

IMPERIAL COLLEGE LONDON

INDIVIDUAL PROJECT

COMPUTING - BENG

---

# Packet Processing in Java

---

ASHLEY HEMINGWAY

*Supervisor:*  
Peter PIETZUCH

*2nd Marker:*  
Wayne LUK

February 2015

## Todo list

explain parameters and mapping for jString to char* etc . . . . .	9
Figure: Gantt chart would be very useful here . . . . .	11
Add references . . . . .	14
Add figure labels . . . . .	14
Do final read through . . . . .	14
Put through spell checker . . . . .	14
Other useful applications . . . . .	14
What font, size and line spacing? . . . . .	14

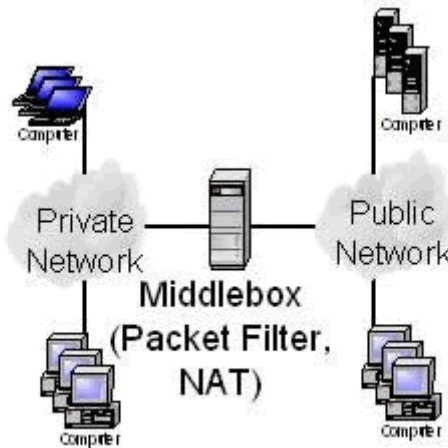
# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Objectives . . . . .	3
1.3	Solution Idea . . . . .	4
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Network Components . . . . .	5
2.1.1	Network Packets . . . . .	5
2.1.2	Network Address Translator (NAT) . . . . .	6
2.1.3	Firewall . . . . .	7
2.2	Java Features . . . . .	8
2.2.1	JVM Networking . . . . .	8
2.2.2	Java Native Interface (JNI) . . . . .	8
2.2.3	ByteBuffer . . . . .	10
2.3	I/O API's . . . . .	10
2.3.1	DPDK . . . . .	10
2.3.2	Netmap . . . . .	10
<b>3</b>	<b>Project Plan</b>	<b>11</b>
3.1	Time Plan . . . . .	11
3.1.1	Key Milestones . . . . .	11
3.1.2	Time Estimations . . . . .	11
3.1.3	Completed Milestones . . . . .	11
3.2	Fall Back Position . . . . .	12
3.3	Possible Extensions . . . . .	12
3.3.1	Off Heap Java Memory . . . . .	12
3.3.2	Limit Testing . . . . .	12
<b>4</b>	<b>Evaluation Plan</b>	<b>13</b>
4.1	Experiments & Outcomes . . . . .	13
4.2	Functionality & Usability . . . . .	13

# 1 Introduction

## 1.1 Motivation

As modern computing techniques advance, people are trying to find more generic solutions to problems which have been solved by native applications in the past. A main area of focus has been network middleboxes (network appliances), which are developed to manipulate network packets. Common examples of middleboxes are firewalls, network address translators (NATs) and load balancers, all of which inspect or transform network packets in the middle of a connection between a public and private network. In recent years, people have been developing a number of programmable middleboxes which allow these generic solutions to be used on a wide scale basis.



As middleboxes are mainly used for networking purposes, they are required to process network packets at line rate (i.e. at speeds which allow packets to be processed as they are received). This requires the application to retrieve the packet from the network line, inspect and transform the packet in the desired way and then insert the packet back onto the network line, all within a time period sufficient enough to not cause a backlog. High performance implementations of such applications are available, but are written in native languages, predominately in C/C++. However, more and more high performance computing projects are being developed in Java and have succeeded in performing at similar speeds to C/C++ applications.

The main challenge is actually getting the I/O system for the Java application to run at line rate speeds, due to challenges with how the JVM (Java Virtual Machine) interacts with memory and the computer's kernel. Once this challenge has been overcome, there are no reasons why programmable middleboxes written in non-native languages such as Java can exist within networking systems.

## 1.2 Objectives

The main objective for this project is to research, develop and test a new application which is written in a non-native language such as Java, but can process packets at a high performance level. This also requires that the application can perform I/O operations at line rate in order to pass on data packets without slowing the line rate. The aim is to match current native applications which are written in C/C++.

With the main objective stated above, I set out a few initial, smaller objectives in order to divide the project into more manageable objectives:

- Understand similar applications and API's written in C/C++ which process at line speeds and how the implementations can be exploited for Java applications
- Exploit two Java features which allows for better I/O performance: ByteBuffer's and the Java Native Interface (JNI)
- Implement basic middlebox applications in Java such as a firewall and a NAT
- Compare Java implementations to those which are pure Java and pure C/C++

### 1.3 Solution Idea

The idea is to implement a new library in Java using a few techniques in order to speed up network access. Firstly, no networking will be done via the JVM and the operating system kernel. Instead, the Java application will bypass the kernel altogether using a combination of direct memory access and high speed I/O operations via a C library which we interact with the application via the Java Native Interface (JNI).

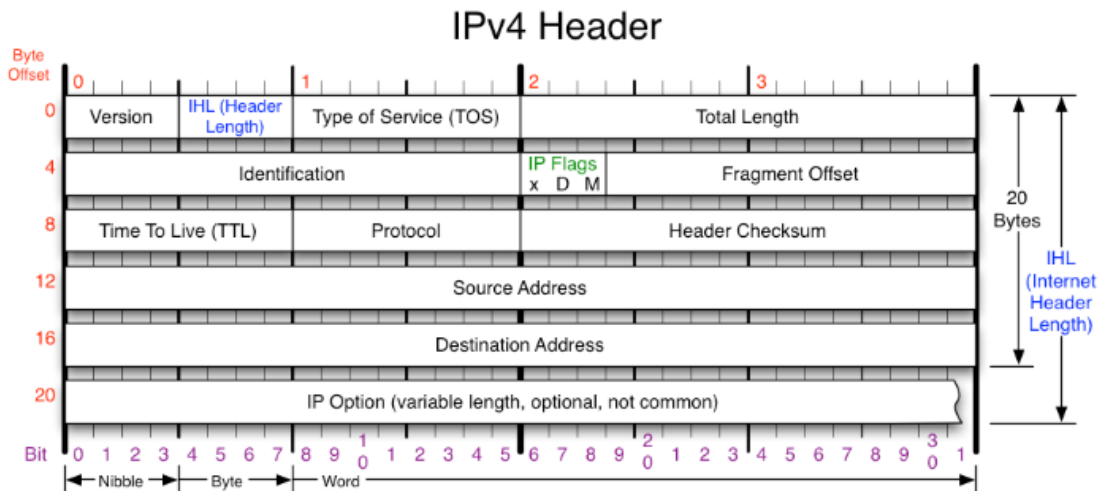
This eliminates the need for the JVM to call system calls from the kernel in order to do the network access, which can be relatively slow compared to direct network access achievable from native applications.

## 2 Background

### 2.1 Network Components

#### 2.1.1 Network Packets

A network packet is responsible for carrying data from a source to a destination. Packets are routed, fragmented and dropped via information stored within the packet's header. Note: in this report packets and datagrams are interchangeable.



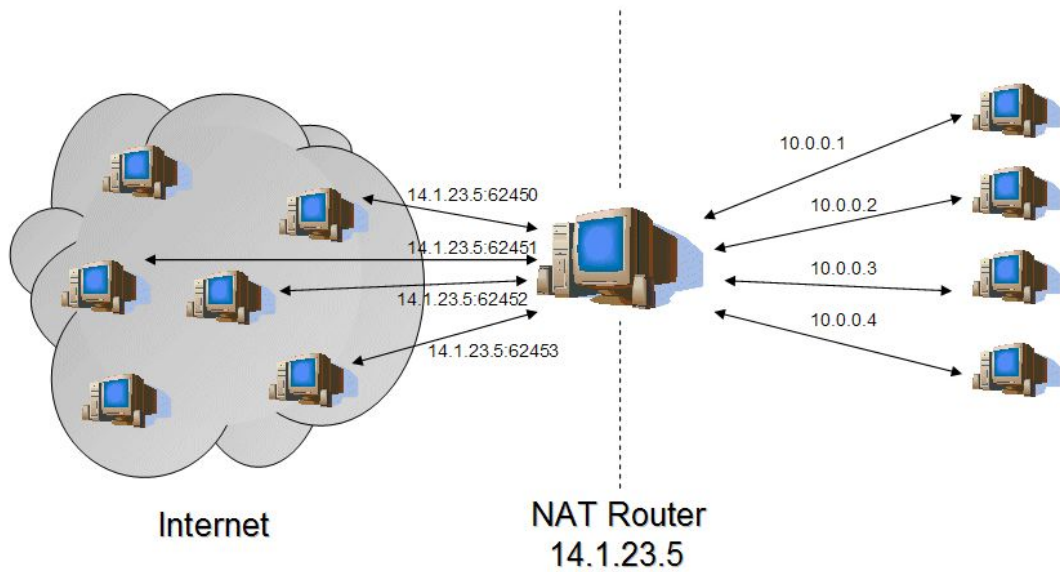
- Version - IP version number (set to 4 for IPv4)
- Internet Header Length (IHL) - Specifies the size of the header since a IPv4 header can be of varying length
- Type of Service (TOS) - As of RFC 2474 redefined to be differentiated services code point (DSCP) which is used by real time data streaming services like voice over IP (VoIP) and explicit congestion notification (ECN) which allows end-to-end notification of network congestion without dropping packets
- Total Length - Defines the entire packet size (header + data) in bytes. Min length is 20 bytes and max length is 65,535 bytes, although datagrams may be fragmented.
- Identification - Used for uniquely identifying the group of fragments of a single IP datagram
- X Flag - Reserved, must be zero
- DF Flag - If set, and fragmentation is required to route the packet, then the packet will be dropped. Usually occurs when packet destination doesn't have enough resources to handle incoming packet.
- MF Flag - If packet isn't fragmented, flag is clear. If packet is fragmented and datagram isn't the last fragment of the packet, the flag is set.
- Fragment Offset - Specifies the offset of a particular fragment relative to the beginning of the original unfragmented IP datagram

- Time To Live (TTL) - Limits the datagrams lifetime specified in seconds. In reality, this is actually the hop count which is decremented each time the datagram is routed. This helps to stop circular routing.
- Protocol - Defines the protocol used the data of the datagram
- Header Checksum - Used for to check for errors in the header. Router calculates checksum and compares to this value, discarding if they don't match.
- Source Address - Sender of the packet
- Destination Address - Receiver of the packet
- Options - specifies a number of options which are applicable for each datagram. As this project doesn't concern these it won't be discussed further.

### 2.1.2 Network Address Translator (NAT)

As a routing device, a NAT is responsible for remapping an IP address to another by altering the IP datagram packet header. NAT's have become extremely important in modern networking systems due to IPv4 address exhaustion, allowing a single public IP address to map to multiple private IP addresses. This is particularly useful in large corporations where only a limited public network connection is required, meaning that all private IP addresses (usually associated with a single machine) are mapped to the same public IP address. A NAT will make use of multiple connection ports to identify which packets are for which private IP address and then re-assign the packet header so the internal routers can forward the packet correctly. As can be seen by Figure (number) each internal address is mapped to via the port number associated with the external address. NAT's are generally implemented as part of a network firewall as they inspect the datagram packets for malicious data and sources.

Private IP Address	Public IP Address
10.0.0.1	14.1.23.5:62450
10.0.0.2	14.1.23.5:62451
10.0.0.3	14.1.23.5:62452
10.0.0.4	14.1.23.5:62453

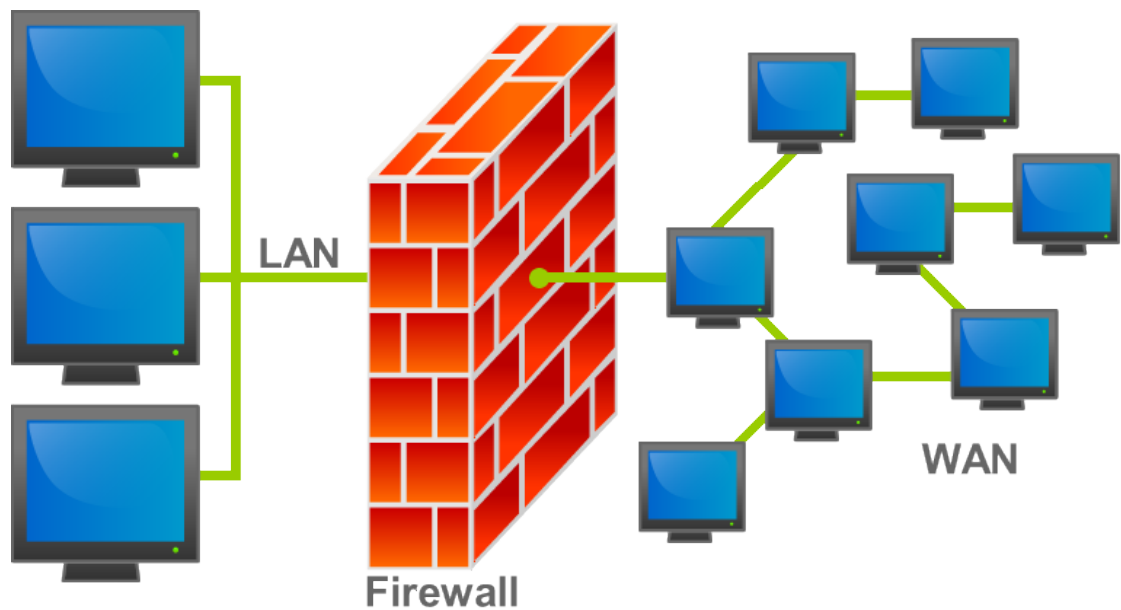


### 2.1.3 Firewall

Firewalls are generally the major applications which sits between the public and private network of a system. They provide packet filtering which controls which packets can enter the private network via establishing a set of rules which packets have to adhere to. Filtering can be based on a number attributes of the packet such as the source and destination IP address and port and the destination service. Firewalls can also offer a number of other useful features such as NAT's or dynamic host configuration protocol (DHCP). As well as providing protection on a network level, application layer firewalls exist which stop certain applications from sending or receiving a packet.

Within this project, the term 'firewall' is used for a network layer firewall which filters packets dependent on the source IP address. Any changes in this will be mentioned within the relevant sections.





## 2.2 Java Features

As the Java language and the JVM implementation is known by most, this section will focus on the more specific sections related to this project, instead of discussing the overall implementation.

### 2.2.1 JVM Networking

### 2.2.2 Java Native Interface (JNI)

Provided by the Java Software Development Kit (SDK), the JNI is a native programming interface that lets Java applications use libraries written in other languages. The JNI also includes the invocation API which allows a JVM to be embedded into native applications. This project and therefore this overview will only focus on Java code using C libraries via the JNI on a linux based system

In order to call native libraries from Java applications a number of steps have to be undertaken as shown below, which are described in more detail later:

1. Java code - load the shared library, declare the native method to be called and call the method
2. Compile Java code - compile the Java code into bytecode
3. Create C header file - the C header file will declare the native method signature and is created automatically via a terminal call
4. Write C code - write the corresponding C source file
5. Create shared library - create a shared library file (.so) from C code
6. Run Java program - run the Java program which calls the native code

```

1 public class Hello {
2     public native void sayHi(String who, int times);
3
4     static { System.loadLibrary("Hello"); }
5
6     public static void main(String[] args) {
7         Hello hello = new Hello();
8
9         hello.sayHi(args[0], Integer.parseInt(args[1]));
10    }
11 }

```

The code above shows a simple Java program which uses some native C code from a shared library. Line 4 indicates which shared library to load into the application, which is by default lib\*.so where \* indicates the name of the C file. Line 2 is the native method declaration which specifies the name of the method and the parameters which will be passed to the corresponding C method. Line 9 is where this native method is called.

```
1 $ javac Hello.java
```

Compiling the above Java class is shown above and should be self explanatory.

```
1 $ javah -jni Hello.java
```

In order to generate the C header file the command 'javah' is used with the flag 'jni' which tells Java that a header file is required which is for use with the JNI. It will then produce method signatures which correspond to the native method declared within Hello.java. The auto generated C header file is shown below.

```

1 /* DO NOT EDIT THIS FILE - it is machine generated */
2 #include <jni.h>
3 /* Header for class Hello */
4
5 #ifndef _Included_Hello
6 #define _Included_Hello
7 #ifdef __cplusplus
8 extern "C" {
9 #endif
10 /*
11  * Class:      Hello
12  * Method:     sayHi
13  * Signature:  (Ljava/lang/String;I)V
14  */
15 JNIEXPORT void JNICALL Java_Hello_sayHi
16     (JNIEnv *, jobject, jstring, jint);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif

```

```

1 #include <stdio.h>
2 #include "Hello.h"
3
4 JNIEXPORT void JNICALL Java_Hello_sayHi
5 (JNIEnv *env, jobject obj, jstring who, jint times) {
6     jint i;
7     jboolean iscopy;
8     const char *name;
9     name = (*env)->GetStringUTFChars(env, who, &iscopy);
10    for (i = 0; i < times; i++) {
11        printf("Hello %s\n", name);

```

explain parameters and mapping for jString to char\* etc

```
12     }  
13 }
```

The C source file implementation is shown above. Line 9 is the interesting line, where the code is retrieving the string stored at pointer 'who' from the Java environment and copying it into the pointer 'name' for use later on.

```
1 $ cc -c -I/System/Library/Frameworks/JavaVM.framework/Headers Hello.c  
2 $ cc -dynamiclib -o libhello.so Hello.o -framework JavaVM
```

The 2 commands above will first compile the C source code into an object file (.o), requiring a pointer to the jni.h file provided with the Java framework. The second line then generates the required shared object file (.so) which the Java application looks for.

```
1 $ java Hello Ashley 5  
2 Ashley  
3 Ashley  
4 Ashley  
5 Ashley  
6 Ashley
```

Running the Java application will give the above output.

As can be seen in the above code, there are 2 extra parameters in the method that weren't defined by me. The 'env' pointer is a structure that contains the interface to the JVM, therefore providing numerous functions required to interact with the JVM and the Java objects. Examples include converting native arrays to Java arrays and throwing exceptions, meaning that anything that can be done in Java can be done via this interface.

The 'obj' argument refers to the Java object which the native method had been declared within.

Although the JNI provides a very useful interface to interact with native library code, there are a number of issues that users should be wary of before progressing:

- The Java application that relies on the JNI loses its portability with the JVM as it relies on natively compiled code.
- Errors within the native code can potentially crash the JVM, with certain errors been very difficult to reproduce and debug.
- Anything instantiated with the native code won't be collected by the garbage collector with the JVM, so freeing memory should be a concern.
- If using the JNI on large scale, converting between Java objects and C structs can be difficult

### 2.2.3 ByteBuffer

## 2.3 I/O API's

### 2.3.1 DPDK

### 2.3.2 Netmap

## 3 Project Plan

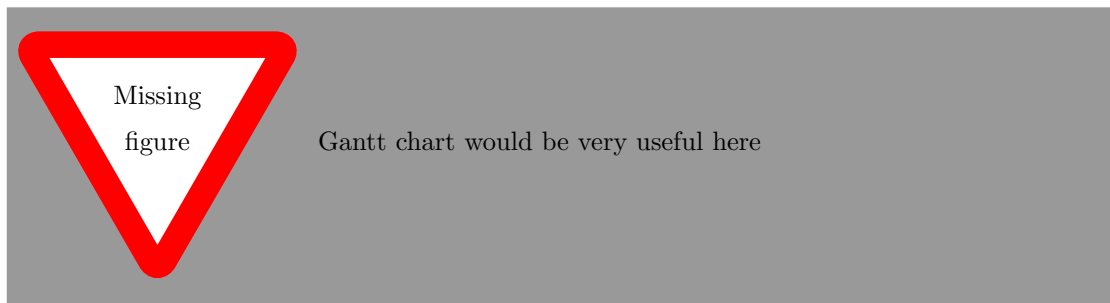
### 3.1 Time Plan

The given time plan outlined below provides me with a useful indication of how to measure the progress of the project, as well as dividing the whole project into smaller, more manageable sections which can be assessed individually. Some of the milestones have already been completed, while the others are expected to be finished within the given time period.

#### 3.1.1 Key Milestones

- Done - Background reading on DPDK & Netmap
- Done - Understand workings off Java Native Interface (JNI) and Java memory
- Done - Get DPDK (and associated programs) working on local machine (or VM)
- In Progress - Link DPDK library with shared object files for JNI
- Awaiting - Implement basic IP address firewall
- Awaiting - Implement basic NAT
- Awaiting - Find similar implementations of firewall and NAT in C/C++ and Java
- Awaiting - Set up 2 independent machines to use for testing purposes
- Awaiting - Run tests on firewall and NAT to produces performance measurements
- Awaiting - Analyse results

#### 3.1.2 Time Estimations



#### 3.1.3 Completed Milestones

As indicated, the background reading on the DPDK and Netmap libraries has been completed and the overview of the implementation has been assessed. The pros and cons has been outlined in the background section (give number). From this, I was able to get DPDK fully working on a virtual machine running locally on a personal machine, which allowed for a more in depth analysis of the implementation of DPDK.

I'm currently working on a way to generate shared object files from the C code used the link

DPDK with a Java application. Due to the complex build system which DPDK makes use of, there has been a number of issues which have arisen.

### 3.2 Fall Back Position

If a number of unforeseen issues arise which impact on the progress of the project, I've outlined a few of the key milestones which can be omitted in order to allow the project to still be evaluated:

- Implement basic NAT
- Find similar implementations of firewall and NAT in C/C++ and Java
- Run tests on firewall and NAT to produce performance measurements

Obviously, the omission of the above milestones will result in a less accurate evaluation of the overall project, but it's better than having nothing to evaluate in the end. As seen above, limiting the testing to one type of middlebox (only the firewall) can still give good results. This therefore reduces the need to look for an implementation for the NAT in Java and C/C++, while meaning less tests will have to be carried out. I feel that the remaining milestones are definitely key to the project in order for an adequate evaluation to be carried out.

### 3.3 Possible Extensions

If time allows, there are a number of possible extensions which can be undertaken to advance the project further forward and produce more data which can be evaluated later.

#### 3.3.1 Off Heap Java Memory

In order for DPDK to handle packets correctly, it has to insert these packets onto the outgoing 'ring'. To do this, the Java application has to place its serialised objects into memory which DPDK can access. Under the current implementation, the Java application will directly place the packets onto the 'ring' using the direct ByteBuffer allocate feature. The other option is to use off heap java memory which DPDK can access and place onto the 'ring' automatically. This off heap memory is slightly slower than on heap memory but much faster than disk memory, while off heap memory stores object serialised and therefore ready for transmission in packets.

Altering the application to make use of off heap Java memory and then testing the results could potentially offer a new and faster way of high performance packet processing in Java.

#### 3.3.2 Limit Testing

Another interesting extension could be to test the limitations of the NAT and firewall applications implemented within this project, and therefore limitations of the underlying implementation. In terms of the project, the limitations could be on the max packet throughput which it can handle without having to buffer the incoming packets.

## 4 Evaluation Plan

The sections below outline how the project will be evaluated in order to determine whether the initial objectives have been met and whether the final outcome can be deemed a success, even if it provides unexpected results.

### 4.1 Experiments & Outcomes

The main measure of success will be from the experiments which are to be carried out. Using the high performance Java packet I/O application which is currently been developed as part of this project, it will be compared to similar, already existing applications written in pure Java and in C/C++. This comparison will consist of 2 parts, firstly an application for a NAT and secondly a simple IP address firewall. These similar applications will then be benchmarked against each other and the results analysed.

Benchmarking will consist of setting up 2 independent computers, most likely standalone machines, in order to take advantage of the ability to overwrite the Intel network card drivers as required by the DPDK API. Testing can be carried out on the firewall and NAT implemented in the 3 different contexts, measuring the latency between the time sent and time received of the network packet. As the testing will be carried out on the same machines, linked to the same network there is unlikely to be much variation in the network latency. This means that the variation in the times will be from the system processing speeds. To account for minor variations in the computing performance of the system, numerous iterations of the same test will be carried out, then taking statistical averages will provide the best final results in order to analyse correctly.

Analysis of the results will be mainly carried out via the use of graphs which allow for easy comparison of results. The expected outcome is that the application developed from within this project is of similar speeds to applications coded directly in C/C++ and much faster than those coded in Java. This is mainly because the kernel will be bypassed, but the use of the Java Native Interface is expected to slow down certain aspects of the application. However, as long as speeds which coincide with those of the line rate are reached, this will be acceptable. There is the possibility that expected speeds are much faster than those that are actually measured. Although this isn't ideal, the project shouldn't be considered a failure as it will still provide very useful feedback as to where future investigations should be focussed on.

### 4.2 Functionality & Usability

Although the objectives didn't specify any requirement to make the final application highly usable, my personal preference would be to have a product that could be used by anybody in the future. This obviously requires well commented and documented code. As the previous evaluation was all quantitative measurements, this evaluation will consist of qualitative measurements mainly undertaken by myself.

The functionality and usability can also be checked by the project supervisor & co-supervisor to check if the original specifications were met. This can allow for an appraisal of the final application to be carried out which will be a good indication of the success of the project.

- Add references
- Add figure labels
- Do final read through
- Put through spell checker
- Other useful applications
- What font, size and line spacing?

## References

<http://www.neclab.eu/Projects/midcom.htm>

<http://windowsitpro.com/networking/what-types-network-address-translation-nat-exist>