1 Background

1.1 Network Components

1.1.1 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.

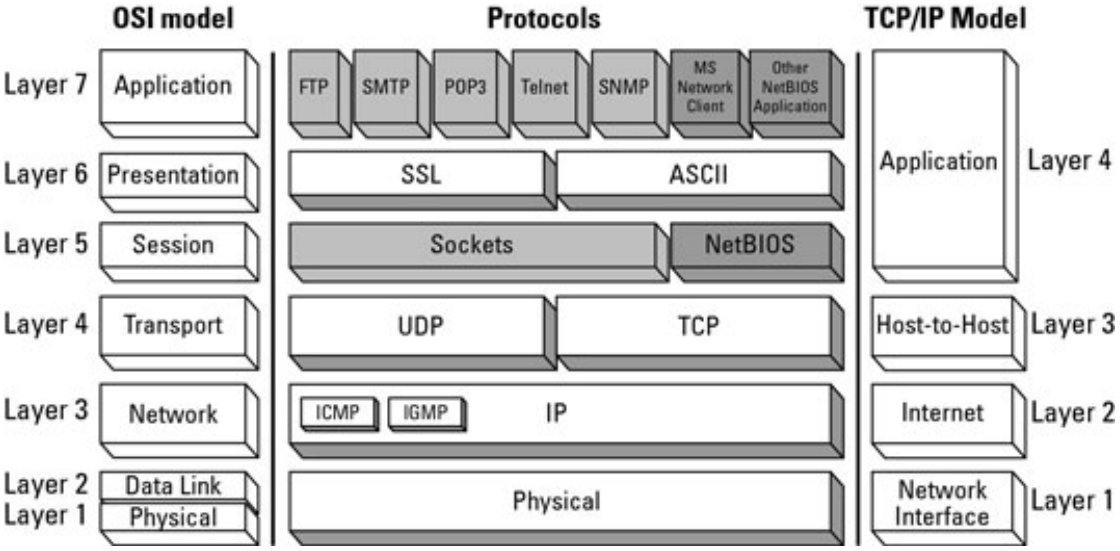

Figure 1: OSI vc TCP/IP Model

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

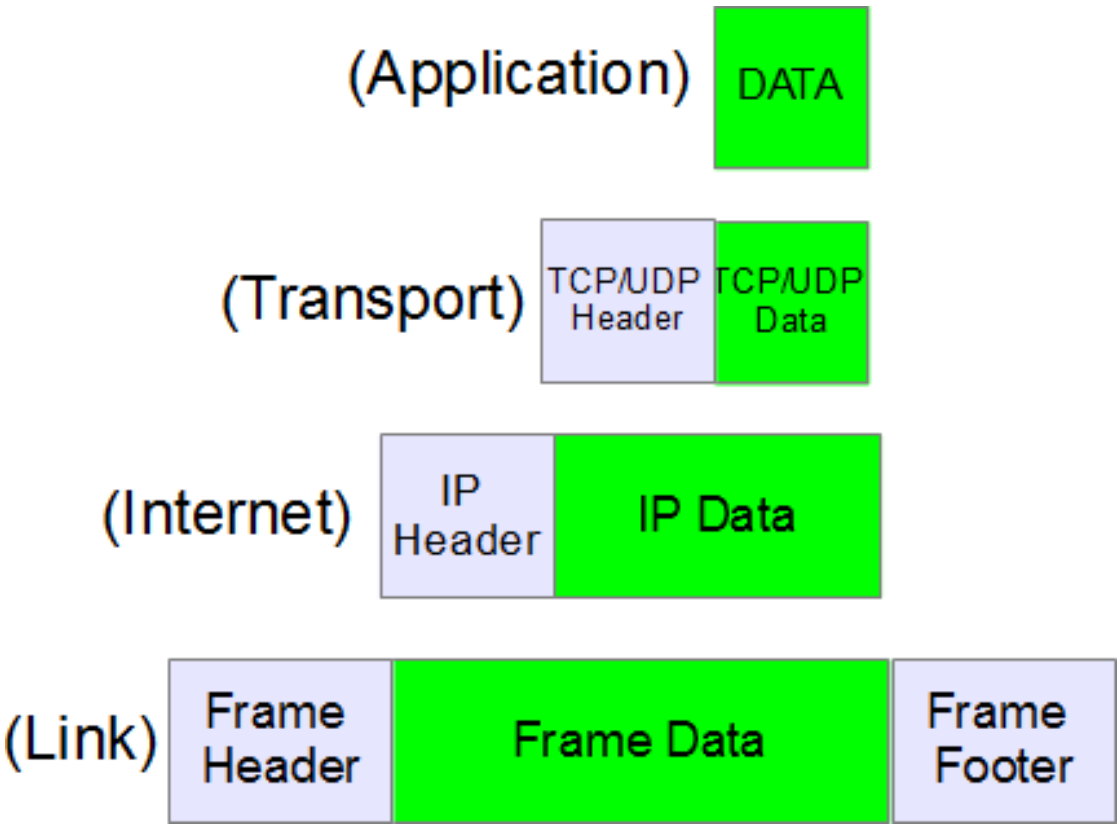

Figure 2: Network model headers [**?**]

1.1.2 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

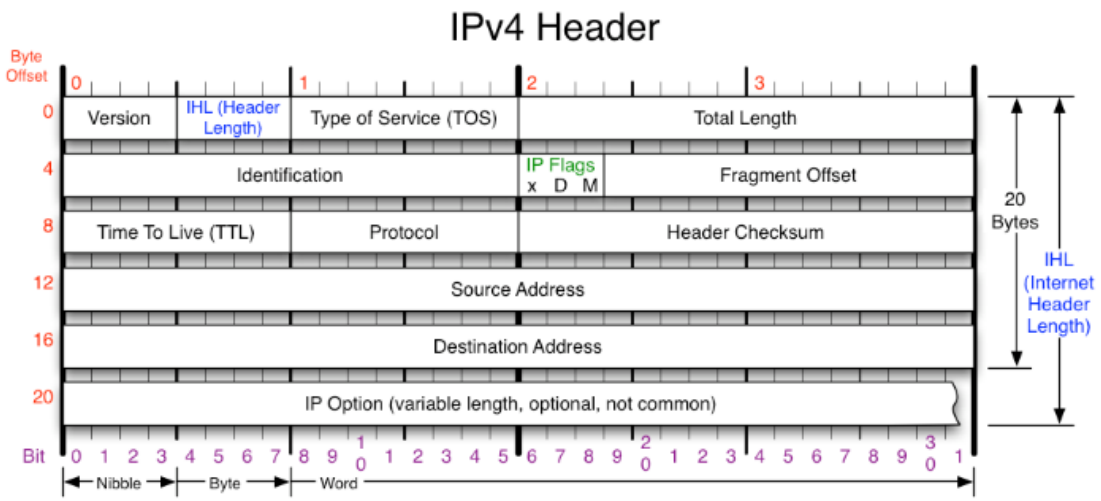

Figure 3: IPv4 Packet Header [?]

- 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.

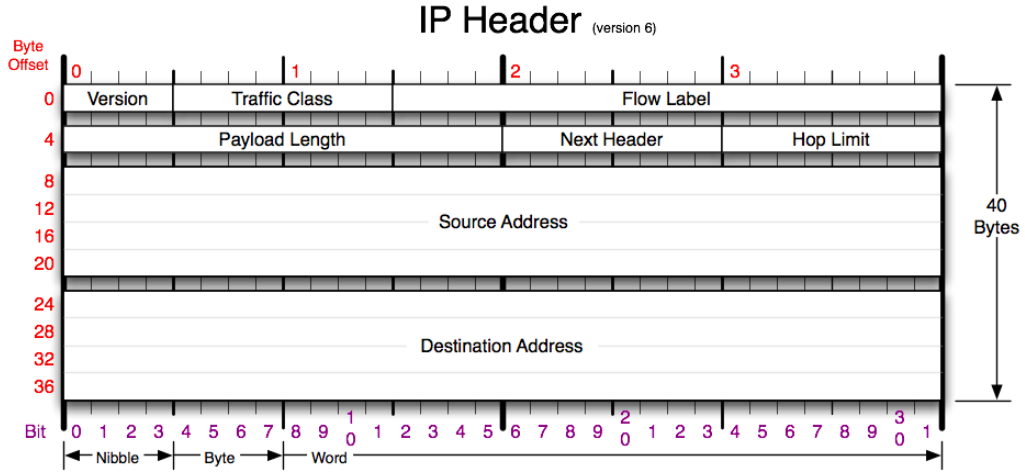

Figure 4: IPv6 Packet Header [?]

- 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.

1.1.3 Packet Handling

1.1.4 Network Interface Card (NIC)

1.1.5 Network Address Translator (NAT)

1.1.6 Firewall

1.2 Java Features

1.2.1 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.

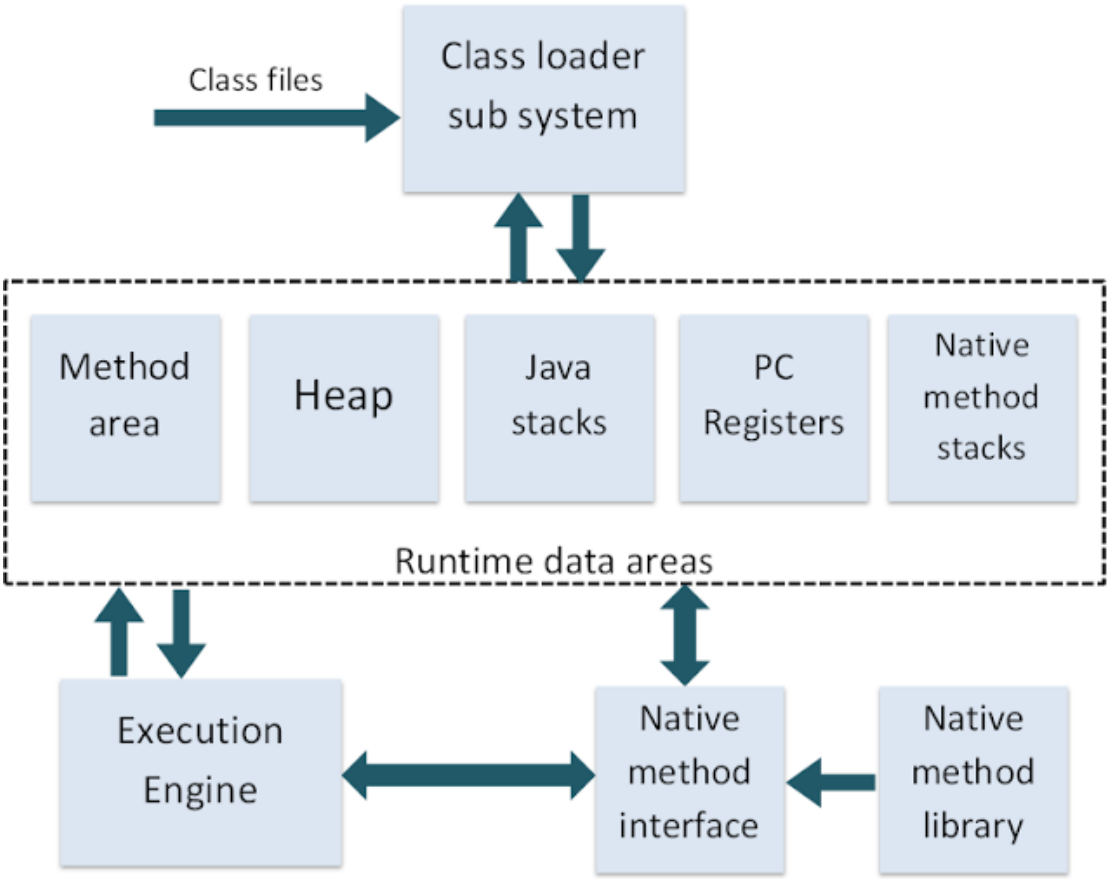

Figure 5: Java Virtual Machine interface [**?**]

- 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.

1.2.2 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.

```
1 public class Jni {
2
3   int x = 5;
4
5   public int getX() {
6     return x;
7   }
8
9   static { System.loadLibrary("jni"); }
```

```
10
11    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.

```
1 $ 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.

```
1 $ 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.

```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class Jni */
4
5  #ifndef _Included_Jni
6  #define _Included_Jni
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Class:     Jni
12  * Method:    objectPrint
13  * Signature: (LJni;)V
14  */
15 JNIEXPORT void JNICALL Java_Jni_objectPrint
16   (JNIEnv *, jclass, jobject);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #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.

```
1 $ 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.

```
1 $ java -Djava.library.path="." Jni
2 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

1.2.3 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.

1.2.3.1 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.

1.2.3.2 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 1.2.3.1.

1.2.3.3 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.

1.3 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.

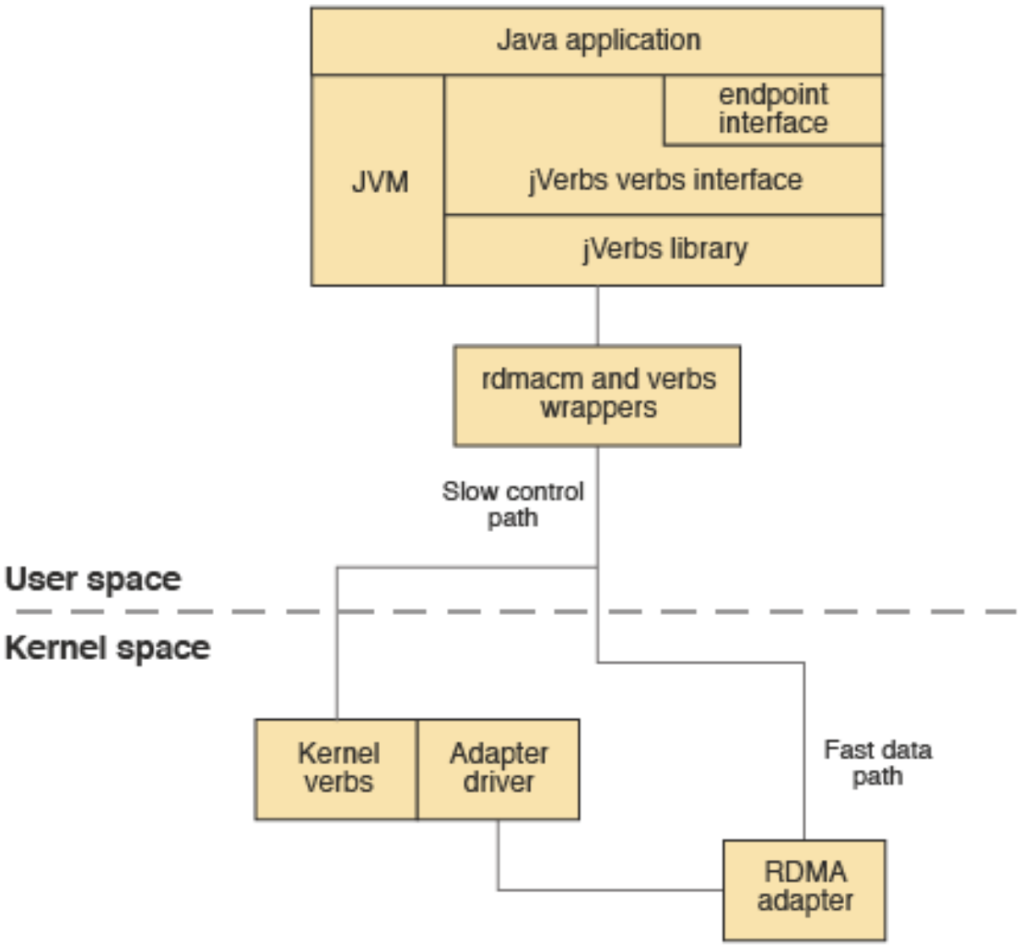

Figure 6: jVerbs architecture - shows how the framework bypasses the kernel and JVM [**?**]

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.

1.4 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 1.4.1) is one of the tools that is open source and publicly available for use.

1.4.1 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.

1.5 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 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

1.5.1 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 problem-

atic. Other factors can be introduced when deciding on the optimisation level and the compiler of JIT used.

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

1.5.1.1 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:

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

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

better phrasing off that sentence needed

More here

get paper

explain difference with GC etc

1.5.1.2 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

1.5.1.3 Using Economics

About economics paper

1.5.1.4 Which is better?

1.5.2 Intra-Language Techniques

1.5.3 Applications