6182944

*Todo list

Things to talk about

- Is app portable - it uses unints so why not?, packed structs a problem?
- spell check in excalibur

Imperial College London

Individual Project

Computing - BEng

Packet Processing in Java

Ashley Hemingway

Supervisor:
Peter Pietzuch

2nd Marker:
Wayne Luk

June 2015

Abstract

This project aims to implement a fast packet processing framework in Java for use within network middleboxes, which are required to inspect, transform and forward packets at line rate of speeds in excess of 10Gbps. Such applications are normally custom designed for the specific task and struggle with scaling and evolving technologies. Programmable middleboxes using medium to high level languages offers a better solution, considering the emerging cloud computing support which utilise dynamic virtual machine allocation.

In this report, a number of techniques for data sharing between native memory and Java are explored, performance tested and evaluated to determine the most suitable. This technique is then utilised to develop a fast packet processing framework written in Java which uses the existing Data Plane Development Kit (DPDK).

Middleware applications are then implemented using the same algorithm in Java and C and tested to compare performance in packet and data throughput of the application. Through this, further improvements are suggested and implemented to fully maximise memory and cpu usage.

final results - use numbers and conclusion

Acknowledgements

Contents

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 which are developed to process 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 the past, middleboxes have been purpose built hardware and software solutions which perform one task, but do this task extremely well. However, looking forward for network architecture, constant advancements such as 4G and the start of development for 5G for mobile communication has given rise to an ever increasing internet usage. This increase in data traffic provides a problem to the purpose built solutions which don't scale well to new technologies and increase demand. 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) which generally is 10Gbps (gigabits per second), but speeds can reach 40Gbps and even 100Gbps. Performance increases for networking hardware have vastly outshone those of the software, opening up a new area of research to find new techniques to overcome this hurdle. 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 or dropped packets. 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 been developed in Java and have succeeded in performing at similar speeds to C/C++ applications. These high performance projects typically utilise a distributed system, so it makes sense to have the middleboxes written in Java as well for easy scaling.

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, computer's kernel and the network interface controller/card (NIC). Once this challenge has been overcome, there are no reasons why programmable middleboxes written in none native languages such as Java can't exist within networking systems.

1.2 Contributions

This project contributions can be broadly split up into 5 categories:

- Research into existing Java technologies which can be exploited to dramatically increase throughput of data between a native language and one which runs on a virtual machine. Statistical analysis of these technologies and an evaluation of the suitability for the project.
- A new Java based fast packet processing framework named DPDK-Java. This is based upon an existing native framework but abstracts low level complications into an easy to use Java framework allowing quick implementation and deployment of applications.

- Multiple middlebox implementations using the fore-mentioned DPDK-Java framework focussing on the ease of use.
- Comparison testing between existing native middlebox implementations and the new Java based middleboxes. This focusses on analysing bottlenecks within the applications and suggested improvements
- Abstracting away key components and techniques of fast packet processing to give rise to more generic implementations using different languages.

1.3 Potential Applications

standard high volume (shv) servers - aim to deliver services much more rapidly and give network which is scaliable and flexible. aim for software to be run on shv servers which can be scaled and are flexible. should work on VM

relate back to quick boot up vm servers on cloud

1.4 Report Structure

Section 2 provides background information for a number of components which are part of the networking model and are useful to gain a full understanding of implementation details. The Java Virtual Machine and the Java Native Interface are explained in detail in sections 2.2.1 and 2.2.2 respectively and are needed to fully understand the techniques discussed. Finally, related works and applications are mentioned while briefly touching on performance testing which is used throughput the report.

As DPDK was chosen for the framework to be utilised, section 3 looks further into the implementation, most of which will be abstracted into the Java framework later on. Key concepts surrounding what enhances the performance over standard frameworks allow for techniques to be used later.

Section 5 primarily focusses on testing certain techniques and outlines the general design consideration needed to implement the new framework. The actual implementation is discussed in section 6, which also touches on the middlebox applications as well.

The new framework performance is fully compared to that of native applications in section 8. It looks through analysis of the applications performance in terms of memory and CPU usage and improves upon the initial implementation in a number of key ways to aim for matched performance.

Finally, a conclusion is drawn on the effective of the new framework in section 9 and touches upon future works surrounding this project. An abstraction of key concepts is drawn from the full report in the later stages to allow for generic implementations regardless of language choice.

cite ALL images

2 Background

This section focusses on a number of background concepts which will help to gain further understanding of networking practices and protocols which the solution must adhere to. It then gives an overview of the architecture of the Java langugage and explores on some of the key techniques which will be exploited. As this report focusses on performance testing throughout, this section will then cover techniques to test custom programs. It finally looks at related works within the research area and existing frameworks which could be exploited.

2.1 Network Components

intros for background section

2.1.1 Models

Generally there are 2 well known network models. The Open System Interconnection (OSI) model in Figure 1 represents an ideal to which all network communication should adhere to while the Transmission Control Protocol/Inter Protocol (TCP/IP) model represents reality in the world. The TCP/IP model combines multiple OSI layers into 1 of its layers simply because not all systems will go through these exact stages depending on the required application.

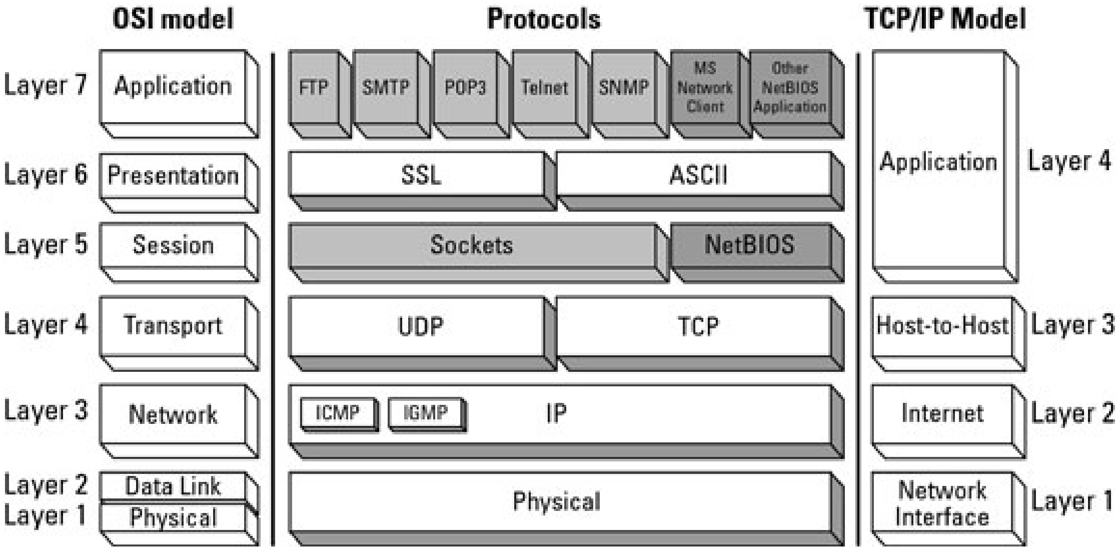


Figure 1: OSI vs TCP/IP Model

The TCP/IP application layer doesn't represent an actual application but instead its a set of protocols which provides services to applications. These services include HTTP, FTP, SMTP, POP3 and more. It acts as an interface for software to communicate between systems (e.g. client retrieving data from server via SMTP).

The transport layer is responsible for fragmenting the data into transmission control protocol (TCP) or user datagram protocol (UDP) packets, although other protocols can be used. This layer will attach its own TCP or UDP header to the data which contains information such as source and destination ports, sequence number and acknowledgement data.

The network/internet layer attaches a protocol header for packet addressing and routing. Most commonly this will be an IPv4 or IPv6 header . This layer only provides datagram networking functionality and it's up to the transport layer to handle the packets correctly.

ref this

The network interface or link layer will firstly attach its own ethernet header (or suitable protocol header) to the packets, along with an ethernet trailer. This header will specify the destination and source of the media access control (MAC) address which are specific to network interfaces. The next step is to put the packet onto the physical layer, which may be fibre optic, wireless or standard cables.

This will eventually build a packet of data which include the original raw data along with multiple headers for each layer of the model (Figure 2).

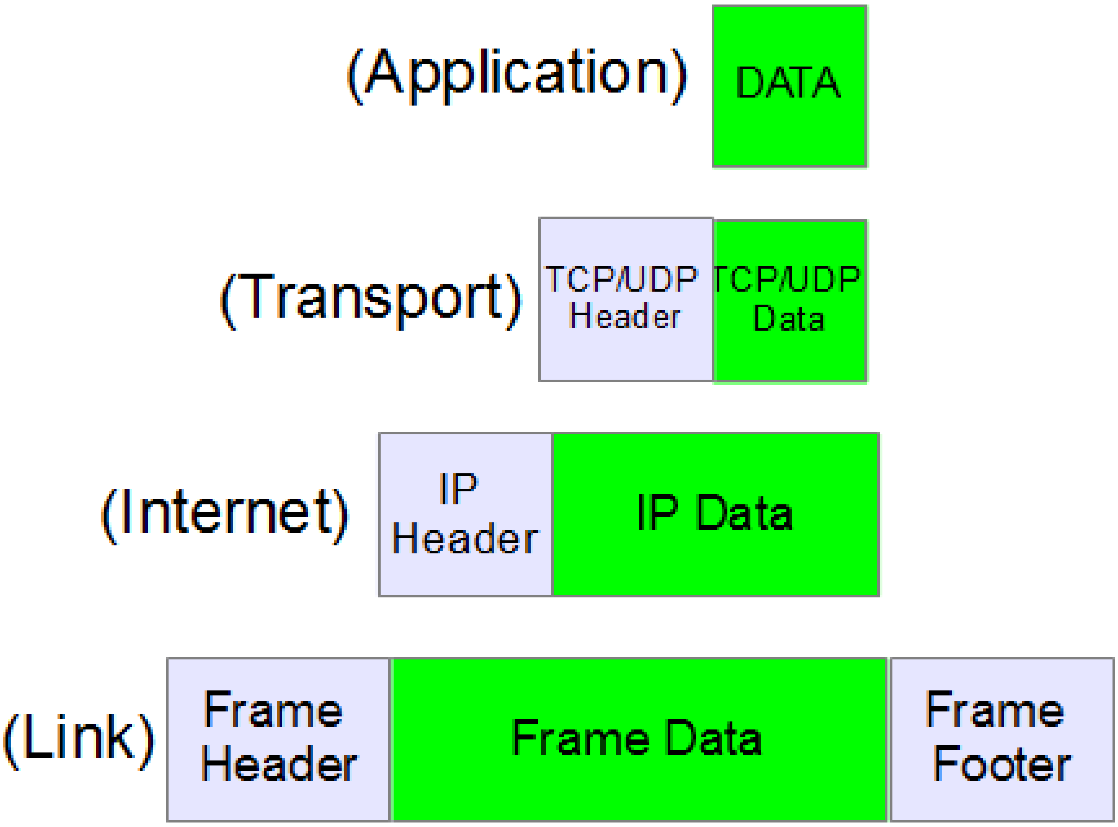


Figure 2: Network model building up of data headers

2.1.2 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. Data within the packets are generally input from the application layer, and headers are appended to the front of this data depending on the network level described in Section 2.1.1. Packets are routed to their destination based on a combination of an IP address and MAC address which corresponds to a specific computer located within the network, whether that is a public or private network. In this project we will only be concerned with the Internet Protocol (IP) and therefore IPv4 (Figure 3) and IPv6 (Figure 4) packet headers.

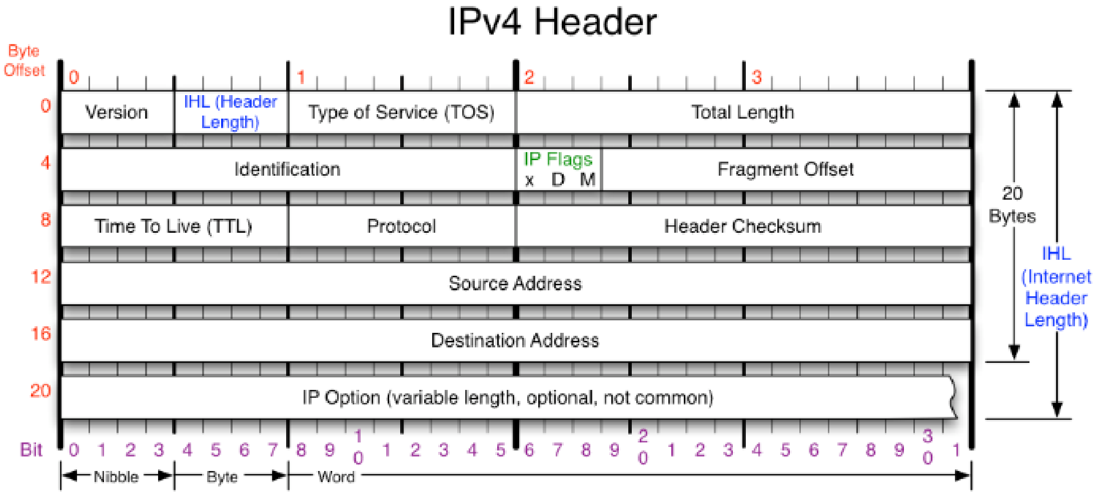


Figure 3: IPv4 Packet Header

- 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.

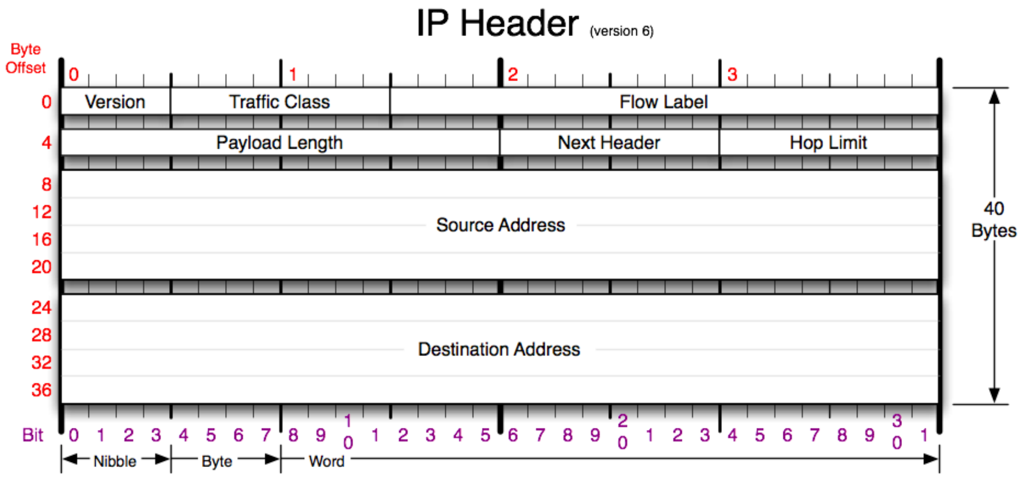


Figure 4: IPv6 Packet Header [17]

- Version - IP version number (set to 6 for IPv6)
- Traffic Class - Used for differentiated services to classify packets and ECN as described in IPv4.

ref this

- Flow Label - Used by real-time applications and when set to a non-zero value, it informs routers and switches that these packets should stay on the same path (if multiple paths are available). This is to ensure the packets arrive at destination in the correct order, although other methods for this are available.
- Payload Length - Length of payload following the IPv6 header, including any extension headers. Set to zero when using jumbo payloads for hop-by-hop extensions.

- Next Header - Specifies the transport layer protocol used by packet's payload.
- Hop Limit - Replacement of TTL from IPv4 and uses a hop value decreased by 1 on every hop. When value reaches 0 the packet is discarded.
- Source Address - IPv6 address of sending node
- Destination Address - IPv6 address of destination node(s).
- Extension Headers - IPv6 allows additional internet layer extension headers to be added after the standard IPv6 header. This is to allow more information for features such as fragmentation, authentication and routing information. The transport layer protocol header will then be addressed by this extension header.

2.1.3 Packet Handling

Once the kernel of the given operating system has received data to transmit from a given application, the data is then placed into a packet with the correct header and these packets are placed on a IP stack (Figure 5). Through a few intermediate steps the packets arrive at the driver queue, also known as the transmission queue. This queue is implemented as a ring buffer, therefore has a maximum capacity before it starts to overwrite packets which are still to be transmitted. As long as the queue isn't empty, the network interface card (NIC) will take packets from the queue and place them on the transmission medium. A similar process occurs when receiving packets, but in the opposite direction. For each NIC, there is a receive and transmit queue which are independent of each other allow communication to be bidirectional, although this depends on the kernel and its handling of events associated with packets.

is this section right

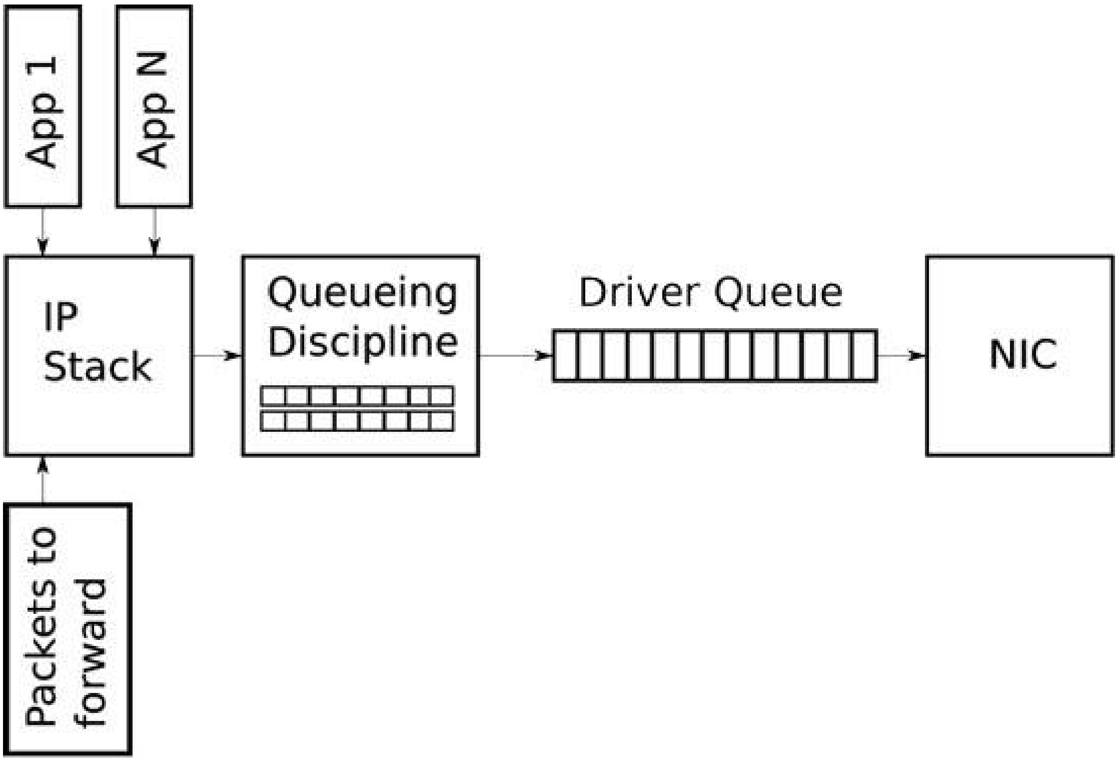


Figure 5: Linux packet handling [12]

2.1.4 Network Interface Controller (NIC)

Also known as a Network Interface Card, they provide the ability for a computer to connect to a network through a variety of mediums such as wireless, ethernet and fibre optics. They provide both the data link and physical layer of the network model (Figure 1), allowing a protocol stack to communicate with other computers on the local area network (LAN) or over a wider area via the IP protocol using IP addresses.

NICs can run at speeds of up to 100Gbps but more commonly run at 10Gbps for servers and 1Gbps for standard computers. The kernel or other applications retreive packets via the NIC by polling or itterupts. Polling is where the kernel or application will periodically check the NIC for received packets while the use of interupts allows the NIC to tell the kernel or application that it has received packets. Generally NICs provide the ability for 1 or more recieve and transmit queues to be assigned per port, allowing for increased performance by assigning queues to different threads.

2.1.5 Middleboxes

Middleboxes are a device within a network that inspect, alter and forwats packers them depending on certain rules and the intended functionaility of the middlebox. A few different types are described below and can be combined into 1 single application:

Firewall Firewalls (Figure 6) are generally the major applications which sit 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 (DCHP) to allow dynamic assignment of IP addresses within a network. As well as providing protection on a network level, application layer firewalls exist which stop certain applications from sending or receiving a packet.

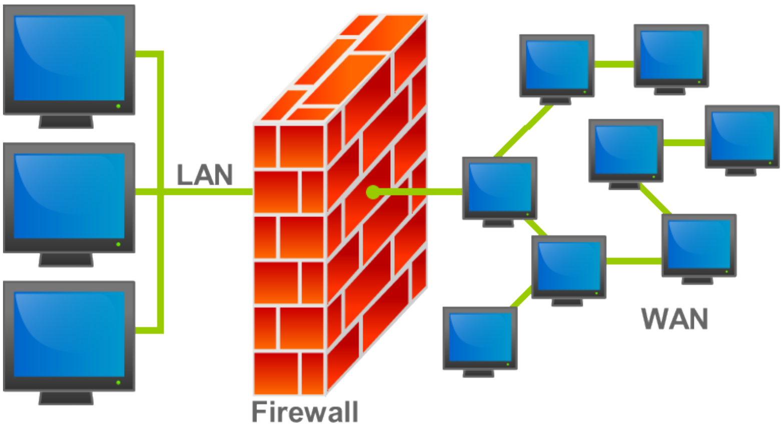


Figure 6: Firewall intercepting packets as a security measure [21]

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 Table 1 and Figure 7, 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.

complete this with help from links

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

Table 1: Example of public IP address and ports mapping to private IP address

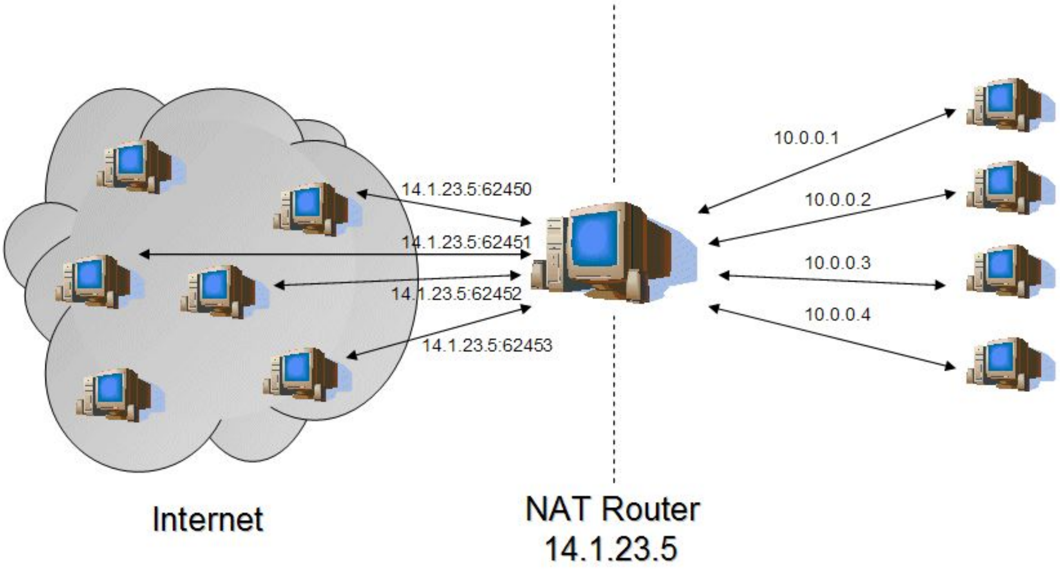


Figure 7: NAT translating public IP addresses into private IP addresses [22]

any more middle-boxes - packet cap?

2.2 Java

2.2.1 JVM

The Java Virtual Machine is an abstract computer that allows Java programs to run on any computer without dependant compilation. This works by all Java source code been compiled down into Java byte code, which is interpreted by the JVM's just in time (JIT) compiler to machine code. However, it does require each computer to have the Java framework installed which is dependant on the OS and architecture. It provides an appealing coding language due to the vast support, frameworks and code optimisations available such as garbage collections and multithreading. Figure 8 shows the basic JVM architecture with the relevant sections explained in the list below.

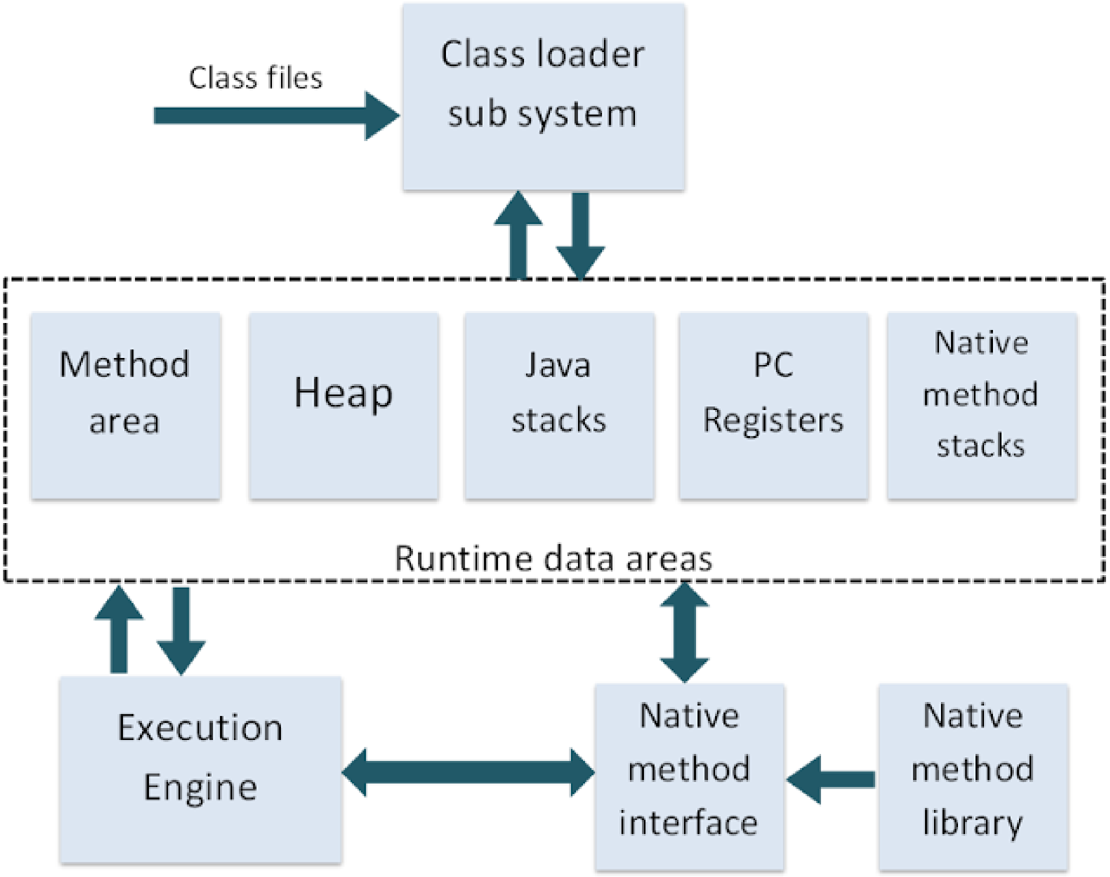


Figure 8: Java Virtual Machine interface [6]

- Class loader sub system - Loads .class files into memory, verifies byte code instructions and allocates memory required for the program
- Method area - stores class code and method code
- Heap - New objects are created on the heap
- Stack - Where the methods are executed and contains frames where each frame executes a separate method
- PC registers - Program counter registers store memory address of the instruction to be executed by the micro processor
- Native method stack - Where the native methods are executed.
- Native method interface - A program that connects the native method libraries with the JVM
- Native method library - holds the native libraries information

- Execution engine - Contains the interpreter and (JIT) compiler. JVM decides which parts to be interpreted and when to use JIT compiler.

Typically, any network communication from a Java application occurs via the JVM and through the operating system. This is because the JVM is still technically an application running on top of the OS and therefore doesn't have any superuser access rights. Any network operation results in a kernel system call, which is then put into a priority queue in order to be executed. This is one of the main reasons why network calls through the JVM and kernel can be seen as 'slow', in relative speeds compared to network line rate speeds.

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

The Java Framework provides a number of typedef's used to have equivalent data types between Java and C, such as jchar, jint and jfloat, in order to improve portability. For use with objects, classes and arrays, Java also provides others such as jclass, jobject and jarray so interactions with Java specific characteristics can be undertaken from native code run within the JVM.

```
public class Jni {

  int x = 5;

  public int getX() {
    return x;
  }

  static { System.loadLibrary("jni"); }

  public static native void objectCopy(Jni o);
12
13    public static void main(String args[]) {
14      Jni jni = new Jni();
15      objectCopy(jni);
16    }
17
18  }
```

Code 1: Basic Java class showing native method declaration and calling with shared library loading

Code 1 shows a simple Java program which uses some native C code from a shared library. Line 9 indicates which shared library to load into the application, which is by default lib*.so where * indicates the name of the library identifier. Line 11 is the native method declaration which specifies the name of the method and the parameters which will be passed to the corresponding C method. In this case, the method name is 'objectCopy' and a 'Jni' object is passed as a parameter. Line 15 is where this native method is called.

```
$ javac Jni.java
```

Code 2: Compiling basic Java program

Code 2, run from a terminal, compiles the Java class and create a class file which can be executed.

```
$ javah -jni Jni
```

Code 3: Generating C header file

In order to generate the C header file the command 'javah' (Code 3) 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 Jni.java. The auto generated C header file is shown in Code 4.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Jni */

#ifndef _Included_Jni
#define _Included_Jni
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Jni
 * Method:    objectPrint
 * Signature: (LJni;)V
 */
JNIEXPORT void JNICALL Java_Jni_objectPrint
  (JNIEnv *, jclass , jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Code 4: Auto-generated C header file

```
1 #include "Jni.h"
2
3 JNIEXPORT void JNICALL Java_Jni_objectPrint(JNIEnv *env, jclass class, jobject obj) {
4   jclass cls = (*env)->FindClass(env, "Jni");
5   jmethodID method = (*env)->GetMethodID(env, cls, "getX", "()I");
6   int i = (*env)->CallIntMethod(env, obj, method);
7   printf("Object x value is %i\n", i);
8 }
```

Code 5: C source file corresponding to auto-generated header file

The C source file implementation is in Code 5. The method signature on line 3 isn't as first declared in the Java source file. The 'env' variable is used to access functions to interact with objects, classes and other Java features. It points to a structure containing all JNI function pointers. Furthermore, the method invocation receives the class from which it was called since it was a static method. If the method had been per instance, this variable would be of type 'jobject'.

Line 4 shows how to find a class identifier by using the class name. In this example, the variables 'class' and 'cls' would actually be equal. In order to call an objects' method, a method id is required as a pointer to this method. Line 5 shows the retrieval of this method id, whose parameters are the jclass variable, the method name and the return type, in this case an integer (represented by an I). Then the method can be called on the object via one of the numerous helper methods (line 6) which differ depending on the return type and static or non-static context.

```
$ gcc -shared -fpic -o libjni.so -I/usr/java/include -I/usr/java/include/linux jni.c
```

Code 6: Terminal commands to generate shared library file (.so)

The command in Code 6 will create the shared object file called 'libjni.so' from the source file 'jni.c'. This output file is what the Java program uses to find the native code when called. It requires pointers to the location of the Java Framework provided jni.h header file.

```
$ java -Djava.library.path="." Jni
Object x value is 5
```

Code 7: Output from running Java application calling native C methods

Running the Java application, pointing to the location where Java can find the shared library (if not in a standard location) will output the above in Code 7.

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 and slow

2.2.3 Current Java Networking Methods

http://haumacher.de/publ/parallel/p086.pdf http://www.javacoffeebreak.com/articles/javarmi/javarmi.html For high performance computing in Java, a number of existing programming options are available in order for applications to communicate over a network. These can be classified as: (1) Java sockets; and (2) Remote Method Invocation (RMI); (3) shared memory programming. As will be discussed, none of these are capable of truly high performance networking, especially at line rate speeds.

Java Sockets Java sockets are the standard low level communication for applications as most networking protocols have socket implementations. They allow for streams of data to be sent between applications as a socket is one end point for a 2 way communication link, meaning that data can be read from and written to a socket in order to transfer data. Even though sockets are a viable option for networking, both of the Java socket implementations (IO sockets & NIO (new I/O) sockets) are inefficient over high speed networks [27] and therefore lack the performance that is required. As discussed previously, the poor performance is due to the JVM interacting with network cards via the OS kernel.

Remote Method Invocation (RMI) Remote Method Invocation (RMI) is a protocol developed by Java which allows an object running in a JVM to invoke methods on another object running on a different JVM. Although this method provides a relatively easy interface for which JVM's can communicate, its major drawback relates to the speed. Since RMI uses Java sockets as its basic level communication method, it faces the same performance issues as mentioned in section **??**.

Shared Memory Programming Shared memory programming provides high performance JVM interaction due to Java's multithread and parallel programming support. This allows different JVM's to communicate via objects within memory which is shared between the JVM's. However, this technique requires the JVM's to be on the same shared memory system, which is a major drawback for distributed systems as scalability options decrease.

Even though these 3 techniques allow for communication between JVM's and other applications, the major issue is that incoming packets are still handled by the kernel and then passed onto the corresponding JVM. This means that packets are destined for certain applications, meaning that generic packets can't be intercepted and checked, which is a requirement for common middlebox software.

2.3 Related Works

2.3.1 jVerbs

Ultra-low latency for Java applications has been partially solved by the jVerbs [25] framework. Using remote direct memory access (RDMA), jVerbs provides an interface for which Java applications can communicate, mainly useful within large scale data centre applications.

RDMA is a technology that allows computers within a network to transfer data between each other via direct memory operations, without involving the processor, cache or operating system of either communicating computer. RDMA implements a transport protocol directly within the network interface card (NIC), allowing for zero copy networking, which allows a computer to read from another computer and write to its own direct main memory without intermediate copies. High throughput and performance is a feature of RDMA due to the lack of kernel involvement, but the major downside is that it requires specific hardware which supports the RDMA protocol, while also requiring the need for specific computer connections set up by sockets.

As jVerbs takes advantage of mapping the network device directly into the JVM, bypassing both the JVM and operating system (Figure 9), it can significantly reduce the latency. In order to have low level interaction with the NIC, jVerbs has a very thin layer of JNI calls which can increase the overhead slightly. However, jVerbs is flawed, mainly because it requires specific hardware to run on, firstly limited by the RDMA protocol reliant hardware and further by the required RDMA wrappers which are implemented by the creators. Also, it can only be used for specific computer to computer connection and not generally packet inspection.

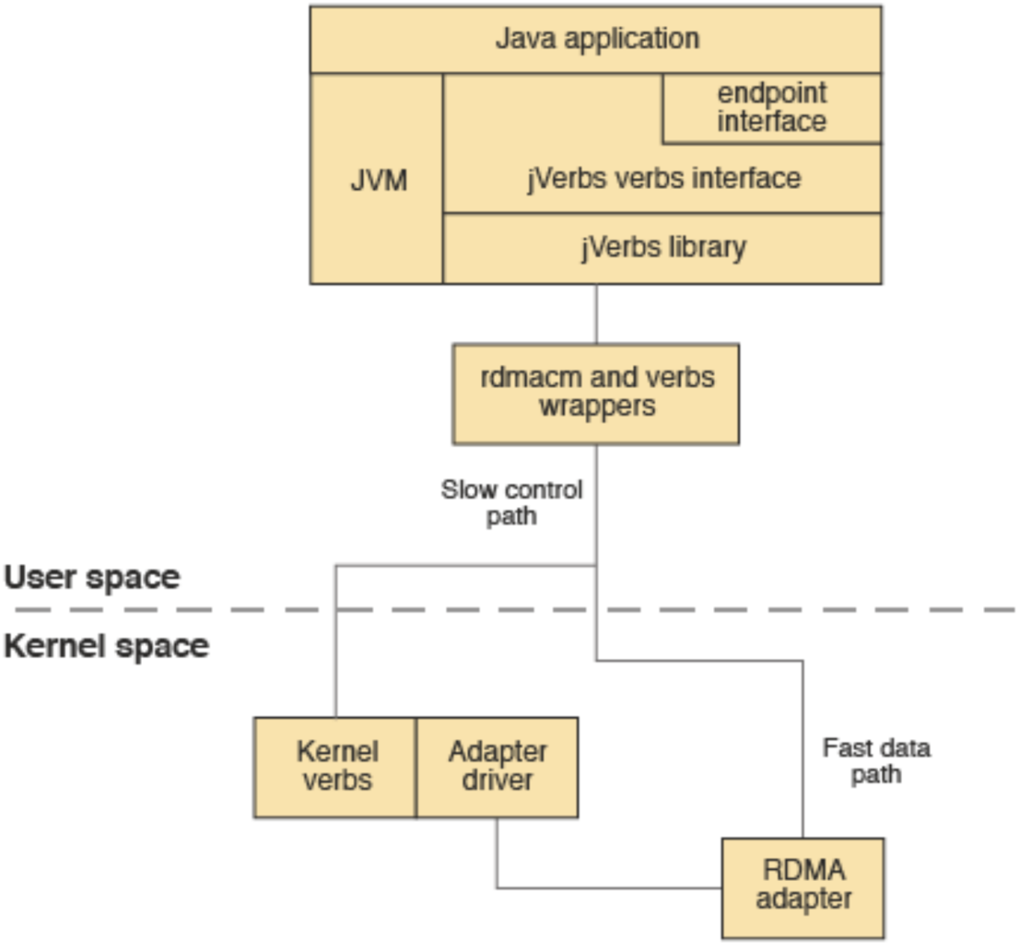


Figure 9: jVerbs architecture - shows how the framework bypasses the kernel and JVM [4]

jVerbs provides a useful example framework which re-emphasises that packet processing in Java is very possible with low latency, while assisting in certain implementation and design choices which can be analysed in more detail.

2.3.2 Packet Shader

The CPU can be a potential bottleneck in high speed software based routers and Packet Shader which aims to eliminate this. It takes advantage of graphic processing units (GPU) which offer greater execution power than typical CPUs due to their ability to run thousands of threads in parallel. By using the GPU and running packet processing threads on the GPU packet shader can increase performance to routers up to 40Gbps. However, the research behind this project has been suspended in favour of DPDK (2.3.3).

2.3.3 Data Plane Development Kit (DPDK)

Data Plane Development Kit (DPDK) [2] is a set of libraries and drivers which enables fast packet processing reaching speeds of 80 million packets per second on certain system set ups. Since DPDK is developed by Intel, it only supports Intel x86 CPU's and certain network interface controllers (NIC). DPDK binds the NIC's to new drivers meaning that the operating system doesn't recognise the network cards and therefore the network stack associated with the ports don't work. It make use of drivers run in user space allowing it to interact with certain memory locations without permission from the kernel or even involving it in any way.

DPDK makes use of an environment abstraction layer (EAL) which hides the environmental specifics and provides a standard interface which any application can interact with. Due to this, if the system changes in any way, the DPDK library needs to be re-compiled with other features been re-enabled in order to allow applications to run correctly again.

In order to use the DPDK libraries for the intended purpose, data packets have to be written into the correct buffer location so they are inserted onto the network. I similar approach is used when receiving packets on the incoming buffer ring, but instead of the system using interrupts to acknowledge the arrival of a new packet, which is performance costly, it constantly polls the buffer space to check for new packets. DPDK also allows for multiple queues per NIC and can handle multiple NIC's per system, therefore scalability is a major bonus of the libraries.

DPDK is very well documented on a number of levels. Firstly there is a online API which gives in depth details about what the methods, constants and structs do. There are a number of well written guides which give step-by-step details of how to install, set-up and use DPDK on various platforms and finally, there are many sample programs included with the build which give understanding of how the overall library works. DPDK is discussed in much more detail in Section 3.

2.4 Performance Testing Techniques

Benchmarking is a process of testing hardware, individual components or full end to end systems to determine the performance of the application or hardware . Generally, benchmarking should be repeatable under numerous iterations without only minor variations in performance results. This is firstly to allow minor changes to be made to the application/component with re-runs of the benchmark showing the performance changes. Secondly, it allows accurate comparisons to be drawn between similar software or hardware with different implementations in order to derive a better product.

Examples of hardware comparisons

better word for derive

why I need benchmarking

2.4.1 Programming Languages

It is well known that different programming languages can provide a radical change in execution for a given program. However, direct comparisons can't truly be trusted as certain languages are suited for for specific tasks and finding a benchmarking program to incorporate this is problematic. Other factors can be introduced when deciding on the optimisation level and the compiler

show example of optimising java and how different it looks, also different depending on architecture of JIT used.

Numerous attempts have been made to compare languages, most noticeably the 'Benchmark Game' and Google.

More here

2.4.1.1 Loop Recognition

Google inducted their own experiment on this problem, testing only C++, Java, Scala and Go on the loop recognition algorithm . Implementations made use of standard looping constructs and memory allocation schemes without the use of non-orthodox optimisation techniques. Selected results of this are shown below:

get paper

explain difference with GC etc

sentence needed

Benchmark	Time [sec]	Facter
c++	23	1.0x
Java 64-bit	134	5.8x
Java 32-bit	290	12.8x
Java 32-bit GC	106	4.6x
Scala	82	3.6x
Go 6g	161	7.0x

Table 2: Results from Loop Recognition benchmarking

2.4.1.2 Benchmark Game

The Benchmark Game is an online community which aims to find the best programming language by using multiple benchmarking algorithms running on different architecture configurations to determine the outcome. Again, even this community regard the best benchmark application to be your application. A few selected results are shown below between Java and C (those used in this report) for a few different benchmarks.

get data

2.4.1.3 Using Economics

About economics paper

2.4.1.4 Which is better?

2.4.2 Intra-Language Techniques

2.4.3 Applications

3 DPDK

This section will go into much more detail about the Data Plane Development Kit (DPDK), focussing on the basic concepts used by the fast packet processing framework for use within custom applications. DPDK does have a number of more advanced features which can be exploited but aren't discussed in this report.

ref manual

3.1 Environment Abstraction Layer (EAL)

The EAL abstracts the specific environment from the user and provides a constant interface in order for applications to be ported to other environments without problems. The EAL compilation may change depending on the architecture (32-bit or 64-bit), operating system, compilers in use or network interface cards. This is done via the creation of a set of libraries that are environment specific and are responsible for the low-level operations gaining access to hardware and memory resources.

On the start-up of an application, the EAL is responsible for finding PCI information about devices and addresses via the igb_uio kernel module, performing physical memory allocation using huge pages (section 3.3) and starting user-level threads for the logical cores (section 3.2) (lcores) of the application.

3.2 Logical Cores

Also know as lcores within DPDK, logical cores shouldn't be confused with processor cores. Lcores are threads which allow different applications functions to be run within different threads. This can allow different lcores to have access to different ports and queues while processing packets as well.

Within DPDK, lcores are implemented with POSIX threads (on linux) and make use of processor affinity (CPU pinning). This allows lcores to be only run on certain processing cores which reduces context switching and cache memory swapping and therefore increasing overall performance. However, this only works of the number of lcores is equal or less than the number of available processing cores so the number of lcores which are allowed to be initiated are limited. Generally a processor can only run 1 thread per core, but hyper-threaded CPUs can work on 2. Hyper-threading gives the core extra registers and execution units two allow the core to store the state of 2 threads and work on them simultaneously.

Machines with multiple sockets with multiple processing units add extra complication to lcores. As a core only exists on one socket, any memory associated with this socket (section 3.5.4) is only fastly accessible from that socket. This means that any lcore accessing a port must be on the same socket or else performance is greatly reduced. This needs to be manually assigned on the start-up of the application.

3.3 Huge Pages

Huge pages are a way of increasing performance when dealing with large amounts of memory. Normally, memory is managed in 4096 byte (4KB) blocks known as pages, which are listed within the CPU memory management unit (MMU). However, if a system uses a large amount of memory, increasing the number of standard pages is expensive on the CPU as the MMU can only handle thousands of pages and not millions.

The solution is to increase the page size from 4KB to 2MB or 1GB (if supported) which keeps the number of pages references small in the MMU but increases the overall memory for the system. Huge pages should be assigned at boot time for better overall management, but can be manual assigned on initial boot up if required.

DPDK makes uses of huge pages simply for increased performance due to the large amount of packet throughput in memory. This is even the case if the system memory size is relatively small and the application isn't processing an extreme number of packets.

3.4 Ring Buffer

A ring buffer is used by DPDK to manage transmit and receive queues for each network port on the system. This fixed sized first-in-first-out queue allows multiple objects to be enqueued and dequeued from the ring at the same time from any number of consumers or producers. A major disadvantage of this is that once the ring is full (more of a concern in receive queues), it allows no more objects can be added to the ring, resulting in dropped packets or packet caches. It is therefore imperative that applications can processes packets at the required rate.

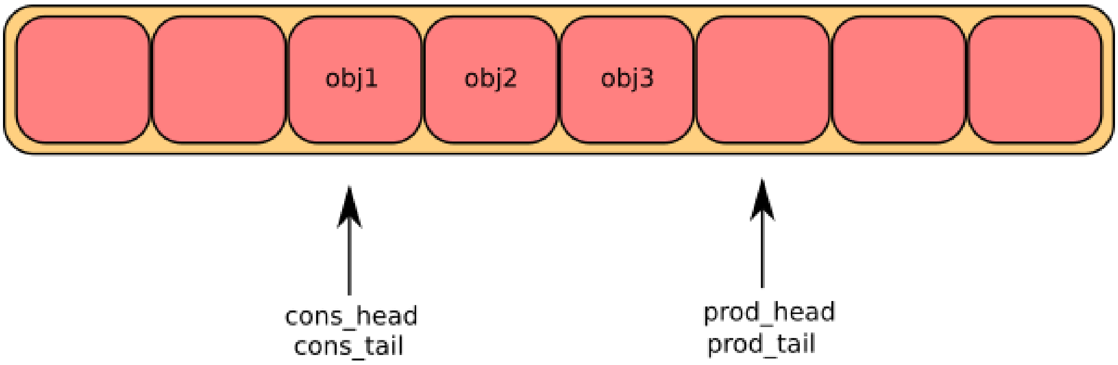


Figure 10: Ring Buffer

cite this

3.5 Memory Usage

DPDK tries to regulate memory usage in order to be more efficient and therefore handles its own memory management. However, it does allow application to allocate memory blocks for specific port queues and for data sharing between lcores. The major memory techniques used are described below.

3.5.1 Allocation

Since DPDK make use of huge pages, it provides its own memory allocation (malloc) library to allow memory blocks of any size, which also improves the portability of applications. However, even the DPDK malloc library is slow compared to pool memory access due to synchronisation constraints. Generally this library should only be used at initialisation time but does support NUMA for specific socket memory access and memory alignment.

3.5.2 Pools

Memory pools allow for fixed size allocation of memory which uses a ring to store fixed sized objects. These a almost guarenteed to be used for message buffer storage for receive and transmit queues for ports. They are initialised with a number of paremeters to increas performance such as cache sizes and NUMA socked identification as well as a name identify.

Pools offer increased performance over the standard memory allocation since the object padding is optimised so each object starts on a different memory channel and rank. Furthermore, a per core cache can be enabled at initialisation. This offers performance advantages, as without caching per core locks are required for every pool access. Caches offer cores lock free access to data, while bulk requested can be carried out on the pool to reduce locking on the pool.

3.5.3 Message Buffers

Message Buffers (mbufs) are stored within a specified memory pool and are used to carry data between different processes within the application and are primarily used for carrying network packets through the application. Mbufs also contain metadata about the information it is carrying which includes the data length, message type and offsets for the start of the data. The headroom shown in figure 11 shows empty bytes between the metadata and start of the data which allows the data to be memory aligned for quicket access. Mbufs can also be chained together to allow for longer data, more commonly jumbo packets.

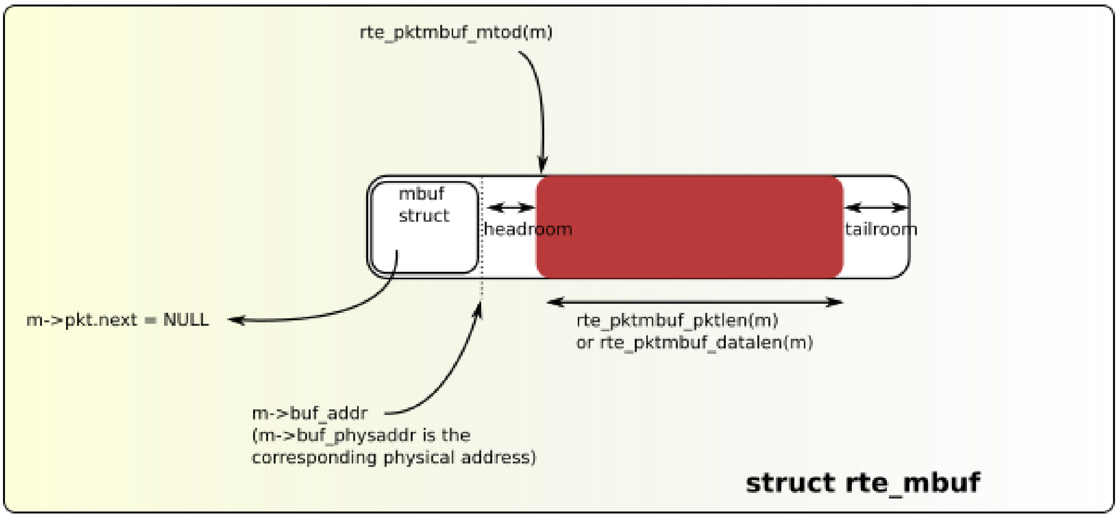


Figure 11: Message Buffer

cite this

3.5.4 NUMA

DPDK can make use of non-uniform memory access (NUMA) if the system supports it. NUMA is a method for speeding up memory access when multiple processors are trying to access the same memory and therefore reduces the processors waiting time. Each processor will receive its own bank of memory which is faster to access as it doesn't have to wait. As applications become more extensive, processors may need to share memory, which is possible via moving the data between memory banks. This somewhat negates the need for NUMA, but NUMA can be very effective depending on the application. DPDK can make extensive use of NUMA as each logical core is generally responsible for its own queues, and since queues can't be shared between logical cores (although dedicated ring buffers can), data sharing is rare.

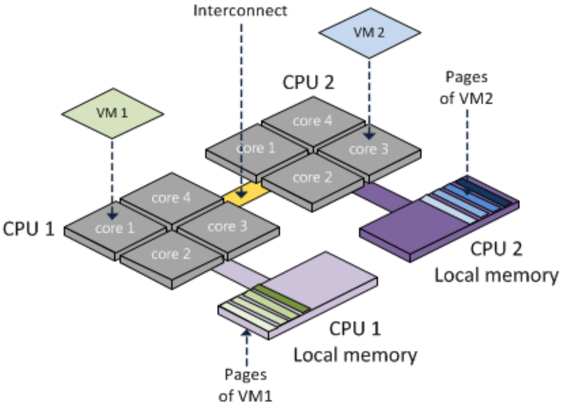


Figure 12: NUMA

cite this

3.6 Poll Mode Driver (PMD)

A Poll Mode Driver (PMD) is a user space device which allows configuration of network interface cards/controllers and their associated queues. Consequently this means that the network stack typicallly associated with the port isn't used. To solve this either a custom network stack needs to be implementated or an open source stack like IwIP can be layered onto of DPDK. PMD's work without interrupts which allow for quicker recieving, processing and transmitting of packets and are therefore lock free. Generally anything which can be achieved by interrupts can also be achived via using rings and continuous polling of the rings. This means that 2 lcores running in parallel can't receive from the same queue on the same port. However, 2 parallel cores can receive from the same port on different queues.

ref this

There a number of consideration for application design depending on the hardware in use. Optimal performance can be achieved by carefully considering the hardware properties such as caches, bus speed and bandwidth along side the software design choices. For example, NIC's are more efficient at transmitting multiple packets in a burst rather than individually but consequently overall throughput may be reduced.

3.7 Models

DPDK supports 2 methods for packet processing applications:

Pipe-line This is an asychronous model where lcores a designated to perform certain tasks. Generally certain lcores will receive packets via the PMD api and simply pass those packets to other lcores via the use of a ring. These other lcores will then process the packets depending the requirements and then either forward the packets to the PMD for transmitting or pass them onto other lcore via a ring.

Run-to-completion This is a synchronous model where each lcore will retrieve the packets, process them individually and then output them for transmission. Each lcore should be assigned its own receive and transmit queue on a given port in order to negate the need for locks. This report will focus on the run-to-completion model.

4 Is Java really slow?

Before any implementation or specific design considerations were undertaken, an evaluation of the performance of C, Java and the Java Native Interface (JNI) was carried out. Although data from existing articles and websites could be used for Java and C, there was no existing direct comparisons between them and the JNI, therefore custom tests were carried out.

The JNI is inherently seen as a bottleneck of an application (even after its vast update in Java 7).

article on this

reasons why JNI is slow

As this application would be forced to use the JNI, numeric values of its performance was helpful to evaluate the bridge in speed required to be overcome.

4.1 Benchmarking Algorithm

As discussed previously , there are always advantages and disadvantages of any algorithm used for benchmarking. In order to minimise the disadvantages, an algorithm was used which tried to mimic the procedures which would be used in the real application, just without the complications. Algorithm 1 shows that the program basically creates 100,000 packets individually and populates their fields with random data, which is then processed and returne in the ’result’ field. This simulates retrieving low-level packet data, interpreting and acting upon the data, and then setting data within the raw packet.

ref this

Algorithm 1 Language Benchmark Algorithm

```
function MAIN
    for i = 1 to 100000 do
        $p \leftarrow$ Initialise Packet
        POPPACKET(p)
        PROPACKET(p)

function POPPACKET(Packet p)                         ▷ Set data in a packet
    $p.a \leftarrow randomInt()$
    $p.b \leftarrow randomInt()$
    $p.c \leftarrow randomInt()$
    $p.d \leftarrow randomInt()$
    $p.e \leftarrow randomInt()$

function PROPACKET(Packet p)                         ▷ Process a packet
    $res \leftarrow p.a * p.b * p.c * p.d * p.e$
    $p.result \leftarrow res$
```

For the JNI version, the same algorithm was used, however, the *PopPacket* method was carried out on the native side to simulate retrieving raw packet data. The *ProPacket* method was executed on the Java side with the result been passed back to the native side to be entered back into the packet structure.

Timing within the algorithm for all variations was carried out between each iteration. This firstly eliminated any initial start-up time associated with the application which is common with the JVM. Secondly, any calls for time stamps to the system would be miminised as 100,000 iterations would occur in-between them.

4.2 Results

Each language had the algorithm run 1,000 times in order to minimise any variations due to external factors. Figures show that C was considerably quicker than Java, while Java using the JNI was extremely slow.

ref this

expand on this

4.3 Further Investigation

Due to the very poor performance of the JNI compared to other languages, further investigations were carried out to find more specific results surrounding the JNI.

Is this relevant

5 Design Considerations

5.1 Data Sharing

The proposed application will be sharing data between the DPDK code written in C (low level) and the Java (medium level) side used for the highly abstracted part of the application. This requires a large amount of data, most noticeably packets, to be transferred between 'sides' in a small amount of time. This section will explore possible techniques for this data sharing, discuss the advantages and disadvantages and finally reason about a decision.

5.1.1 Memory Usage

Programs written in most languages make use of memory on heap, off heap and in a stack. Stack memory is used for local, short term variables which this report isn't concerned with. Heap memory is a block of memory associated with a program for it to store long lasting data on which can be shared among multiple threads. Some languages require allocation of data on heap memory to be freed.

Within Java, heap memory is subjected to a garbage collector which scans the heap when the JVM is running out of memory and dereferences any objects which aren't in use any more while compact the heap by moving references around. Although memory management is taken away from the user, the garbage collector can slow down an application as it traverses memory at random intervals. Off heap memory isn't subjected to the garbage collector which allows fast processing applications to take advantage of this while memory can be referenced directly.

The application needs to assign memory efficiently to have any chance of fast performance. Figure 13 shows the problem which needs to be overcome using memory management with different techniques. The data will follow a pipeline flow, starting on the NIC receive queue and been retrieved as a Mbuf struct containing all of the packet data. From there a 'technique' needs to be implemented to efficiently map the data in the struct to a Java packet object. The data will then be processed depending on the application requirements and then will be forwarded via been put on the transmit queue of the NIC. Between these stages, the data needs to be mapped back from the Java object to the original struct using a 'technique'

There are a number of possible 'techniques' available in order pass data efficiently between native structs and Java objects. The implementations are discussed below along with the advantages and disadvantages.

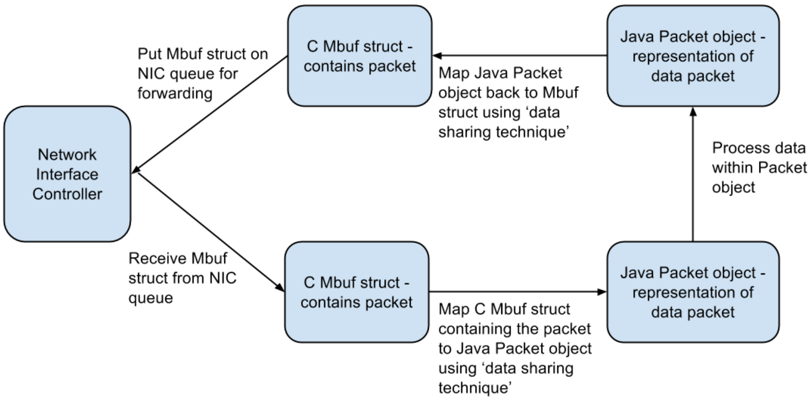

Put Mbuf struct on NIC queue for forwarding
C Mbuf struct - contains packet
Java Packet object - representation of data packet
Map Java Packet object back to Mbuf struct using 'data sharing technique'
Network Interface Controller
Process data within Packet object
Receive Mbuf struct from NIC queue
C Mbuf struct - contains packet
Map C Mbuf struct containing the packet to Java Packet object using 'data sharing technique'
Java Packet object - representation of data packet

Figure 13: Server

5.1.2 Objects and JNI

By far the simplest technique available is using the Java Native Interface (JNI) in order to interact with native code and then retrieve the required data via object method accessing provided by the JNI environment pointer. This can be done 2 ways, either by creating the object and passing it as a parameter to the native methods or creating an object on the native side. Both ways require the population of the fields to be done on the native side. From then on, any data manipulation and processing could be done on the Java side. Unfortunately, this does require all data to be taken from the object and placed back into the structs before packets can be forwarded. Obviously this results in a lot of unneeded data copying, while the actual JNI calls can significantly reduce the speed of the application as shown in .

ref this

There are a few frameworks which aim to solve the problem of mapping structs to objects while trying to increase the overall speed of the JNI using different techniques.

Java Native Access (JNA) The JNA is a community developed framework which aims to abstract the JNI and native code away from developers who want to use native shared libraries. It uses its own data mapping between native and Java data types with automatic conversion of string and object to character arrays and structs. Although this framework provides much of the functionality within this project, it focusses on correctness and ease of use and therefore neglects performance.

Javolution Javolution is similar to the JNA but focusses more on real-time performance witin Java. It provides a number of high performance utilities for this purpose while allowing access to native libraries. For this purpose, struct and union classes are created automatically for mapping between objects. Javolution provides the functionality reqruied for this project but it requires a lot initial set-up which can be heavy, especially for small applications like middleboxes.

Swig Simplified Wrapper and Interface Generator(Swig) differs from JNA and Javolution simply because the framework only focusses on object and struct mapping. It abstracts all creation of objects, structs, access methods and memory allocation away from the user via an interface. Although very useful, it lacks performance when handling numerous objects in quick succession.

5.1.3 Byte Buffers

Byte buffers are a Java class which allow for memory to be allocated on the Java heap (non direct) or outside of the JVM off heap (direct), while abstracting pointer arithmetic and boundary checks away from the application. Non direct byte buffers are simply a wrapper for a byte array on the heap and are generally used as they allow easier access to the data as multiple byte reading and data type conversion is handled by the class. Direct byte buffers allocate memory outside of the JVM in native memory which means that the only limit on the size of byte buffers is memory itself.

The performance characteristics of direct and non-direct byte buffers are very similar. Direct byte buffers can be improved by aligning the bytes with the native endian (normally little endian) format since the class makes use of the Java Unsafe native access which allows for the methods to be in-lined with machine code.

The Java garbage collector doesn't have access to native memory and therefore direct byte buffers allow the application to manage memory and reduce memory usage. Garbage collection bottlenecks within high performance system are therefore reduced. For direct byte buffers, the actual object is stored on the heap and but the memory banks are off heap meaning that the pointer to the memory bank will not change, even if the byte buffer objects gets moved around by the garbage collector. This can be vital as it allows direct referencing of the memory for quicker data access.

Direct byte buffers would be an ideal choice for data sharing if it was possible to redirect DPDK mbuf structs onto the byte buffer directly, consequently allowing the Java side easy access to the data using the byte buffer API. However, since DPDK makes use of huge pages and handles its own memory management, it's not possible to use byte buffers directly as the memory for the NIC queues. Instead, data would have to be copied from structs and extracted into objects and vice versa.

5.1.4 Java Unsafe

The Java Unsafe class is actually only used internally by Java for its memory management. It generally shouldn't be used within Java since it makes a safe programming language like Java an unsafe language (hence the name) since memory access exceptions can be thrown which will ultimately crash the JVM. Even so, it has a number of uses such as:

Object initialisation skipping This is where any instance of an object can be created from the class, but no constructors are used meaning that the object created without any of the fields set. This has a number of uses including security check bypassing, creating instances of objects which don't have a public constructor and allowing multiple objects of a singleton class.

Intentional memory corruption This allows the setting of private fields of any object. It is a common way of bypassing security features as private fields to allow access to certain situation can be overwritten to gain access.

Nullifying unwanted objects This has a common use of nullifying passwords after they have been stored as strings. If a password is stored as a string in Java, even setting the field to *(*null) will only dereference it. The original string will still be in memory after the dereferencing up until it is garbage collected. Even rewriting the field with a new field won't work as strings are immutable in Java. This makes it susceptible to a timing attack to retrieve the password. Using unsafe allows for the actual memory location to be overwritten with random values to prevent this.

Multiple inheritance Java doesn't allow multiple inheritance within its class declaration of casting. However, using Unsafe any object can be cast to any other object without a compiler of run-time error. This obviously only works if data fields are compliant with each other and any method invocations are referenced.

Very fast serialization The Java *Serializable* abstraction is well known to be slow, which can be a major bottleneck in fast processing applications over a network. Even the *Externalizable* functionality isn't much factor and that requires an class schema to be defined. However, custom serializing can be extremely fast using the Unsafe Class. Basically allocating memory and then putting/getting data from the memory requires little JVM usage and can be done with machine instructions.

Obviously without proper precautions any of these actions can be dangerous and can result in crashing the full JVM. This is why the Unsafe class has a private constructor and calling the static Unsage.getUnsafe() will throw a security exception for untrusted code which is hard to bypass. Fortunately, Unsafe has its own instance called 'theUnsafe' which can be accessed by using Java reflection :

ref this

```
Field f = Unsafe.class.getDeclaredField(”theUnsafe”);
f.setAccessible(true);
Unsafe unsafe = (Unsafe) f.get(null);
```

Code 8: Accessing Java Unsafe

Using Unsafe then allows direct native memory access (off heap) to retrieve data in any of the primitive data formats. Custom objects with a set structure can then be created, accessed and altered using Unsafe which provides a vast increase in performance over traditional objects stored on the heap. This is mainly thanks to the JIT compiler which can use machine code more efficiently by inlining certain memory access directly with assembly code. This also removes the need for copying of data between memory locations, structs and objects, therefore meaning it is zero-copy.

5.2 Packing Structures

Structures (structs) are a way of defining complex data into a grouped set in order to make this data easier to access and reference as shown in Code 9. They are heavily used with C applications and can be seen as Java object without the associated methods.

```c
struct example {
    char *p;
    char c;
    long x;
    char y[50];
    int z;
};
```

Code 9: Example C Struct

On modern processors all commercially available C compilers will arrange basic C datatypes in a constrained order to make memory access faster. This has 2 effects on the program. Firstly, all structs will actually have a memory size larger than the combined size of the datatypes in the struct as a result of padding. However, this generally is a benefit to most consumers as this memory alignment results in a faster performance when accessing the data.

Explain why it has faster performance

Nested padding in struct?

C struct field always in given order

Inconsistencies with datatype length so using uint32t etc

Code 12 shows a struct which has compiler inserted padding. Any user wouldn't know the padding was there and wouldn't be able to access the data in the bits of the padding through conventional C dereferencing paradigm (only via pointer arithmetic). This example does assume use on a 64-bit machine with 8 byte alignment, but 32-bit machines or a different compiler may have different alignment rules.

```c
struct example {
    char *p;        // 8 bytes
    char c;         // 1 byte
    char pad[7];    // 7 byte padding
    short x;        // 2 bytes
    char pad[6];    // 6 byte padding
    char y[50];     // 50 bytes
    int z;          // 4 bytes
};
```

Code 10: Example C Struct with compiler inserted padding

Mention this is on 64-bit machine and obviously you don't notice padding and order of elements can pay an important part in this

```c
struct __attribute__((__packed__)) example {
    char *p;        // 8 bytes
    char c;         // 1 byte
    short x;        // 2 bytes
    char y[50];     // 50 bytes
    int z;          // 4 bytes
};
```

Code 11: Example C Struct stopping padding

Since the proposed application in this report requires high throughput of data, the initial thought would be that this optimisation is a benefit to the program. Generally this is the case, but for data which is likely to be shared between the C side and Java side a large amount, data accessing is far quicker on the Java side if the struct is packed (no padding). This results in certain structs been forced to be packed when compiled, more noticeably, those used for packet and protocol headers.

Proof on speed

Packed structures mean there are no gaps between elements, required alignment is set to 1 byte. Also __attribute__((packed)) definition means that compiler will deal with accessing members which may get misaligned due to 1 byte alignment and packing so reading and writing is correct. However, compilers will only deal with this misalignment if structs are accessed via direct access. Using a pointer to a packed struct member (and therefore pointer arithmetic) can result in the wrong value for the dereferenced pointer. This is since certain members may not be aligned to 1 byte. In the below example, unint32 is 4 byte aligned and therefore it is possible for a pointer to it to expect 4 byte alignment therefore resulting in the wrong results.

```c
#include <stdio.h>
#include <inttypes.h>
#include <arpa/inet.h>

struct packet {
    uint8_t x;
    uint32_t y;
} __attribute__((packed));

int main ()
{
    uint8_t bytes[5] = {1, 0, 0, 0, 2};
    struct packet *p = (struct packet *)bytes;

    // compiler handles misalignment because it knows that
    // "struct packet" is packed
    printf("y=%"PRIX32", ", ntohl(p->y));

    // compiler does not handle misalignment - py does not inherit
    // the packed attribute
    uint32_t *py = &p->y;
    printf("*py=%"PRIX32"\n", ntohl(*py));
    return 0;
}
```

Code 12: Example C Struct with compiler inserted padding

On an x86 system (which does not enforce memory access alignment), this will give y=2 and *py=2 which is as expected. Conversely, other systems using the SPARC architecure or similar will give y=2 and *py=1 which isn't what the user would expect.

However, since a packed struct is much easier to traverse from Java than a padded struct, the decision was made to make certain structs packed within the DPDK framework and then recompile the libraries. This decision could be made since other structs within the DPDK framework were also packed and therefore consideration of this was already made.

Note that if a struct contains another struct, that struct should be packed recursively as-well to ensure the first struct has no padding at all.

Char doesn't have alignment and can start on any address. But 2-byte shorts must start on an even address, 4-byte ints or floats must start on an address divisible by 4, and 8-byte longs or doubles must start on an address divisible by 8. Signed or unsigned makes no difference.

Self-alignment makes access faster because it facilitates generating single-instruction fetches and puts of the typed data. Without alignment constraints, on the other hand, the code might end up having to do two or more accesses spanning machine-word boundaries. Characters are a special case; they're equally expensive from anywhere they live inside a single machine word. That's why they don't have a preferred alignment.

Casting to an odd pointer will slow down code and could work. Other architectures will take the word which the pointer points to and therefore the problem occurs above.

5.3 Performance testing

In order to evaluate the most suitable data sharing technique, performance testing on 4 different implementation options for sharing data between Java and native memory were considered. Since the ultimate aim of the implementation is to maximise throughput of packets, the performance test tried to mimic this by processing data on 1 million packets per iteration. This processing involved retrieving data from the native packet struct, loading that data into a Java object, changing the data and then setting the data back into the original struct. Various techniques to do this were used to try and find the best performance possible. All of the techniques made use of a static native struct which acted like a new packet been received. The changed data was then set back into this struct.

Algorithm 2 Data Sharing Performance Test Algorithm

```
function Main
    for i = 1 to 10 do
        startTimer
        performTest( )
        stopTimer
        outputTime

function performTest
    for i = 1 to 1000000 do
        retreiveData
        setNewData
        saveData
```

Considering there were 4 different data sharing techniques tested, the *retrieveData*, *setNewData* and *saveData* methods were different and are described below:

Object Passing The object technique involved creating a packet and sending its reference through the JNI to the native code. From there, this packet could be populated with data from a given struct through the Java environment pointer. For each setter method called, the method id of that method must be retrieved for the given class so the combination of that and the packet reference could set the data.

From there, new data is input into the packet from the Java side and passed back through to JNI so the struct can be set with the new data. This is done using getters for the objects' fields via the Java environment pointer.

Byte Buffer The technique involved declaring a direct byte buffer to assign off heap memory of the size of the packet struct. From there, the pointer to this memory location was sent to the native code, where data from the struct was directly copied into the byte buffer, therefore populating the byte buffer with duplicate data. The Java code then pulled the data from this via the byte buffer's inbuilt methods and set the data into a new packet object. From there the packet object could be used whenever desired.

To save the data back into the original struct, the data was copied from the packet object back into the byte buffer in the order of the members of the struct. The data was then pulled from the memory of the byte buffer and set back into the original struct.

Unsafe Using the Unsafe class allows for direct access to members of the struct. To take account for this this technique first allocates off heap memory for the pointer to the struct to be stored. This pointer is put into the memory location natively and accessed via Unsafe methods. From this, data can be accessed directly from the struct and input to a new packet object for later use. New data is then set in the object.

To set the data in the struct, the data is removed from the object and directly put into the struct. This is done using the pointer and the correct byte offsets depending on the previous data type inputted.

Direct Direct accessing relates to not storing the struct members in Java at all. Instead a different type of Packet object is used which just stores pointers to the struct. From there any accessing and setting of data is simply done using the Unsafe class to directly get or set the values within the struct. This different packet object also contains offset information for the struct so the correct values are accessed.

5.3.1 Expectations

Of the 4 techniques, it is expected that the object method would be by far the slowest, mainly due to the excessive number of JNI calls used which significantly slows an application down. The Byte Buffer method will most likely be the slowest of the other 2 techniques due to the large number of data copying which goes on. The other 2 techniques (unsafe and direct) will likely be very close in performance, mainly because they both use direct accessing into the struct. The unsafe technique should be slightly slower however due to the setting and getting from the packet object.

evaluate each technique on ease and number of data copies and results and say why we picked 1 of the others

5.3.2 Results

The graphs in figures 14, 15 & 16 show the results from the different techniques run on different systems. Considering there were 10 iterations of 1 million packets per technique the results show the averages of the times. However, certain times were disregard and seen as anomalies since they were well above the average . These readings were generally the initial values after the 1st iteration and can be either be related to the just in time compiler warming up after switching to a different class or garbage collection on the vast number of objects left over from the previous technique.

stats for this

The graphs show the time in nanoseconds which it took to process an individual packet, but the scale is logarithmic due to the excessive size of the object technique. For easier reading, the numbers at the top of the columns show the times factor for each technique compared to the fastest. For example, on Figure 14 the byte buffer takes 2.74 times the direct technique to process the packet.

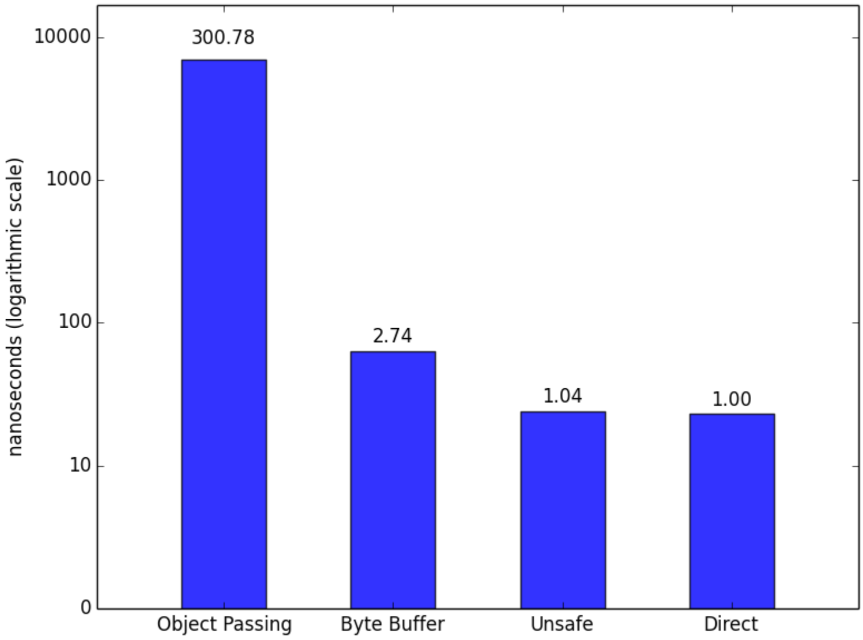


Figure 14: Linux - Intel Xeo E5-2690 at 2.90GHz with 32 cores

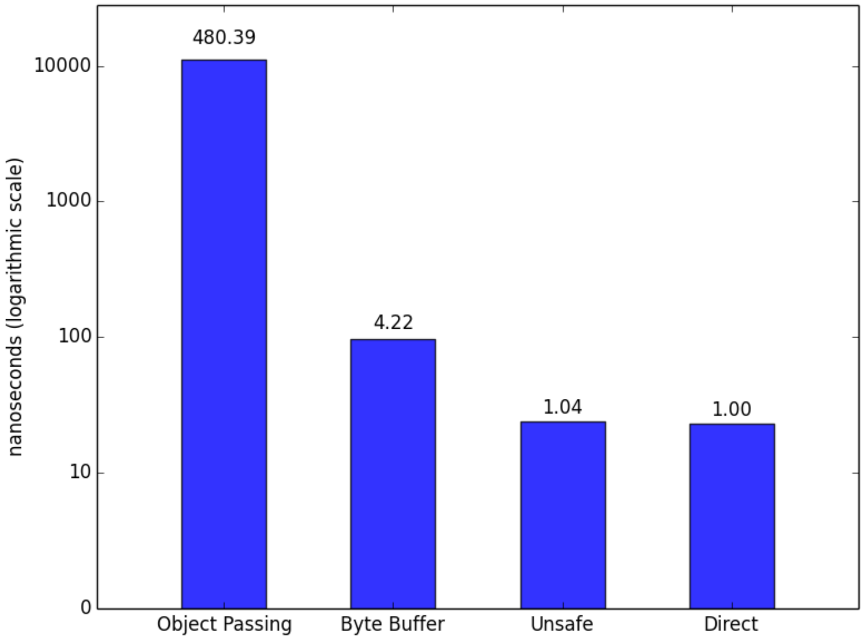


Figure 15: Mac OS X - Intel i7-3615QM at 2.30GHz with 8 cores

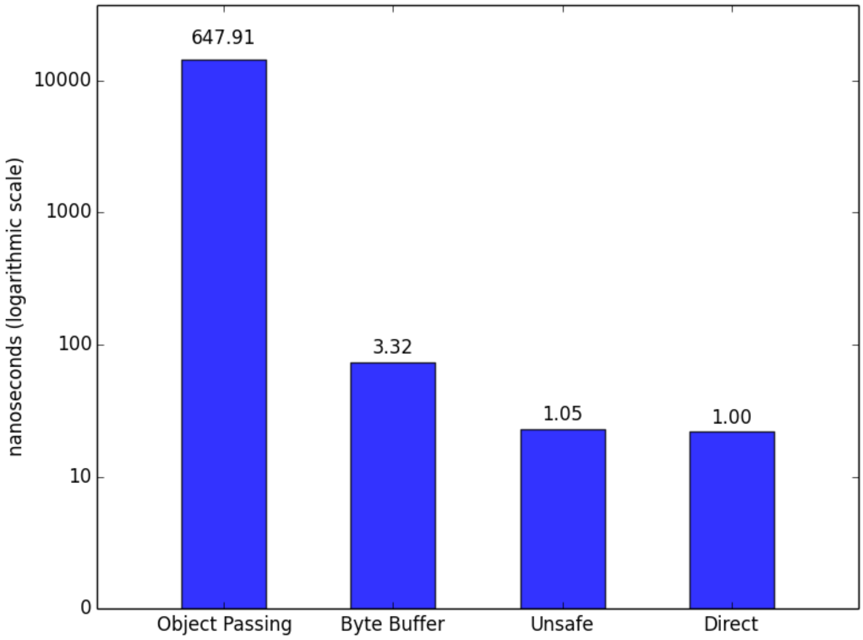


Figure 16: Linux VM on Mac OS X - Intel i7-3615QM at 2.30GHz with 4 cores

5.3.3 Evaluation

As the graphs show, the exceptions declared earlier have been met. The object technique should be completely disregard, even though it offers the easiest solution. Due to its abysmal performance, been 300 - 650 times slower than the faster technique, its not going to help with packet throughput.

For similar reasons Byte buffer should be disregarded as well simply because its too slow compared to the other methods. The amount of data copying with this technique is the downfall, although this could have potentially worked if structs could be directly written into the buffers.

cant be done and reference earlier

The difference between the direct and unsafe methods is minimal on all 3 machines. The direct technique is slightly faster since it doesn't copy the data into the object although Java is surprisingly fast at this as proven by the unsafe results. Furthermore, in general middleware software, there isn't the requirement to access all data fields of the packets generally, therefore storing them all in objects can be seen as needless copying. This is where the direct access technique excels since it only accesses data which it needs to been zero-copy. Direct access was therefore chosen as the technique to use for the implementation which goes into further details about how this was done.

5.4 Thread affinity

Normally as a thread gets a time slice (a period in which to use the core), it is granted whichever core is determined to be most free by the operating system's scheduler. However, due to the scheduler and context switching of threads, a thread may move around a number of different cores during its execution cycle.

Thread affinity aims to reduce this context switching which can be expensive, by limiting threads to 1 or a subset of the available cores. Even if the thread is scheduled to have a 'break', since it doesn't change cores there is no need to copy data between core caches which ultimatily improves performance.

Therefore, for primarily single (or limited) thread applications, it is sometimes best to set the CPU affinity to a specific core, or subset of cores. This will allow the 'Turbo' processor frequency scaling to kick in and be sustained (instead of skipping around to various cores that may not be scaled up, and could even be scaled down).

```
cpu_set_t cpuset; \\ structure used to manage cpu affinity settings
pthread_t thread = pthread_self();  \\ get own system wide thread id
CPU_ZERO(&cpuset); \\ zero cpuset structure
CPU_SET(5, &cpuset); \\ set cpuset structure use with the 6th core (0-5)
int res = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset); \\ set
    affinity of thread
\\ error handling here of res
res = pthread_getaffinity_np(thread, sizeof(cpu_set_t), &cpuset); \\ check
    affinity of thread
\\ error handling here of res
```

Code 13: Example of setting thread affinity

ref this

In Linux, Java threads uses the native thread(i.e, thread provided by Linux which is normally the POSIX thread). This means the JVM creates a new native thread when the Java code creates a new Java thread. Java provides access to the thread which is currently running via the *Thread.currentThread().getId()* call but this actually only gets the JVMs id number of the thread which is local. For affinity threading to be set the system thread id us required.

Since the Java standard library does not provide functionality for affinity threading, then this function need to be provided through custom native methods. A native thread can be bound to a core through the pthread_setaffinity_np() function. Code 13 shows the basic steps of how to achieve affinity threading.

5.5 Endianness

This describes the order in which bytes of data types are stored in memory relative to pointers. In big-endian systems, the most significant bytes are stored at the pointer with every successive data bytes stored in successive memory locations. Conversely, in little-endian systems, the least significant byte is stored at the pointer. There is no advantage or disadvantage to either endian types, its simply a matter on convention for certain systems. Figure 17 shows how different endian systems store the value of the hexadecimal value *0x0A0B0C0D*.

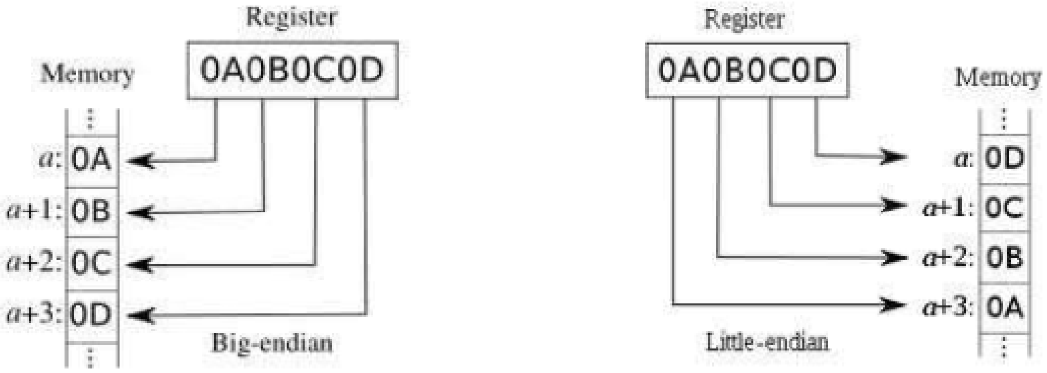


Figure 17: Difference between little and big endian

This becomes a problem when using the Java Native Interface with native code code on Intel architectures since the Java Virtual Machine uses big-endian format while Intel uses little-endian format. Normally the JNI environment would handle this byte ordering conversion but since the JNI has been proven to be too slow to meet the requirement, this needs to be handled by the application.

ref this

5.6 Data type conversion

Between different languages the same data type can be represented by varying lengths in bytes and whether unsigned or signed (in the case of numerical values). It can also be the case that data type lengths in native languages differ depending on the architecture and whether it is 32-bit or 64-bit.

This becomes more of an issue when sharing data between C and Java. Java always uses a signed integer representation with the most significant bit representing whether the number is negative or positive. C uses both signed and unsigned representation depending on the requirements with varying byte length. Again, any conversion between the differences is normally handled via the JNI, but since the implementation aims to bypass the JNI as much as possible, data type conversion will have to be handled.

Lets take the integer representation as an example. In C, and integer can vary between 2, 4 and 8 bytes in length depending on the architecture and compiler. To solve this, standard integer types are used (e.g. uint8_t & int32_t) which guarantee that the representation is at least the length of the type definition stated. In Java, an integer is guaranteed to be 4 bytes long regardless of the system. However, DPDK uses unsigned integers while Java uses signed integers. This requires conversion between the unsigned and signed, but since unsigned has a higher upper bound on the value it can store due to the extra bit (MSB) there can be an overflow error when converting to Java. This requires that Java uses a long (8 byte) representation to hold the C 4 byte unsigned integer.

Conversion from Java to C then could then result in an underflow error if a value which can be represented by a Java long can't be represented by a C integer. This means bound checking is required on the Java side for any number conversions.

get a table of this

5.7 Protocol undertaking

just IP (4 and) supporting udp and tcp (i think), no ARP etc

5.8 config files

6 DPDK Java

Since the Data Plane Development Kit is a framework, it was decided that instead of implementing separate middleboxes, a Java framework would be implemented. This would allow any number of other applications to be designed with ease. The DPDK Java framework only supports a subset of the original DPDK features, however, it provides suitable functionality to produce a fully working application which can make use of the key concepts. A number of example applications are implemented in section 7 using the DPDK Java framework, all of which are easy to extend into more complicated applications which support features like packet fragmentation and packet payload inspection.

6.1 Overview

As proven earlier in the report, for fast packet processing in Java the number of JNI calls should be minimised as possible due to their large overhead. However, since DPDK is a native library some JNI calls were obviously mandatory, mainly those initialising the application and packet interactions. This was minimised by making the majority of the JNI calls at the start of the application, before individual processing threads had been started. From then on, JNI calls would only be made at vital times within the application.

An overview of the DPDK Java structure is shown in figure 19 which shows the key concepts of the framework. Any application requires 2 user defined class:

Application Main This class is responsible for starting the DPDK framework indirectly, via a number of method calls to the application starter, passing the required configurations as parameters. Some of this is done via the config file and other properties have to be passed as parameters to the application starter class by the user. It also initialises the custom processor unit class described below and passes them to the application starter. A statistics module can be initialised at the start as well which runs on its own thread.

Processor Unit This class is the main unit of the application which is eventually threaded by the framework. It therefore has to extend the Runnable class. It's responsible for requesting packets, processing them and then sending or freeing the packets. These classes can share the Packet Free-er and Packet sender classes but most have their own receive poller class which is associated with a given port and queue.

As figure 19 shows, most of these classes interface with the native methods, either on a regular basis or on initial start-up. Most of these native methods interact with the DPDK framework in some capacity. However, these native methods are abstracted away from the users application requiring no knowledge of the JNI calls or the underlying memory layout.

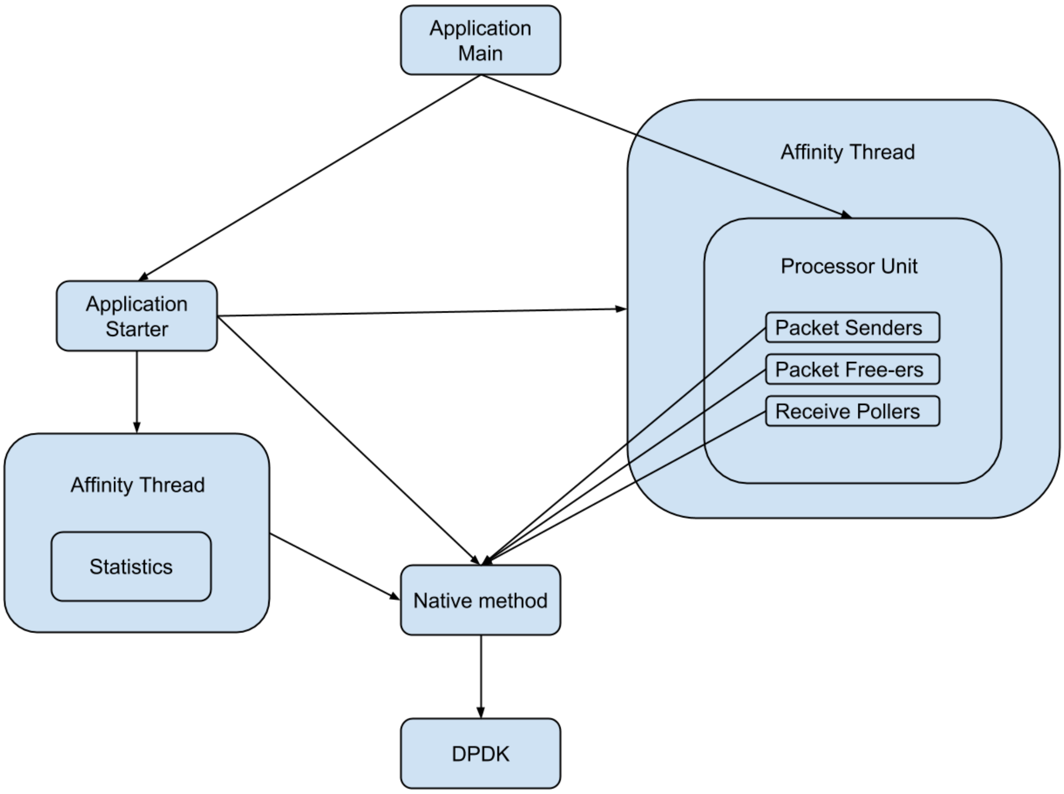


Figure 18: DPDK Java basic class overview

6.2 Native Libraries

For the Java framework to work, it requires a native shared library which includes all of the native methods to be called. This shared library can be placed anywhere as long as the correct Java path is set to its location, however, the common place is to install it within the users library directory (/usr/lib/) in Linux.

This shared library must also be dynamically linked to other shared libraries which contain the other non-JNI methods called, whether this is system calls, standard system libraries or the actually compiled DPDK libraries. For this, there a number of tools which can be used, all with their own advantages and disadvantages.

6.2.1 Library Compilation Tools

GNU Linker The GNU linker on linux is a basic linker which can be used with any number of shared libraries, although as the number increases so does the number of number of commands needed, as well as the number of flags needed. Considering the large amount of shared libraries which DPDK uses, this option is less attractive.

GNU Libtool The GNU Libtool uses the GNU linker underneath but abstracts away the compilations into a build unit, similar the makefiles used. It also has the advantage of only recompiling those libraries required if certain source files are updated. However, it does require a significant amount of set-up, and problems can occur when migrating to other machines.

DPDK Makefile By far the best option is to use the DPDK makefile. It is generally used when the application is written in C and uses the DPDK libraries which is why a lot of documentation is provided on this. However, when wanting to compile shared libraries (which don't include a main method) there is very limited documentation on this and certain flags and processes can only be found by delving into the build process of DPDK. Even so, this is the easiest option to use on a regular basis and ports perfectly well to other systems as it installs the shared library for you as well.

6.3 Initialisation

DPDK requires a number of initialising procedures to create the environment abstraction layer, start ports, allocate memory pools and setup the port specific queues. Each of these procedures is directly mapped to a Java implementation of them, which also handle error checking. Generally any native applications using DPDK create a number of threads depending on the number of available cores to split the work and maximise throughput. However, since the Java side creates its own affinity threads for this, DPDK is always initialised with only 1 thread to be used for the set-up.

The native initialisation functions make use of a number of memory allocation (malloc) calls, which goes against the recommended procedures. These calls are used to dynamically create variables such as those used for initialisation parameters for the EAL. After further research into the memory usage, malloc calls are supported and are only discouraged due to the reduced speed of accessing the memory locations over those stored within the huge pages. However, since they are only used at initialisation, performance isn't a factor here.

When writing custom applications, all initialising should be complete before starting the threads. This is to stop any DPDK errors when trying to access uninitialised ports, memory pools or queues. For this reason, the last thing which should be called is 'startAll()' which starts all threads and affinity threads one by one in quick succession to allow the actual packet processing to take place.

6.4 Processing Threads

The framework supports multiple processing procedures which can run simultaneously and even with different implementations depending on the requirements. This allows an application to be built where certain processing objects do the polling of packets, and then pass these objects to be inspected, and then onto another object for sending and freeing of the packets. This basically creates a pipelined application, although a more simple processing object would receiver, process and forward packets in the same thread. This is instead of multiple processing units doing exactly the same job.

Each of the processing units are automatically threaded by the framework and put into an 'Affinity Thread'. This follows the same logic of the DPDK framework where each thread is set to only run on 1 core of the machine's processor using affinity cores . This provided a few complications as Java and the JVM doesn't provide functionality for assignment of threads to cores. This is mainly as because JVM abstracts away the complications of this and allows the kernel to do its own thread scheduling. However, on Linux, Java does utilise the native POSIX Threads (pthread) and assigns a Java thread to 1 pthread. This meant via a few JNI calls and native system calls , each thread could in fact be associated with certain cores. As with DPDK, this limits the number of threads to be equal or less than the number of available cores on the machine (of hyper-threaded cores) as to fully maximise the application speed.

ref this and say thread can be set to run on multiple cores - need to fix code for this

example?

Each of the processing units must be an extension of the 'PacketProccesor' class which provides an underlying abstraction so statistics of packet data can be retrieved as mentioned below.

Describe 3 objects and fix code to use more than 1

6.5 Packet Data Handling

DPDK and therefore the Java framework version is primarily used for packet processing, meaning that efficient handling of the packet's data and header information are a necessity. Investigations outlined in section supported that copying packet data from the native side to Java and back again was a very poor choice in terms of processing speed. To solve this, all packet information is left in native memory (not copied to the Java heap) and accessed directly from the Java application in order to read and write data to/from specific packets.

ref this

In order to do this, the Java Unsafe class was used extensively as it provided then functionality. However, as discussed previously, the Unsafe class directly accesses native memory and is therefore inherently unsafe to use, as opposed to Java itself. This memory accessing was therefore abstracted away into the 'UnsafeMemory' class which handled the type conversion between native unsigned and Java signed, pointer arithmetic and big/little endian conversion. It also checks whether values are going to be out of data type number representation range when converted from signed to unsigned, since unsigned values are represented as larger data types to account for the signed bit.

Since little-endian to big-endian conversion and vice-versa was required to pass data between the JVM and native memory it was decided that all handling of this would be done on the native side. This firstly abstracts the certain complications of this away from the Java side to provide a cleaner interface and secondly the native implementation provides faster byte shifting. To accomplish this, whenever data was read into a memory location assigned from Java Unsafe, it would flip the data's byte order and put the data into big-endian format for the JVM to interpret correctly. Whenever data was been passed from Java to the native side, the opposite would occur .

elaborate on this, diagram? how did pointers work?

Each individual network packet is assigned its own Packet object representation. Even though object usage on the Java heap can be relatively slow compared with native structure handling due to various reasons, it was required or else the application would be wondering away from the object orientated side of Java. However, the Packet object only tracks 3 fields in order to minimise data copying:

- protected long mbuf_pointer - points to memory location of the start of mbuf header for the packet
- protected long packet_pointer - points to memory location of the start of the packet header (either IPv4 or IPv6)
- protected UnsafeAccess ua - packets own unsafe memory accessing object for secure navigation around packet data

Since the native mbuf and ipv4/ipv6 structures are forced to be packed, this means that all fields can be traversed and therefore read from and written to via pointer arithmetic. The packet_pointer allows for simple getters and setters which relate to the standard IP packet headers, even if the actual fields don't exist. The mbuf_pointer is generally used to gain access to the raw packet data and used later for freeing and forwarding the packet.

ref this

6.6 Packet Polling

Packet polling as the act of receiving data from the memory buffers used to store packets received by the network interface card/controller. When an application requests new packets from the received queue of a specific port, it does so using native methods via the JNI. These are the only JNI calls used after the initial start up (i.e used in processing threads) in order to increase performance. The native methods' parameters specify the port id, queue id and a memory location pointer, while defaulting to fetching the default number of packets set on the application initialisation. Before this call happens, Java DPDK allocates a set number of bytes within the native memory and holds its pointer in memory in long format. This pointer location is then passed as one of the parameters to the native method.

Within the native method, the number of received packets is put as a short data type into the memory location of the pointer. Every subsequent value inserted into the memory location is first a pointer to the mbuf location of a given packet and then a pointer to the location of the packet header. This happens for all of the packets received by the call.

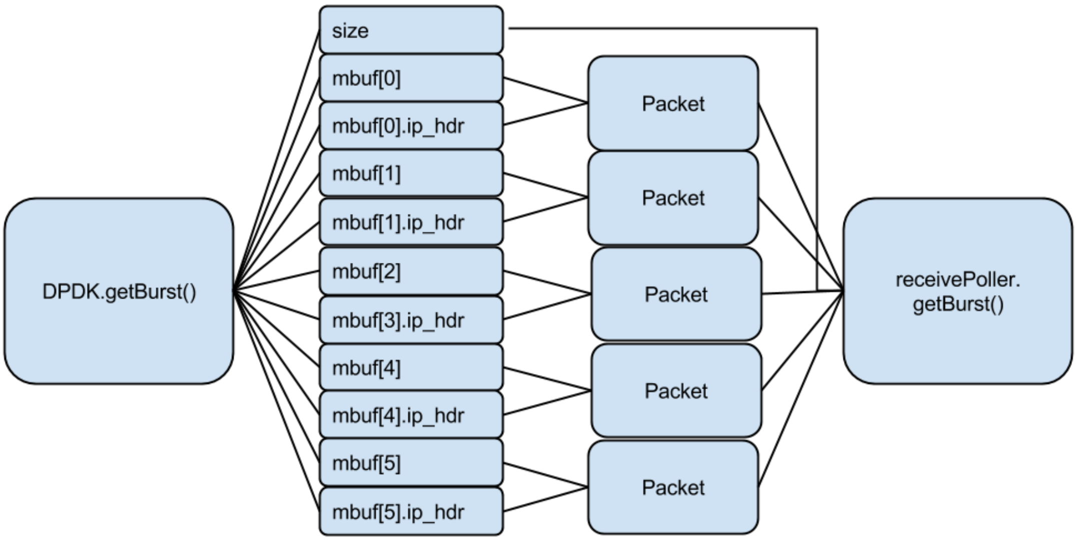


Figure 19: Example memory setup for getting burst of packets

Within the Java side, using the UnsafeMemory class, the number of packets is pulled from the memory location. From there, the pointers can be pulled within a loop and added to their own packet class, depending if they are IPv4 or IPv6 headers. A list of packets is then returned from the method.

6.7 Packet Sending

Packet sending involves taking processed packets and putting them on the port queues ready to be sent by the network interface card/controller. When ever a packet needs to be sent, it is passed to a PacketSender object which initiates the process. Within PacketSender, packets are stored within a list ready to be sent. Since packet sending within DPDK is more efficient when sending multiple packets at the same time, whenever the list reaches the default value for a sending burst (n), the first n packets in the list are sent. However, there is also a time-out functionality which is used whenever the receiving of packets is slow. Since the program doesn't want to wait for n packets which could take a long time, the time-out allows any number of packets to be sent if a time reaches a user set time period away from the last send burst.

Sending of packets is very similar to the polling of packets but in the opposite direction. Firstly a memory location of the required size (depending on the number of packet to send) is allocated in native memory, and that pointer, along with port id and queue id are passed as parameters to the native code. Each packets' mbuf pointer is then added to the memory location in increasing order. This pointer is then directly passed as a parameter to the DPDK packet sending methods as the memory is already set up in the correct format.

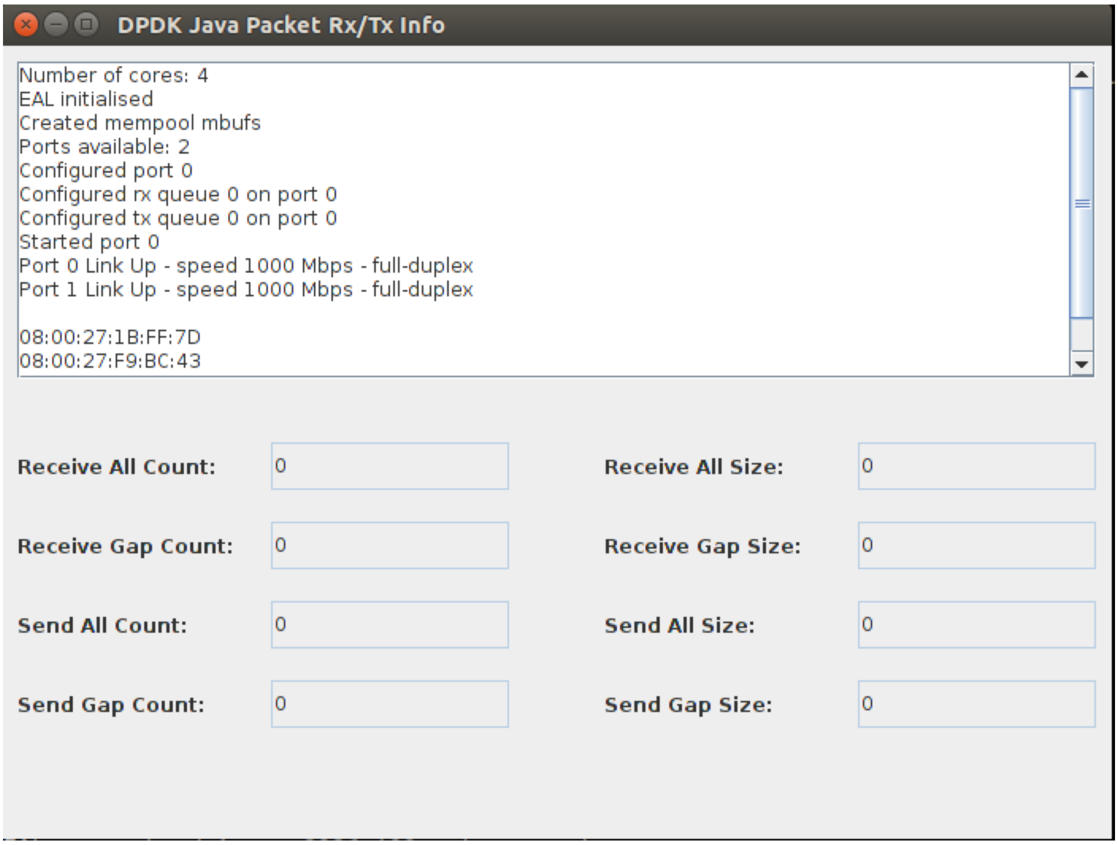
ref this

Once the packets have been put onto the queue, a loop if made through the pointers to the mbuf's to free the memory.

6.8 Statistic Profiling

The statistic profiling is an optional part of the framework which allows statistics to be gathered on the number and sizes of packets received and packets sent. It gathers data every second by default, but can be user set to find information about the number of packets sent/received for a given time unit. It's up to the user to set which ReceivePoller and PacketSender objects to gather information on. In order to be most efficient for the application, it runs on its own thread, although not an affinity thread, so it can be context switched into any of the cores.

It also has the option of running a graphical user interface (GUI, figure 20) which displays the data in better format instead of constant prints to the console. It also redirects other console information on the running of the application to a user console on the GUI for easier use as DPDK outputs a lot of information which isn't that useful.



```
Number of cores: 4
EAL initialised
Created mempool mbufs
Ports available: 2
Configured port 0
Configured rx queue 0 on port 0
Configured tx queue 0 on port 0
Started port 0
Port 0 Link Up - speed 1000 Mbps - full-duplex
Port 1 Link Up - speed 1000 Mbps - full-duplex

08:00:27:1B:FF:7D
08:00:27:F9:BC:43
```

Receive All Count: 0
Receive All Size: 0
Receive Gap Count: 0
Receive Gap Size: 0
Send All Count: 0
Send All Size: 0
Send Gap Count: 0
Send Gap Size: 0

Figure 20: Statistics GUI showing packet data and console output

The implementation for the statistic collector simply involves each packet sender and receive poller iterating counters on the required data. This can potentially have synchronisation issues, which aren't solved to prohibit performance reduction. This means the statistics aren't fully accurate, although they do provide good estimations of the data.

To solve this issue, another statistics profiler was implemented which makes use of some DPDK libraries and native timer libraries to continously poll the NIC for its very accurate on-board data collection. Again, this process is run in a separate thread but it doesn't allow for interaction with the GUI.

6.9 Correctness Testing

Not testing performance but packet data movement and pddk set-up setting u VM since needed specific environments.

6.10 Initial Testing

The initial testing of applications was carried out an a local Mac OS X machine running Ubuntu 14.04 LTS 64-bit on a VirtualBox virtual machine. Although this set-up didn't provide the ability to load test on very high speeds (anything above 1Gbit/s), it allowed for basic testing to check that the application was running as expected. Load testing of speeds up to roughly 700Mbit/s were also possible which have a basic testing platform without the need to move code to servers.

which ones?

ref this and ubuntu

important to check for errors at every stage to stop jvm crashes

6.10.1 Set-up

Testing could be carried out using the 2 available 1Gbit NICs of the machine via a bridged network from the host to guest machine which severely reduced transmission speed. This allowed an ethernet cable to be looped back and connected between the ports, meaning anything transmitted via 1 port was guaranteed to be received by the other port.

Pktgen and the custom application were booted up simultaneously running in parallel. Careful memory allocation, port addressing and processor core assignment had to be carried out to stop shared resourced impacting the overall performance of either application. This allowed Pktgen to send packets and the application to receive and process them.

6.10.2 Methods

6.10.3 Results

7 Applications

Since a Java DPDK framework was designed, any applications are relatively easy to create and only use Java code so no interaction with the native DPDK is necessary from a users point of view. Below are 2 simple applications created using this framework, which both show the small amount of code required to make a working application. However, either of these could be heavily extended to use multiple threads, different processing objects, shared ports in order to design more complex application such as packet fragmentation, reassembly, multi-casting or ordering.

7.1 Firewall

7.2 Network Address Translator (NAT)

8 Evaluation

This section focusses on the comparison testing which was carried out between the C and Java implementations of basic middleboxes. Evaluation of the results will then be discussed and further improvements to the Java solutions will be considered.

In order to fully understand the capabilities of middleboxes, testing was carried out on Imperial College's Large Scale Distributed Systems (LSDS) test-bed. Although this system consists of numerous machines, tests were carried out using just 2. The first machine was used to host the middlebox application and receive the packets. The other machine was used as the client and ran the pktgen software allowing it to generate packets at up to 10Gbps in order to take advantage of the machines network interface controllers. Each machine had 2 Intel Xeon ES-2690 2.90Ghz processor which when hyer-threaded, resulted in 16 cores and had 32GB of memory.

8.1 Packet Generating

Packet generating is the act of creating packets with random payloads to be sent to certain MAC addresses on the network. This can either be done via the use of specialised hardware or using software. They are used for load testing of packet processing applications to test the amount of data which applications can process per second. This can reveal whether limitations on a system is software or hardware based.

Talk about pktgen module in kernel - does dpdk version use this

Pktgen is open source software tool, maintained by Intel, which aims to generate packets using the DPDK framework. It can generate up to 10Gbits of data per second with varying frame sizes ranging from 64 to 1518 bytes, and send the data in the form of packets across a compatible network interface card/controller. It has a number of benefits which include:

- Real time packet configuration and port control
- Real time metrics on packets sent and received
- Handles UDP, TCP, ARP and more packet headers
- Can be commanded via a Lua **??** script

Pktgen's transmitting performance can be reduced depending on the frame size. This was negated to keep it a constant within the tests by running the application with multiple lcores on the same port with different queues, which allow the transmitting speed to match those of the NIC. With the packet size varying, obviously the number of packets transmitted per second is dependant on this. The differences are shown in Table 3.

Packet Size	Packet/s (millions)	MBit/s
64 (min)	14.9	9999
128	8.4	9999
256	4.5	9999
512	2.3	9999
1024	1.2	9999
1518 (max)	0.8	9999

Table 3: Pktgen performance for varying packet sizes transmitting in 32 packet burst.

8.2 Further Testing

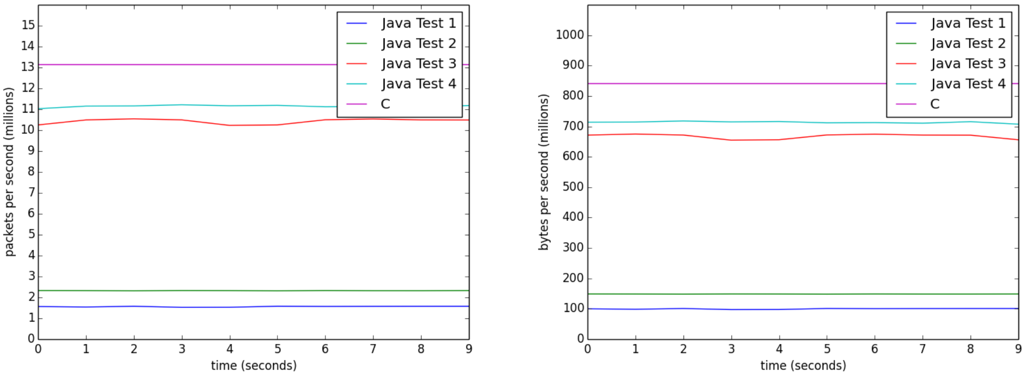
The first test on the LSDS test-bed involved comparing the C and Java implementations of a packet capturing application which simply received the packets and freed them straight away without any further transmission. The C implementation is used to give the optimal readings possible from this and further tests, since very limited processing is carried out between receiving and dropping the packet. The figure below shows the performance of the application at varying packet sizes.

Missing figure

Graph of c packet capture, 1 core, different frame sizes, will need to reference transmit speed of pktgen - are results of this correct?

what does the results show

The same testing was carried out with an identical packet capture algorithm which was instead implemented in Java using the DPDKJava framework described in section 3. All testing was done using 64 byte frames to make sure a consistency was kept throughout. Frame size of 64 bytes was chosen as this size is the hardest for applications to process simply because of the sheer number of packets per second which are received. This is an ultimate test of an applications packet throughput, and generally if it can handle packets at 64 bytes, it can handle much larger packets since the packet rate will be reduced. These results are shown in figure **??**.



(a) Millions of packets per second

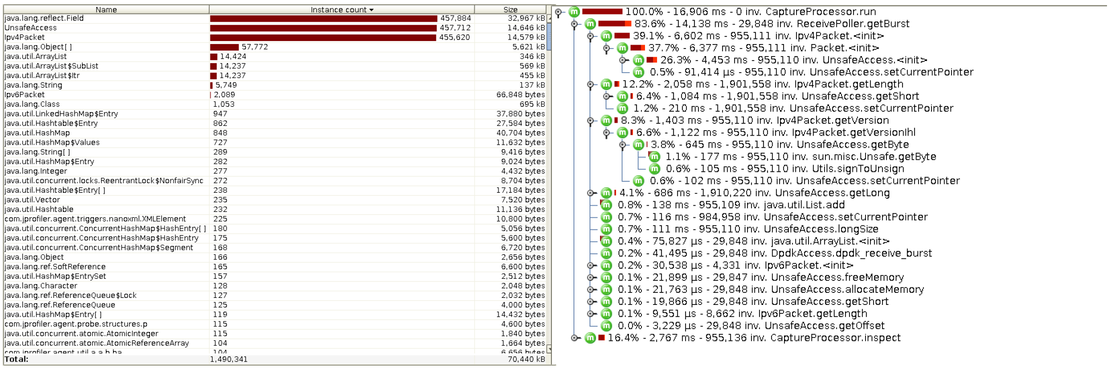
(b) Millions of bytes per second

Figure 21: Performance results of C vs Java implementations of packet capture application for all rounds of testing

different frame sizes?

Figures 21a and 21b indicate that after the first testing using the initial DPDKJava framework that the packet and data throughput was extremely low compared to the C version. Obviously these results are extremely closely related to each other resulting in the C version been roughly 8 times quicker after the first test. Although the native implementation was expected to be slightly quicker, this gap was way too high considering the design considerations and framework implementation.

To check where the problems lied within the code, a Java profiler (JProfiler) was used to check multiple parts of the code including memory usage, cpu usage and the number of method calls and the average time per method call. This provided invaluable analysis of where the problems were, although the profiler significantly reduced the performance of the application since it connects to the JVM and reads the data itself. Even so, obvious bottlenecks could be spotted allowing for performance improvement implementations to be made.



Name	Instance count ▾	Size
java.lang.reflect.Field	457,884	32,967 kB
UnsafeAccess	457,712	14,646 kB
Ipv4Packet	455,620	14,579 kB
java.lang.Object[ ]	57,772	5,621 kB
java.util.ArrayList	14,424	346 kB
java.util.ArrayList$SubList	14,237	569 kB
java.util.ArrayList$Itr	14,237	455 kB
java.lang.String	5,749	137 kB
Ipv6Packet	2,089	66,848 bytes
java.lang.Class	1,053	695 kB
java.util.LinkedHashMap$Entry	947	37,880 bytes
java.util.Hashtable$Entry	862	27,584 bytes
java.util.HashMap	848	40,704 bytes
java.util.HashMap$Values	727	11,632 bytes
java.lang.String[ ]	289	9,416 bytes
java.util.HashMap$Entry	282	9,024 bytes
java.lang.Integer	277	4,432 bytes
java.util.concurrent.locks.ReentrantLock$NonfairSync	272	8,704 bytes
java.util.Hashtable$Entry[ ]	238	17,184 bytes
java.util.Vector	235	7,520 bytes
java.util.Hashtable	232	11,136 bytes
com.jprofiler.agent.triggers.nanoxml.XMLElement	225	10,800 bytes
java.util.concurrent.ConcurrentHashMap$HashEntry[ ]	180	5,056 bytes
java.util.concurrent.ConcurrentHashMap$HashEntry	175	5,600 bytes
java.util.concurrent.ConcurrentHashMap$Segment	168	6,720 bytes
java.lang.Object	166	2,656 bytes
java.lang.ref.SoftReference	165	6,600 bytes
java.util.HashMap$EntrySet	157	2,512 bytes
java.lang.Character	128	2,048 bytes
java.lang.ref.ReferenceQueue$Lock	127	2,032 bytes
java.lang.ref.ReferenceQueue	125	4,000 bytes
java.util.HashMap$Entry[ ]	119	14,432 bytes
com.jprofiler.agent.probe.structures.p	115	4,600 bytes
java.util.concurrent.atomic.AtomicInteger	115	1,840 bytes
java.util.concurrent.atomic.AtomicReferenceArray	104	1,664 bytes
com.jprofiler.agent.util.a.a.b.ba	104	6,656 bytes
Total:	1,490,341	70,440 kB

```
100.0% - 16,906 ms - 0 inv. CaptureProcessor.run
  83.6% - 14,138 ms - 29,848 inv. ReceivePoller.getBurst
    39.1% - 6,602 ms - 955,111 inv. Ipv4Packet.<init>
      37.7% - 6,377 ms - 955,111 inv. Packet.<init>
        26.3% - 4,453 ms - 955,110 inv. UnsafeAccess.<init>
        0.5% - 91,414 µs - 955,110 inv. UnsafeAccess.setCurrentPointer
    12.2% - 2,058 ms - 1,901,558 inv. Ipv4Packet.getLength
      6.4% - 1,084 ms - 1,901,558 inv. UnsafeAccess.getShort
      1.2% - 210 ms - 1,901,558 inv. UnsafeAccess.setCurrentPointer
    8.3% - 1,403 ms - 955,110 inv. Ipv4Packet.getVersion
      6.6% - 1,122 ms - 955,110 inv. Ipv4Packet.getVersionIhl
        3.8% - 645 ms - 955,110 inv. UnsafeAccess.getByte
          1.1% - 177 ms - 955,110 inv. sun.misc.Unsafe.getByte
          0.6% - 105 ms - 955,110 inv. Utils.signToUnsign
        0.6% - 102 ms - 955,110 inv. UnsafeAccess.setCurrentPointer
    4.1% - 686 ms - 1,910,220 inv. UnsafeAccess.getLong
    0.8% - 138 ms - 955,109 inv. java.util.List.add
    0.7% - 116 ms - 984,958 inv. UnsafeAccess.setCurrentPointer
    0.7% - 111 ms - 955,110 inv. UnsafeAccess.longSize
    0.4% - 75,827 µs - 29,848 inv. java.util.ArrayList.<init>
    0.2% - 41,495 µs - 29,848 inv. DpdkAccess.dpdk_receive_burst
    0.2% - 30,538 µs - 4,331 inv. Ipv6Packet.<init>
    0.1% - 21,899 µs - 29,847 inv. UnsafeAccess.freeMemory
    0.1% - 21,763 µs - 29,848 inv. UnsafeAccess.allocateMemory
    0.1% - 19,866 µs - 29,848 inv. UnsafeAccess.getShort
    0.1% - 9,551 µs - 8,662 inv. Ipv6Packet.getLength
    0.0% - 3,229 µs - 29,848 inv. UnsafeAccess.getOffset
  16.4% - 2,767 ms - 955,136 inv. CaptureProcessor.inspect
```

(a) Object allocation in memory

(b) Call tree of most cpu intensive methods

Figure 22: Profiler output when analysing the first test of packet capture

Figure 22 shows some of the output from the profiler which indicated where the main problems were in terms of memory usage and performance. From this, a number of performance upgrades were implemented:

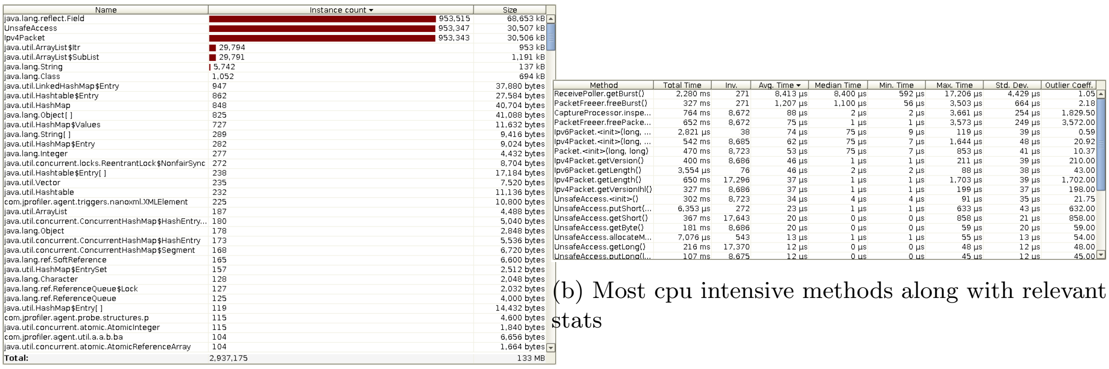
Capture Processor fields The first improvement were to the Packet Capture application itself. The associated processor originally stored the ReceivePoller and PacketFreeer objects within a list to allow for easy iteration if there were numerous objects. Since the Packet Capture application only used 1 of each, seperate fields could be used for the objects which removed the list accessing times and iterations.

Object lifespan The other improvements were made to the actual DPDKJava framework. These involved utilising objects throughout the application lifespan instead of creating new ones on every loop. This dramatically reduced the number of initializing methods invoked for the objects and reduced the memory usage in the heap, which reduced the number of times the Java garbage collector was invoked. The class which caused the most problem with this was the ArrayList, which were created on every loop to pass packets through the Java system. Since the framework uses threads without the need to synchronise objects, an ArrayList could be created on initialisation and simply cleared before been used again.

Off heap allocation Finally, instead of allocating new off heap memory to receive the packet pointers through on every loop, a memory bank was allocated on initialisation and the same memory was simply overwritten on every loop iteration. This stopped expensive calls to the allocate and free methods.

Again, the same tests were carried out to check the performance increase of the new framework implementation. Shown in figure 21 by comparing the blue and green lines, there was a slight performance increase of roughly 1 millions packets per second just by these simple changes, although it was still significantly down on performance to the C version.

Further profiling of the application was undertaken on the improved application. Figure 23 shows the analysis which outlined 2 further major improvements which could be made. Each one of these improvements were undertaken and performance tested individual.



Name	Instance count ▾	Size
java.lang.reflect.Field	953,515	68,653 kB
UnsafeAccess	953,347	30,507 kB
Ipv4Packet	953,343	30,506 kB
java.util.ArrayList$Itr	29,794	953 kB
java.util.ArrayList$SubList	29,791	1,191 kB
java.lang.String	5,742	137 kB
java.lang.Class	1,052	694 kB
java.util.LinkedHashMap$Entry	947	37,880 bytes
java.util.Hashtable$Entry	862	27,584 bytes
java.util.HashMap	848	40,704 bytes
java.lang.Object[ ]	825	41,088 bytes
java.util.HashMap$Values	727	11,632 bytes
java.lang.String[ ]	289	9,416 bytes
java.util.HashMap$Entry	282	9,024 bytes
java.lang.Integer	277	4,432 bytes
java.util.concurrent.locks.ReentrantLock$NonfairSync	272	8,704 bytes
java.util.Hashtable$Entry[ ]	238	17,184 bytes
java.util.Vector	235	7,520 bytes
java.util.Hashtable	232	11,136 bytes
com.jprofiler.agent.triggers.nanoxml.XMLElement	225	10,800 bytes
java.util.ArrayList	187	4,488 bytes
java.util.concurrent.ConcurrentHashMap$HashEntry...	180	5,040 bytes
java.lang.Object	178	2,848 bytes
java.util.concurrent.ConcurrentHashMap$HashEntry	173	5,536 bytes
java.util.concurrent.ConcurrentHashMap$Segment	168	6,720 bytes
java.lang.ref.SoftReference	165	6,600 bytes
java.util.HashMap$EntrySet	157	2,512 bytes
java.lang.Character	128	2,048 bytes
java.lang.ref.ReferenceQueue$Lock	127	2,032 bytes
java.lang.ref.ReferenceQueue	125	4,000 bytes
java.util.HashMap$Entry[ ]	119	14,432 bytes
com.jprofiler.agent.probe.structures.p	115	4,600 bytes
java.util.concurrent.atomic.AtomicInteger	115	1,840 bytes
com.jprofiler.agent.util.a.a.b.ba	104	6,656 bytes
java.util.concurrent.atomic.AtomicReferenceArray	104	1,664 bytes
Total:	2,937,175	133 MB

Method	Total Time	Inv.	Avg. Time ▾	Median Time	Min. Time	Max. Time	Std. Dev.	Outlier Coeff.
ReceivePoller.getBurst()	2,280 ms	271	8,413 µs	8,400 µs	592 µs	17,206 µs	4,429 µs	1.05
PacketFreeer.freeBurst()	327 ms	271	1,207 µs	1,100 µs	56 µs	3,503 µs	664 µs	2.18
CaptureProcessor.inspe...	764 ms	8,672	88 µs	2 µs	2 µs	3,661 µs	254 µs	1,829.50
PacketFreeer.freePacke...	652 ms	8,672	75 µs	1 µs	1 µs	3,573 µs	249 µs	3,572.00
Ipv6Packet.<init>(long, ...	2,821 µs	38	74 µs	75 µs	9 µs	119 µs	39 µs	0.59
Ipv4Packet.<init>(long, ...	542 ms	8,685	62 µs	75 µs	7 µs	1,644 µs	48 µs	20.92
Packet.<init>(long, long)	470 ms	8,723	53 µs	75 µs	7 µs	853 µs	41 µs	10.37
Ipv4Packet.getVersion()	400 ms	8,686	46 µs	1 µs	1 µs	211 µs	39 µs	210.00
Ipv6Packet.getLength()	3,554 µs	76	46 µs	2 µs	2 µs	88 µs	38 µs	43.00
Ipv4Packet.getLength()	650 ms	17,296	37 µs	1 µs	1 µs	1,703 µs	39 µs	1,702.00
Ipv4Packet.getVersionIhl()	327 ms	8,686	37 µs	1 µs	1 µs	199 µs	37 µs	198.00
UnsafeAccess.<init>()	302 ms	8,723	34 µs	4 µs	4 µs	91 µs	35 µs	21.75
UnsafeAccess.putShort(...	6,353 µs	272	23 µs	1 µs	1 µs	633 µs	43 µs	632.00
UnsafeAccess.getShort()	367 ms	17,643	20 µs	0 µs	0 µs	858 µs	21 µs	858.00
UnsafeAccess.getByte()	181 ms	8,686	20 µs	0 µs	0 µs	59 µs	20 µs	59.00
UnsafeAccess.allocateM...	7,076 µs	543	13 µs	1 µs	1 µs	55 µs	13 µs	54.00
UnsafeAccess.getLong()	216 ms	17,370	12 µs	0 µs	0 µs	48 µs	12 µs	48.00
UnsafeAccess.putLong(l...	107 ms	8,675	12 µs	0 µs	0 µs	45 µs	12 µs	45.00

(b) Most cpu intensive methods along with relevant stats

(a) Object allocation in memory

Figure 23: Profiler output when analysing the first test of packet capture

Packet creation Of course, for every packet entering the application a packet object is created for easy referencing further down the application pipe-line. However, not using packet objects would significantly reduce the usability and scalability of applications. It was decided not to alter this. However, for each packet initialization a new UnsafeAccess object was been created for use with accessing packet header information. However, since each thread controls its own packets and therefore can only process 1 packet at a time, 1 UnsafeAccess object could be shared between all packets which significantly reduced the number of objects on the heap. This improvement created a vast performance increase, rising the number from 2.4 million to just over 10 million packets per second as indicated with the red line in figure 21.

Send and Free lists Further problems were identified with the lists of packets awaiting to be sent and freed. This list was been iterated over with the Packet's mbuf pointer then been stored in an off heap memory bank waiting to be freed. This was pointless since the packets mbuf pointer could directly be put into the off heap memory, therefore eliminating the need for the list while also allowing the packet objects to be dereference quicker. Again, this bottleneck was fixed and resulted in a further improvement of roughly 1 million packets per second as indicated with the turqoise line in figure 21.

The results show that the performance was further improved and pretty close to that of the C implementation. After further profiling, there was only 1 obvious improvement which could be made, This would be to replace the list storing packets received from the poller. However, replacing this with a custom implementation using off heap memory would firstly reduce the usability of the code and would also mean that the code was drifting away from the Java language. It was decided not to implement this fix and instead to continue with testing of other applications, assuming that the would be negated by applications. This initial series of testing also proved that a dramatic increase in performance can be achieved simply by programming the applications efficiently, by trying to reduce the number of objects created.

8.2.1 Set-up

8.2.2 Methods

8.2.3 Results

8.3 Software Design

Mention somewhere about the limitations of pktgen

8.3.1 Portability

8.4 Possible Improvement

9 Conclusion

extension to remove packet pointer passing and do it directly from struct array - would have to dynamically pack array though - how????

9.1 Future Work

9.2 extrapolate for other non-native languages

A number of other programming languages such as Visual Basic (.NET) and C# use similar a virtual machine just like Java to make them portable to other machines. This gives the opportunity to extrapolate certain high-level implementation details and apply them to other non-native programming languages, allowing for other application not written in Java to take advantage of fast non-native packet processing.

continue this, where should it go?

10 References

[1] Byte alignment and ordering. http://www.eventhelix.com/RealtimeMantra/ByteAlignmentAndOrdering.htm#.VXSljlxVikp. Accessed: 2015-06-10.

[2] Dpdk - about. http://dpdk.org.

[3] Dzone - heap vs off heap memory usage. http://java.dzone.com/articles/heap-vs-heap-memory-usage. Accessed: 2015-06-14.

[4] Ibm knowledge centre - rdma jverbs. http://www-01.ibm.com/support/knowledgecenter/#!/SSYKE2_7.0.0/com.ibm.java.lnx.71.doc/diag/understanding/rdma_jverbs.html. Accessed: 2015-02-27.

[5] Java code geeks - which memory is faster heap or bytbuffer or direct. http://www.javacodegeeks.com/2013/08/which-memory-is-faster-heap-or-bytebuffer-or-direct.html. Accessed: 2015-06-14.

[6] Java servlets jsp web - jvm architecture. http://www.javaservletsjspweb.in/2012/02/java-virtual-machine-jvm-architecture.html#.VPAQAkJrjKA. Accessed: 2015-02-27.

[7] Javolution. http://javolution.org/. Accessed: 2015-06-14.

[8] Jenkov - nio buffers. http://tutorials.jenkov.com/java-nio/buffers.html. Accessed: 2015-06-14.

[9] Jna - java naive access. https://github.com/twall/jna. Accessed: 2015-06-14.

[10] Kdgregory - bytebuffer. http://www.kdgregory.com/?page=java.byteBuffer. Accessed: 2015-06-14.

[11] Kernel.org - kernel hugepages. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt. Accessed: 2015-06-14.

[12] Linux journal - queueing linux network stack. http://www.linuxjournal.com/content/queueing-linux-network-stack?page=0,0.

[13] Mind prod - bytebuffer. http://mindprod.com/jgloss/bytebuffer. Accessed: 2015-06-14.

[14] Nec europe network laboratories - middlebox traversal. http://www.neclab.eu/Projects/midcom.htm. Accessed: 2015-02-27.

[15] Random paper. http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1409136&url=http%3A%2F2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1409136. Accessed: 2015-06-10.

[16] Redhat - memory hugepages. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html. Accessed: 2015-06-14.

[17] Route my world - ipv6. http://routemyworld.com/2009/02/01/bsci-ip-version-6/. Accessed: 2015-02-27.

[18] Structure packing. http://www.catb.org/esr/structure-packing/. Accessed: 2015-06-10.

[19] Toms hardware - what core thread. http://www.tomshardware.co.uk/forum/306079-28-what-core-thread. Accessed: 2015-06-14.

[20] When cpu affinity matters. http://bitsum.com/pl_when_cpu_affinity_matters.php. Accessed: 2015-06-10.

[21] Wikipedia - firewall(computing) (png image). http://en.wikipedia.org/wiki/Firewall_(computing)#mediaviewer/File:Firewall.png. Accessed: 2015-02-27.

[22] Windows it pro - what types of network address translation (nat) exist? http://windowsitpro.com/networking/what-types-network-address-translation-nat-exist. Accessed: 2015-02-27.

[23] Wim H. Hesselink. A crossing with java threads and posix threads. *Science of Computer Programming*, 2011.

[24] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, 2012.

[25] Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. jverbs: ultra-low latency for data center applications. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013.

[26] Guillermo L Taboada, Sabela Ramos, Roberto R Exposito, Juan Tourino, and Ramon Doallo. Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013.

[27] Guillermo L Taboada, Juan Touriño, and Ramón Doallo. Java fast sockets: Enabling high-speed java communications on high performance clusters. *Computer Communications*, 31(17):4049–4059, 2008.

11 User Guide

should I be putting all data in appendix

http://plvision.eu/blog/deploying-intel-dpdk-in-oracle-virtualbox/ https://www.virtualbox.org/manual/ch08.html