Aaron Lo, Jeriah Yu
CS 380L

Comparing User, Kernel, & Kernel-Bypass HTTP Services

Abstract

Low-latency networking is essential in modern computer services. Distributed networking and online gaming represent use cases that require data to be sent, processed, and received with very low overhead to achieve maximum performance. Our experiments seek to analyze the empirical performance characteristics of servicing network requests at different boundary levels of the operating system, in particular servicing packets with user-level processes, kernel-level modules, and direct kernel-bypass NIC communication in the data plane.

Background and Problem

Network communication is ubiquitous in modern computing. For standard user-level applications in a monolithic kernel, network communication requires the process to communicate with the OS abstraction layer, which then interacts with the network interface card (NIC) driver, which then sends instructions to the NIC hardware to send a packet. The reverse process happens to receive packets. These layers of indirection increase overall latency to process and respond to packets, adversely affecting performance in critical low-latency high-communication applications such as grid computing, competitive gaming, or high-frequency trading.

Many system designs have been proposed, implemented, and tested to reduce latency and improve performance while still maintaining an abstraction layer to share the network resource and interface among multiple processes. JITSU treats incoming network packets as signals for servers to rapidly spin up unikernels to process the packet and send the proper response, but faces higher latency due to kernel boot delay. Arrakis sets up access for a process to have direct dataflow to the NIC for a time quantum before interposing control to another process, allowing fast response for the process with exclusive packet access, but requiring other processes to wait their turn. Exokernel uses injected kernel code to process packets based on a filter without having to switch on the user boundary, prioritizing speed over security. SnapOS uses a user-level library to bypass the kernel with a separate channel to the hardware NIC, ensuring the path to network access is short, but requiring explicit support from the NIC driver through SR-IOV and explicit cooperation of user processes.

Our goal is to investigate the performance of the underlying methods used in these systems in terms of throughput and latency in responding to HTTP requests. We look at the performance of some modern userspace web servers, Apache and Flaskr, as a baseline. We then compare this to a simple web server daemon implemented within the Linux kernel as a kernel module (khttpd) and an application that interfaces directly to the NIC via SR-IOV using the Data Plane Development Kit (DPDK) (with a lot of help from tinyhttpd-lwip-dpdk).

Hypothesis

Before starting, we hypothesized that in terms of latency, Flaskr will perform worst, followed by the kernel module server, with the DPDK server performing the fastest, as each layer removes a level of abstraction overhead from userspace to kernel, then from kernel to hardware interface. While Apache is theoretically also slower, we reserve the possibility that it could perform better than anticipated and potentially better than a kernel module due to its widespread usage and continuous optimization under the worse-is-better principle.

The Journey and Setbacks

For the kernel module, we first found existing implementations of khttpd as experimental code (Github). This did not have the ability to read files or serve concurrent requests with multiple threads, so we found a forked version that implemented that functionality (mostly) to use for testing (Github). We first attempted to compile and run the module on WSL, but WSL's kernel does not come with Linux headers that are needed for the kernel module interface. Next, we tried to directly inject the installed module into the kernel used in lab0 so it starts on boot. That also did not work (the server did not start on boot), but using the insmod install module command did, enabling curl commands to be sent on localhost and receiving directory listings. However, the module crashes whenever a file is attempted to be read, and the kernel messages that resulted were difficult to parse due to a lack of error messages.

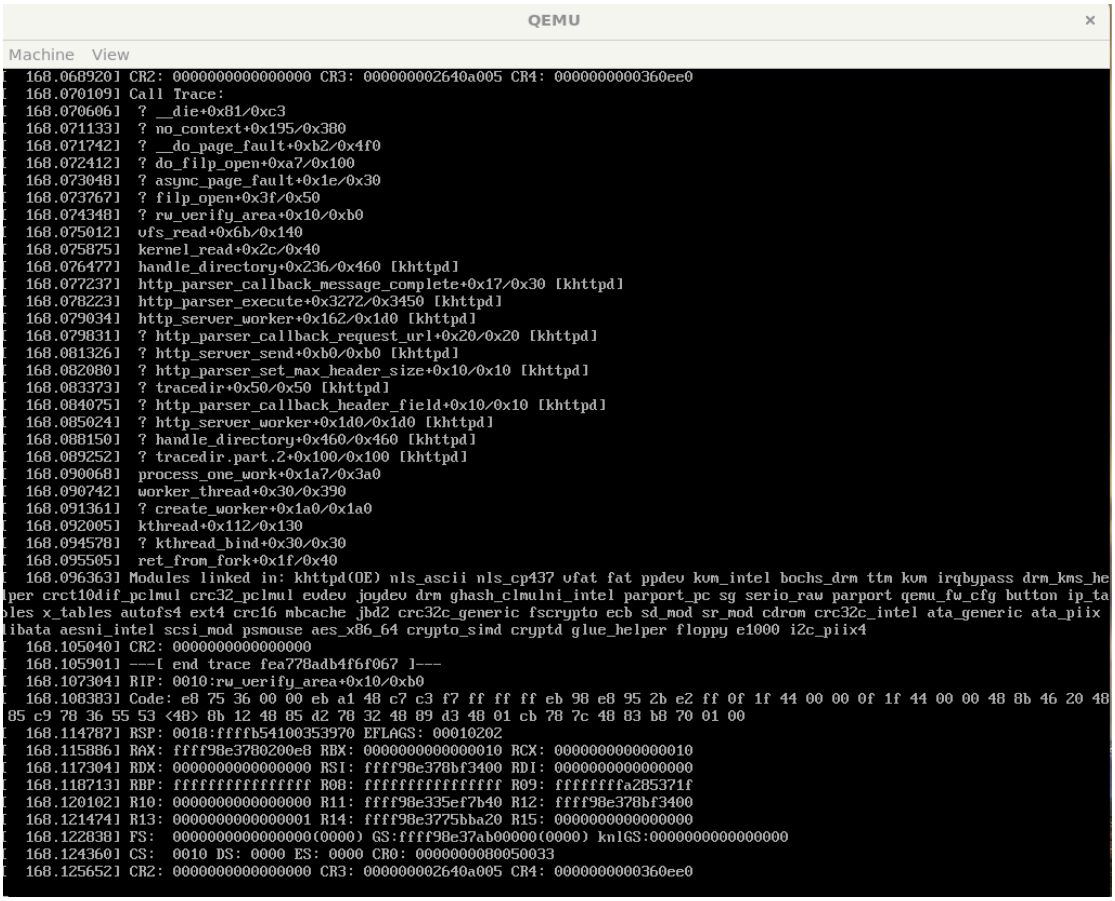

Image 1: Kernel messages from khttpd crashing on file GET request.

We eventually found the root cause that separated the behavior of reading directories from reading files was the kernel_read call, which requires a change to pass a pointer to 0 rather than a 0 (which is incorrectly interpreted as a null pointer). We have not pushed this change upstream (so good luck to everyone else figuring it out). With a functional server on the VM, we worked to port it to the physical machine serving as the benchmarking environment. This presented a new challenge in the form of security: because kernel modules have broad unrestricted access, installing a module was blocked even with superuser privileges.

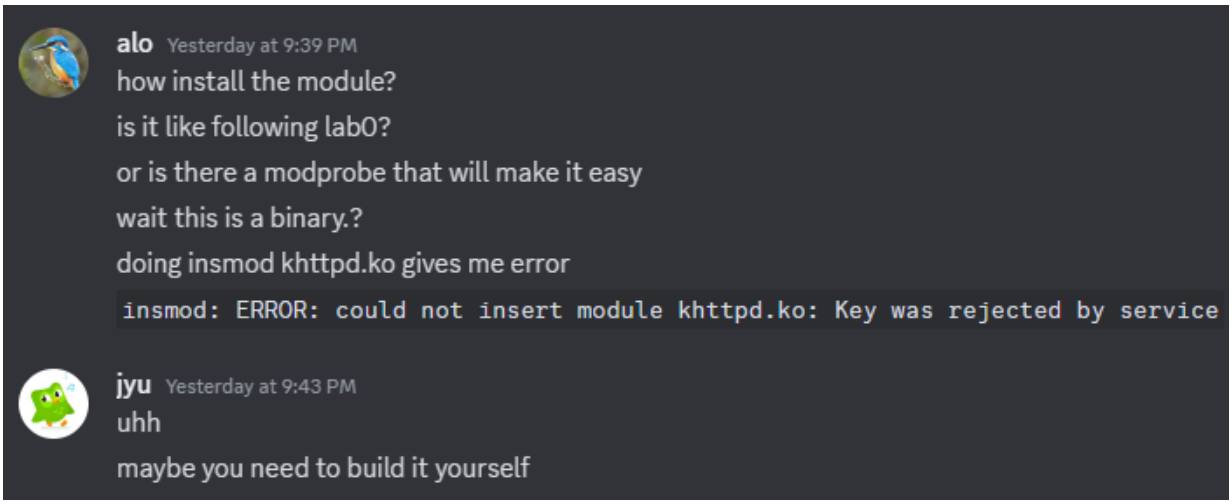

Image 2: Attempting to resolve the insmod error. Building it again did not help.

After further research, we found the error was due to secure boot. We also found along the way that the VM Linux and physical machine Linux versions differed, and the rebuild process required additional patching to handle gcc version differences that resulted. After temporarily disabling the option in the BIOS, the module could then be loaded into the kernel while running and we could benchmark it. A limitation of this kernel server is that the entire file being read is first copied into kernel memory and then sent via HTTP, so there is a maximum limit to the size of the file in this prototype that commercial web servers address.

For the implementation of an HTTP server with DPDK as the network backend, the first step was to set up the system so that the NIC is able to communicate with the internal system. In order to do this, the NIC must be placed in its own IOMMU group in order to bind the ports to VFIO for DPDK usage. However, it was discovered that the placement of the Intel NIC was actually in an IOMMU group that contained multiple other PCI devices, including the storage device that contained the currently running OS. Deciding that that was unwise (and finding other ways of breaking IOMMU groups questionable), we eventually decided to take apart the machine and strategically move PCI devices around to different points on the motherboard until everything was arranged in a way such that the NIC was in its own IOMMU group.

Image 3: DPDK complaining about IOMMU groups

After that, we were able to not-easily set up a DPDK script that forwarded packets received from SR-IOV NIC to an internal script that polled continuously for packets. While this seemed to work great in the sense that packets could be sent back and forth through this bridge, and the NIC was forwarding packets to the application, we could not intentionally ping/send packets to said application through the NIC (internal DPDK packet generators could communicate but not external). This probably occurred due to the lack of backend or in-depth network issues that remain outside our knowledge. It was then at this point that we realized that DPDK does not provide a TCP/IP stack, requiring the user to build it themselves. As this would be insurmountable for us to complete in the short time frame we had, we turned to external libraries that promised to help. One that we chose was SeaStar (Github), which promised to be a high-performance networking library. After spending much time trying to get this to work, it still throws countless errors trying to compile. We remain very confused as to how others got it working as even the mere import of their libraries causes gcc to complain about a myriad of issues and missing variables. Finally, we settled on another project tinyhttpd (Github) that combines both DPDK with lwIP, a lightweight TCP/IP stack, to build a simple web server. As we were still managing the entire TCP/IP stack, dealing with HTTP splitting into multiple TCP packets was out of scope, thereby limiting it to a single TCP packet and limiting the size of a possible HTTP response. DPDK still remains very confusing to us and perhaps was not the best thing to attempt in this project.

Limitations: Limited HTTP size therefore our HTTP server testing was restricted to small files.

Environment

These experiments were all performed on a PC that was running Ubuntu-22.04. The PC was running an AMD Ryzen 5 5600X 6-Core Processor that has support for AMD-V virtualization for an x86_64 architecture and 12 CPUs. In addition, it was running on top of a B550-A Pro motherboard that had support for SR-IOV PCie. The two network cards within the system were

a Realtek RTL8111 PCI Express Gigabit Ethernet Controller and an NIC with an Intel 82576 card (that supports SR-IOV as well as is supported by DPDK).

Experimental Design

Due to the limitations discussed above, we were only able to test for latency on small files. Stemming from this, we decided to use the HTTP benchmarking tool *wrk* to test the latency of retrieval of a certain file. We utilize *wrk* to run a benchmark that runs for 10 seconds and keeps a varied number of HTTP connections open to the server. The number of these connections go from 1 to 64 in powers of 2 and we measure the average latency, 90% latency, and request throughput of the server.

Results

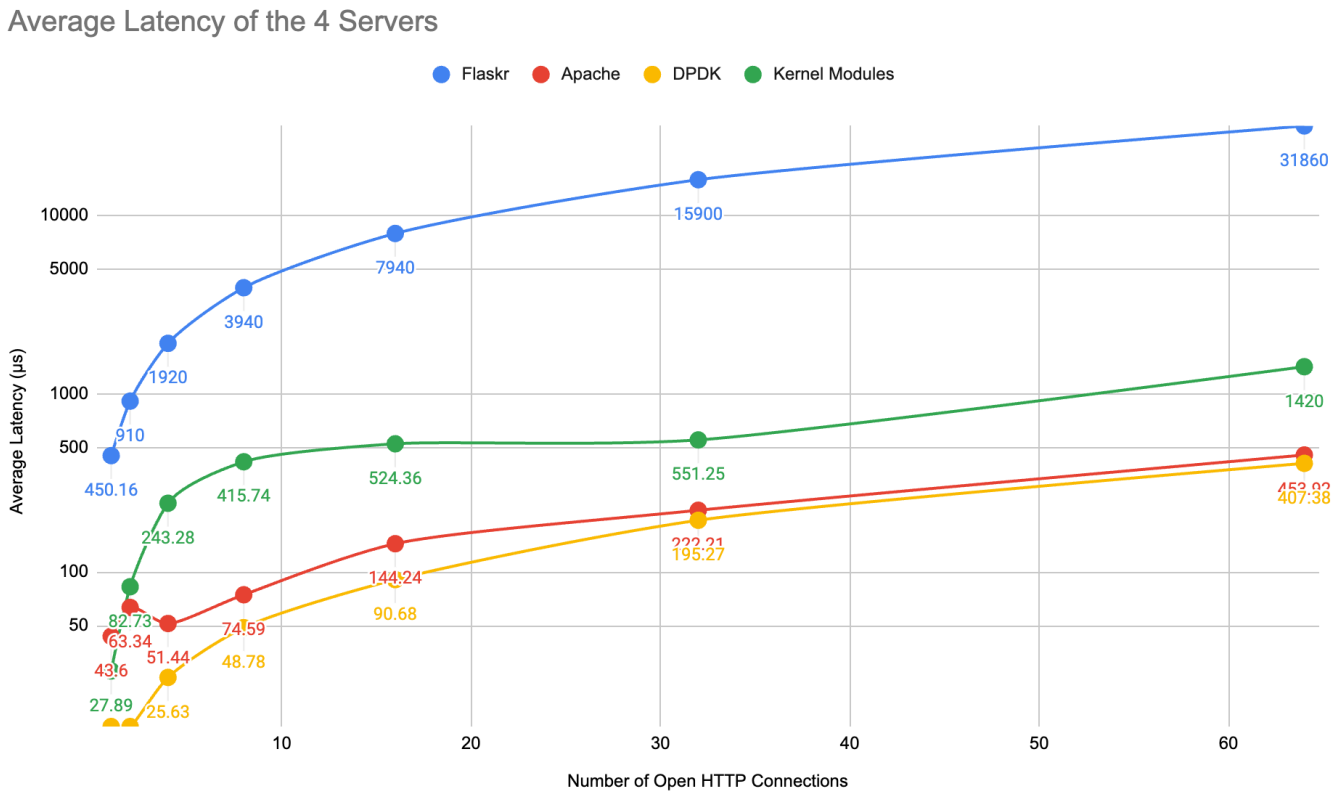

Figure 1: Average Latency of the Servers given Number of Connections (µs)

When comparing average latency, the results (conveniently) support our hypotheses, with Flaskr having the worst latency followed by the kernel module and the DPDK service. As anticipated, due to Apache's long usage and continuous development, its performance is very good, better than the kernel server we used, and gets close to matching the DPDK setup at high connection

amounts. This suggests that Apache is optimized to scale to high numbers of requests at once and handle load-stress scenarios.

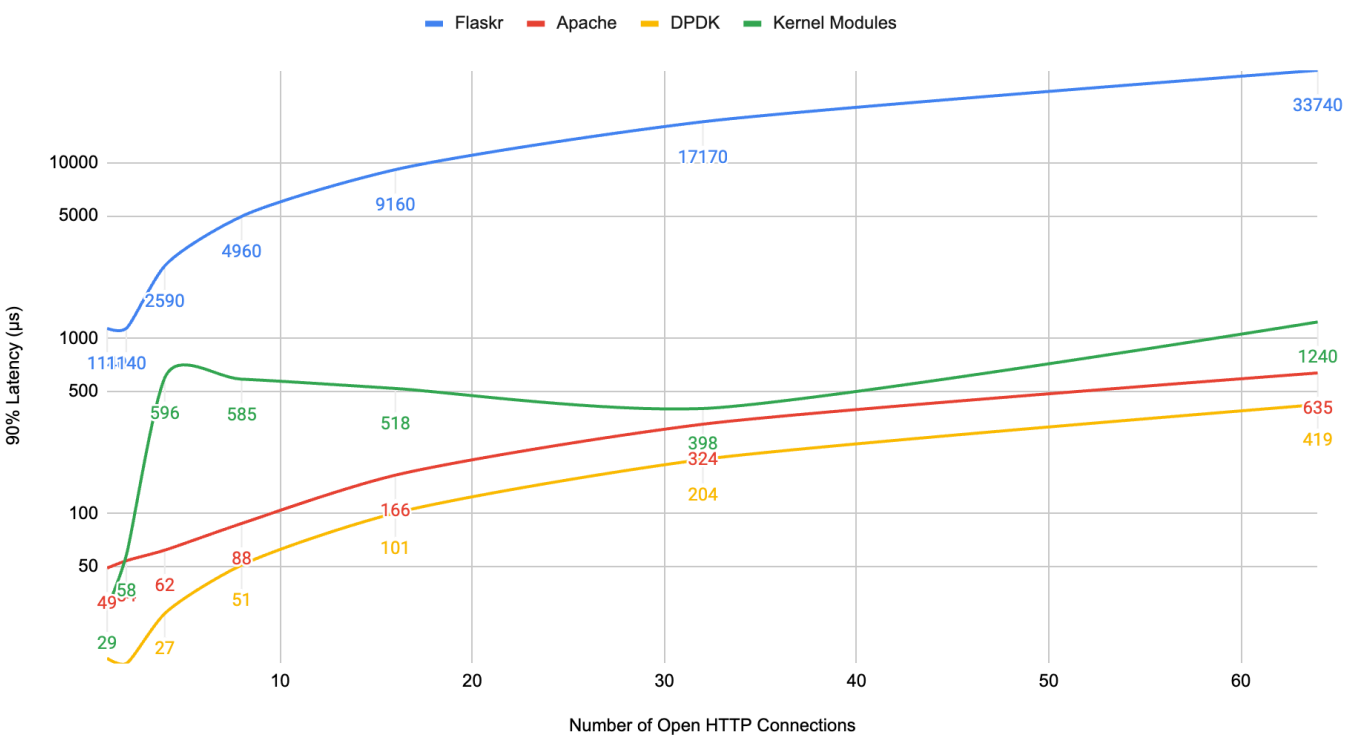

Figure 2: 90th Percentile Latency of the Servers given Number of Connections (µs)

The results of the 90th percentile latency response we received from the servers matches up very well with our previous results of the overall average latency. If a server experiences a lower average latency, so will its 90th percentile latency response. However, there were still several interesting things to note. Comparing Apache and DPDK, the difference between DPDK average latency and 90th percentile latency is quite low, suggesting packets experience a very uniform treatment through the pipeline, which makes sense as the interrupts that normally occur are mostly removed, whether that be an interrupt from new packet received or syscalls required to interact with network stack. Comparatively, Apache's is much higher, given the reasons above. Another thing to note is that the Kernel Modules implementation faced absolutely horrible 99% that hit up to 30 times worse latency compared to the average. Why this is an issue specific to the kernel implementation compared to userspace and hardware implementations remains a mystery.

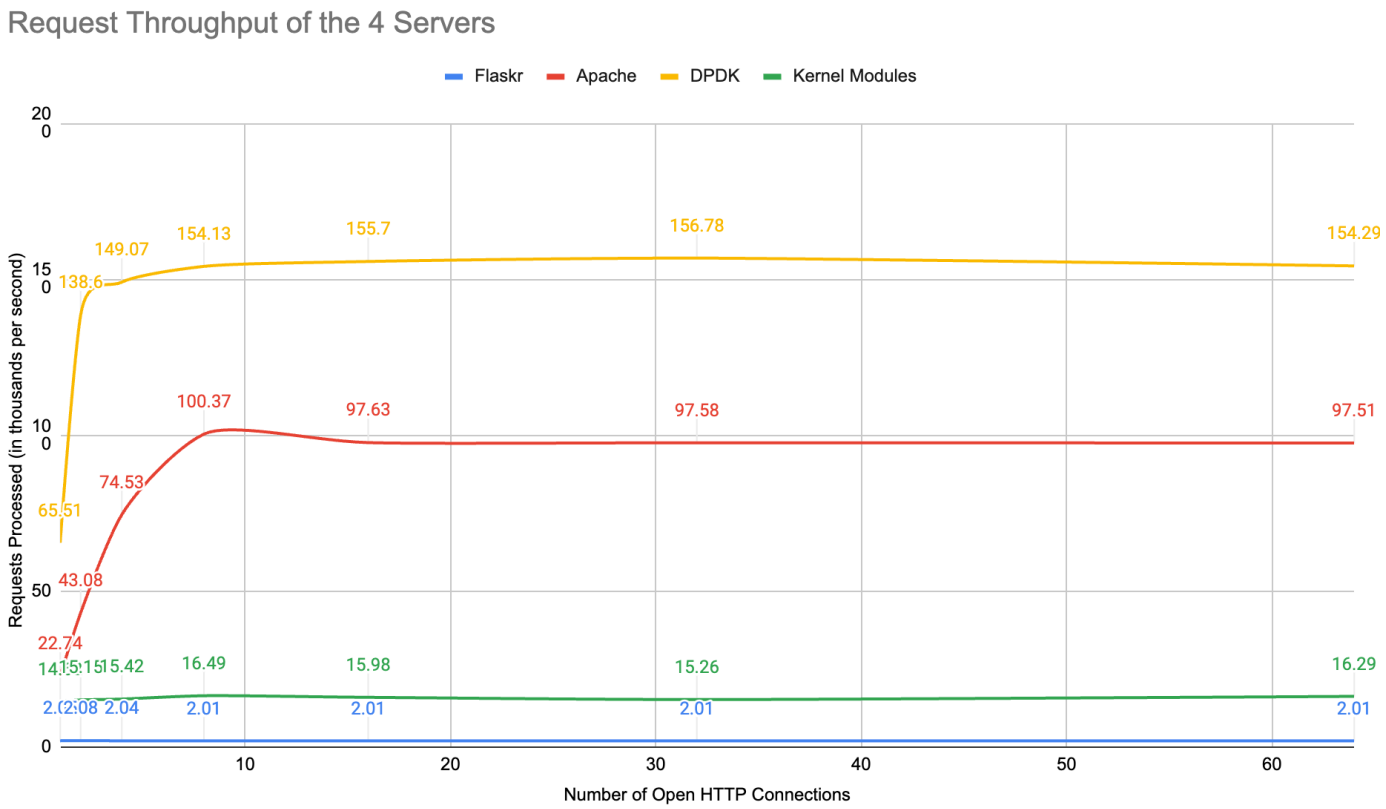

Figure 3: Request Throughput of the Servers given the Number of Connections

The results here matched what we expected from the other measures of average latency and 90% latency. The implementations that performed well there perform well here. One interesting thing to note is how quickly it took to saturate the system given the open HTTP connections for DPDK versus Apache. DPDK very quickly got saturated at around 3 connections while Apache lasted up until 10 connections before reaching saturation. This again suggests that Apache has been optimized for high requests, given its wide adoption in server environments.