EEN1011 Project Design Plan

# Purpose

The purpose of this project is to investigate the possibilities to improve speed and processing network packets operating over high-speed modern network interfaces by monitoring the performance of packet processing in the Linux kernel and optimizing it.

# Project Scope

Up to now, the Internet has developed and is developing exponentially, and with it, increases the number of network applications. However, this has also led to the increase in need for speed and high performance for the networking subsystem of the Operating System (OS).

The Hardware (HW) has advanced in the past decade. This includes the increase of speed of the Network Interface Card (NIC), modern NICs supporting multiple RX and TX hardware queues such as RSS and flow direct, and the increase of available cores for the CPUs. This improved the rate of transmitting data using the physical link.

However, there are several drawbacks in the processing of packets. In user space, a socket API summaries a network communication channel, using the kernel TCP/IP stack underneath. In networking, sockets are created, initialized, connected, then wait for a connection, and in the end, close.

Packet processing involves 2 main mechanisms:

* Polling: CPU checks another device’s status to determine whether or not the device needs attention. It involves a loop in which queries are sent to the device and awaiting a response repeatedly until the loop ends with the obtained data or preferred outcome.
* Interruption: the device informs the CPU its need of attention by sending an interrupt signal to the CPU when an event occurs or when data is ready. This causes the CPU to halt any activities it does currently and attend to the device that interrupted it. Traditional packet processing involves interruption.

Problems associated with packet processing:

* High Interruption Frequency problem: Instead of concentrating on processing packet, the processor would concentrate more on handling interruption due to the high rate of incoming packets.
* Cross-Core Scheduling problem: transmitting a packet from one core’s local cache to the local cache of another core.
* Uneven Distribution problem: when the number of tasks is spread unevenly among the cores. Some cores process much more tasks than the other cores.

Several techniques were developed to speedup packet processing:

* Batch processing: instead of one packet per system call, a group or batch of packets can be processed. Used in Netmap, DPDK and Netslices. Reduces the number of system calls.
* Zero-copy: Zero-copy is a technique used to avoid copying packets from kernel space to user space. While Netmap, PF RING and DPDK use it, however, NAPI, BPS and Netslices still copy packets.
* Parallelism: used in Netslices, where CPU cores and NIC queues are divided into slices. Parallelism is also included in DPDK.
* Polling: polling is used to keep both the processors and threads active when the rate of incoming packets is low, in order to avoid interrupt processing overhead. Widely used within DPDK.
* Direct Cache Access (DCA): transmits packets directly to the cache. The processor then reads it not from the memory, but from the cache, reducing the memory access frequency and make processing more efficient.

There are various solutions such as NAPI, Busy Poll Socket, Netmap and DPDK.

NAPI

It is packet-processing API that involves both polling and interruption. The main idea of NAPI is that instead of handling one packet for each interruption, take multiple. It handles a batch of packets for each interruption instead of for each packet and reduces the frequency of high interruption, giving CPU more time to process other packets. However, at low rate of packet arrival, the frequency of cores sleeping increases. This increases the time for cores to wake up when new packets arrive and increase packet processing latency.

BPS

BPS (**B**usy **P**oll **S**ocket) is an improvement to NAPI in regards to processor cores sleep frequency. Threads do not fall asleep often and constantly poll. Its disadvantage – excessive polling when there no incoming packets – wasted CPU cycles.

DPDK

DPDK (**D**ata **P**lane **D**evelopment **K**it) consists of libraries for fast and efficient data packet processing functions. Environment Abstraction Layer (EAL) is used to gain access to hardware directly from user-space. The Poll Mode Drivers (PMD) deliver packets straight to user programs. Its advantage is that the user-space I/O prevents encountering the massive process of interruption and copying data between the two spaces, kernel and user. Its disadvantage – complete kernel bypass – prevents processing services and the security guarantees which are provided by the kernel protocol stack.

XDP

The next solution is XDP (E**x**press **D**ata **P**ath), an eBPF-based high-performance network data path which enables custom packet processing at the lowest level of the kernel network stack. XDP works by allowing user-defined programs to be loaded into the kernel, which can then process incoming packets before they reach the networking stack. This is achieved using the eBPF (extended Berkeley Packet Filter).

# Design Approach

XDP is chosen as the most efficient solution for this project and for numerous reasons:

* Speed and Low Overhead: Operates directly in the NIC driver, bypasses much of the kernel’s overhead, enabling faster packet processing.
* Customizability: Allows developers to create custom packet-processing programs with eBPF, providing more flexibility and granularity than legacy tools like iptables.
* Resource Efficiency: Does not require dedicating entire CPU cores to packet processing, unlike user-space solutions like DPDK.
* Kernel Integration: XDP works within the Linux kernel, allowing seamless interaction with the existing kernel network stack and tools.

# Detailed Timeline

The next steps are to explore XDP usages to speedup early-stages network processing. Create (or reuse) an XDP based input packet filter (similar to Linux iptables INPUT tables).

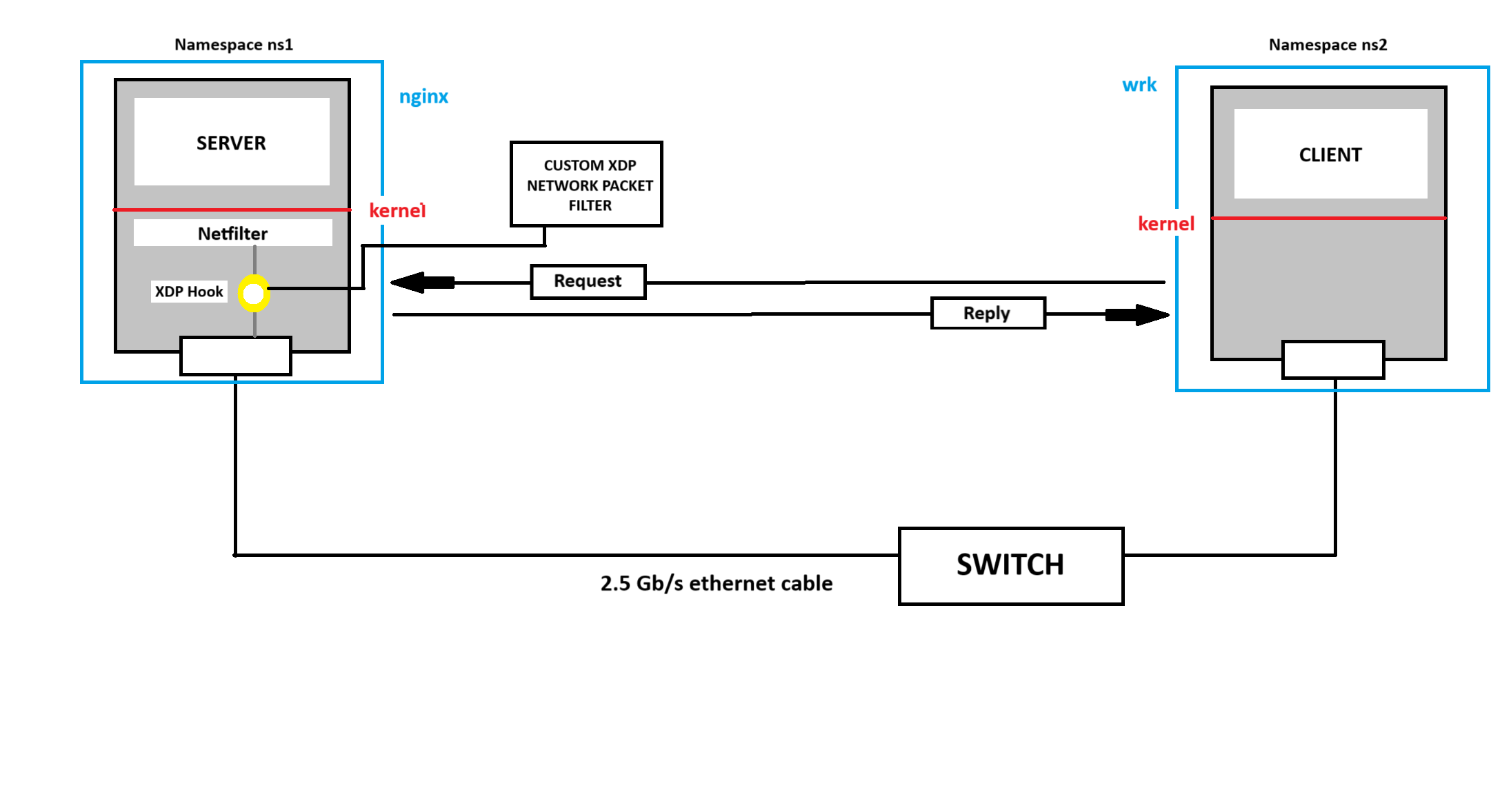
Next is to examine its performance to see how feasible it is to achieve 2.5 Gb/s line-speed and to compare with conventional iptables performance. Investigate impact of number of filter rules (and its complexity) on overall performance. Try to optimize, if necessary, its performance for high number of rules (10K+)

## Milestones

* Setup testing and development environment in the campus lab
  + 2-4 2.5Gb/s NICs (PCIe cards) required.
* Get familiar with XDP and implement XDP based packet filter.
* Measure system performance for different set of rules.
* Investigate optimisation opportunities and provide final report.

## Timeline

* Task 1: Test Setup
  1. First it is vital to setup 2.5 Gb/s NICs, configure them within Linux kernel and make sure they work properly. In particular, it must be ensured that packets are flowing between the two
  2. To test the end-to-end user experience, I plan to use the http server/client scenario. The plan is to use nginx[[1]](#nginx), as HTTP web server serving static content. The wrk[[2]](#wrk), a tool generating load, will be used as a client to generate a load of packets sent to the server.
  3. For measuring performance, I plan to use
     1. Throughput (Gb/s, Query-per-Second (QPS))
     2. Average Query Latency (ms)
  4. To generate this maximum load on the server, I plan to use small size request/response over short-lived connections.



* 1. If this set could not provide enough load for testing, it might be possible to use small UDP packets. A memcached server[[3]](#memcache), might be a good fit for it due to its high performance in handling a large number of requests simultaneously. By caching data in memory, it allows rapid access and reduces latency.

As alternative scenarios for generating high network load might switch my test suite to use other network services such as FTP server, video streaming server, etc.

* Task 2: Define Netfilter Rules
  1. First it is to measure the maximum possible performance for System Under Testing (SUT) without any netfilter rules. Take note of observations and results. That will be our baseline.
  2. Next step is to define a set of netfilter rules on the SUT via iptables. The test will be repeated for every different sets of iptables’ rules, for example 100 rules, 1K, and 10K rules.
  3. The SUT performance will be measured for each netfilter rule set. It is expected that the performance will degrade as the number of rules increases – the greater is the number of rules, the greater is the time required for the server to search through them.
* Task 3: XDP

1. Task 3 involves using the existing XDP project – “pcn-iptables: a clone of iptables based on eBPF”[[4]](#pcn) to run an XDP program with the same set rules used for netfilter testing in task 2.
2. First step in task 3 is to setup the PCN, building and making it work. Next is measuring the SUT performance with XDP program for the same set of rules.
3. After achieving measurements and comparing them with the results achieved both with and without netfilter, the final task is to try improving the performance of XDP based program (if necessary). The ultimate goal is to demonstrate better performance done by XDP than by a conventional netfilter.

# Success Criteria

The main aim of the project is to investigate ability of XDP based programs to improve performance of Linux packet filtering over high speed networks (2.5 GbE) comparing to traditional netfilter approach, while providing the same security mechanisms. This project will be considered successful if the following criteria are met:

* Investigate ability for XDP based approach to provide comparable or even better performance results either higher throughput or reduced latency or both) then conventional netfilter approach.
* If the XDP based approach will provide worse performance than the netfilter, try to figure out the main reasons for that behavior.
* Compile a research report, which summarizes all my findings for given subject.

# References

1. Nginx main page <https://nginx.org/>
2. “wrk – a HTTP benchmarking tool” <https://github.com/wg/wrk>
3. Hussein Nassar “Memcached Architecture” [Online] <https://medium.com/@hnasr/memcached-architecture-af3369845c09>
4. “pcn-iptables: a clone of iptables based on eBPF [Online] <https://polycube-network.readthedocs.io/en/latest/components/iptables/pcn-iptables.html>