1 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.

1.1 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

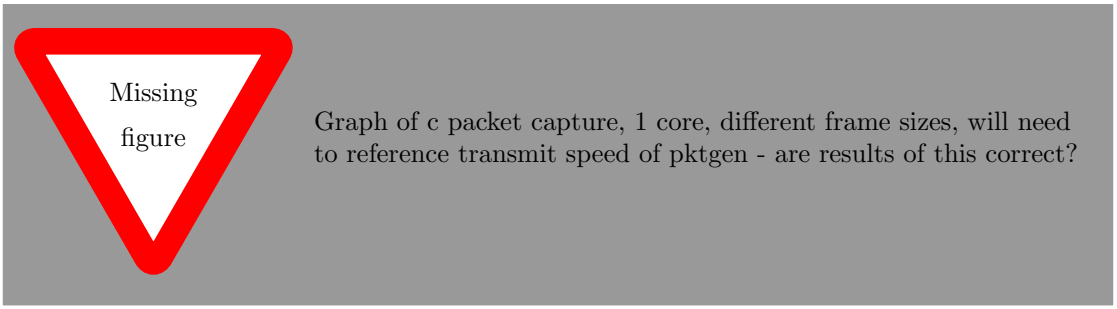
Pktgen's transmitting performance can be reduced depending on the frame size. This was negated to keep it a constant within the tests by running the application with multiple lcores on the same port with different queues, which allow the transmitting speed to match those of the NIC. With the packet size varying, obviously the number of packets transmitted per second is dependant on this. The differences are shown in Table 1.

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

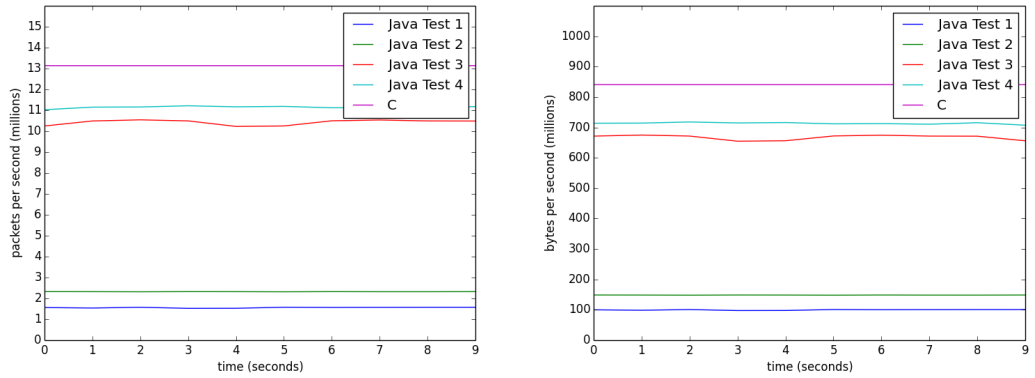
1.2 Further Testing

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.



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 **??**. 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

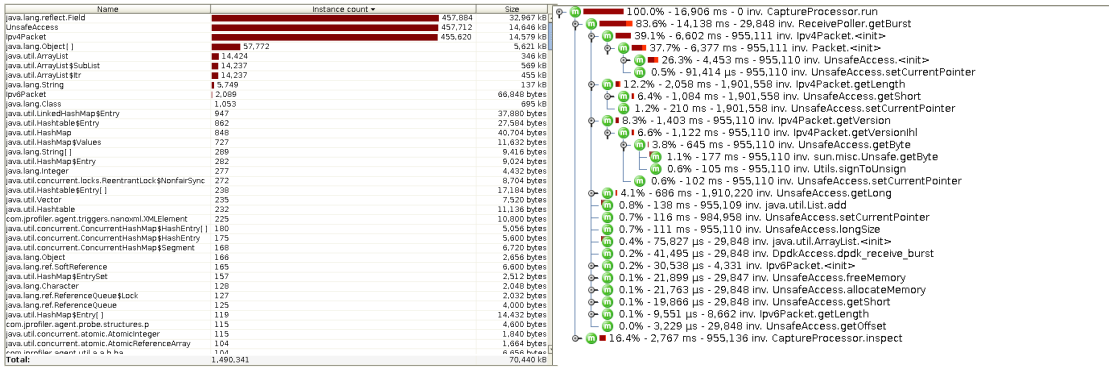
(b) Millions of bytes per second

Figure 1: Performance results of C vs Java implementations of packet capture application for all rounds of testing

different frame sizes?

Figures 1a and 1b 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.



(a) Object allocation in memory

(b) Call tree of most cpu intensive methods

Figure 2: Profiler output when analysing the first test of packet capture

Figure 2 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:

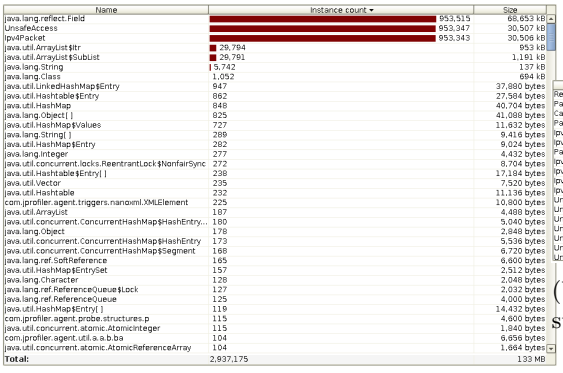
Capture Processor fields The first improvement were to the Packet Capture application itself. The associated processor originally stored the ReceivePoller and PacketFreeer objects within a list to allow for easy iteration if there were numerous objects. Since the Packet Capture application only used 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 1 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 3 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	1,161 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	41,098 bytes
java.util.HashMap$Values	727	11,632 bytes
java.lang.String[ ]	289	9,416 bytes
java.util.HashMap$Entry	292	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

(a) Object allocation in memory

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	3,681 µs	254 µs	1,826.50
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(...	6,359 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...	7,078 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(I...	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

Figure 3: 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 1.

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 1.

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.

1.2.1 Set-up

1.2.2 Methods

1.2.3 Results

1.3 Software Design

Mention somewhere about the limitations of pktgen

1.3.1 Portability

1.4 Possible Improvement