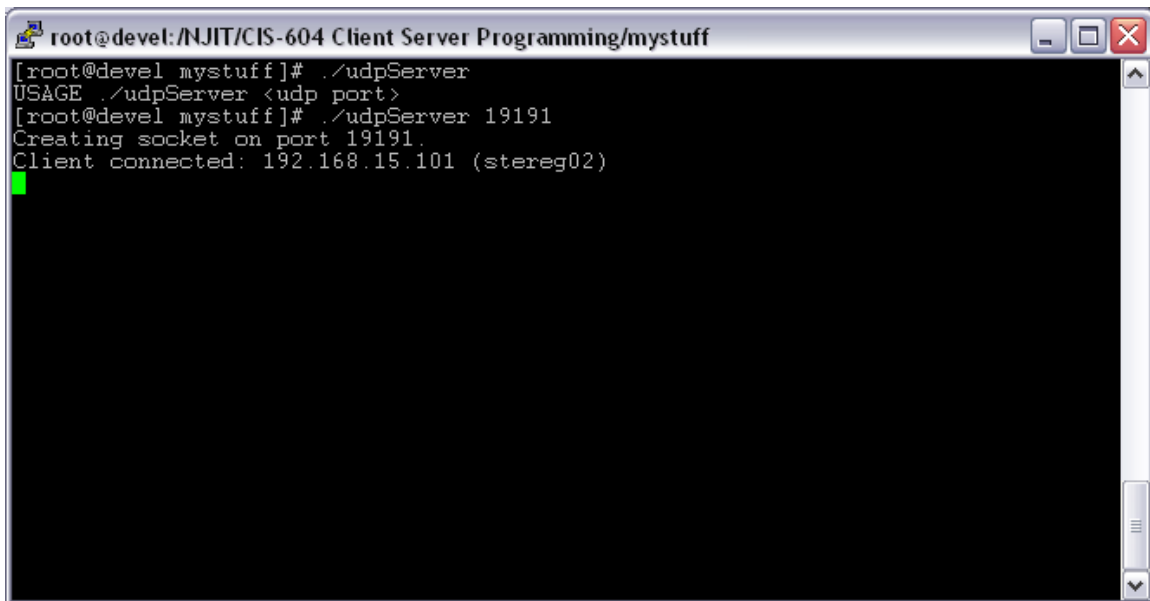
Finally, there are the things I learned as a result of problems during the project (sometimes mentioned earlier):

- 8) To compile my Linux socket programs on SunOS I need to use the “-lxnet” flag.
- 9) `Varargs.h` is basically deprecated and instead `stdargs.h` should be used.
- 10) Utilize the Sun .BIN files for best results when upgrading a JAVA SDK, as they are not Linux distribution specific.

## V. Screen Shots

**a. UDP C++ Client/Server**

In this section I present screenshots to show the reader what it looks like from the perspective of a user of any of the applications documented here. The first screenshot (Fig. 1) is a terminal window representing an SSH session to my development machine. First the program is executed without any command line parameters, at which point the program does not initiate the server but instead points out the proper execution syntax to the user. Immediately following the syntax response I have executed the UDP C++ server by typing “./udpServer 19191” to set the listening port to 19191. The server then responds by letting the user know that the “Socket is being created” and begins waiting for connections. At last when a client connects the server echo’s the following to STDERR “Client Connected:” followed by the IP Address of the user who has connected, and also the client’s name. In this case, the client is my laptop. At the point where you see this message, the server returns its hostname, as well as the client name back to the client.



```
root@devel:/NJIT/CIS-604 Client Server Programming/mystuff
[root@devel mystuff]# ./udpServer
USAGE ./udpServer <udp port>
[root@devel mystuff]# ./udpServer 19191
Creating socket on port 19191.
Client connected: 192.168.15.101 (stereg02)
```

Figure 1: C++ UDP Server in Execution

To show that the server is actually returning its name to the requestor, the client program produces more output. The client program in fact returns output including a “Hello <client name>”, and also a “From: <server name>.” Also, you’ll notice in the screenshot listed as Fig. 2, a packet capture was performed above to further prove that a network connection actually occurred.

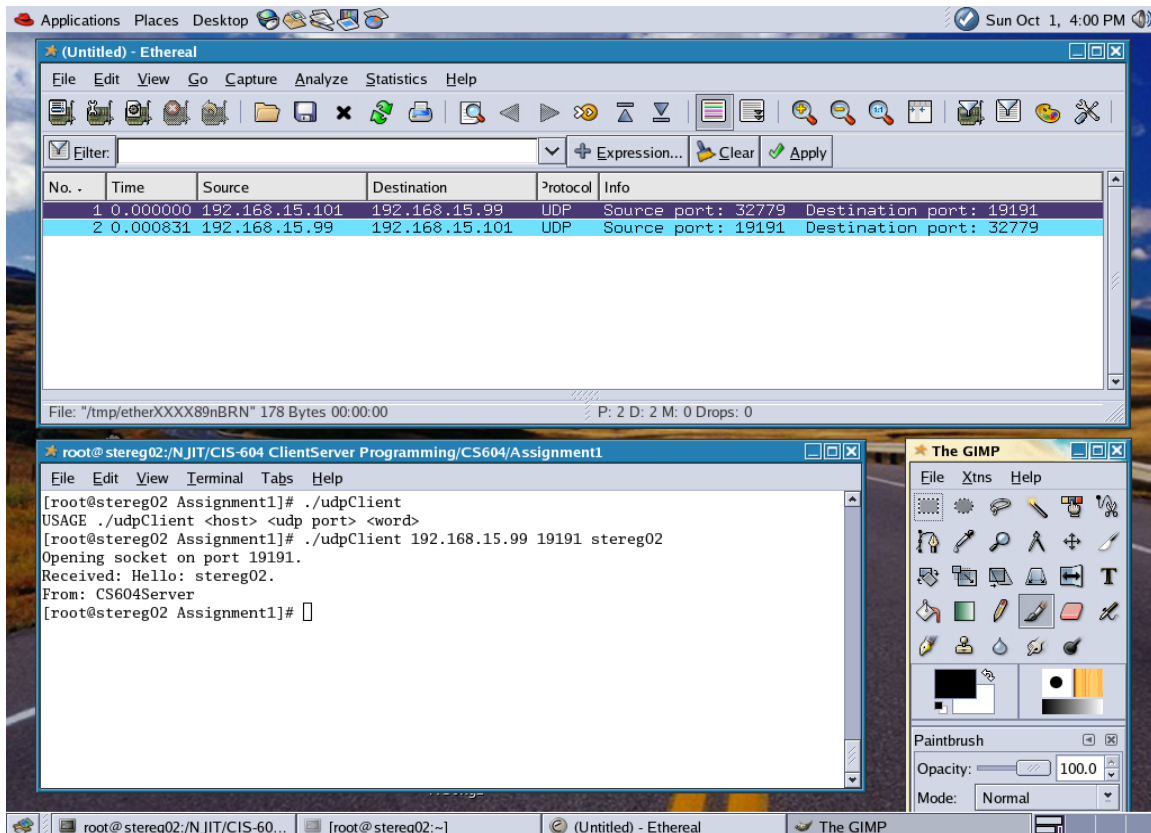
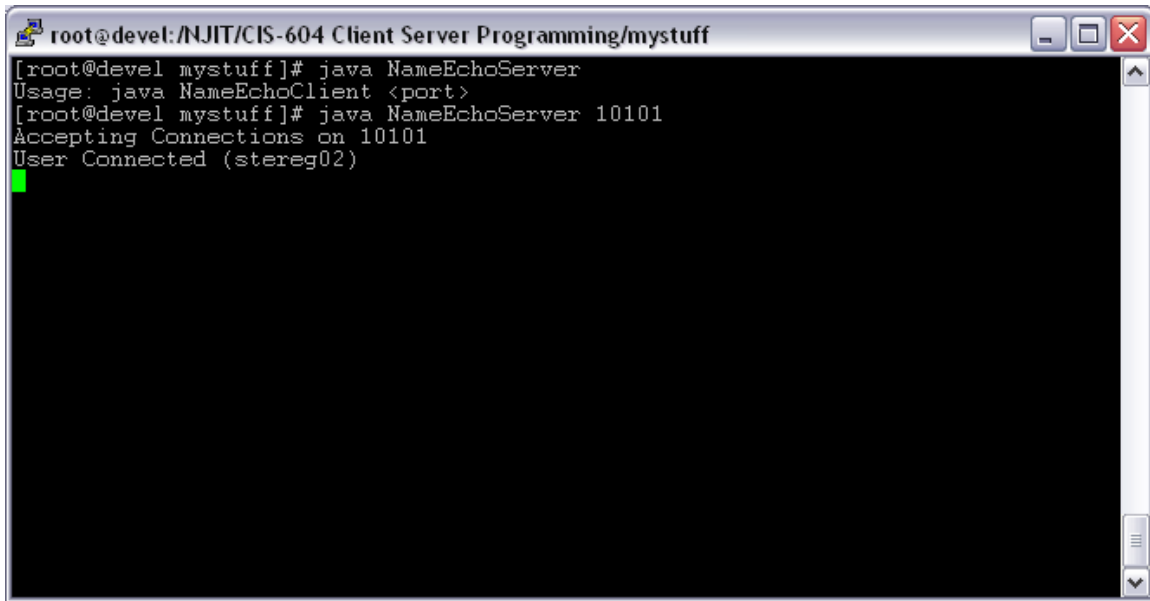


Figure 2: Client execution in bottom terminal window as well as packet capture above

### b. TCP JAVA Client/Server

In addition to the UDP C++ screenshots, it is also necessary to document the working JAVA TCP socket implementation. I have refined the JAVA version to function very similarly to the C++ implementation. In fact, doing this allowed me to compare traffic based on packet captures between the UDP version as well as TCP version. This was a learning bonus. In figure 3, the server program is first executed with no command line parameters. When this occurs, the server prompts the user on usage syntax (to include a port number). Finally, the user kicks off the JAVA server by typing “java NameEchoServer 10101” to start the server listening on TCP port 10101. The program then tells the user that it is listening on the chosen port and waits for connections. Eventually a client named “stereg02” connects and the server produces output based on this fact. Just after this output, the server sends back a “Hello: stereg02” and a “From: <servername>” message to the client.

A terminal window titled "root@devel:/NJIT/CIS-604 Client Server Programming/mystuff" with standard window controls. The terminal output shows the following commands and responses:

```
[root@devel mystuff]# java NameEchoServer
Usage: java NameEchoClient <port>
[root@devel mystuff]# java NameEchoServer 10101
Accepting Connections on 10101
User Connected (stereg02)
```

A green cursor is visible on the line following "User Connected (stereg02)".

Figure 3: JAVA TCP Server in Execution

Finally, the last screenshot required is the one displaying the execution of the JAVA TCP client. Figure 4 shows the client first executing the program with no command line parameters. In response to this, the program responds with the syntax necessary to connect to a server. At last I execute the client with the required command-line parameters that specify the host to connect to, the port that it is listening on, and my client machine name. Above the terminal window in the screenshot is the packet capture of the TCP stream that occurred during the client/server communication. You'll notice there is more traffic with the TCP version than the UDP version presented in Figure 2.

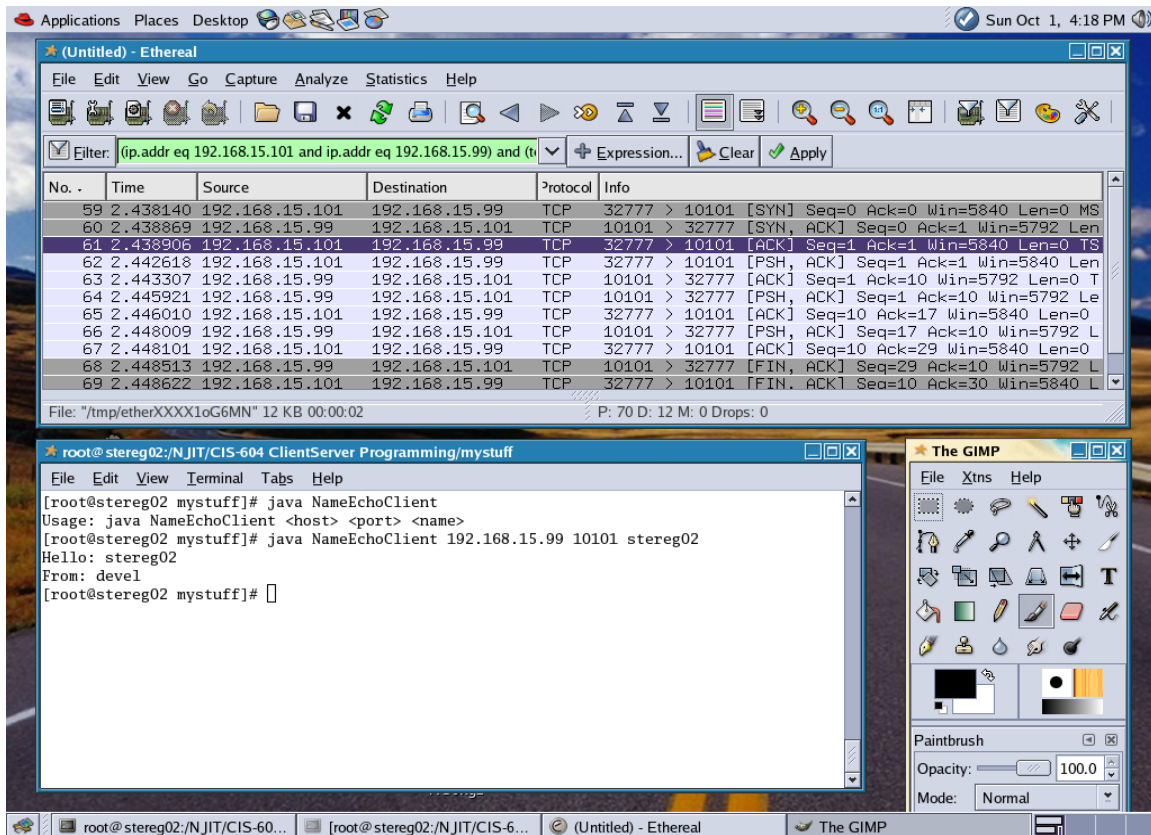


Figure 4: Execution and Packet Capture of JAVA TCP client