*Todo list

Things to talk about

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

References

- http://www.catb.org/esr/structure-packing/
- http://www.eventhelix.com/RealtimeMantra/ByteAlignmentAndOrdering.htm#.VXSljlxVikp
- http://bitsum.com/pl_when_cpu_affinity_matters.php

Imperial College London

Individual Project

Computing - BEng

Packet Processing in Java

Ashley Hemingway

Supervisor:
Peter Pietzuch

2nd Marker:
Wayne Luk

June 2015

Abstract

Acknowledgements

Contents

1 Introduction

1.1 Motivation

1.2 Objectives

1.3 Solution Idea

2 Background

2.1 Network Components

2.1.1 Network Models

Generally there are 2 well known network models. The Open System Interconnection (OSI) model 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 TCP/IP layer simply because not all systems will go through these exact stages depending on the use of the system.

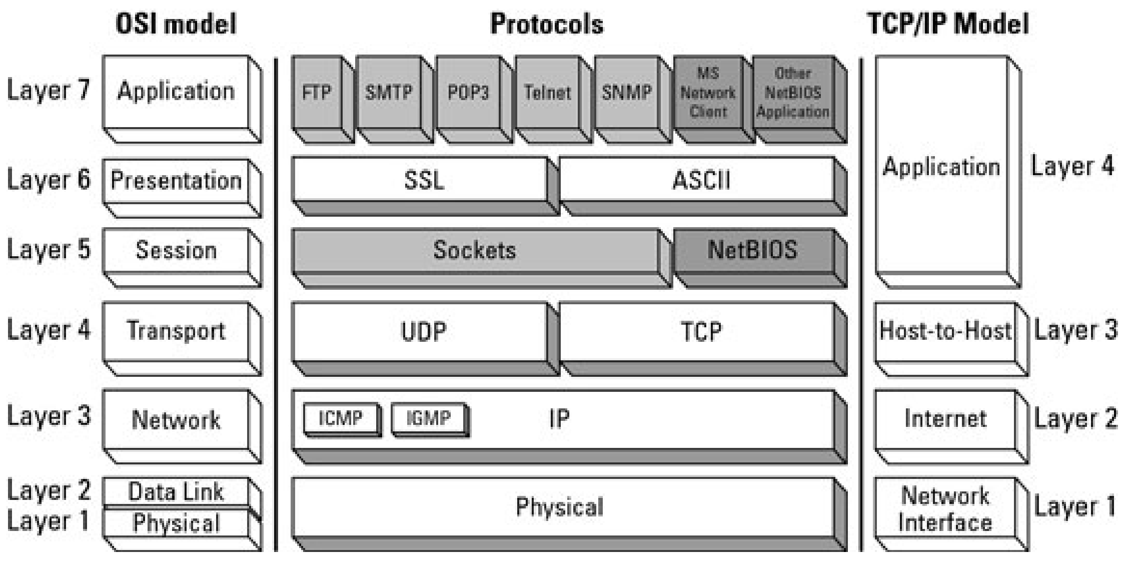

Figure 1: OSI vc TCP/IP Model

cite this

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.

ref image and source

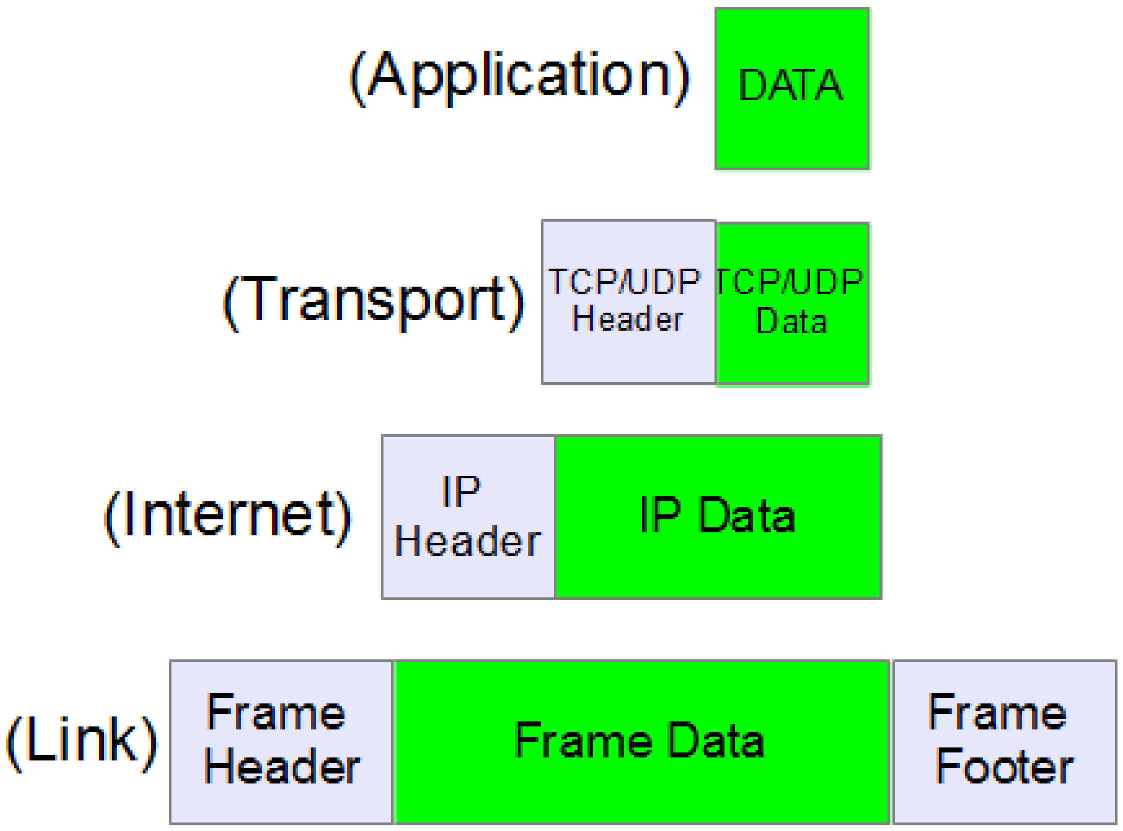

Figure 2: Network model headers [?]

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 . 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 and IPv6 packet headers.

ref this

update software for ipv6

ref packet headers below

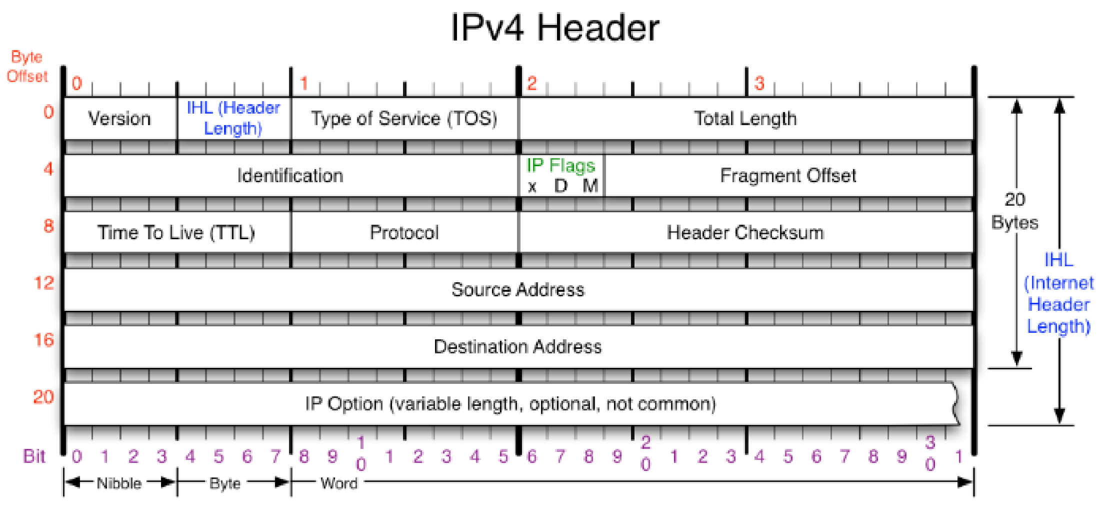

Figure 3: IPv4 Packet Header [?]

- 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.

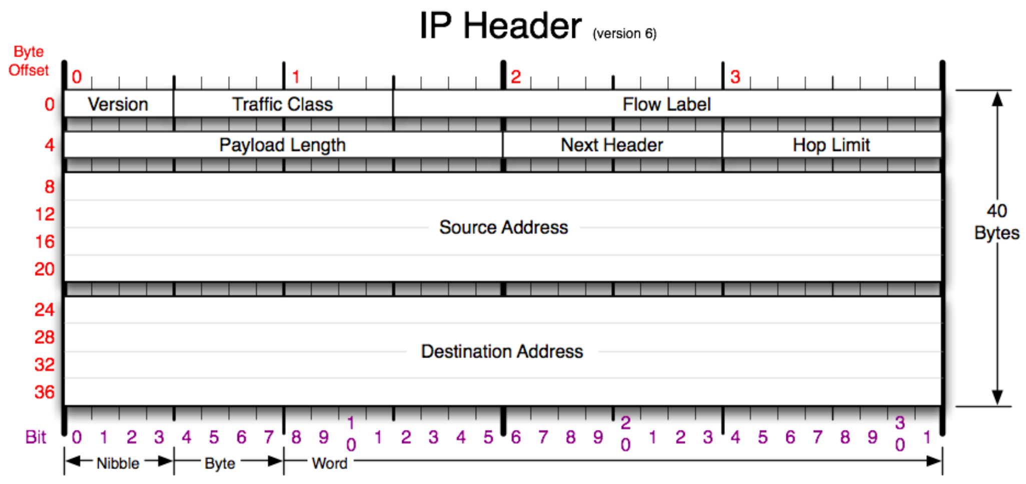

Figure 4: IPv6 Packet Header [?]

- 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

2.1.4 Network Interface Card (NIC)

2.1.5 Network Address Translator (NAT)

2.1.6 Firewall

2.2 Java Features

2.2.1 Java Virtual Machine (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 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 5 shows the basic JVM architecture with the relevant sections explained in the list below.

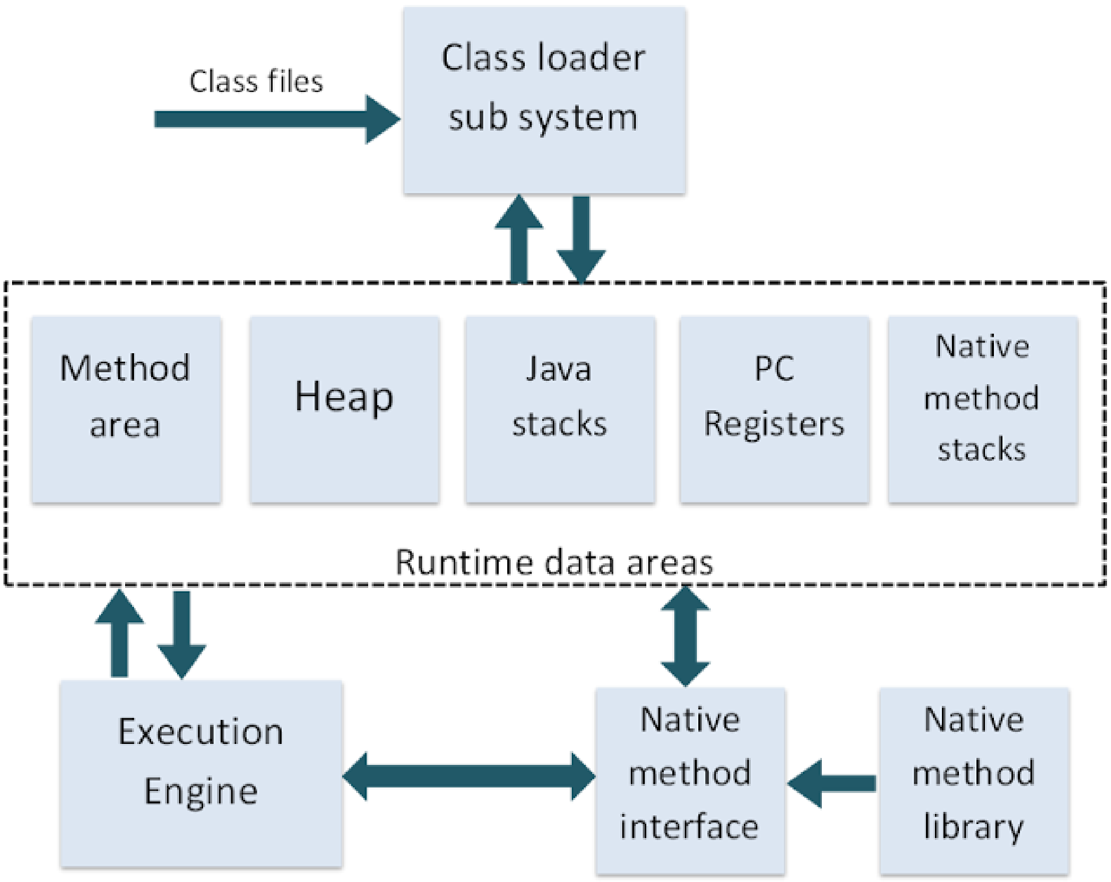

Figure 5: Java Virtual Machine interface [?]

- 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.

```java
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

```c
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

2.2.3 Current Java Networking Methods

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.

2.2.3.1 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 [?] 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.

2.2.3.2 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 2.2.3.1.

2.2.3.3 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 jVerbs

Ultra-low latency for Java applications has been partially solved by the jVerbs [?] 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 6), 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.

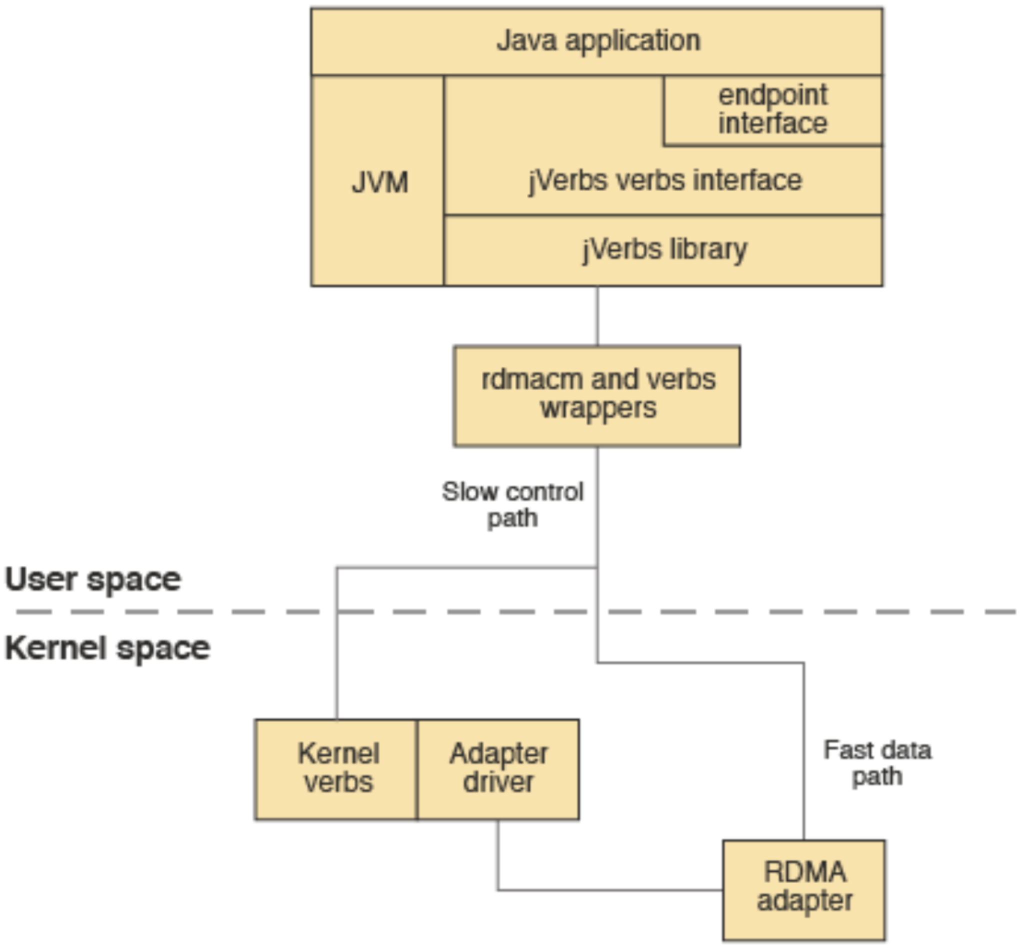

Figure 6: jVerbs architecture - shows how the framework bypasses the kernel and JVM [?]

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.

2.4 Native I/O API's

Currently available native networking API's are capable of reading and writing packets to the NIC transmission and receive queues at line rate. This is due to a number of techniques which tend to alter the kernels understand of the underlying NIC, therefore requiring specialist hardware and software to use such tools. DPDK (Section 2.4.1) is one of the tools that is open source and publicly available for use.

2.4.1 Data Plane Development Kit (DPDK)

Data Plane Development Kit (DPDK) [?] is a set of libraries and drivers which enabled fast packet processing with certain system set ups. Since DPDK is developed by Intel, it only supports Intel x86 CPU's and certain network interface controllers (NIC). DPDK overwrites the NIC's drivers meaning that the operating system doesn't recognise the network cards and can't interact with them. It installs its own drivers allowing it to interact with certain memory locations without permission from the kernel or even involving it in any way.

refer to in-depth section - also is this right?

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.

2.5 Benchmarking

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

Examples of hard-ware com-parisons

be drawn between similar software or hardware with different implementations in order to derive a better product.

better word for derive

why I need benchmarking

2.5.1 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 of JIT used.

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

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

2.5.1.1 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:

better phrasing off that sentence needed

More here

get paper

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 1: Results from Loop Recognition benchmarking

explain difference with GC etc

2.5.1.2 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

2.5.1.3 Using Economics

About economics paper

2.5.1.4 Which is better?

2.5.2 Intra-Language Techniques

2.5.3 Applications

3 Initial language comparison

Before any implementation or specific design considerations were undertaken, an evaluation of the performance of C, Java and Java using 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.

3.1 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 return 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

```
 1: function Main
 2:     for i = 1 to 100000 do
 3:         $p \leftarrow$ Initialise Packet
 4:         PopPacket(p)
 5:         ProPacket(p)

 6: function PopPacket(Packet p)                    ▷ Set data in a packet
 7:     $p.a \leftarrow randomInt()$
 8:     $p.b \leftarrow randomInt()$
 9:     $p.c \leftarrow randomInt()$
10:     $p.d \leftarrow randomInt()$
11:     $p.e \leftarrow randomInt()$

12: function ProPacket(Packet p)                    ▷ Process a packet
13:     $res \leftarrow p.a * p.b * p.c * p.d * p.e$
14:     $p.result \leftarrow res$
```

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.

3.2 Results

Each language had the algorithm run 1000 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

3.3 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

4 Design Considerations

4.1 Data Sharing

The proposed application will be sharing data between the DPDK code written in C and the Java side used for the high functionality 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.

Diagram of packets from NIC using c through 'technique' and then processing packets in java and then back

A few techniques for this are available with Java and C, all with different performances and ease-of-use.

4.1.1 Objects and JNI - using heap and lots of jni calls

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 via this. 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 via the Java environment parameter. 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

4.1.2 ByteBuffers - Non-heap and heap memory

ByteBuffers are a Java class which allow for memory to be allocated on the Java heap (non direct) or outside of the JVM (direct). Non direct ByteBuffer's are simply a wrapper for a byte array on the heap and are generally used as they allow easier access to bytes, as well as other primitive data types.

Direct ByteBuffers allocate memory outside of the JVM in native memory. This firstly means that the only limit on the size of ByteBuffers is memory itself. Furthermore, the Java garbage collector doesn't have access to this memory. Direct ByteBuffers have increased performance since the JVM isn't slowed down by the garbage collector and intrinsic native methods can be called on the memory for faster data access.

4.1.3 Java Unsafe - non-heap

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. It can be used for a number of things such as:

- Object initialisation skipping
- Intentional memory corruption
- Nullifying unwanted objects
- Multiple inheritance
- Dynamic classes
- Very fast serialization

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

```java
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 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.

4.1.4 Evaluation

4.1.5 JNA?

4.1.6 Packing C Structs

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.

```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 11 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

Example making sure compiler doesn't pad

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 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 11: Example C Struct with compiler inserted padding

On an x86 system (which does not enforce memory access alignment), this will produce

y=2, *py=2

as expected. On the other hand on my ARM Linux board, for example, it produced the seemingly wrong result

y=2, *py=1

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; theyre equally expensive from anywhere they live inside a single machine word. Thats why they dont 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.

4.1.7 Javolution

4.1.8 Performance testing

4.1.9 Thread affinity

Thread.currentThread().getId(); just gets id of thread relative to jvm.

It keeps a process limited to certain a certain core or cores. Process will still be taken out of use and switched back in but without the problem of moving cache between cores.

Normally as a thread gets a time slice (a period in which to use the core), it is granted whichever core [CPU] is determined to be most free by the operating system's scheduler. Yes, this is in contrast to the popular fallacy that the single thread would stay on a single core. This means that the actual thread(s) of an application might get swapped around to non-overclocked cores, and even underclocked cores in some cases. As you can see, changing the affinity and forcing a single-threaded CPU to stay on a single CPU makes a big difference in such scenarios. The scaling up of a core does not happen instantly, not by a long shot in CPU time.

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).

core thrashing - ust by the name, you know this is a bad thing. You lose performance when a thread is swapped to a different core, due to the CPU cache being 'lost' each time. In general, the *least* switching of cores the better. One would hope the OS would try to avoid this, but it doesn't seem to at all in quick tests under Windows 7. Therefore, it is recommended you manually adjust the CPU affinity of certain applications to achieve better performance.

Another important issue is avoiding placing a load on a HyperThreaded (non-physical) core. These cores offer a small fraction of the performance of a real core. The Windows scheduler is aware of this and will swap to them only if needed. As of mid Jan 2012 the Windows 7 and Windows 2008 R2 schedulers have a hotfix for AMD Bulldozer CPUs that see them as HyperThreaded, cutting them down from 8 physical cores to 4 physical cores, 8 logical cores. This is for two reasons: The AMD Bulldozer platform uses pairs of cores called Bulldozer Modules. Each pair shares some computation units, such as an L2 cache and FPU. To spread out the load and prevent too much load being placed on two cores that have shared computational units, the Windows patch was released, boosting performance in lightly threaded scenarios.

Processor affinity takes advantage of the fact that some remnants of a process that was run on a given processor may remain in that processor's memory state (for example, data in the CPU cache) after another process is run on that CPU. Scheduling that process to execute on the same processor could result in an efficient use of process by reducing performance-degrading situations such as cache misses. A practical example of processor affinity is executing multiple instances of a non-threaded application, such as some graphics-rendering software.

put code showing cpu affinity assignment

In Linux, Java thread uses the native thread(i.e, thread provided by Linux). This means the JVM creates a new native thread when the Java code creates a new java thread. So, the Java threads can be organised in any way the native threads can be organised.

A native thread can be bound to a core through the sched_setaffinity() function. So, a Java thread can be bound to a core. If Java standard library does not provide a function to do so, then this function need to be provided through JNI.

In Linux, multi-threading is same as parallel threading. Linux kernel distribute threads among processors to balance the cpu load. However the individual threads can be bound with any core as wished. So, in Linux Java multi-threading is same as parallel threading.

5 General Implementation

5.1 Shared Libraries

5.2 Handling Errors

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.

mention applications below for demonstrations of ease of use of framework, also mention only subset of availability of dpdk, and mention further extensions easy to do like packet fragmentation etc

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

6.1 Overview (better name?)

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.

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.

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

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.

example?

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 apposed 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.

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.

get image of setup of memory location, ie number then list of packet pointers

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.

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) 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.

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, multicasting or ordering.

7.1 Firewall

7.2 Network Address Translator (NAT)

8 Evaluation

9 Conclusion

10 User Guide