# 1 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

## 1.1 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.

## 1.2 Native Libraries

### 1.2.1 Library Compilation Tools

## 1.3 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.

## 1.4 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

## 1.5 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

## 1.6 Packet Polling

Packet polling as the act of receiving data

## 1.7 Packet Sending

## 1.8 Statistic Profiling