**EF_VI TCP Implementation**
**Team 4**
**Kevin Xue**

## Introduction

My project was an EF_VI TCP Implementation. EtherFabric Virtual Interface, a library produced by Solarflare, is a high-performance networking API developed by Solarflare. It gives direct user-space access to the Network Interface Card (NIC) for sending and receiving raw packets, bypassing the kernel. This is extremely important in high-frequency trading as elevation from user mode to kernel mode often requires microseconds if not tens to hundreds of microseconds. By doing direct-memory access with the NIC, we can eliminate the large overheads associated with this, potentially allowing for quicker response times to grab market opportunities. This version allocates memory region buffers for zero copy DMA in user space to allow for low latency TCP transmission without kernel elevation. EF_VI forms the basis for already existing applications, such as OpenOnload, but for pure speed optimization, this implementation assumes fixed IPs and ports as in a real scenario and does less safety checks. This allows it, in theory, to operate fast. If a different destination is wanted, this information needs to be modified directly in the source code - for filtering, and for the packet headers.

## Technical Details

EF_VI provides a driver handle that enables the registration of various components within the EF_VI ecosystem. One of the first steps is creating a protection domain, which establishes a security boundary for access control. Next, a virtual interface (VI) is allocated; this VI is mapped to a receive ring, a transmit ring, and an event queue. Memory regions are then registered—these are contiguous, page-aligned blocks of memory designated for direct memory access (DMA). These regions are often organized into a “free pool,” which typically consist of 2KB buffers which are page-aligned for minimal latency during paging operations. Each buffer reserves the first ~500 bytes for NIC metadata and optional application metadata, with the remainder available for actual data transmission. Because the number of buffers is limited, memory must be explicitly freed after a buffer is used for a transmission. Additionally, filters are applied to define which types of network traffic—identified by attributes such as ports or packet types—should be directed to the VI. These filters ensure that relevant packets are DMAed into the appropriate memory regions for both transmission and reception. To send these packets, which we initialize in buffers from the free pool, we post a tx_descriptor to the transmit ring and indicate a transmit through ef_vi_transmit. The

tx_descriptor is handled automatically by this function. However, for receives, a rx_descriptor must be posted to the receive ring, which indicates an open buffer for transmission. When a receive happens, the virtual interface will select a random open buffer from the ring and return an event, which indicates the id and buffer in which it put the new data. Currently, the system handles these through user reads, which parse the events in the event queue and returns an appropriate number of bytes to the user. Since ef_vi does not allow for nonconsumption of events, a pointer to leftover data is pushed to a data_queue, and will be read on the next read in sequential order. Due to the purpose of TCP with trading systems, as data being read is typically server acknowledgements of some sort, this will be changed in the future to allow for event-driven reads, applying information to the data structures in the system. Currently, the system supports the following

- An initialization function analogous to socket, through ef_init_tcp_client()
- A "connect" function, through ef_connect();
- A "send" function, through ef_send(char *buf, int len)
- A "read" function, through ef_read(char *buf, int len)

This allows for client-server directed communication and server-client directed communication. However, duplex communication has not been fully tested, but this will also be a focus of future efforts.

**Challenges/Design Pivots**

One of the biggest challenges encountered in this is not necessarily following TCP standards but rather handling the underlying management given the complexity of ef_vi. Packets have to be constructed manually and with that, proper infrastructure must support the allocation of buffers, construction of packets, and freeing of buffers. This included much thought into when and how the system would manage these resources which we have to programmatically give it. Computing and passing in accurate offsets was also a challenge, given all the different metadata offsets needed to be computed. This led to weird, unexplainable errors which were quite hard to debug given the low level that all of this code operated on. This required thinking through how the NIC actually interacted with the switch. I found that I often had to dive into the hex dumps to truly understand what was going on. Overall, it was a complex project that required a deep understanding of networking as well as good, old fashioned persistence.

Although there weren’t too many design pivots, as the main idea had already been mostly fleshed out in talking to the instructor, Matt, I did note one significant compromise that had to be made. This was regarding the way to read, that was, we needed to be able to apply market updates and acknowledgements that come from the exchange in

an actual trading setting. Matt suggested that I create a parser which has a callback to apply the data to the system, which would be much faster than having a user read into a buffer (which is not zero-copy). However, due to the time constraints and complexity concerns, especially with a possible explosion of bugs, we did not actually hook this up to the actual trading system, so I found it likely more interesting to have this work with plain text, hence the decision to provide a traditional “read” function. In this sense, it is not truly zero-copy, as it does a single copy into the user’s buffer, but if this were to be attached to an actual trading system, a truly zero-copy interface could be achieved. However, as mentioned previously, this was a decision that had to be made for the sake of presentability and simplification.

Overall, I learned a lot from this project. I also gained a newfound appreciation for and interest in networking, especially in TCP and understanding of network hardware. Formerly, I took TCP mostly for granted, but unraveling much of the complexity (such as ports not really existing in the sense most people think of them) has been rewarding. I think there’s a lot of cool things that can be learned and built in this field which require solving hard problems. Because of this, I want to delve more into networks and focus my software engineering knowledge on this. Furthermore, I want to eventually produce a more refined, polished version of this project itself so that I can open source it. This will be my summer goal, and I have some cool ideas brewing for how I will continue to develop and refine it to support more functionality. Thank you to Matt for always being very responsive to my constant Slack messages, thank you to my team for an amazing learning environment, and thank you to everyone else in the class for a great semester. I appreciate you all and your support throughout both this project and the trading system, and I hope I’ll be able to take something from this into a future career.