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.

mention machine specs

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 64 byte frame sizez, 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

how does pktgen work and why we used it

Pktgen has limitations though, since 10Gbps transmitting is only possible with 64 byte frames. Frames of greater size up to 1024 bytes can be transmitted although total bytes per seconds is reduced, therefore limiting the testing ability.

possibly run more ports on more than 1 core?

1.2 Initial Testing

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

which ones?

ref this and ubuntu

1.2.1 Set-up

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

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

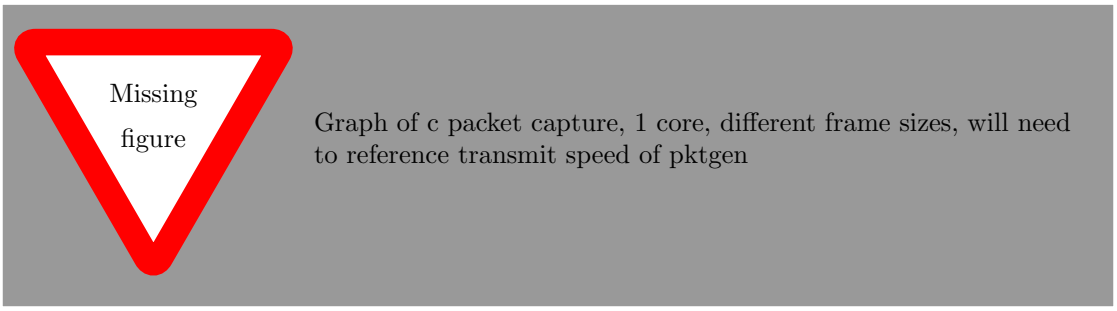
1.2.2 Methods

1.2.3 Results

1.3 Further Testing

In order to fully understand the capabilites of the middleboxes, testing was carried out on Imperial College's Large Scale Distributed Systems (LSDS) testbed. Although this system consists of numerous machines, tests were carried out using just 2. The first machines was used to host the middlebox application and received 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.

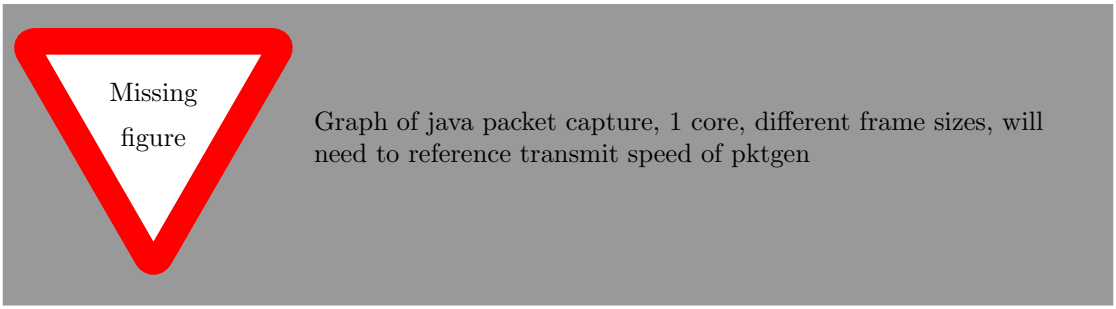
The first test on the LSDS testbed involved comparing the C and Java implementations of a packet capturing application which simply received the packets and freed them straight away without forwarding them. The C implementation is used to give the optimal readings possible from this and further tests, 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.



Graph of c packet capture, 1 core, different frame sizes, will need to reference transmit speed of pktgen

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 descirbed previously. These results are shown below.

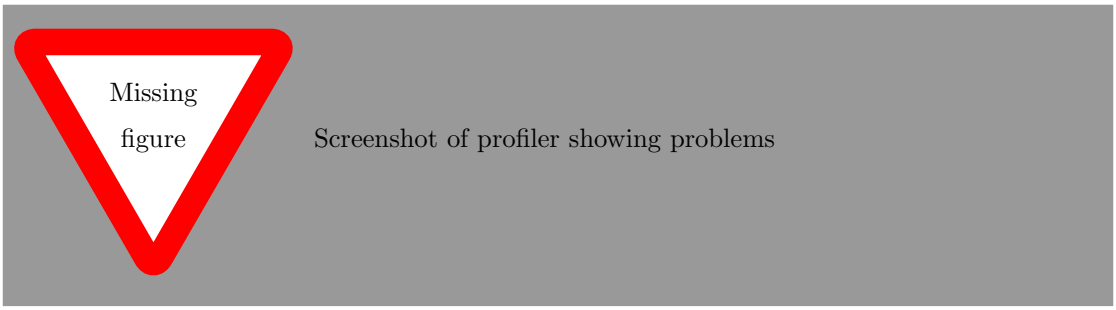


Graph of java packet capture, 1 core, different frame sizes, will need to reference transmit speed of pktgen

The results above are very low to what was expected for such a simple application.

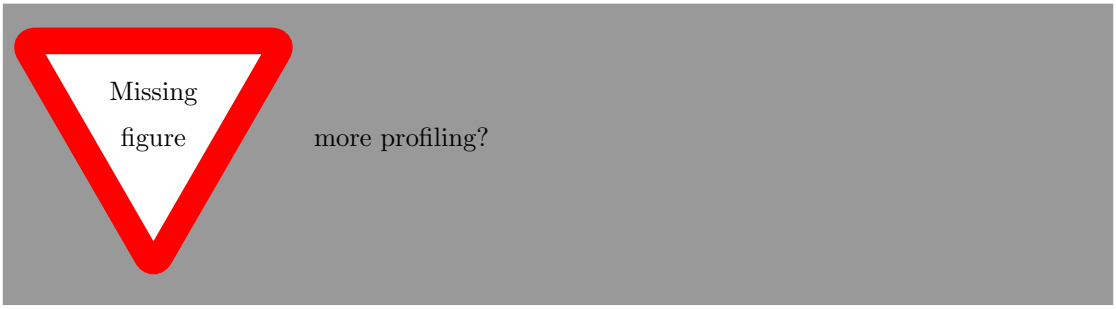
expand on this

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 avergae time per method call. This provided invalueable analysis of where the problems were. As can be seen by figure ??, the major problem lied with???



Screenshot of profiler showing problems

The first improvements were to the PacketCapture application itself, by storing the ReceivePoller objects within individual fields instead of lists. This removed the unnessary list accessing, especially since the application was only using a single receive poller per processor. 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. 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. These improvements resulted in the graph below.



explain results compated to previous ones and c version

Again, the results didn't fully reflect the expected results. Further profiling was done to identify more bottlenecks within the application. This showed that for every Packet creation (which was unavoidable), there was also a new UnsafeAccess object been initialized. 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 signifantly reduced the number of objects on the heap. 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 elminiating the need for the list while also allowing the packet objects to be dereference quicker.

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 implemenation 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 impplement 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.

1.3.1 Set-up

1.3.2 Methods

1.3.3 Results

1.4 Software Design

Mention somewhere about the limitations of pktgen

1.4.1 Portability

1.5 Possible Improvement