Survey of state-of-the-art Network Stacks

Seminar Report
submitted in partial fulfillment of the
requirements for the degree of
PhD
by

K Ashwin Kumar

under the guidance of

Prof. Mythili Vutukuru

Department of Computer Science and Technology
Indian Institute of Technology, Bombay
Mumbai 400 076

Contents

List of Figures

Abstract

Linux network stack is one of the most popular network stack out there. There has been a constant effort to make it more efficient over the years, so that the applications running on top it can deliver better metrics. Despite all the efforts, it has not been able to keep up with advances in network hardware. One of the major reasons for this reduced capability has been that it is still part of a monolithic kernel which for ease-of-use for the programmer has to keep the network stack API consistent with other parts of the kernel which results in a lot of systemic issues in the network stack which hinder performance.

Kernel bypass accelerators like netmap and DPDK and recent virtualization technology enable user applications to get control the networking hardware without any intervention from the Linux kernel. This development has made it possible to decouple the data path of a network stack from the monolithic Linux kernel, giving rise to highly tuned user level network stacks. As data paths of network stacks move closer to the user application, researcher have started thinking of network stacks as an important module of the application itself. In such a case, the network stack will be highly customised to work in consort with the user application to deliver optimal performance. Kernel bypass techniques have not completely eliminated the role of the kernel from the network stack and kernel is still responsible for the control path of the network stack.

In this seminar, we present a survey of existing state-of-the-art in network stack research. We start off with exploring in-kernel techniques and then move on to advanced user level network stacks. We also look at network stacks which offload some of the processing capability on to the NIC hardware. To conclude, we explore future work in the field of network stacks in relation to 5G Network Functions.

Chapter 1

Introduction

A network stack, which is sometimes also known as protocol stack is nothing but an implementation of a networking protocol suite like the TCP/IP protocol suite [12]. In other words, a protocol suite is an implementation of the communication protocols laying down rules which both the sender and receiver follow to communicate with each other, and a network stack is just the software implementation of those protocols. Because the network stack is free to implement the protocols in any which way it pleases, we actually end up with a plethora of network stacks implementing the same set of protocols (protocol suite).

The Linux network stack which was developed as part of the Linux kernel is an open source networking stack, and is one of the most widely used network stacks today. For public Internet servers, Linux is generally counted as a dominant operating system, powering well over twice the number of hosts as Windows Server operating system – which is trailed by many smaller players including traditional mainframe OSes. [12].

Because of such popularity, the Linux network stack is constantly being worked upon to improve the performance of the application running on top of the network stack. One of the major patches in the Linux networking stack was the NAPI patch, where the networking stack gravitated towards an interrupt + polling model instead of solely depending on interrupts. Another popular patch was the SO_REUSEPORT patch which helped facilitate connection locality in the Linux networking stack, and we discuss this patch at length in the next chapter. Alongside these improvements in the networking stack, the networking hardware has also been constantly improving, where at the start of the decade we had 10Gbps NICs and now we have state-of-the-art NICs delivering 100Gbps. There has been a 10 fold increase in networking capacity in a very short period of time. Despite all the improvements in the Linux networking stack it has not been able to keep up with the improvements made in networking hardware because of some systemic issues. The Linux networking stack is part of a monolithic kernel which imposes some restrictions on its performance. For example, to give a consistent view of the kernel API to the user writing applications to run on top of the Linux kernel, every resource inside the kernel is treated as a file and accessed via the virtual file system, and the Linux kernel networking stack is no different. But, as we point out in the next chapter, that this indirection is completely unnecessary. The granularity of a system call for network I/O in the Linux kernel is very fine as well, where a single system call tends to only a single I/O request, and this is done to again give a consistent view of the Linux system to the user, where a system call only performs a single operation. There are other issues which hurt Linux network stack's performance, and we discuss them in detail in the next chapter.

1.1 Network Stack Research Timeline

Citing multiple bottlenecks in the Linux networking stack, researchers have also been trying to improve the network stack's performance independent of the improvements made in the Linux networking stack. The timeline of development of modern network stacks is presented in figure 1.1.

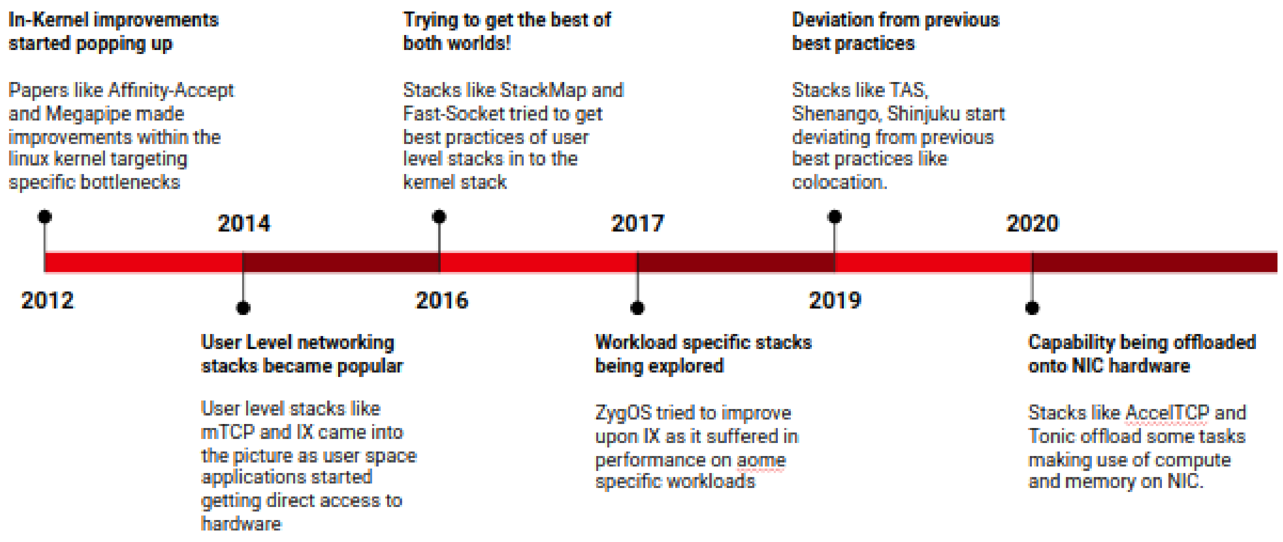

In-Kernel improvements started popping up
Papers like Affinity-Accept and Megapipe made improvements within the linux kernel targeting specific bottlenecks
Trying to get the best of both worlds!
Stacks like StackMap and Fast-Socket tried to get best practices of user level stacks in to the kernel stack
Deviation from previous best practices
Stacks like TAS, Shenango, Shinjuku start deviating from previous best practices like colocation.
2012
2014
2016
2017
2019
2020
User Level networking stacks became popular
User level stacks like mTCP and IX came into the picture as user space applications started getting direct access to hardware
Workload specific stacks being explored
ZygOS tried to improve upon IX as it suffered in performance on aome specific workloads
Capability being offloaded onto NIC hardware
Stacks like AccelTCP and Tonic offload some tasks making use of compute and memory on NIC.

Figure 1.1: Timeline of Novel Network Stacks

Researchers early on started with trying to improve upon the Linux network stack by staying within the kernel framework. Papers like Affinity-Accept [6] and Megapipe [9] targeted specific bottlenecks inside the Linux networking stack.

Kernel bypass accelerators like netmap [11] and DPDK [2] started gaining traction around the same time, which prompted researches to explore user level networking stacks with papers like mTCP [10] which ran completely in the user space, bypassing the Linux kernel. Other ways of bypassing the kernel were also explored which included making use of the virtualization technology to give direct access of NIC hardware to networking stacks running as guest OSes in the system, and papers like IX made use of this technology.

In 2016, we had stacks like Fast-Sockets [13] and StackMap [17] which tried to bring in some ideas from the user level network stacks in to the kernel framework. In 2017, ZygOS [19] tried to improve upon IX to deliver better tail latencies which they identified to be suffering in the case of IX for a specific set of workloads.

In 2019, network stacks like Shinjuku [15] started targeting more specific workloads, and the networking stacks in general started deviating from previously believed best practices like colocation.

In 2020, with the advent of compute and memory resources on NICs, stacks like AccelTCP [1] and Tonic [3] start offloading some of the capability of the networking stack on to the NIC hardware.

1.2 Importance of network stacks

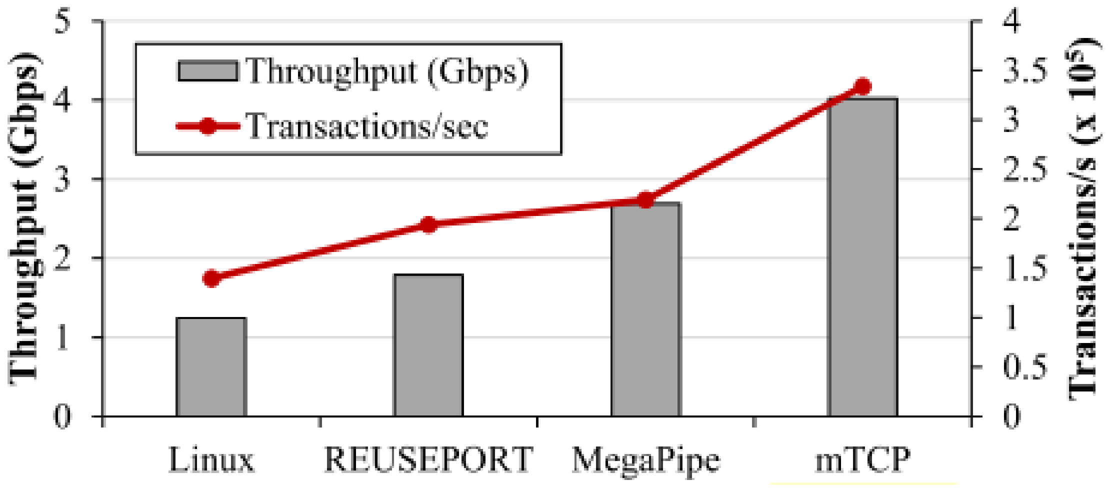


Figure 1.2: Performance of an application server running on top of four different network stacks

Choosing the correct network stack for an application can have a significant impact on the application's performance. Figure 1.2 shows the performance of lighttpd application which is a simple web server, running on top of four different network stacks when subjected to a standard load pattern known as the static file workload extracted from SpecWeb 2009. We clearly see an improvement of around 3.2 times when the same application runs on top of the mTCP network stack as opposed to the Linux network stacks, and this signifies the role a network stack can play in an application's performance [10].

The rest of the report includes more in-depth analysis of state-of-the-art network stacks. Chapter 2 describes the Linux network stack and the issues related with it, which acts a motivation for us to look at better network stack designs for 5G network functions. Chapter 3 includes a summary of existing research on various popular state-of-the-art network stacks. And finally, in chapter 4 we propose research work which we plan to carry out in the future.

Chapter 2

Motivation

Having gone through an introduction of network stacks in the last chapter, in this chapter we will briefly review the Linux network stack [8] which is one of the most widely used open source network stack out there. Then we will move on to some of the inefficiencies which have been identified in the Linux network stack which places a limit on the network stack's performance and ease of scalability. Finally, we will present the direction in which research related to network stacks is headed, and how this research path motivates the work we plan to do.

2.1 Linux Network Stack

Before diving into the specifics of the Linux network stack let us first take a brief look at interrupts. Hardware devices such as the Network Interface Card (NIC) can send interrupts to processors and this forces the processor to stop what it was doing and execute a pre decided piece of code which the NIC device driver had pre registered with the processor, and this pre decided piece of code is what we call the **Interrupt Handler**. The interrupt handler cannot run for too long because of two reasons, one being that during the time the interrupt handler is running on the CPU, interrupts from that particular hardware device are disabled by the CPU, and hence all the interrupts which the device raises on the CPU while the interrupt handler is executing will be lost. And, the second reason is that interrupts quite literally interrupt the application which was running on the CPU, and we would not want the application to be interrupted for long as it can hurt the application's performance. Because of these reasons the interrupt handling code is divided into two parts:

1. **Top half** is the piece of code which runs when the device sends an interrupt to the CPU, much like the interrupt handler. This part is said to execute in hardirq context and includes code which acknowledges the interrupt, and sets up the execution of the rest of the interrupt handling code in bottom half at some later point in time.

2. **Bottom Half** is the part which does the heavy lifting of actually processing the interrupt which was received earlier. This piece of code is scheduled on to the CPU by the Linux scheduler and is said to run in softirq context. Because this piece of code is voluntarily scheduled by the Linux operating system, it can be more expansive when compared to the interrupt handler, but as we will see later, the execution of code in softirq context is also bounded.

With this information in mind let us move on to the working of the Linux network stack on the reception of a packet, in other words, steps involved in the RX processing of the Linux network stack. Figure 2.1 shows a pictorial representation of the whole process.

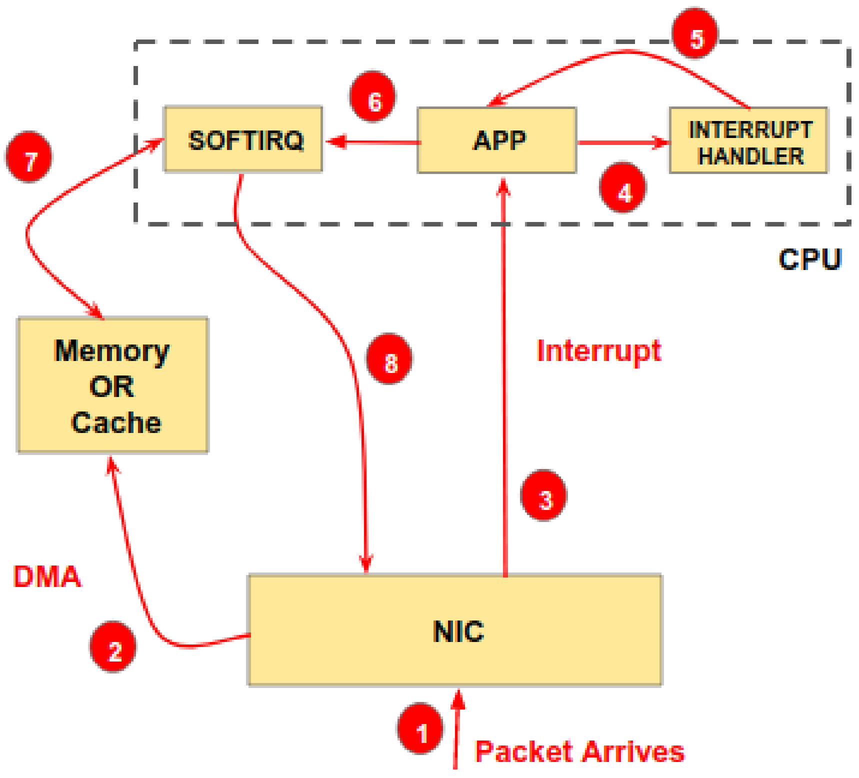


Figure 2.1: Linux network stack

The process is kick started as soon as a packet arrives on the NIC. The NIC driver maintains a queue of RX descriptors, each of which includes information on where to copy a packet in memory if and when a packet a received on the interface. So, after receiving the packet, NIC hardware picks up an entry from this RX descriptor queue, and via Direct Memory Access (DMA), copies the contents of the received packet into the specified memory location. After doing so, the NIC hardware raises an interrupt to a CPU, notifying it of packet reception and telling it to take necessary steps to process this packet. The CPU would be running some application, and on reception of this interrupt from the NIC hardware, it switches to running the interrupt handler related to that particular interrupt, which has been registered on the CPU by the NIC driver beforehand. This interrupt handler does only two things:

1. It enables softirq, or the part of the code which actually processes the received packet. This step enables the scheduler to schedule the softirq at a later point in time on the very CPU on which the interrupt handler runs. The important point to note here is that **softirq is scheduled to run on the same CPU which receives the interrupt from the NIC hardware**.

2. It disables further interrupts from the NIC on this CPU, and this shifts the network stack into polling mode, where polling will happen inside softirq. It is important to note here that even thought the interrupt handler disables interrupts from the NIC, this does not stop the NIC from DMAing packets into memory as and when it receives them.

After performing these two steps, the interrupt handler is context switched out and the application which was executing when the CPU had received that interrupt starts off from where it had left. After some time, the scheduler schedules the softirq or the bottom half to run on the CPU. The softirq polls the RX descriptor queue for packets, and performs two steps on each of the packets. First, it wraps each packet in a structure known as sk_buff, which is the standard representation of a packet in the linux network stack, and the second thing it does is pass this sk_buff structure to the network stack for network and transport layer processing. It is important to note here that the **sofitrq execution is also bounded by both time and the number of packets it can process**, and as soon as softirq hits one of those limits, it is scheduled out by the scheduler, and the last thing that the softirq does before being scheduled out is to enable the interrupts from the NIC to the CPU on which softirq is executing, so that the whole cycle can be repeated again.

2.2 Issues with the Linux network stack

The Linux kernel has a lot of issues which act as a bottleneck for high speed I/O, and some of them are listed here.

1. **File system overheads:** As we know that sockets, which contain actual packet and some metadata for a network connection, are accessed via the file system. This is done to promote consistency in the API exposed by the Linux kernel, where every resource in the kernel is treated as a file. But, this indirection gives rise to a bottleneck in terms of multicore scalability because to access a resource via the file system, lot of locks have to be obtained first. This indirection is completely unnecessary in the case of sockets because of two reasons, (1) **sockets are rarely shared** especially established TCP sockets hence you don't need the indirection which is provided in the form of the file descriptor table, then the open file table and so on, and (2) **sockets are ephemeral** unlike files, so creating a bunch of structures in all these tables for just a couple of packets in the case of short connections does not make sense. [9]

2. **System Calls:** To perform network I/O, the application has to make use of system calls to pass I/O commands requesting the kernel to carry out I/O on its behalf. On every system call, a privilege level switch has to take place which has a penalty. First, there is the time it takes to save all the registers of the currently executing code and then elevate the privilege level of the CPU, and load the registers to execute the system call. This time is known as **mode switch time**. Second, there is the footprint of the system call execution itself. The system call while it was executing would have made use of the L1 cache, the instruction cache, translation lookaside buffers (TLB), branch prediction tables, prefetch buffers, hence replacing most of the application state on these resources. And, this replacement of user-mode processor state by kernel-mode processor state is referred to as the **processor state pollution caused by a system call**. [4]

3. **Packet Copy:** When NIC receives a packet, it DMAs the packet to a memory location which is controlled by the Linux kernel, and the application level code does not have access to this part of memory. So, naturally the kernel after

successfully processing the packet, has to copy the contents of the packet to a memory location specified by the application which the application code has access to. This process has to be repeated for every packet, and hence acts as a bottleneck to achieving high network I/O speeds. [8]

4. **Memory Management:** The Linux kernel treats memory as a scarce resource, and hence it seldom over allocates memory. For this reason, memory for connection state, sockets, packet content and various other data structures related to the network stack have to allocated and deallocated dynamically. This leads to invoking the memory management module of the linux kernel very frequently, which naturally makes use of locks to give a consistent view of free memory to all cores in the system, and hence becomes a bottleneck for multi core scalability of the linux network stack. [8]

5. **Scheduling:** As discussed before, packet processing takes place in the softirq context which is scheduled on the CPU by the Linux kernel scheduler. But, the granularity at which the scheduler switches out tasks replacing them with new ones is not enough to handle certain load patterns or even short bursts in load, and this leads to an increase in latency which lasts for quite some time. [8]

2.2.1 Connection Locality

This issue deserves a separate subsection of its own as most of the effort in devising better in-kernel network stacks majorly focus on this issue. Let us start by giving a brief description of the connection locality issue.

As we already know RX processing of a connection will happen on a core to which the NIC decides to deliver an interrupt, and the processing will happen in a softirq context. This is shown to happen on core X in figure 2.2. Similarly TX processing of a connection will happen on a core where the application thread decides to do the write system call, which is usually the same core from where the application reads the packet and does application processing. The application is shown to run on a separate core Y in figure 2.2. Now connection state for each TCP connection in the network stack is maintained in a TCP control block (TCB), and both RX and TX processing of a connection need to access and update state in that TCB. So, if these two cores happen to be different which is shown in figure 2.2, then some sort of synchronization will be needed to maintain serial access to the TCB which will lead to one core waiting and thus wasting precious CPU cycles.

Also, the TCB state would be bouncing between the caches of core X and core Y as code running on both of these cores will be updating state of the same TCB, and this would in-turn lead to a lot of cache line invalidations and hence will compound the issue.

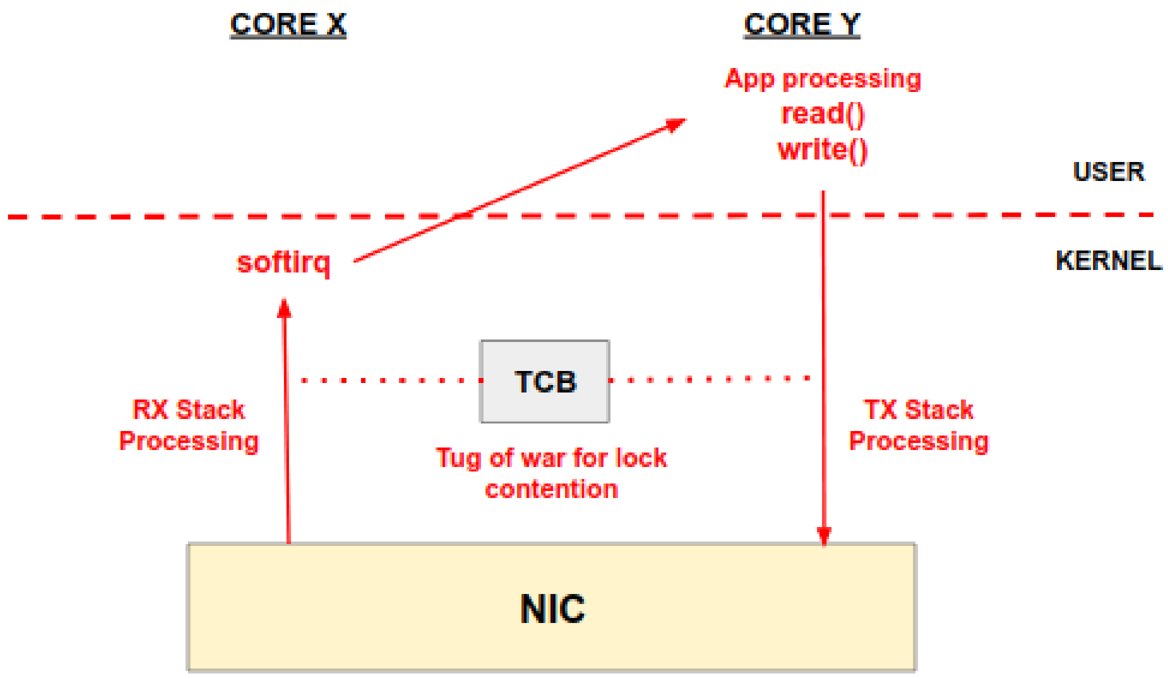


Figure 2.2: Connection Locality Problem

An easy solution to the problem would be to direct packets from the NIC to only those specific cores on which the application is running. In this case each core on which the application runs would handle a subset of the total number of connections, and all packets related to those particular connections would be processed by the Linux network stack on the very core responsible for application level processing of that particular TCP connection. For example, lets say we have a TCP network application running on some cores on the machine, and by using some logic the application decides that all application level processing for a particular connection X will be done on core Y, which is one of the CPU cores on which the application is running. Now, the NIC hardware can be set up in such a way that any packets corresponding to connection X will be redirected to core Y in the system. Hence, both network stack processing and application level processing will happen on the same core, which would eliminate the connection locality issue described before.

Curse of accept

As we saw earlier, transferring packets to specific cores and pinning application threads on those cores to process these packets was said to deliver connection locality, but as we will see now it does not completely solve the connection locality problem.

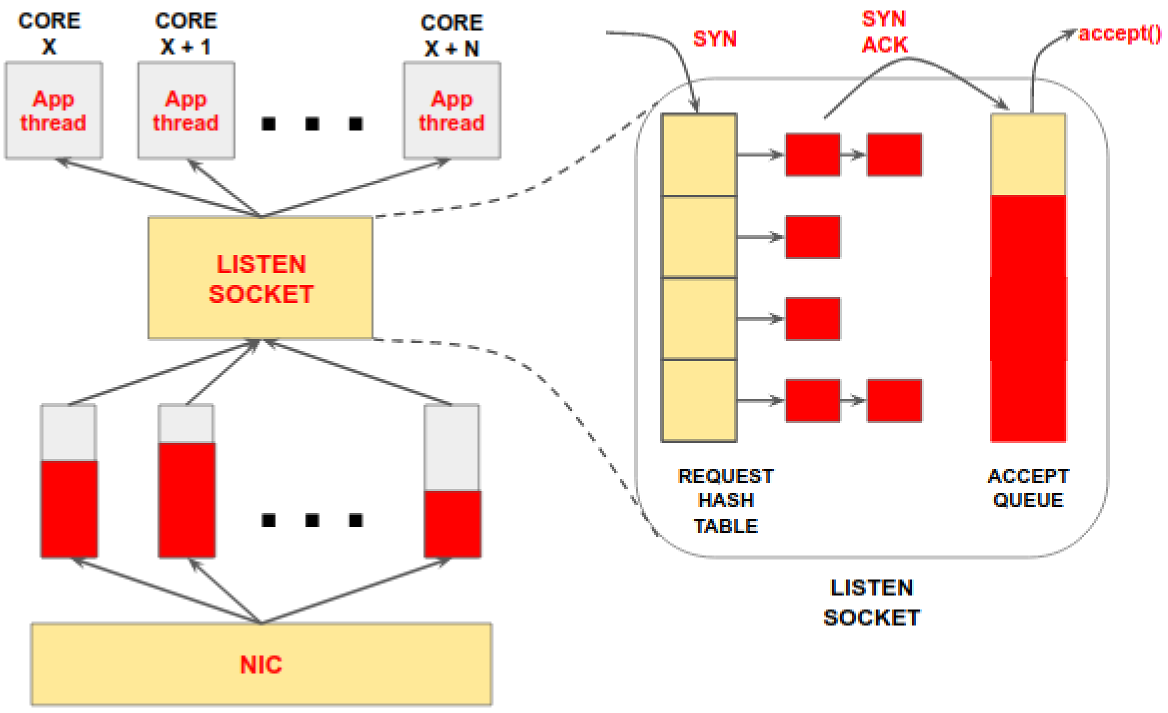


Figure 2.3: Curse of Accept

As you can see in figure 2.3, all of these packets will have to share a single listen socket which is the abstraction of a port in the Linux kernel network stack. The application will have to call accept on this listen socket to get an established socket connection on which data transfer can take place. Let's take a closer look at the listen socket data structure. It consists of two data structures, a request hash table and an accept queue. As soon as a SYN packet is received for that port, an entry is made into the request hash table which is made to record the SYN packet. Subsequently, a corresponding SYN-ACK packet is sent by the kernel, and in response the other TCP application running on the other side will send an ACK packet. After receiving an ACK packet, the corresponding SYN entry is moved from the hash table to the accept queue and this is from where the accept system call picks up an established TCP connection, and returns file descriptor pointing to the that new established socket. To make matters worse the lock granularity on the listen socket is really bad and the complete listen socket is protected by a single lock. So, only one operation can happen at a time on the listen socket which is a serious bottleneck especially for short connections.

SO_REUSEPORT

Linux came up with a solution to the issue discussed in the last section in the form of SO_REUSEPORT [16]. Although it is important to note that the motivation behind this patch was not connection locality, it was to distribute requests between multiple accept queues because of high request build up in the request queue. The solution was to basically allow multiple listen sockets to bind to the same port, and this feature can be used in a way to deliver connection locality. Listen sockets are

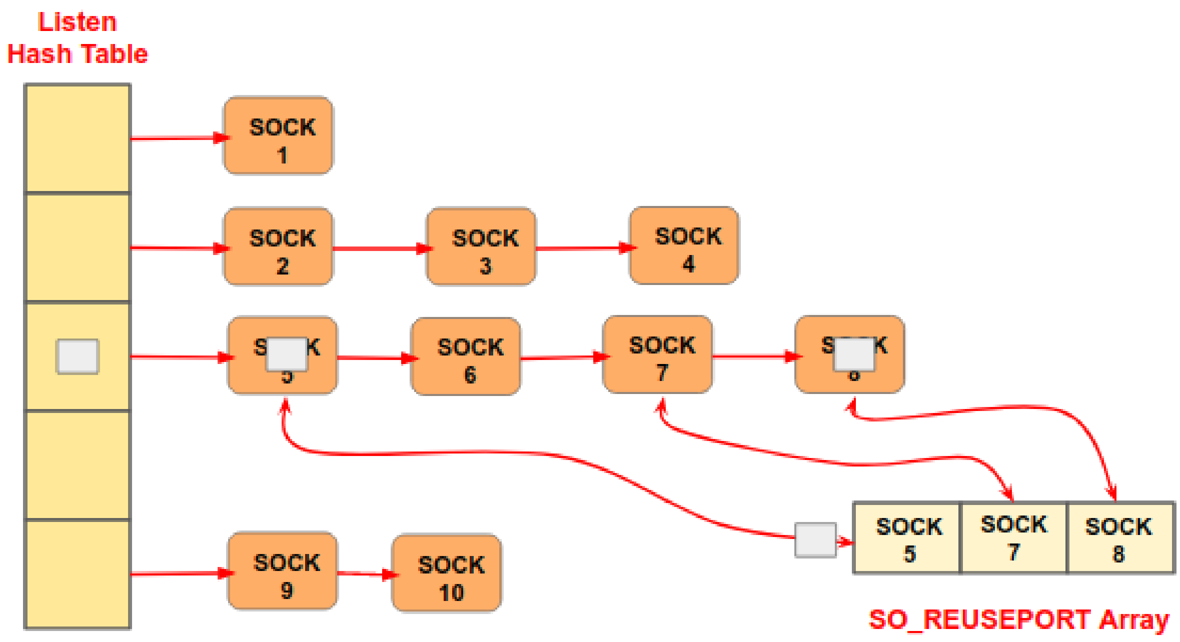


Figure 2.4: SO_REUSEPORT

placed in a global listen hash table as shown in figure 2.4. Listen sockets are hashed in to the global listen hash table by just the port on which the socket is listening for connections. Now, with SO_REUSEPORT the kernel allows for multiple listen sockets to bind to the same port. So, multiple sockets hashed in to the same slot in the global listen hash table can now be listening for connections on the same port. All listen sockets listening on the same port are called to be part of the same SO_REUSEPORT group. In figure 2.4, listen sockets SOCK5, SOCK7 and SOCK8 are part of the same SO_REUSEPORT group. All sockets part of the same

SO_REUSEPORT group point to an array which is simply an array of pointers to listen sockets which are part of this SO_REUSEPORT group. This array is called the SO_REUSEPORT array and we will look at why this data structure is needed.

As soon as a TCP handshake packet comes into the system it must be associated with a listen socket, and the kernel looks up for that listen socket in this global listen hash table. So, SOCK5, SOCK7 and SOCK8 which you see in figure 2.4 here are listen sockets belonging to the same SO_REUSEPORT group or let's say all of these sockets are listening on some port X, and the incoming handshake packet comes on the same port X. The kernel associates this handshake packet with the first listen socket it finds in the global hash table bucket, which is SOCK5 in figure 2.4. Then, the kernel finds out that SOCK5 is part of a SO_REUSEPORT group, so the kernel accesses the SO_REUSEPORT array and then associates this particular handshake packet with one of the listen sockets which are part of the SO_REUSEPORT group, as shown in figure 2.4.

Now the application can have a listen socket for each core. The NIC hardware will distribute handshake packets among the cores on which which the application is running. On each core, the listen socket listening for connections on a particular port X on that core will receive that handshake packet and then the application thread pinned to that core will call accept on that very listen socket extracting established connections. Packet and application processing will also take place completely on a core for each TCP connection as explained before, and hence the Linux kernel is able to deliver complete connection locality in this way.

2.3 Network Stack Research

The current Linux network stack has a lot of issues which we touched upon briefly in the last section. Researchers have been constantly coming up newer network stacks which directly tackle some of the issues which we touched upon in the last section, and some try and fix some of the broader issues with network stacks. A good example would be the Shenango network stack which focuses on maintaining good CPU efficiency and not compromising tail latency of latency sensitive applications in doing so. But, modern network stacks have one thing in common, all of them focus on very high speed network I/O, and hence have removed Linux kernel's involvement from the data path as it is widely believed that the kernel is simply not designed to handle high packet I/O rates.

As shown in the figure 2.5, modern network stacks expose two types of API to the application running on top these stacks. One part of the API handles the control path which involves things like provisioning resources like CPU cores, memory etc. to the network stack or application. The control path tasks as shown in figure 2.5 are delegated to be handled by the kernel as they are not so frequent in nature, and the kernel already exposes API to carry out these particular tasks, and it does not make sense to implement the same thing twice. [5]

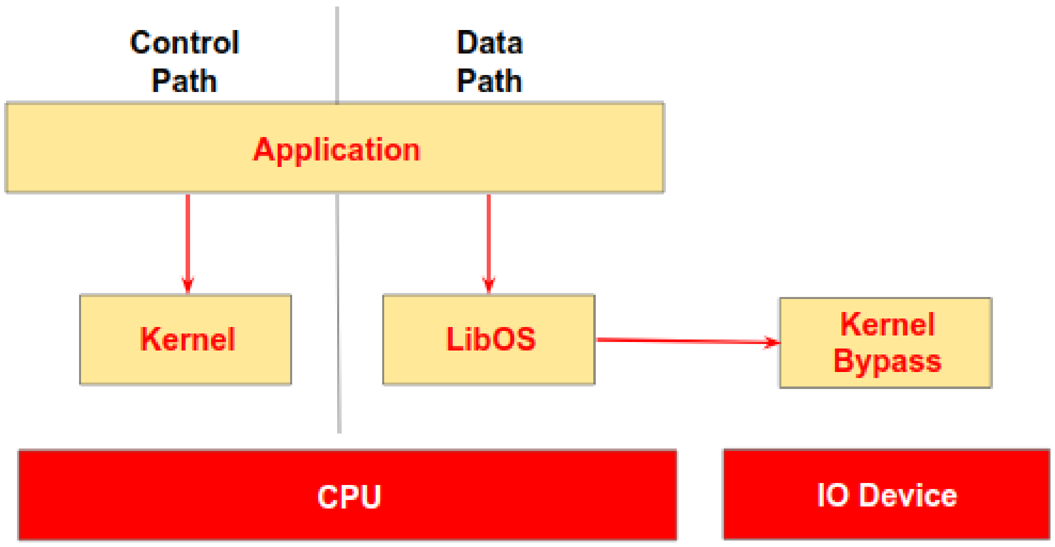


Figure 2.5: Modern Stack Architecture

The data path APIs map into a LibOS, and the LibOS covers all OS functionality that handles I/O, including reading and writing to networking devices. The data path is both performance-critical and CPU-intensive because it must execute on every I/O operation. Each libOS is customized to a kernel-bypass accelerator like DPDK, which implements some OS functionality while the libOS provides the rest.

LibOS is the part where a lot of innovation happens, and researchers implement new techniques to make the data path faster so that applications running on top of these stacks are able to support high network I/O rates. There is another part in the modern stack architecture story which is not shown in figure 2.5, and that is offloads. Modern network hardware now has resources like on-chip CPU, and some amount of on-chip memory which is similar to the resources that any application running in the system has. So, some work of the application and/or the libOS can now be offloaded onto the network hardware, and we will look at some such solutions where researchers have offloaded some part of the TCP processing onto the NIC hardware to reduce load on the CPU on which the application is running and hence the application is now able to spend more time processing requests.

Based on this information, we think that the question to be answered when coming up with a modern network stack architecture is, what features must our libOS support to provide high speed I/O to the application? Although most of the modern network stacks out there support a common set of features like connection locality because that is said to improve the network stack's performance and is considered best practice. But having said that, there a lot of exclusive features which give one network stack an edge over the other. And, another observation we have that these "exclusive" features give the network stack an edge over others on a very specific set of workloads. So, as modern network stacks include more of these complex "exclusive" features, the workloads on which their system does well will also narrow down. In other words, the network stacks will be more or less customised to work for a specific set of applications.

The set of applications we aim to target are 5G network functions, and **come up with designs for network stacks which maximise the 5G network function's performance**. Also, most of the **state-of-the-art research work in the field of network stack design focuses on applications which are I/O intensive**, or in other words the application does not have to do much while processing a request. But, on initial inspection we found that **this was not the case with a lot of the 5G network functions, and many of them were actually CPU intensive**, which means they do a considerable amount of processing on each request. **Network stack designs for CPU intensive applications, like some of the 5G network functions is something that has not been studied in detail yet, and we plan to do so**.

Chapter 3

Existing Network Stack Research

After having gone through the limitations of the Linux kernel stack in the last chapter, we will review some of the state-of-the-art network stacks in this chapter. Most of these network stacks can be neatly categorised into three major categories:

1. **In-Kernel Fixes:** These are optimizations made while working within the Linux kernel network stack.
2. **Kernel Bypass Solutions:** These are network stacks which bypass the Linux kernel completely.
3. **Hardware Offload Solutions:** These are network stacks which offload some of the capability on to the networking hardware.

3.1 In-Kernel fixes

In this section we will review some of the papers which provide improvements to the Linux networking stack while working within the kernel stack framework.

3.1.1 Affinity-Accept

Affinity-Accept [6] tries to solve the single listen socket issue which we looked at in the last chapter. This paper came out in 2012 which was before Linux came up with the SO_REUSEPORT patch in 2014 which provided a solution to the issue. The main idea of the paper is to have core local accept queues, which will prevent any kind of lock contention on the accept queue. And, they also try and have a more fine grained locking mechanism in place where they don't have a single lock protecting both the accept queue and the request hash table data structures inside the listen socket.

Design

It turns out that having per core accept queues and breaking the lock granularity to protect core local accept queues and request hash table by means of separate locks would require a huge amount of modification in the linux kernel network stack. So, instead they choose to create per core clones of the single listen socket as shown in figure 3.1. This would essentially give each core its own accept queue which was their aim all along. In the paper they fail to mention how the listen socket lookup process would look like with the introduction of these per core listen socket clones.

The most easiest solution is to have the original listen socket point to all these per core listen socket clones of itself. Now, during lookup the kernel finds this original listen socket via a process we mentioned in the last chapter, and based on the core on which the packet the packet is being processed on, the kernel will fetch the pointer to the listen socket clone responsible for handling handshake packets for that core. After the kernel is able to locate the core-local listen socket, its business as usual. The kernel obtains a lock on the whole listen socket clone for that particular core and carries on with the operation that it intended to perform on the listen socket. So, as there is a listen socket clone for each core, the kernel is essentially now obtaining locks on per core accept queues.

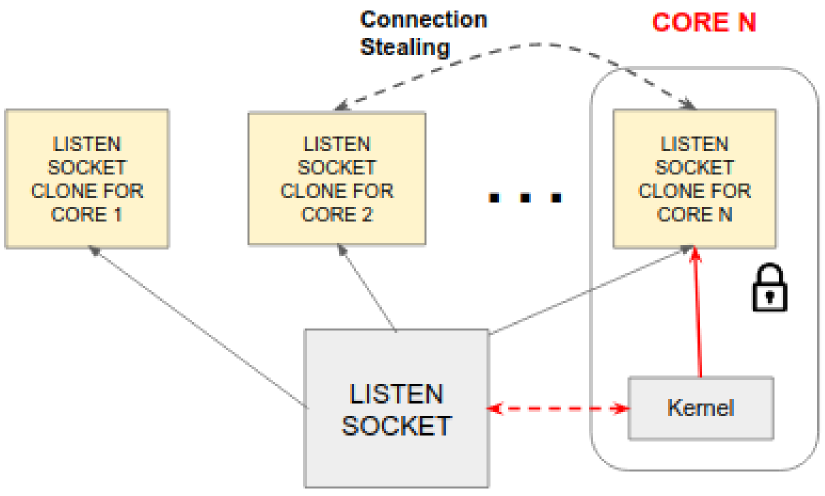

Connection Stealing
CORE N
LISTEN SOCKET CLONE FOR CORE 1
LISTEN SOCKET CLONE FOR CORE 2
. . .
LISTEN SOCKET CLONE FOR CORE N
LISTEN SOCKET
Kernel

Figure 3.1: Affinity-Accept Design

They also allow for connection stealing, where connections from the listen socket clone on one core can be moved to the listen socket clone for some other core, and this is done to balance load between cores better. They made changes in the logic of accept() to facilitate the feature of connection stealing. A call to accept() first looks for established connections on the local core's accept queue, and if does not find any connections there then it looks for connections on some remote busy core's accept queue, and if it does not find any connections there as well, the kernel will then put the thread to sleep. Similarly, they change the logic for poll as well. So, if the kernel gets a connection on an accept queue, it first tries and wake up local threads waiting for connections on that particular listen socket, and if it is not able to find any such threads on the local core then the kernel moves on to threads waiting on remote cores and so on.

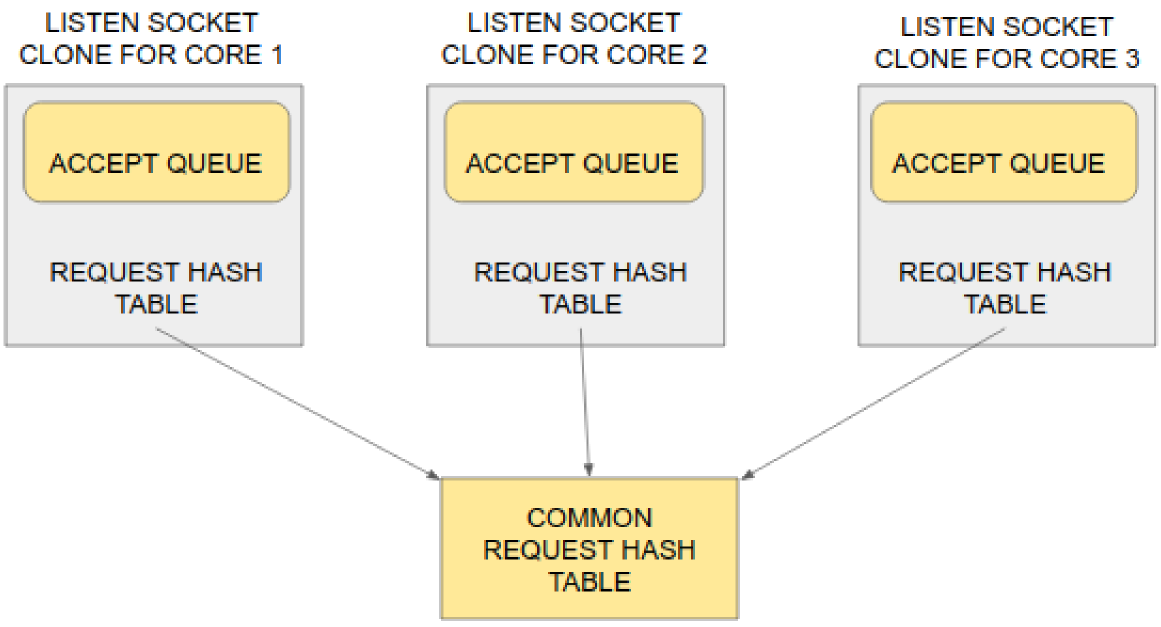

LISTEN SOCKET CLONE FOR CORE 1
LISTEN SOCKET CLONE FOR CORE 2
LISTEN SOCKET CLONE FOR CORE 3
ACCEPT QUEUE
ACCEPT QUEUE
ACCEPT QUEUE
REQUEST HASH TABLE
REQUEST HASH TABLE
REQUEST HASH TABLE
COMMON REQUEST HASH TABLE

Figure 3.2: Listen Socket Clones share a common Request Hash Table

We can see in figure 3.2 that each of the listen socket clone has its own accept queue but all of these listen socket clones share a common request table. The question is why can't you have a per core request hash table when you have a per local accept queue? This is because of the connection stealing which Affinity-Accept supports. Imagine a scenario where you receive a SYN packet on core X, so the kernel finds the listen socket clone responsible for handling packets on core X and makes an entry in the request hash table of that particular listen socket clone, but then this particular connection gets migrated to or stolen by core Y. So, the ACK from the other end would be received on core X, and as the request hash table on core X does not have an entry corresponding to this ACK packet anymore, this connection would be dropped. This issue could be mitigated by first scanning all the request hash tables of all the other listen socket clones for the corresponding SYN packet entry, but a far more easier solution is to have a single request hash table which would contain SYN entries from all listen socket clones, and this is the approach that they choose to take.

3.1.2 Megapipe

Megapipe [9] tries to combine a lot of ideas in a single system. First, it solves the connection locality issue via socket level partitioning which is similar to the approaches adopted by Affinity-Accept and SO_REUSEPORT in Linux. Generally user applications access the sockets in the system via the Virtual File System (VFS). But, they argue that the VFS abstraction is not useful for sockets because of two reasons:

1. sockets are rarely shared especially established TCP sockets hence you don't need the indirection which is provided in the form of VFS
2. sockets are ephemeral unlike files, so creating a bunch of structures in all these tables for just a couple of packets in the case of short connections does not make sense.

So, they eliminate going through the VFS and provide direct access to sockets as shown in figure 3.3.

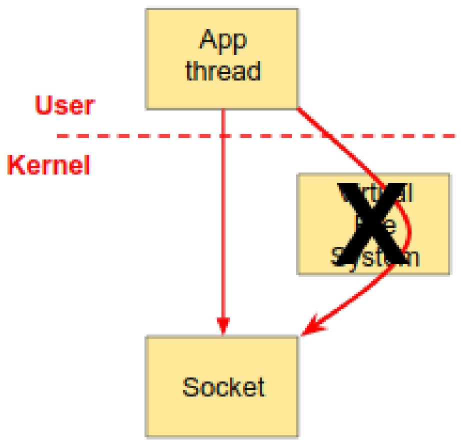


Figure 3.3: Megapipe eliminates VFS abstraction

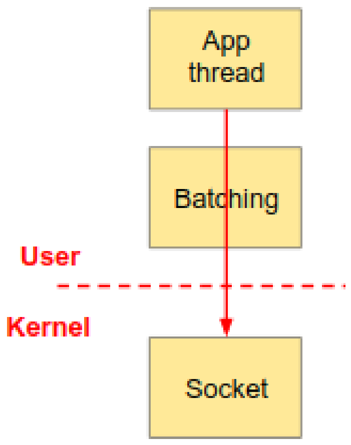


Figure 3.4: Megapipe batches system calls

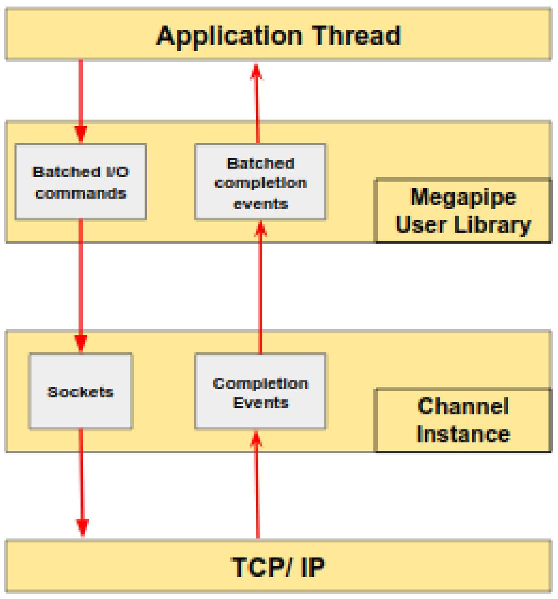
The user application issues I/O requests to the Linux kernel via means of a system call, but there is performance hit related with it due to the privilege level switch as we discussed in the last chapter. So, they implement batching of a bunch of I/O requests made by the user application, and then make a single system call which would service all of these batched I/O requests as shown in figure 3.4. This would essentially amortize the cost of single system calls over a batch of I/O requests.

Their system is based on a new type of network programming model which they refer to as the completion notification model as opposed to the traditional model which they call the readiness model. In the readiness model, we use epoll to check whether the socket is ready for network I/O or not, and only when the call to epoll succeeds we move on to performing the actual I/O. They argue that this check of whether the socket is ready or not is wasteful as most of I/O requests can be carried out instantaneously. So, they subscribe to the I/O completion notification programming model event which is just a fancy term for asynchronous I/O, where the user program gives I/O requests blindly without checking whether the socket is ready for I/O or not, and the system carries out I/O asynchronously, and after I/O is completed it lets the user know via I/O completion notifications which the user constantly polls for.

Design

The first thing you would like to do is to create a channel instance in the Megapipe kernel module, and the call to *mp_create()* at the beginning of the psuedocode in figure 3.5, which is the pseudo code of a ping-pong server working on top of the

Megapipe network stack. A channel instance can be thought of as a container of all the resources needed for network I/O to take place. So, the user application can create multiple channel instances and the user application should ideally create a channel for each core to have core local I/O resources. Each channel consists of two important things as shown in figure 3.5, one is the sockets that are associated with this particular channel instance, and the user application can register a socket to a particular channel instance via the *mp_register()* call as shown in the psuedocode in figure 3.5. The second important data structure inside a channel instance is the completion events data structure which holds a list of notifications of completed I/O requests which were requested by the user the user application to be carried out on one of the sockets in this particular channel instance. The *mp_register()* call returns a handle, which is a unique id for this particular socket inside the channel instance.



```
ch = mp_create( )
handle = mp_register( ch , listen_sd , mask=0x01 )
mp_accept(handle)
while true:
        ev = mp_dispatch(ch)
        conn = ev.cookie
        if ev.cmd == ACCEPT :
                mp_accept(conn.handle)
                conn = new Connection( )
                conn.handle = mp_register(ch, ev.fd, cookie=conn)
                mp_read (conn.handle , conn.buf , READSIZE)
        elif ev.cmd == READ :
                mp_write(conn.handle , conn.buf , ev.size)
        elif ev.cmd == WRITE :
                mp_read (conn.handle, conn.buf , READSIZE)
        elif ev.cmd == DISCONNECT:
                mp_unregister(ch, conn.handle)
                delete conn
```

Figure 3.5: Megapipe Design

The Megapipe user library intercepts I/O requests from the user program and similarly gets I/O completion notification events from the Megapipe Kernel Module which is actually responsible for carrying out I/O requests. The user program issues I/O requests blindly and the Megapipe user library intercepting these I/O requests will not pass these along to the Kernel Module instantly as shown in figure 3.5, rather the user library will batch these I/O requests coming in from the user program and then after some time, flush a batch of I/O requests to the kernel module to carry out I/O. The user library communicates with the kernel module via a special device file by the name of /dev/megapipe with the help of ioctl system calls. On the RX side as well, it gets I/O completion notification events from the Megapipe kernel module in batches but the user application can only get a single I/O completion notification event at a time with the help of the *mp_dispatch()* call as shown in pseudo code in figure 3.5.

Now, having a general understanding of Megapipe's design, let us take a closer look at the psuedo code in figure 3.5. As we already know, the call to *mp_create* creates a channel instance and returns a channel instance identifier to the user program which the user program can include in further Megapipe API calls. Next, you associate an already existing listen socket with this channel instance and supply a *cpu_mask* parameter. So, this would create a new listen socket in the Megapipe system with the *cpu_mask* parameter supplied by the user program, and returns the handle for the newly created listen socket in this particular channel instance. Next, we call *mp_accept* on the listen socket handle which will enqueue an I/O request to accept connections on this listen socket, and then will move on without blocking the user program. Next, the program goes into an infinite loop, and the first call inside the loop is to *mp_dispatch* will fetch a single completion notification from the Megapipe user library, and each completion notification has a cookie attached to it which identifies the connection on which I/O was actually done. Next we have if statements based on the type of I/O completion notification, so, if read was successfully completed on a socket, then you would like to write to that socket because of it being a ping pong server, and vice versa when a write is completed on a socket you would like to read from that socket. And, when an accept is successfully completed on a listen socket, first you make a call to accept again on the same listen socket to get more established connections, and then you would do a read I/O request on this new established socket after registering this new established socket with a particular channel instance.

3.1.3 Fast-Sockets

The main idea of fast sockets [13] is to have a table level partition instead of a socket level partition approach which papers like Affinity-Accept and Megapipe and even SO_RESUEPORT in the Linux kernel take as shown in figure 3.6. They argue that even though the solutions before provide different sockets for each core but that is not enough as all of the sockets are accessed via a global hash table which all of these core share, and hence pose a problem in case of multi core scalability.

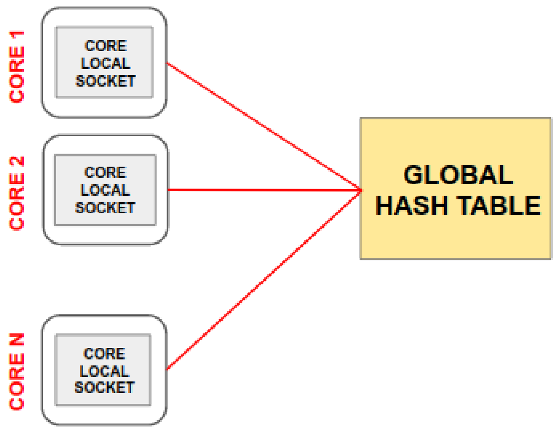


Figure 3.6: Socket level partitioning

The paper lacked any concrete evidence of the fact that a global hash table used to access core local sockets would be a bottleneck in multicore scalabilty. So we did some experiments of our own to establish the fact. We ran a server on one machine and a client on another machine, and the client established some fixed number of connections with the server, 512 in our case. Then, the client continuously sent packets over those established connections. So, in theory, the client establishes X connections with the server and then keeps on sending data. On the server side, we measured the average time it took for the kernel to perform a single socket lookup in the global hash table as we increased the number of cores we ran the server on. Ideally, we should not see an increase in lookup time in a hash table as we increase the number of cores, but as you can see in the graph in figure 3.7, the socket lookup time does increase as we increase the number of cores the server runs on. The average time of performing a single socket lookup in the global hash table is around 212 nanoseconds when the server runs on a single core as opposed to 710 nanoseconds when the server runs on 4 cores. We don't change the number of connections, and the connections are established on the same ports as we wary the number of cores, so we can be sure that the socket placement in the hash table is consistent across all experiments with the only differing factor across the experiments being that the server runs on different number of cores. This essentially means that as we increase the number of cores on which the application runs, the time it takes for the kernel to perform a single socket lookup in a global hash table increased. This acts as some sort of evidence that the global socket hash table is a bottleneck in multicore scalability.

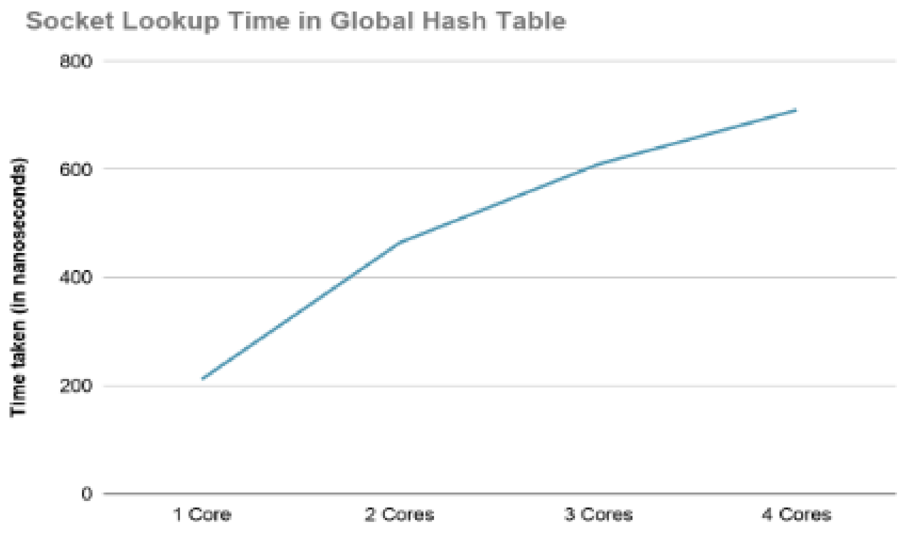


Figure 3.7: Average Socket Lookup Time

Design

As you can see with their design in figure 3.8, the most obvious solution is to have local socket tables for each core the application runs on. Receive Flow Deliver (RFD) is responsible for directing packets to their respective cores on which their complete processing will take place. And with these local socket hash tables containing sockets responsible for connections to be processed on that core, and the Receive Flow Deliver module, they are able to create a complete Per-Core Process Zone as shown in figure 3.8. Just like Megapipe, they also argue that the Virtual File System overhead is unnecessary in case of sockets, but instead of changing the API completely to bypass the VFS which megapipe does, they try to do some plumbing in the VFS itself to fix the issue, and they call this new improved VFS, Fastsocket aware VFS. There are two major components in the VFS, inode and dentry, and sockets don't use both as they are never stored on disk and never accessed through the directory structure, so they skip initializing and maintaining a lot of such state for sockets which reduces the VFS overheads. As shown in figure 3.8, the application accesses these data structures via this Fastsocket aware VFS.

They classify connections as either active or passive. A passive connection is where the connection is initiated by a client. An active connection is where the connection is initiated by the server. Papers like Affinity-Accept, Megapipe only talk about connection locality for passive connections where the server is the receiver of the SYN packet. Fast-Sockets guarantees complete connection locality for both

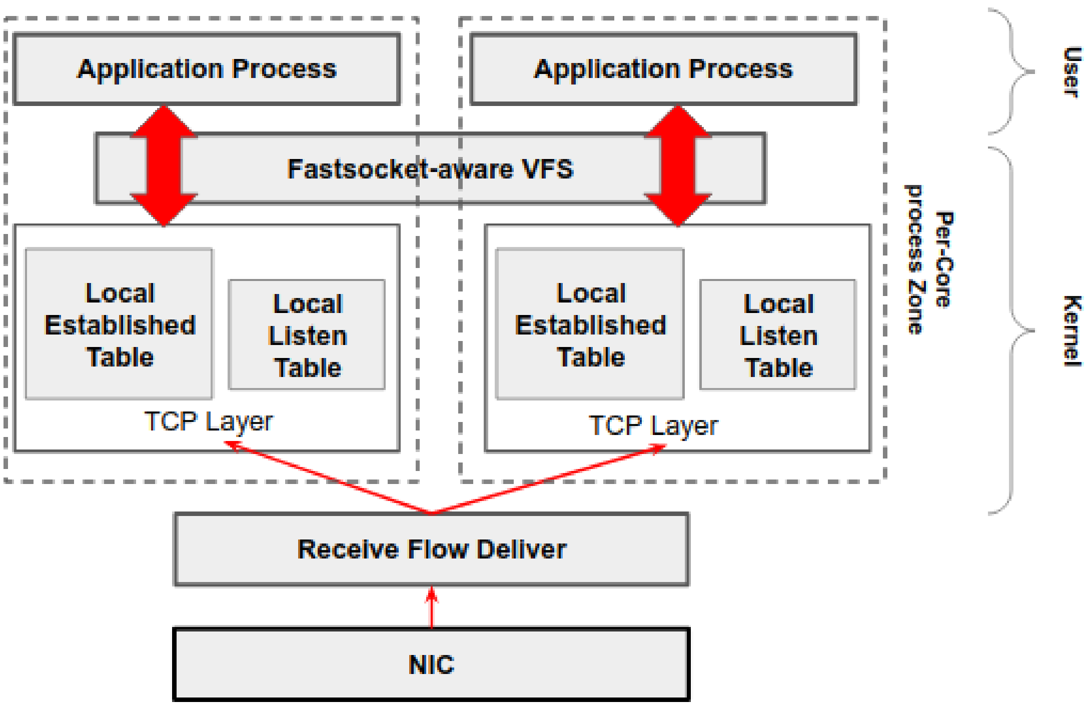


Figure 3.8: Fast Socket Design

passive and active connections. In the case of active connections, the application first consults the RFD module of the system as shown in figure 3.9 regarding the port on which to send the SYN packet on. Basically, if the application tells the RFD the source IP address, destination IP address, and the destination port of the packets you are going to send, and the core on which you wish to process the packets, the RFD can tell you the port on which you have to send the packet on such that the incoming packets are redirected to the core you want to process the packets on. This works because RFD distributes packets among the cores on which the application runs on using the 4-tuple (Source IP, Source port, destination IP, destination port) information embedded in the packet. In this way, Fast-Sockets guarantees connection locality for even active connections.

3.1.4 StackMap

Network stacks can be classified into two categories:

1. **In-kernel workaround**, which are the stacks we have been looking at, and the way they try to improve performance is by improving the socket API, and these solutions make use of the complete TCP kernel stack.

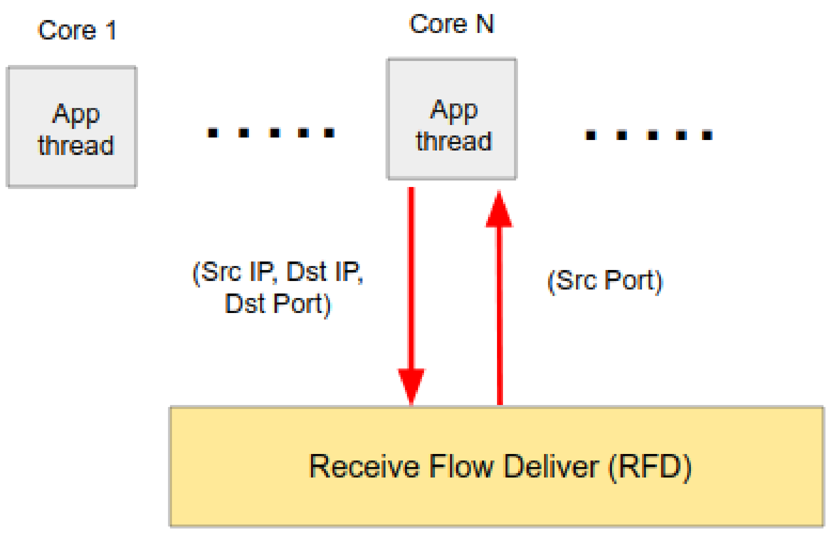


Figure 3.9: Connection Locality for Active Connections

2. Another category is **kernel bypass approach**, which we will look at in the upcoming sections, and these dedicate NICs to one application, and employ different techniques which are made possible by the network stack now running in user space.

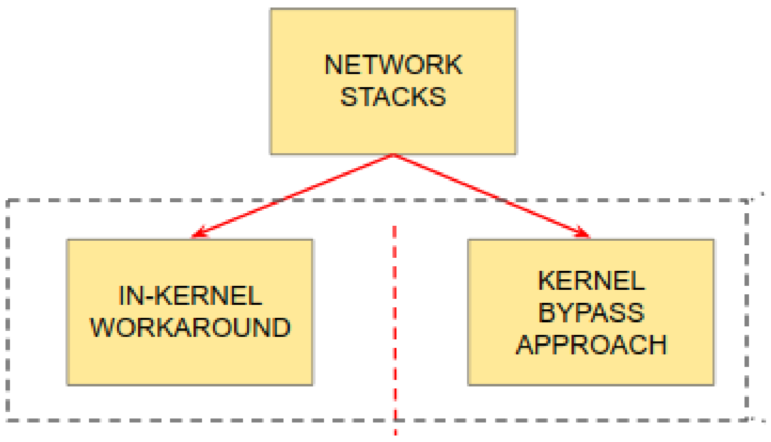


Figure 3.10: Classification of Network Stacks

Stackmap combines both of these approaches where they dedicate NICs to applications using their system (similar to a kernel bypass solution), but also make use of the complete TCP/IP stack (similar to an in-kernel solution). One of the design goals is to make use of state-of-the-art techniques used in recent papers to improve network I/O like zero packet copy I/O and system call batching. Finally, their system is based on an extension of Netmap and we'll look at how when we get to their design.

Netmap Design

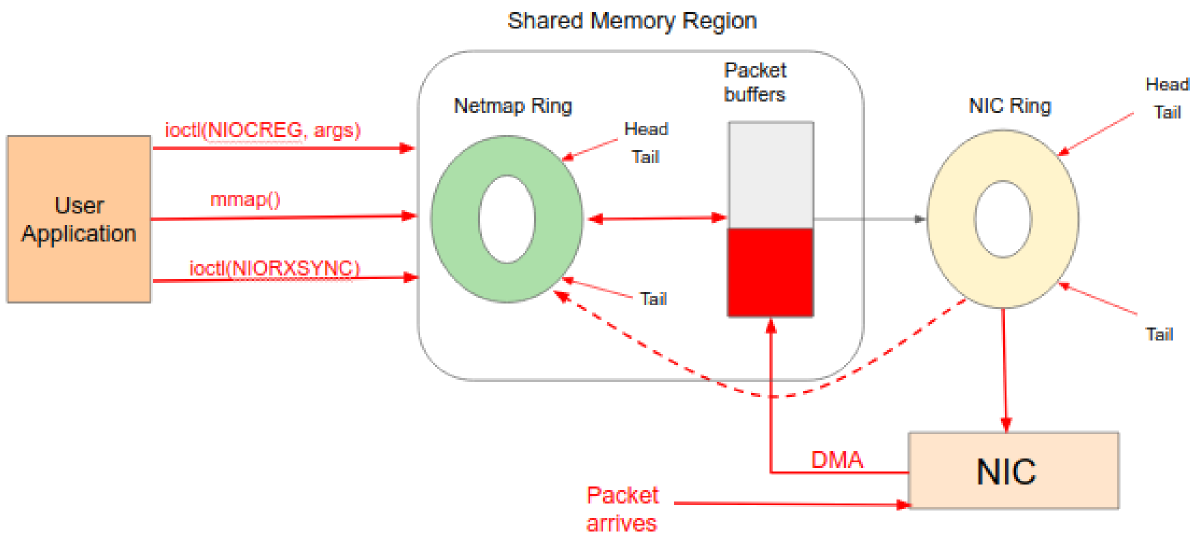


Figure 3.11: Netmap Design

The user application communicates with netmap [11] through a device file with the help of ioctl calls. The first ioctl call is issued with the flag NIOCREG as shown in figure 3.11, which initializes the shared memory region and the data structures inside it using the arguments passed in the ioctl call.The major data structures include a netmap ring + fixed number of preallocated constant size packet buffers. The NIC ring is also filled up with descriptors referring to these preallocated packet buffers. The user application can use mmap to map this shared memory region in its address space. Now, let us look at RX processing of a packet, as soon as a packet comes in on the NIC, the NIC fetches a descriptor from the NIC ring and using the descriptor DMA's the packet to one of the packet buffers. And, this would update the tail of the NIC ring to indicate in-use descriptors. Then after some time, the application will issue a NIOCRXSYNC ioctl call which will update the netmap ring to synchronize it with the NIC ring, so the netmap ring will now contain packets which were DMA'd recently by NIC. Its important to note here that each packet buffer is identified uniquely by an index so during synchronization only packet buffer indexes are transferred from NIC ring to netmap ring, and hence no packet copy takes place. And, as the netmap ring and packet buffers are mapped into the user address space, it can consume these packets directly at the cost of a single ioctl system call. Something similar will happen on packet transmission. It also provides protection in the sense that the application only has access to the shared memory region which it memory mapped in its address space, and the OS won't let it access memory outside of this memory area and as this memory area does not contain any kernel data structures, a misbehaving application can only hurt itself. Netmap is able to support the following features via this design:

- **Zero-Copy** : The application is able to consume the data without having to copy the packet data from the kernel buffer to the application buffer.
- **Amortize cost of a single system call** as multiple packets are consumed in a single ioctl system call.
- **Protection** : Application is still not able to access any critical data structures.

StackMap Design

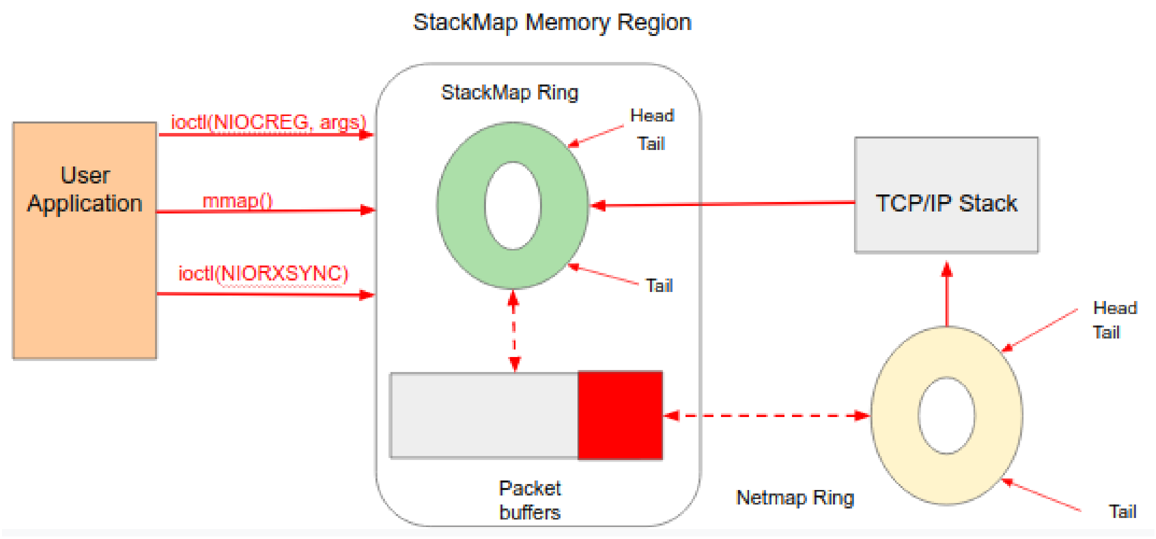


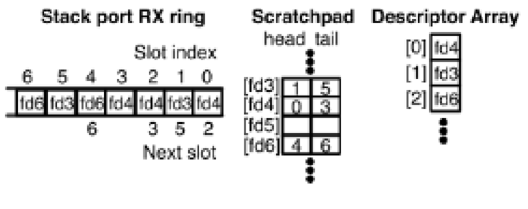
Figure 3.12: StackMap Design

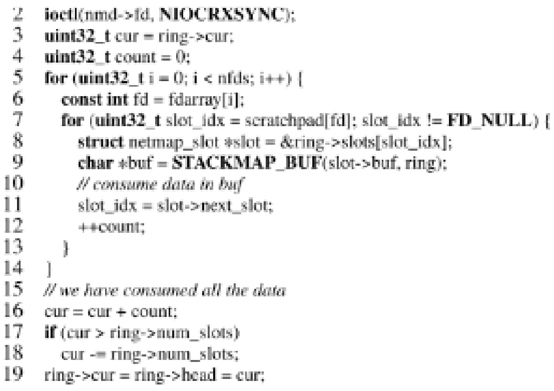
StackMap [17] adds a ring to the netmap design, and this ring is called the stackmap ring as shown in figure 3.12. So, as before application running on top of stackmap will first issue a NIOCREG ioctl call which would initailize the stack map memory region, and then the user can mmap() the stack map memory region into its address space. Let's look at the RX call flow now. As soon as packets come in, the NIC DMAs them into pre allocated packet buffers using descriptors in the NIC ring as before. Then, after some time the user will issue an NIORXSYNC ioctl call, which will synchronise the netmap ring with the NIC ring, process till here is similar to what it was in netmap. But, after getting packets in the netmap ring, it injects these new packets into the TCP/IP stack of the linux kernel for TCP processing and after processing identifies identifies whether a packet is in-order or not, in-order packets are moved to the stack map ring, out-of-order packets are held back and transferred later when the hole is filled. So, the stackmap ring at any given point in time only contains in-order data packets for any connection. As soon as packets get to the stack map ring, user applications can now access them. There are two important points to note here:

1. Pre allocated sk_buff structs are maintained for each packet buffer because packets are represented as sk_buff inside the kernel. Before passing on a packet from a packet buffer to the TCP/IP, the packet is wrapped into the corresponding pre allocated sk_buff structure for that particular packet buffer.

2. Application running on top of stackmap makes use of all socket system calls except for read and write, so TCB is still maintained in sockets inside kernel. The read and write are replaced by stackmap versions, but even these call modify state for that particular connection in the socket corresponding to that connection maintained inside the kernel.

StackMap Data Structures





```
ioctl(nmd->fd, NIOCRXSYNC);
uint32_t cur = ring->cur;
uint32_t count = 0;
for (uint32_t i = 0; i < nfds; i++) {
   const int fd = fdarray[i];
   for (uint32_t slot_idx = scratchpad[fd]; slot_idx != FD_NULL) {
      struct netmap_slot *slot = &ring->slots[slot_idx];
      char *buf = STACKMAP_BUF(slot->buf, ring);
      // consume data in buf
      slot_idx = slot->next_slot;
      ++count;
   }
}
// we have consumed all the data
cur = cur + count;
if (cur > ring->num_slots)
   cur -= ring->num_slots;
ring->cur = ring->head = cur;
```

Figure 3.13: StackMap Data Structures

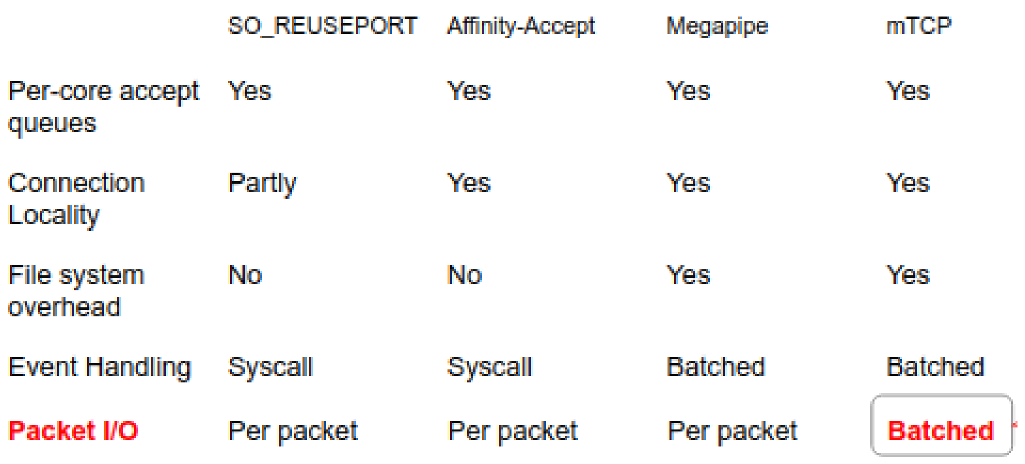
If you look on the extreme left in figure 3.13, each slot in the stackmap ring has 3 things. First, the index of the buffer where packet data is present. Second, the fd of the connection socket containing TCB state with which this packet is associated. Third, next slot index in the ring which contains the next packet received for this connection. There are two ways in which the user can read these packets. The first way a user can read packets is inorder of packet arrival, where they read packets starting from the head of the stackmap ring and keep reading until you reach the tail of the ring. The user application can also read data connection wise as applications usually would do with something like epoll where first the application checks which connections have data available and then read available packets for each connection one by one. Stackmap exports two additional data structures called the descriptor array and scratchpad to facilitate this feature. The descriptor array on the left in figure 3.13 behaves like the epoll interest list which tells us which connections have packets available to read. The scratchpad in the middle in figure 3.13 maintains the head and tail slot indexes in the stackmap ring where packets are placed for each connection present in the array descriptor. So, a user program as shown at line 6 in the pseudo code in figure 3.13 would first pick a fd from the descriptor array, then would identify the head slot where data is present for that connection from scratchpad as it does in initialization part of the for loop at line 7 and then consume all data using the next_slot info until it hits the tail slot for that connection, and this process is shown in the inner for loop from lines 7-13. And, then the outer for loop starting from line 5 would help repeat this process for each fd in the descriptor array.

3.2 Kernel Bypass Solutions

In this section we will review some of the papers which improve upon the Linux kernel stack by designing a data path which completely bypasses the Linux kernel.

3.2.1 mTCP

The main idea of the paper [10] is to have a user level network stack which is optimized for multicore scalability of TCP. So, the first question that we would like to answer is why do you need a user space stack? To answer this question, let's look at a table with different network stacks which we have seen until now and the features they support. mTCP came out in 2014, and Megapipe was state-of-the-art at that time, as it had come out in 2012. StackMap and Fast-Sockets came onto the scene a little later in 2016, so we leave StackMap and Fast-Sockets out of the table in figure 3.14. First, we have per core accept queues and all of the four network stacks listed, Linux with SO_REUSEPORT, Affinity-Accept, Megapipe and mTCP, support this feature. Similarly all of the listed stacks provide connection locality. Next, only Megapipe and mTCP deal with File system overheads. Similarly, multiple events are handled at the cost of a single system call, or a batched event handling technique is present in Megapipe as well. So, if you look at the table it becomes clear that Megapipe already supports a lot of the features even though it is a kernel level network stack.



	SO_REUSEPORT	Affinity-Accept	Megapipe	mTCP
Per-core accept queues	Yes	Yes	Yes	Yes
Connection Locality	Partly	Yes	Yes	Yes
File system overhead	No	No	Yes	Yes
Event Handling	Syscall	Syscall	Batched	Batched
Packet I/O	Per packet	Per packet	Per packet	**Batched**

Figure 3.14: Why a User Level Network Stack?

The major distinguishing factor between a kernel level network stack like Megapipe and a user level network stack like mTCP is that packet I/O is batched in case of user level network stack. Batched packet I/O means that the network stack is able to obtain a batch of packets from the network device driver which allows the network stack to process these batch of packets together which has an impact on performance, and we will also discuss some of the sophisticated features that user space network stack provide are kind of built on top of this batched packet I/O feature. This is made possible by packet I/O libraries like netmap and DPDK. There are other apparent advantages with user space network stacks as well, one of which is that you don't have to make modifications to the kernel which is very difficult to do. It's easier to make modifications to the TCP stack because now the TCP stack is completely decoupled from the kernel.

Design

Let's have a look at the design overview of mTCP. As we can see in figure 3.15, mTCP will pin each of the application threads to a core and will also pin a corresponding mTCP thread to run on the same core. The mTCP thread is where the packet processing happens. So, they ensure connection locality by running the mTCP thread, which does TCP processing and the user application thread which does application level processing on the same core. Each mTCP thread will also initialize its own network stack data structures, in complete isolation from the network stack data structures for mTCP threads running on other cores. This is what we mean when we say that mTCP takes a shared-nothing approach. Essentially, you have a separate instance of a network stack running on each core completely oblivious of other instances of the network stack running on other cores. No locking is required to protect the network stack data structures as each data structure belongs to a separate instance of a network stack, each of which are again running on separate cores. In essence, all state inside the network stack is now only accessed from a fixed single core. The mTCP thread is responsible for network processing of packets in batches, so it'll grab a batch of packets from the NIC, and process them together and generate a bunch of events for the application to consume. Similarly, the mTCP user library runs as part of the application thread, and this user library can be used to pick up events generated by the mTCP thread, process them and generate corresponding packets for the mTCP thread to transmit. These two threads communicate via a bunch of event queues, and there are events queues for each core the mTCP application runs. These event queues again need no locking because at any given point in time both the producer and consumer for an event queue won't be running simultaneously as they are both pinned to the same core. It's important to note here that when switching from application level to network stack processing, the Linux kernel had to go through a system call. In case of mTCP, the system has to go through a context switch, as the network stack itself is running in user space but in a different context(thread). They mention in the paper that a context switch is 17 times more costly than a system call. They try and amortize the cost of the context switch which they are forced to do by generating as many events as they can before switching to the other context.

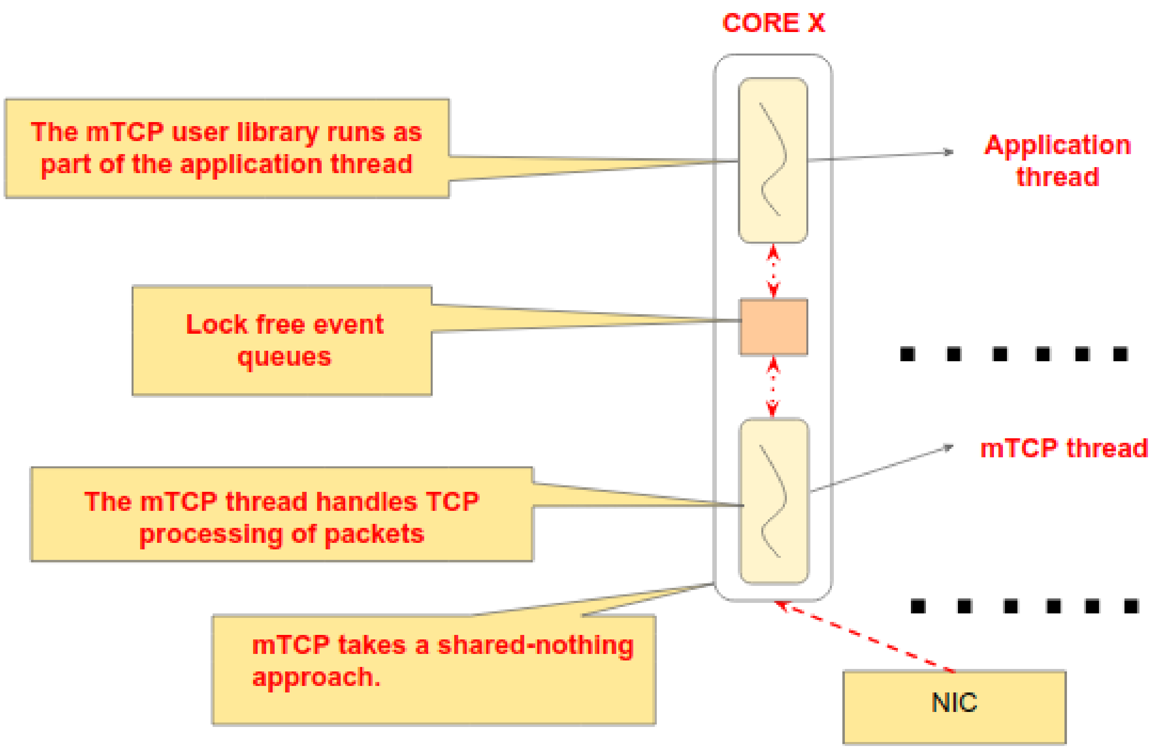

CORE X
The mTCP user library runs as part of the application thread
Application thread
Lock free event queues
mTCP thread
The mTCP thread handles TCP processing of packets
mTCP takes a shared-nothing approach.
NIC

Figure 3.15: mTCP Design

Implementation

Let's take a deeper dive into the mTCP design and start by looking at the rx path and then the tx path. First of all the mTCP thread will receive a batch of packets from the NIC. Now, if the packet is a SYN-ACK packet which essentially means that it has received a new connection, mTCP will do two things:

1. It will add the new connection to the accept queue
2. It will generate a read event for the listening socket and add it to the internal event queue.

If the packet is a normal data packet, mTCP will pass it on the payload handler which would do TCP processing and after that will again do 2 things:

1. It will generate an event for the corresponding established socket and would place it in the internal event queue
2. Would generate corresponding ACK for the packet, and place it in the ACK list for TX, so an important point to note here is that mTCP does not piggy-

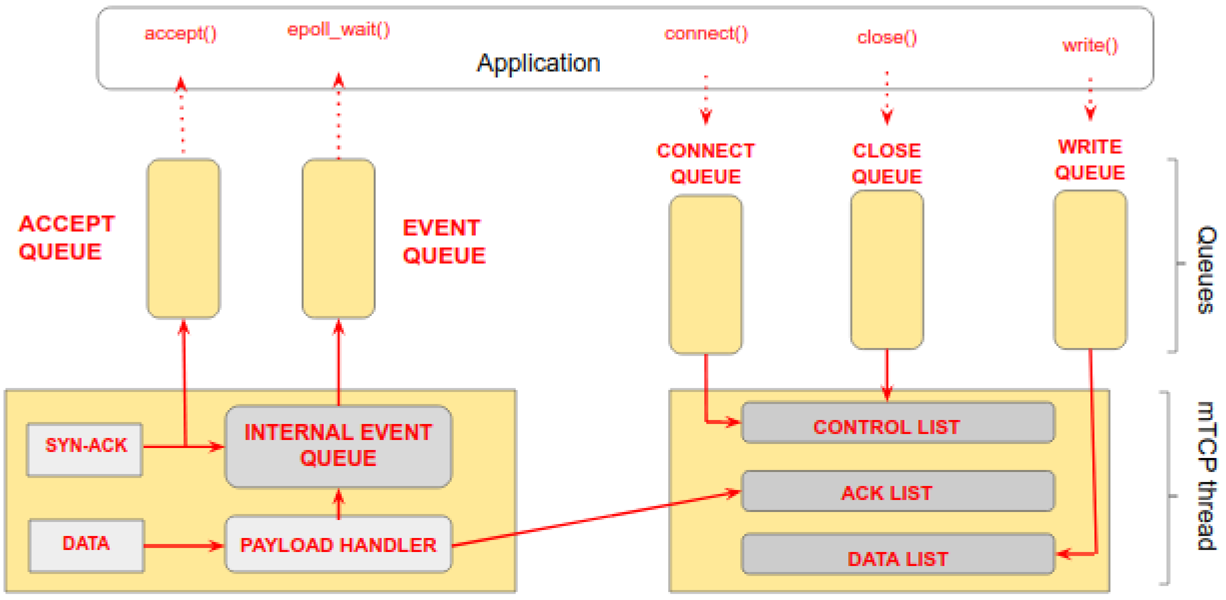


Figure 3.16: mTCP Implementation

back data packets with acknowledgment, but rather sends ACKs as separate packets.

After processing a single batch of packets, the mTCP thread will flush all generated events to the event queue, and will signal the application thread to run. The application thread will process these events generated by the mTCP thread, and in response can do 3 things:

1. It will try and establish new connections which place an event in the connect queue.
2. It will try and close a connection which will place an event in the close queue.
3. It will generate packets for transmission by placing an event in the write queue.

When the mtcp threads gains back control, it'll pick events from connect queue, which will generate packets to send and place those packets in the control list, and events from the close queue will also generate packets into the control list. Similarly, processing the write queue will generate packets into the data list. mTCP optimizes for short connections by sending out control packets from the control list first, then acks from the ack list and data packets from the data list at the end, because control packets are a priority when looking for high performance for short connections and if control packets get lined up behind data packets, then this could have an impact on the throughput of the total number of connections processed.

3.2.2 IX

The main idea of the paper is to break the 4-way tradeoff between high throughput, low latency, strong protection and resource efficiency. IX [7] borrows a lot of network stack design techniques from mTCP, for example IX also follows a shared nothing approach where it effectively creates a new instance of the network stack running on each core and uses hardware to distribute packets among these different instances of the network stack. IX also makes use of batching, similar to mTCP for TCP processing which results in less number of instruction cache misses. But, having said this they also identify a lot of flaws in the system and provide appropriate extensions. They identify three major issues with the system, and we'll mention two of them in this section and then we will mention the third issue a little later when describing their design.

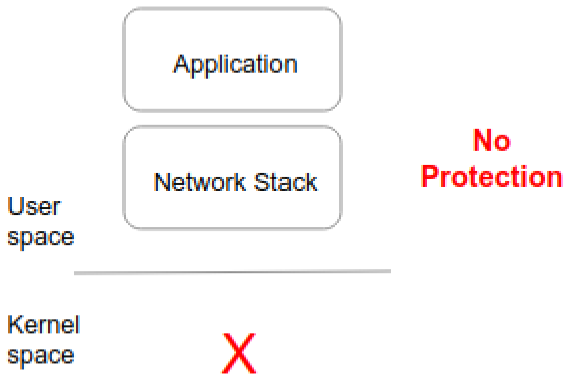


Figure 3.17: User level Network stack not protected

As we can see in figure 3.17 in the case of mTCP, or any other user level stack for that matter, both the user application and the network stack run in the user space, and nothing runs in kernel or protected space which leaves the network stack a little vulnerable as it is not protected, and the user application can tamper with network stack data structures.

Another issue with user space network stacks like mTCP is that resources required for the functioning of the network stack are acquired during initialization and kind of locked in as shown in figure 3.18, and this hurts the efficiency with

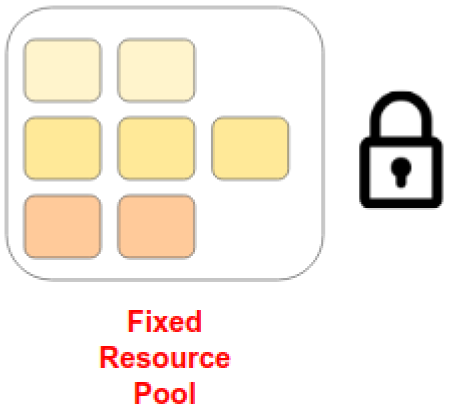


Figure 3.18: Fixed Resource Pool

which said resources are used as there is basically no provision to scale up or scale down the resource pool which the network stack is using.

Design

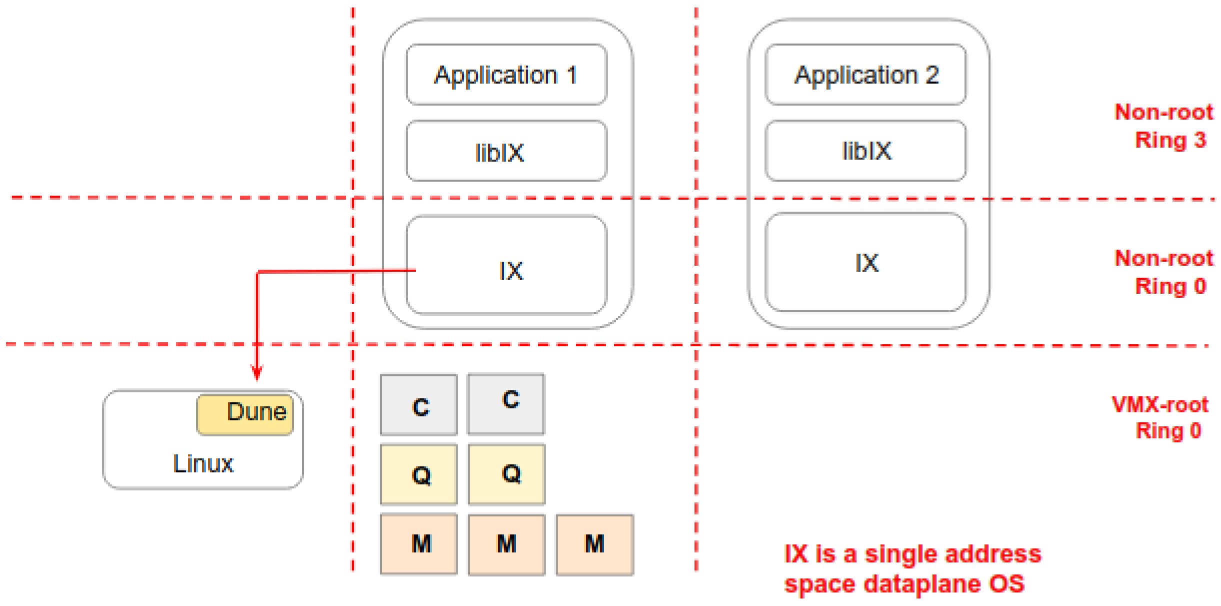


Figure 3.19: IX design

To overcome the first issue, IX makes use of hardware virtualization as we can see in figure 3.19. The IX network stack runs in ring 0 in VMX non root mode, and the application along with the IX user library which functions similar to the mTCP user library as seen before runs in ring 3 in VMX non root mode. So, the

IX network stack is protected from the application code as they run in different privilege levels and the hardware will throw exceptions whenever the application code tries to tamper with the network stack. But, this gives rise to another issue here. As IX is running in non root mode, it'll experience VMExits to ring 0 in VMX root mode on network I/O because code running in VMX non-root mode does not have enough privileges to access NIC registers. This is not acceptable as a VMExit on every network I/O will seriously hurt network stack performance. To overcome this problem, IX makes use of the Dune framework along with KVM, and Dune will give privileged access to some hardware resources to code running in ring 0 in VMX non root mode. So, now with the help of the Dune hypervisor, IX has direct access to NIC hardware and network I/O now will no longer result in VMExits. One important thing to note here is that IX is a single address space Operating System, which means that a single instance of IX will only support a single network application to run on top of it, and there are two obvious advantages of doing so:

1. You don't have to manage memory between multiple applications which saves a lot of time, and consistent switching between page tables of different address spaces could also corrupt the TLB.

2. You don't have to switch address spaces multiple times as this would corrupt the data cache and instruction cache and hence hurt network performance.

A different application will then run on top of a completely different IX instance as shown in figure 3.19. They also allow for the IX network stack to scale their resources up and down. They identify 3 major resources which a network stack uses and might need to scale based on load:

- CPU Cores
- NIC Queues
- Memory pages

It's important to note that these resources are allocated to the network stack at a coarse grained level. Which means that the unit of allocation is not fine grained, for example, a network stack can only be allocated memory of size in multiples of the size of a single memory page. So, the network stack can scale its resources by making a request to the Linux kernel via the Dune hypervisor and the kernel will allocate additional resources, for instance additional memory pages, or an additional NIC queue and respective CPU core.

Run To Completion

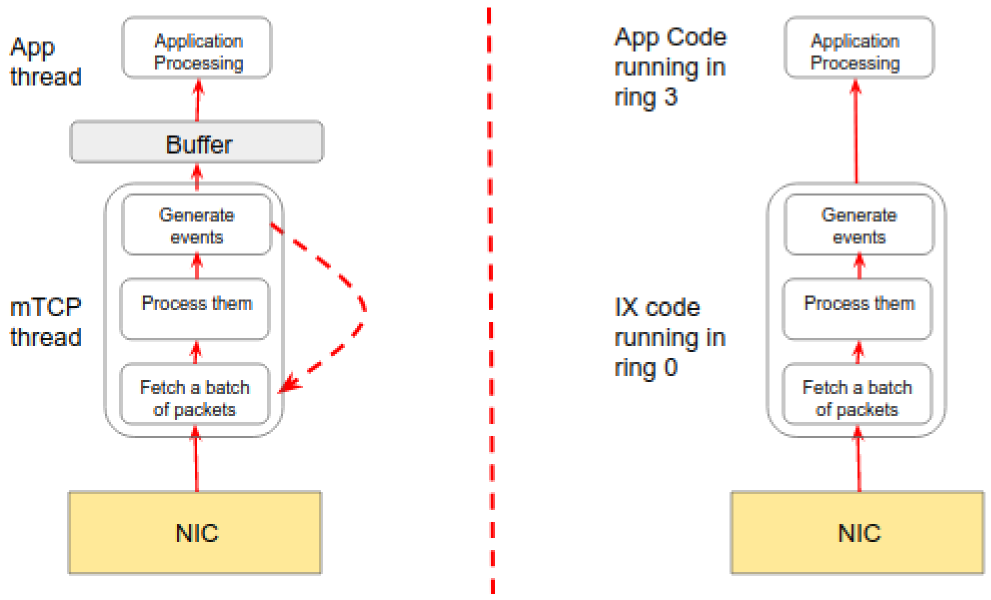


Figure 3.20: Run TO Completion Model

We move onto the third major issue which they had recognised with major user level stacks like mTCP. As we can see on the left of figure 3.20, the mTCP thread does 3 things:

1. It fetches a batch of packets from the NIC.

2. It processes these packets together.

3. Generates events after processing a batch of packets and buffers these events in a queue for the application to later process them.

The mTCP thread will loop many times repeating the same process, and the reason why they loop is to generate as many events as possible before context switching to the application thread such that the cost of the context switch is amortized over a large number of I/O requests. Basically, they try and find a sweet spot so as to process as many packets as possible at the cost of a single context switch, and also they are mindful of the fact that the mTCP thread does not eat up a lot of the CPU cycles as the application thread again runs on the same core. IX says that this approach is not ideal, and even though mTCP might get good throughput number by using this approach, but they will have a latency hit because they buffer events generated by packets for some time while the mTCP thread tries to generate more events. IX comes up with an alternative approach which is run to completion which is shown on the right in figure 3.20. This approach is pretty similar to the on the left but they do not do any kind of buffering of events. In this model a batch of packets are picked up from the NIC, processed and then control is transferred to the application running in ring 3 which processes these generated events and then in-turn generate packets for tx. So, because there is no buffering involved between rx of a packet and corresponding tx, this helps IX attain decent tail latency when compared with other user space stacks like mTCP.

3.2.3 ZygOS

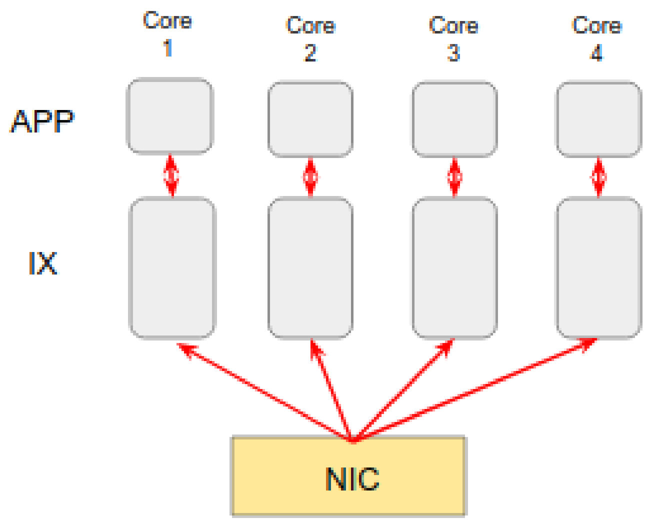


Figure 3.21: IX follows Distibuted FCFS model

ZygOS [19] is a paper by the same research group that worked on IX, so the paper mainly tries to work on improvements to IX. And naturally they also inherit a lot of the features from IX like shared nothing dataplane approach, run-to completion, and making use of virtualization to provide protection. As we can see in the figure 3.21, in the case of IX, the NIC hardware will distribute incoming packets among all the NIC queues present in the system, and then each of these NIC queues will be mapped with a particular CPU core which will pick up packets from this NIC queue and will process them, as shown in the figure. Formally, we can say that IX follows a partitioned-FCFS model, and what we mean by this is that each core in IX has kind of a queue of its own from where it picks requests from and processes them. There are obvious pitfalls of this model, and the major one is load imbalance. If a particular client generates requests at a much higher rate compared to the other clients, one of the cores will be overworked as compared to others and this will have an impact on the overall tail latency of the system. Another common problem is service-time variability, wherein if a request takes significantly higher time to process, then all requests with low service time will be blocked behind this request with high service time and this will again have an impact on tail latency of the system. They conducted some experiments and found that centralized-FCFS model was better than the partitioned-FCFS model in such cases. In the case of a centralized-FCFS model, you have a single queue of requests and the request at the front of the queue can be processed by any of the cores, and this is far better in terms of tail latency when compared to the partitioned-FCFS model because this request at the head of the queue could be serviced as soon as a core becomes available.

Design

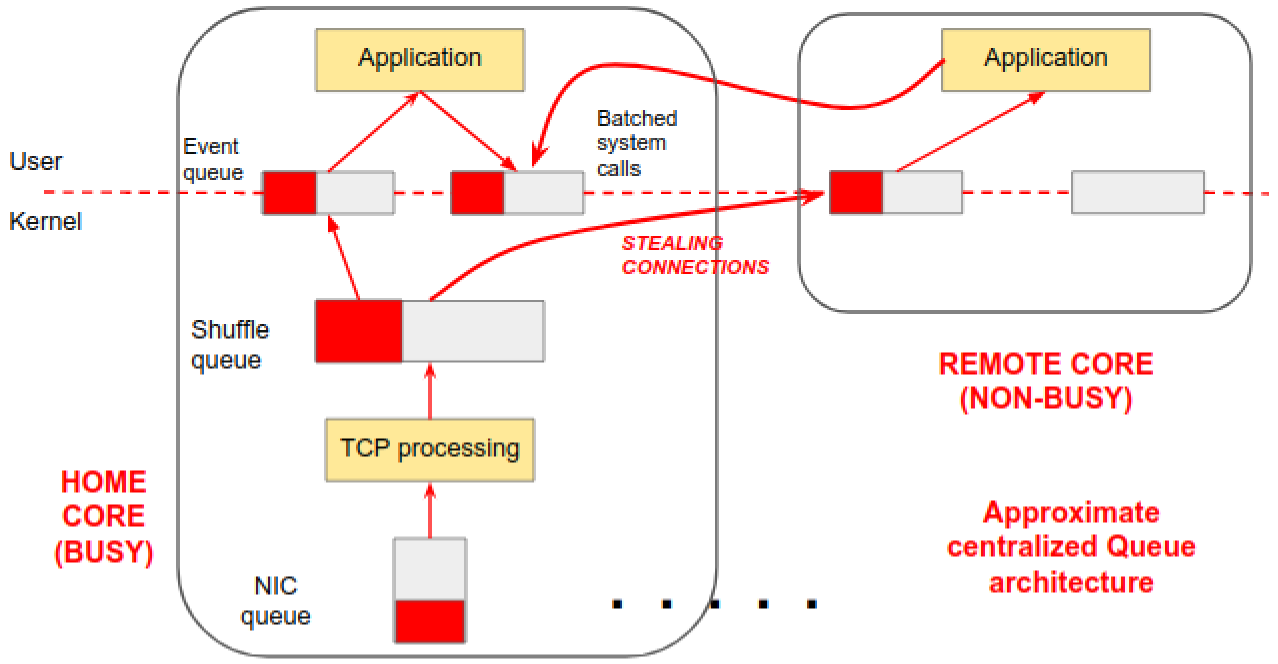


Figure 3.22: ZygOS follows an approximate centralized-FCFS model

Let us look at the ZygOS design more closely. Similar to IX, the ZygOS network stack executes in VMX non-root mode ring 0. The ZygOS network stack is responsible for polling packets present in the NIC queue and then processing them. But unlike IX, where in the network stack directly passes the generated events to the application, in ZygOS, processed connections or TCBs are placed in an intermediate buffer which is called the shuffle queue as shown in figure 3.22. And, after placing TCBs in the shuffle queue after processing a batch of packets, the ZygOS network stack picks TCBs from the shuffle queue one by one, and generates events corresponding to each connection for the application to consume and places them in the event queue as shown in figure 3.22. After this the control is switched over to the application running in VMX non-root mode ring 3. These event queues are the interface through which application and the ZygOS network stack communicate. The application picks events from the event queue, processes them one by one and generates a bunch of system calls for the network stack to carry out, and this loop continues. Let's call this core the home or the busy core which just processes packets which were directed to it by the NIC hardware The process looks a lot like IX in the sense that the network stack processes packets directed to it by the NIC hardware in a run-to-completion fashion. The only apparent difference between IX and ZygOS as shown in figure 3.22 is the shuffle queue where ZygOS performs intermediate buffering of TCBs of the packets which it processed in the last cycle, and this intermediate buffering is missing in IX. It turns out that this is what enables connection stealing to be incorporated into the ZygOS stack which was not possible in IX. So, a non-busy or a remote core can steal connections from the shuffle queue of a busy core, and process those connections, and then the application running on that non busy core will then generate a batch of system calls that will be enqueued on the home core's system call queue, so that both rx and tx processing of a packet happen on the same core so as to maintain connection locality. In a way this architecture is a step closer to the centralized queue architecture. So, ZygOS is said to follow an approximate centralized-FCFS model wherein multiple non busy cores can now steal connections from a single busy core's shuffle queue, and hence is able to report better tail latency numbers in case of load imbalance and service time variability when compared with IX and mTCP.

Short note on Connection Stealing

Connection stealing is great feature, as it helps a network stack to better distribute whenever a load imbalance occurs in the system where some cores in the system are asked to a very high proportion of the work. In such a case, the network stack steps in distributes that load among all the cores in the system. If network stacks use NIC hardware to distribute packets among cores, then there is bound to be load imbalances, and to solve those imbalances you need to steal some work from the heavily loaded core so that connections aren't dropped. But, its important to understand that this feature is not free. If the network stack has to support this feature then it needs to have a common memory location where the stack stores information on which CPU core is relatively busy and non busy. When a non busy core decides to partake in connection stealing, it has to obtain a lock on this data structure to figure out which busy core to steal from. This gives rise to lock contention number 1. After finding a busy core, the network stack code running on the non busy core needs to obtain a lock on the data structure where that busy core stores its local connections. This gives rise to lock contention number 2. This step can force the non busy cores to obtain multiple locks to steal from a remote core if the network stack has not organized the network stack data structures well. The lock contention gets even worse when there are multiple non busy cores trying to steal connections from a single busy core which is a highly likely scenario. Now, if the user application decides to do app processing and network stack processing on this new non busy core then the code running on this non busy core avoids obtaining any more locks, but it breaks the connection locality principle because the NIC hardware will keep sending packets corresponding to this stolen connection to the original busy core, and hence the RX processing will take place on that core. The network stack can force the NIC hardware to redirect packets corresponding to the stolen connection to this new non busy core but re configuring hardware will take some time and will hurt performance.

If the network stack tries to do tx processing on the original busy core, then you end up contending for another lock on the data structure where the original busy core stores its tx packets (ZygOS does this). Again, there might be more than one non busy core trying to get that lock. This is lock contention number 3.

The whole point of distributing packets among cores was to avoid lock contention on network data structures, but as we saw in this section that supporting connection stealing means having to again introduce that lock contention back into the system. So, there is a tradeoff here between fighting lock contention and mitigating load imbalances, and this is the reason why some of the paper just decide to not support the feature because it makes things difficult.

3.2.4 TAS

The main idea of this paper [18] is to optimize the network stack for common case TCP processing. They make an argument that although TCP has grown to be a very complex protocol with a lot of moving parts, but the common case TCP is

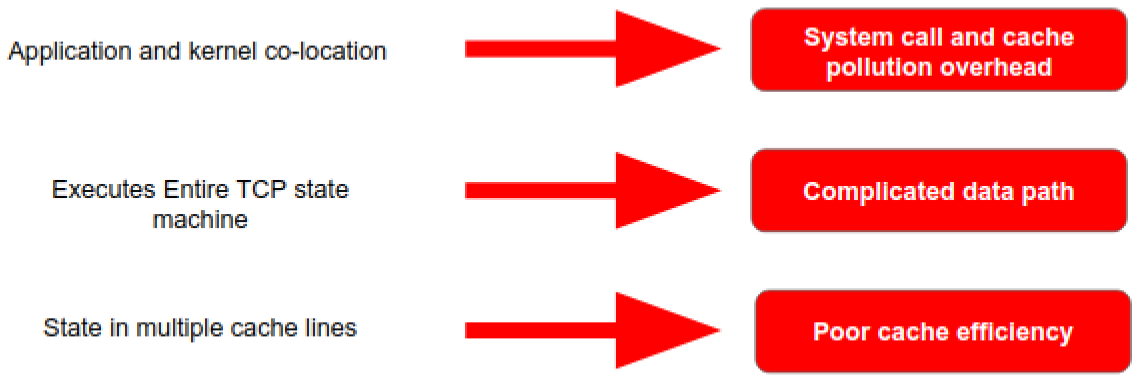

Application and kernel co-location
System call and cache pollution overhead
Executes Entire TCP state machine
Complicated data path
State in multiple cache lines
Poor cache efficiency

Figure 3.23: Issues TAS identifies

still very fast. By common case TCP, they refer to the fragments of TCP which are most likely traversed by most of the packets coming in to the system. First, let us start out by looking at the issues which they have identified in state-of-the-art network stacks as shown inn figure 3.23, and then in a later section we'll look at how they try to overcome these issues:

1. The first issue they identify is application code and the network stack code run on the same core, and this is true for all the stacks we have seen until now. They argue that this model leads to cache pollution as the application code overwrites or in other words pollutes a lot of the data that the network stack had on the cache, and when the network stack runs again it has to encounter a lot of cache misses because of this phenomena. There are system call or privilege level switch overheads in the case of stacks like IX and ZygOS, and context switch overheads in the case of mTCP as well due to colocation, as only one, either the application or the network stack can run at the same time.

2. The next issue they identify is that stacks execute the entire TCP stack machine for each and every packet, and the TCP stack being complex and huge, leads to inefficient usage of the instruction cache, and hence has a huge hit on performance.

3. Another common problem with network stacks is that the connection state maintained by them is spread throughout memory, which leads to a lot of TLB misses, also the connection state is not cache line aware which leads to a lot of false sharing and unnecessary cache line invalidations.

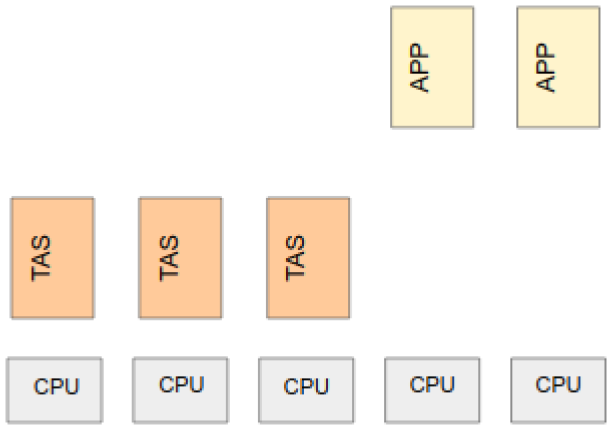


Figure 3.24: TAS Design Overview

Let's look at the design overview which they have come up with to overcome the issue which we discussed on the previous slide. The first one is pretty obvious, they run the network stack and the application on separate cores so that there is no cache pollution and privilege level or context switch overheads. This also makes it possible to scale the TCP stack independent of the application, so for example based on load we can scale the network stack up by adding in cores, or we can scale it down by taking away cores, all the while the application is still running on the same cores as it was before. The next issue they try and deal with is the bulky connection state, and we talked about how that leads to a high memory footprint and false sharing. The obvious thing to do is to minimize the connection state to the bare essentials, for example sequence and ack numbers, remote IP/port, Send rate, and congestion statistics. They are able to bring the connection state down to 100 bytes, which is roughly 2 cache lines and this connection state is cache line aware and hence there is no false sharing. Also, modern servers have L1 cache sizes in the order of a few MBs, which enables them to keep connection state of 10s of thousands of connections in the cache simultaneously. Because they can keep complete connection state of a lot of connection in cache simultaneously and the application cannot pollute the data cache of the network stack as the application runs on a separate core, they opt for aggressive prefetching. Before doing any kind of processing on a packet, the first thing they do is to prefetch the connection state for that packet, so that they never have to hit memory while processing the packet, which speeds up things a lot.

TCP Fast Path

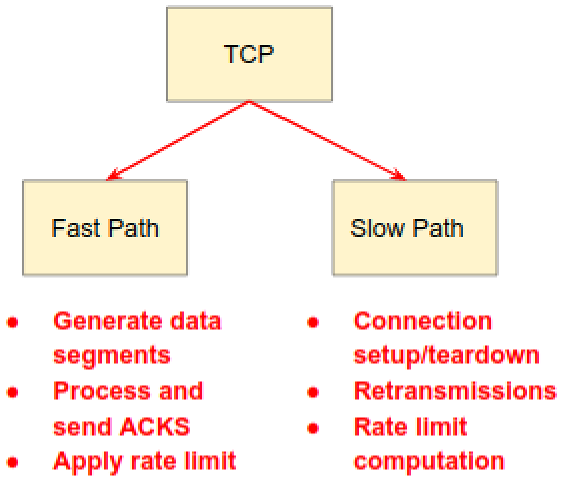


Figure 3.25: TCP Fast and Slow Paths

They identify that although TCP as a whole has become quite complex with many moving parts, but the common case data path remains relatively simple. They argue that the following claims can be made of TCP running in datacenters:

1. Packets sent within the data center are never fragmented at the IP layer.
2. Packets are almost always delivered reliably and in order.
3. Timeouts almost never fire.

They make use of this insight and split TCP processing into 2 stages as shown in figure 3.25. One they call the slow path which takes care of code paths in TCP which they identify as rare, and the slow path consists of things like connection setup/teardown, retransmissions, rate limit computations. The other they call the fast path which takes care of the things which they identify as common case TCP, which includes generating data segments to send, processing incoming packets and then generating corresponding ACKs, and enforcing the rate limit calculated by the slow path.

Design

Let's look at the RX path through their design. We have three players in the system, the slow path thread, the fast path thread, and the application thread, and all of these threads run on separate cores as shown in figure 3.26. The fast path thread

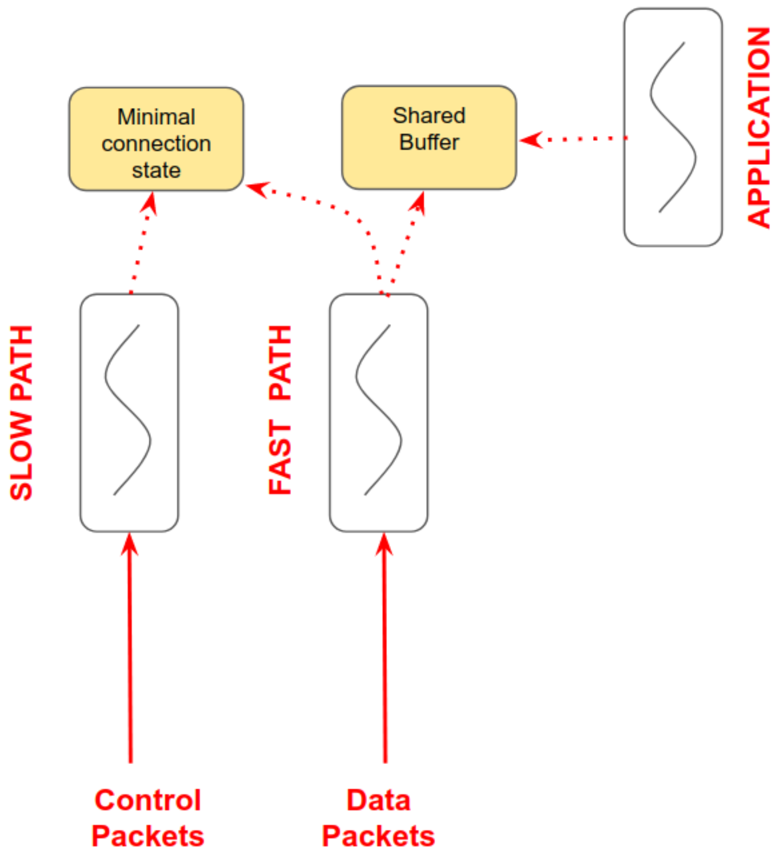


Figure 3.26: TAS design detail

as the name suggests is where the fast path TCP code path is executed and this is the part which is scalable based on the load being generated. Control packets are redirected to the slow path thread, which will takes care of the code path for establishing a connection, and after that the slow path will generate a minimal connection state which the fast path thread will also be given access to. The slow path thread will also set up buffers which the fast path thread and the application thread will have access to. All the data packets will be directed to one of the fast path threads running on the system and the fast path thread will first prefetch the connection state on to the cache of the core on which it is running, and then do TCP processing, after which it will store events and data in to the shared buffers setup by the slow path thread, and the application thread will now pick up events and data from these shared buffers and process them.

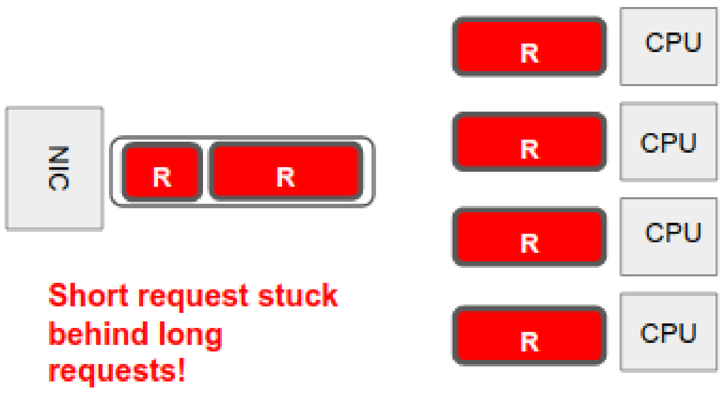


Figure 3.27: Issue with centralized FCFS model

3.2.5 Shinjuku

The main idea of this paper [15] is to ensure tail latencies at the microsecond scale by using preemptive scheduling at a similar granularity. Tail latencies are an important factor in data center performance as servicing a user request typically requires fetching data from multiple nodes, and the servicing of the request is only as fast as the response from the slowest node. ZygOS identified that a centralized FCFS model was the way to go, as in that model a request could be serviced on any available core, which led to better latency numbers, but Shinjuku points to a flaw in the centralized FCFS model as well. As you can see in figure 3.27, all cores are busy servicing a long request(request with a long service time), and a short request(request with a short request time) is stuck behind those long requests. They argue that short requests are penalized in a centralized FCFS system which again drives the tail latency numbers upwards.

They run some simulations on bimodal distribution of requests, where 99.5% of requests take 0.5us, and 0.5% of requests take 500us. They identify this to be a very common request pattern in key value servers, and they find that the 99.9 percentile latency of the shoots up pretty quickly because of the reason we mentioned just now. They argue that processor sharing must be added into the mix along with centralized-FCFS, where a long running request can be preempted which will give short running requests a chance to take control of the CPU. And, again by running simulations on the same workload, they find that processor sharing with a 1ms time slice performs similar to a non preemptive centralized FCFS model, and a 5-15us time slice is ideal where the tail latency only goes up at peak network capacity.

Design

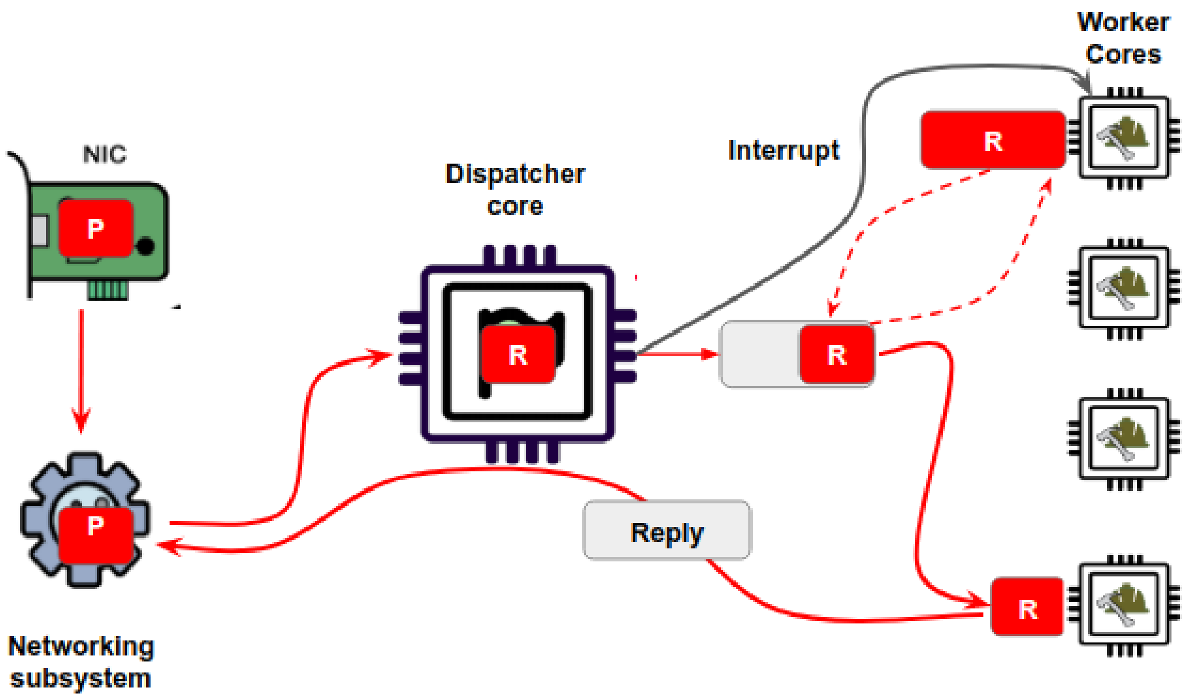


Figure 3.28: Shinjuku Design

In this section we will look at the design strategy of Shinjuku as shown in figure 3.28. Initially we start out with a NIC, as soon as a packet arrives on to the NIC, it is forwarded to the networking system, which is responsible for the network processing of this packet. The networking susbsytem could be a software module or even a hardware module and their design is oblivious to where this module is implemented. The networking subsystem will convert this network packet into an application-level request and pass it on to the central dispatcher core. The networking subsystem can be running on the same core as the central dispatcher or both of them can be running on separate cores, but the important thing to note is that Shinjuku no longer uses NIC hardware to distribute load among cores as we have seen with all network stacks until now. In the case of Shinjuku all the load from the network is sourced to a single core, and then load is distributed among cores in software as we'll see now. Along with the dispatcher core, there'll be a set of worker cores which are responsible for carrying out the actual servicing of the user request. The dispatcher core will maintain a queue of application-level requests, and as soon a worker core signals the dispatcher that its not busy, the dispatcher core will dispatch the request at the head of queue to be serviced at that worker core, and after the worker core is done, it'll send a reply back to the client via the networking subsystem. Now, if a long running request is being serviced on a worker core, the dispatcher core will send an interrupt after a time slice has elapsed and this will essentially do kind of a context switch on the worker core, where the context of the new request will be replaced by the context of the previous long running request. It's pretty evident that their whole design is built on top of user level preemption, and in the next section we will look at how they achieve that.

Fast User Level Preemption

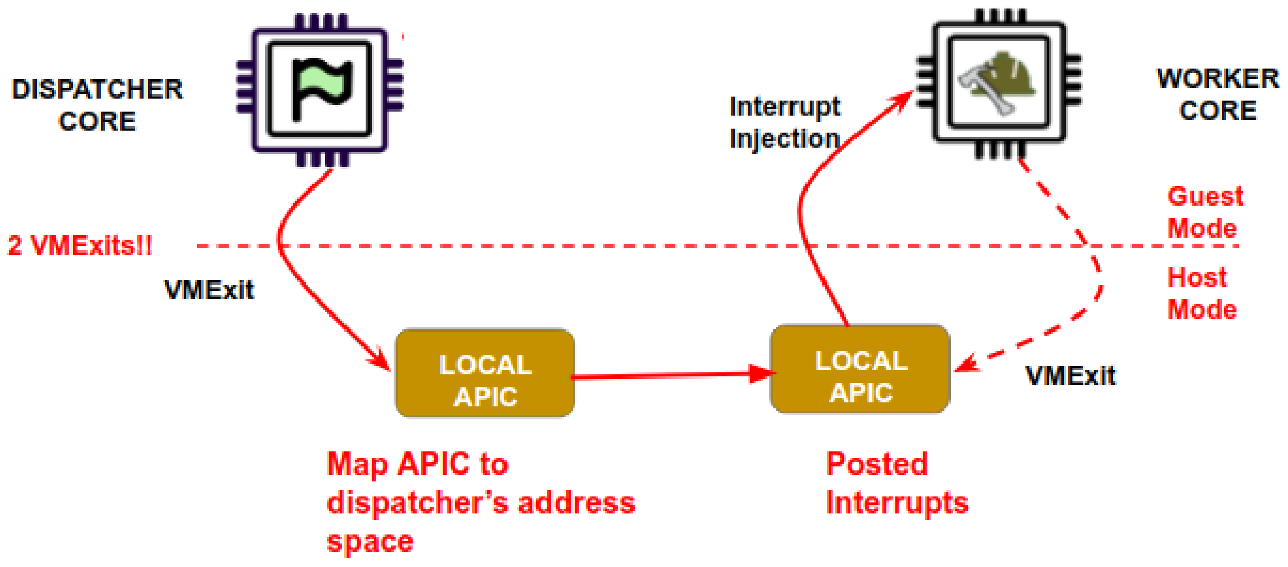


Figure 3.29: Shinjuku Preemption

Shinjuku aims to deliver preemption at the granularity level of a few microseconds, so normal preemption techniques won't work as user level signalling between processes does not operate at that scale in Linux. Shinjuku makes use of inter processor interrupts to drive preemption. As you see in figure 3.29, much like IX and ZygOS, Shinjuku makes use of virtualization hardware and both the dispatcher core and the worker core run in guest mode or VMX non-root mode. For the dispatcher core to send an inter processor interrupt to the worker core, the dispatcher code will first have to write into its local APIC (Advanced Programmable Interrupt Controller), and because this is a privileged operation, it'll result in a VMEXit to the hypervisor running in root mode, which will eventually write into the local APIC. After this an interrupt will be delivered to the worker core, which will again cause a VMExit to the hypervisor running in root mode, which will read the interrupt delivered from its local APIC, and then inject the virtual interrupt into the worker code, which is running in non-root mode. This process still results in VMExits on both the dispatcher and the worker core side. To eliminate VMExit at the worker core side they make use of posted interrupts hardware support which enables interrupts to be injected in non-root mode without any hypervisor intervention. And, at the dispatcher core side they trust that the Shinjuku dispatcher will not misbehave and map the registers of the local APIC into the address space of Shinjuku via Extended Page Table support, and with this the dispatcher code can now directly modify local APIC without any hypervisor intervention. Using these techniques for eliminating VMExits, along with fast context switching they are able to achieve preemption with time slice of the order of a few microseconds.

3.2.6 Shenango

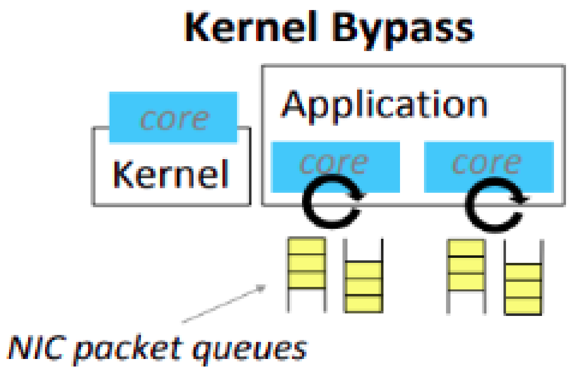


Figure 3.30: Modern Kernel Bypass Architecture

The main idea of the paper [14] is to keep CPU efficiency high while keeping tail latency in the order of microseconds for latency sensitive applications running on servers in the data centre. They define CPU efficiency as the fraction of cycles spent doing application-level work, as opposed to busy-spinning, context switching, packet processing, or other systems software overhead. If we look at modern kernel bypass architectures as shown in figure 3.30, then these employ busy spinning on the core allocated to the network stack where they continuously poll for packets on the NIC queues in a loop. And, this continuous polling lets these network stacks deliver high throughput at low latency. But modern kernel bypass network stack almost in all cases over allocate CPU resources to deal with instantaneous bursts of loads, and this is because tail latency suffers in the case where the load on the system increases suddenly and requires scaling the network stack to more number of CPU cores. Because of this over allocation of resources, CPU efficiency takes a hit, and based on some experiments they estimate that the CPU efficiency on modern data centers is somewhere in between 10 to 66 percent. On top of this you don't have a single application running on the server. They classify applications running on server into two types:

1. Latency sensitive applications which are generally user facing and load varies over time in response to user demand.

2. Batch applications where latency is not of much concern and they just aim for high throughput.

Servers ideally pack both of these types of applications on the same server so that as load varies on latency sensitive applications and dies down, those extra CPU cycles can be used by batch applications. In an ideal case, we would like to have a system which would spin up a core only for the required amount of time required to process a user request and then release the core. Such a system would achieve perfect CPU efficiency, and this is what they try to get close to. For such a system they argue that extremely fast core reallocation is necessary which would function at the granularity of a few microseconds so that an application could be granted cores quickly when necessary and this would not hurt tail latency in case of a quick load burst on the system, and similarly the system would be able to take away cores from the application quickly when the application is at low load, so that that CPU cycles can be used to perform some other batch operation which would maximize CPU efficiency.

Design

Much like Shinjuku had the central packet dispatcher which got packets from the NIC queue, and then distributed those packets among cores in software, in the case of Shenango we have a user process named IOKernel performing a similar function where it continuously polls for packets from the NIC queue as shown in figure 3.31, and then distributes packets among the cores on which the user application is running on. So, similar to Shinjuku, load distribution happens in software as opposed to in hardware. This process runs with root privileges and communicates with the kernel to get complete access to a set of cores, which are then completely controlled by the IOKernel. The IOKernel decides what and when things will run

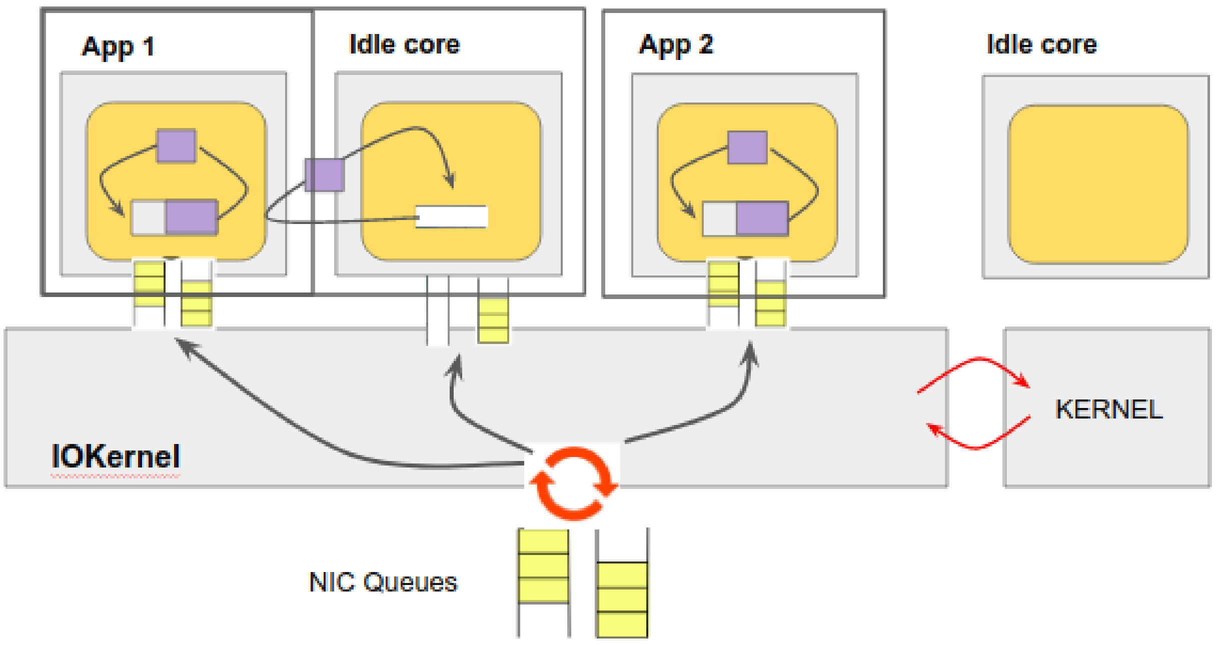


Figure 3.31: Shenango Design

on those specific sets of cores. The IOKernel starts a kernel thread on each of these cores which is never scheduled out, and then different applications will run on top of these cores in user level threads. Each of the cores have their own packet queues to which the IOKernel will direct packets to. If the IOKernel senses congestion in one of the applications, then it can allocate that application another core to run on, and will start directing some of the packets meant for that application towards the new core. Much like ZygOS, if a core in the application has no packets to tend to, then they support work stealing from other cores to balance the load out.

Fast Congestion Detection

They argue the fact that core reallocations need to be fast inorder for the CPU efficiency to be really high, so that the system does not have over allocate CPU to handle sudden bursts in load. This kind of translates into a really fast congestion detection algorithm, so they run this congestion detection algorithm at an interval of 5 microseconds, to detect congestion and then based on the result they either scale the application up or scale it down. And, naturally if the IOKernel has to do this once every 5 microseconds, the congestion detection algorithm must be simple and fast. The IOKernel has access to packet queues on all cores, and they say that if there are any packets remaining in the queue from the last run then there

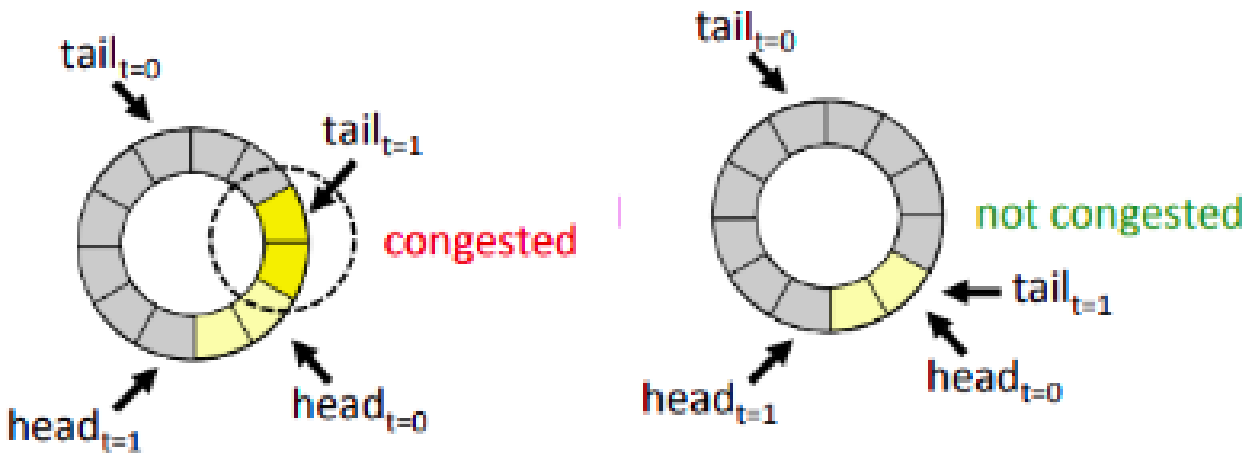


Figure 3.32: Fast Congestion Detection

is congestion in the system and we have to grant the application another core as shown in figure 3.32. The packet queues are actually ring buffers which makes this check very easy as you can in figure 3.32, at t=0 head and tail will be at some position, now if tail at t=1 is less than head at t=0, then that means that there are packets left in the ring from the last run, and hence the system is congested. Similarly on the right in figure 3.32 we see that the tail at t=1 points to the same buffer as head at t=0, which means that no packets from the previous run of the congestion detection algorithm are left in the ring buffer, and hence the system is not congested. Essentially they are able to detect congestion on a core by a simple comparison, which is way faster than the application level metrics that the previous stacks used and this allows them to perform core reallocations at the granularity of a few microseconds.

3.2.7 Deviation from Connection Locality

Let us look at connection locality a little bit more. The newer stacks like TAS, Shinjuku and Shenango seem as if they are not following connection locality. Most of the network stacks we have seen until now are designed as you can see on the left in figure 3.33 where the application and the network stack are running on a single core, and we have already gone through the reasons for having such a design. But, the new generation stacks have a design which is similar to what we see on the right in figure 3.33, where the application and the network stack run independently on separate cores. The application thread and the network stack communicate using shared buffers here as they were doing before. Although, it does appear as if these

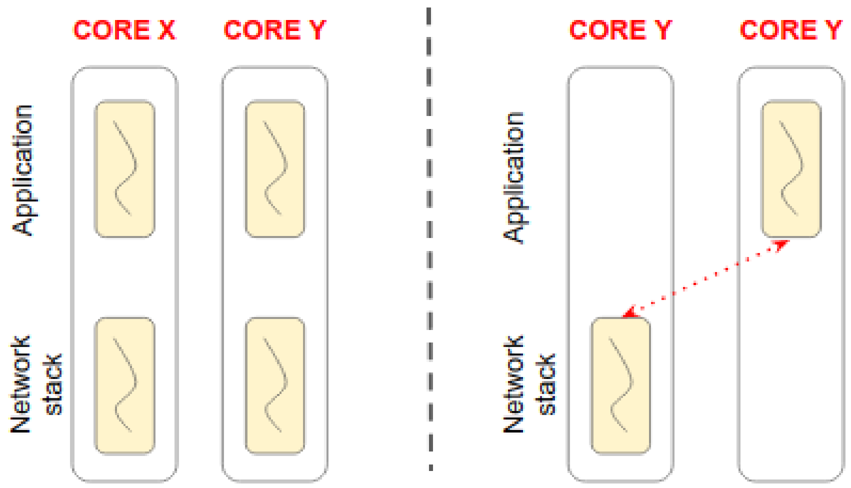


Figure 3.33: Colocation vs Connection Locality

stacks deviate from connection locality but that is not the case. The core which does RX processing of a packet is the same one which does TX processing of the packet, so that there are no cache miss issues and locking. It's just that the application is running on a separate core, but still when the application makes a request to the network stack, the request is handled consistently by the same CPU for a particular connection. So, they have moved away from colocation of application and network stack on the same core but are still maintaining connection locality. Plus it turns out that there are a lot of advantages of doing so. We know for a fact that the CPU is really fast, and looking at some of the evaluation numbers of the newer papers it look as if a network stack running on a single core without being ever scheduled out is able to handle very high packet rates. For example, Shinjuku is able to reach throughput values of close to 40Gbps by running the network stack on a single core. Although, Shinjuku achieves these numbers for UDP processing which is a lot cheaper when compared to TCP processing, but still this shows that a network stack isolated on a single core can support very high packet rates.

And with only the network stack running on the CPU, the application cannot pollute the data cache and instruction cache of the processor, which enables network stacks to hold most of the connection state in the data cache and most of the instructions of the network stack in the instruction cache. And this in turn complements the point made before well with now most of the things present on cache. This is also one of the major design principles of TAS. The network stack and application used to communicate using shared memory, and the same thing is true now as well, the only difference being that now both the application and network stack are not co-located on the same core. The application will have to definitely go through cache misses as it accesses these buffers for data, but the application also has the CPU completely to itself, so it can make up for the CPU cycles wasted on those cache misses.

3.3 Hardware Offload Solutions

In this section we will review some of the papers which improve upon the Linux networking stack by offloading some of the network stack processing on to the NIC hardware.

3.3.1 AccelTCP

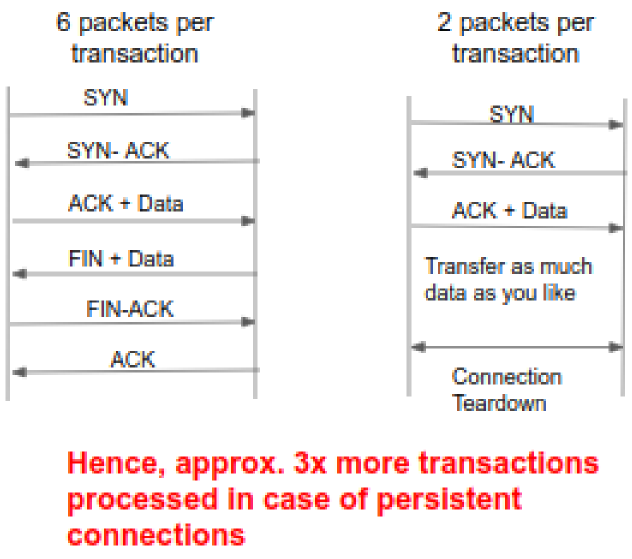

6 packets per transaction
SYN
SYN- ACK
ACK + Data
FIN + Data
FIN-ACK
ACK
2 packets per transaction
SYN
SYN- ACK
ACK + Data
Transfer as much data as you like
Connection Teardown
Hence, approx. 3x more transactions processed in case of persistent connections

Figure 3.34: Short Connections

The paper [1] look at two major cases and tries to devise solutions for these two cases. The first case we look at is short connections. As you can see in figure 3.34 short connections are usually those where after connection setup a single packet exchange takes place between the client and server and then the connection is terminated. We can define this single packet exchange between the client and server as a transaction. On the other hand, we have persistent connections where after connection setup a lot of transactions take place between client and server, and after some time the connection is terminated. In the case of short connections we have 6 packets per transaction as opposed to 2 packets per transaction in the case of persistent connections. They conducted some experiments on mTCP and found that approximately 3x more transactions were processed in persistent connections as opposed to short connections which is proportional to the number of packets processed per transaction in persistent vs short connections.

The second case is of load balancers, where the main job of the application is to map a client connection with a server. This is achieved by basically just switching some headers without tinkering with any of the packet content. They argue that most of time is spent in getting packet from client connection through the network stack to the load balancer application and then sending the packet out on server connection where it'll traverse the network stack again. There are some solutions which come to mind which are as follows:

1. We could use partial TCP offloads like segmentation offloads which reduce DMA overheads by clubbing multiple packets together. But, they argue that in the case of short connection there are not enough packets to club in the first place and hence this would not improve performance in any significant way.

2. Complete TCP stack can be offloaded on to the NIC as well. But, they argue that this solution is not scalable as the NIC has limited compute and memory resources.

3. We could use zero copy stack to help with load balancing. This is something that is netmap's selling point but they argue that there still there would be DMA overheads associated with each packet.

Design

AccelTCP divides TCP operations in two pools: central and peripheral TCP operations as shown in figure 3.1. Central TCP operations contain operations which are logically relevant to the application at hand. It includes things like congestion control because its important to the application that TCP packets are not dropped due to congestion because it would hurt the application's performance. Similarly reliable data transfer and buffer management are also important to applications performance. Then in the Peripheral TCP operations bucket we have operations which are logically independent of the application and have to be done to maintain

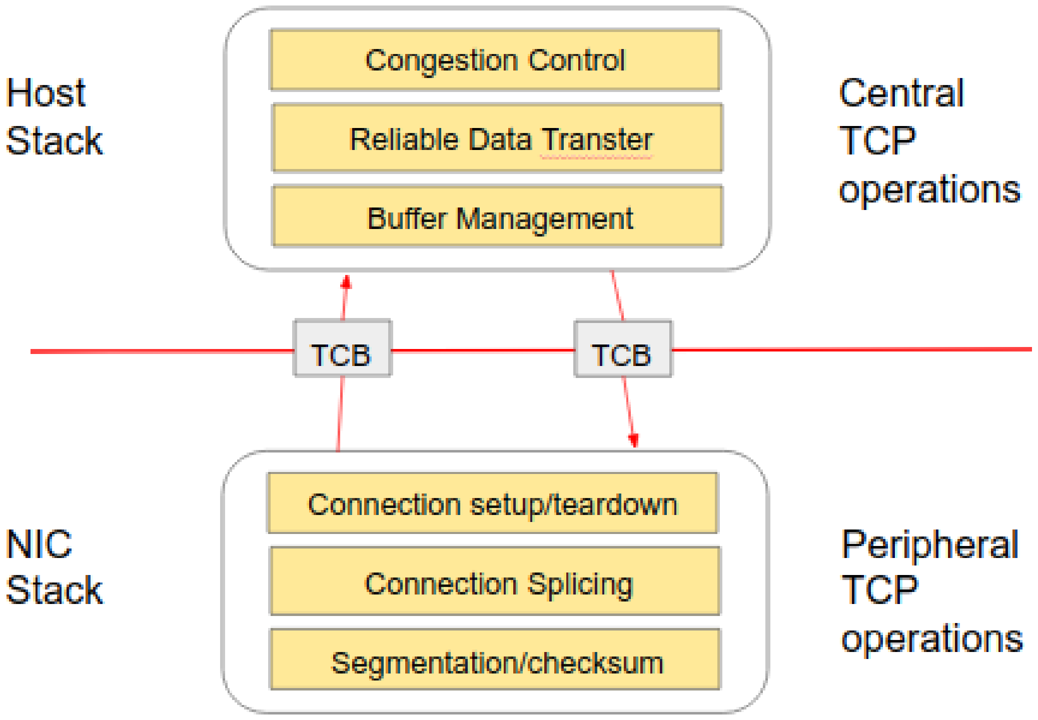


Figure 3.35: AccelTCP Design

adherence to the protocol. This includes things like connection setup/teardown, connection splicing(relaying of packets between two connections), and segmentation and checksum calculation to be performed on each packet. As they have divided the TCP operations into two groups, they have come up with a dual network stack model where one of the network stack named host stack handles central TCP operations and the other stack named NIC stack handles peripheral TCP operations. As the name suggests, the host stack runs on top of general CPU cores on the system and the NIC stack runs on the Flow Processing Cores present on a SMARTNIC. We could think of them as two different instances of network stacks running on two different pieces of hardware simultaneously. Now, a basic question would arise: how could two different network stack instances handle different operations of the same TCP connection? The basic building block of a TCP connection is a data structure called the TCB, and whichever stack has the TCB has control over the connection. For example, during connection setup the NIC stack will have the TCB, and as soon it is done, it'll hand the TCB over to the host stack. Now, the host stack will perform central TCP operations on TCB, and as soon as it is done, it'll pass the TCB to the NIC stack again for connection teardown. We say that the host stack offloaded the connection setup/teardown responsibility onto the NIC stack. There are some design guidelines that they follow:

1. Which TCP connection and what operation for that connection to offload is decided exclusively by the host stack. The host may choose to do some operations on its own even though it can be offloaded.

2. The two stacks don't share a single TCB, rather there is exclusive ownership to avoid any kind of synchronization

3. NIC has limited processing capability so complex operations are not offloaded onto the NIC.

Connection Setup Offload

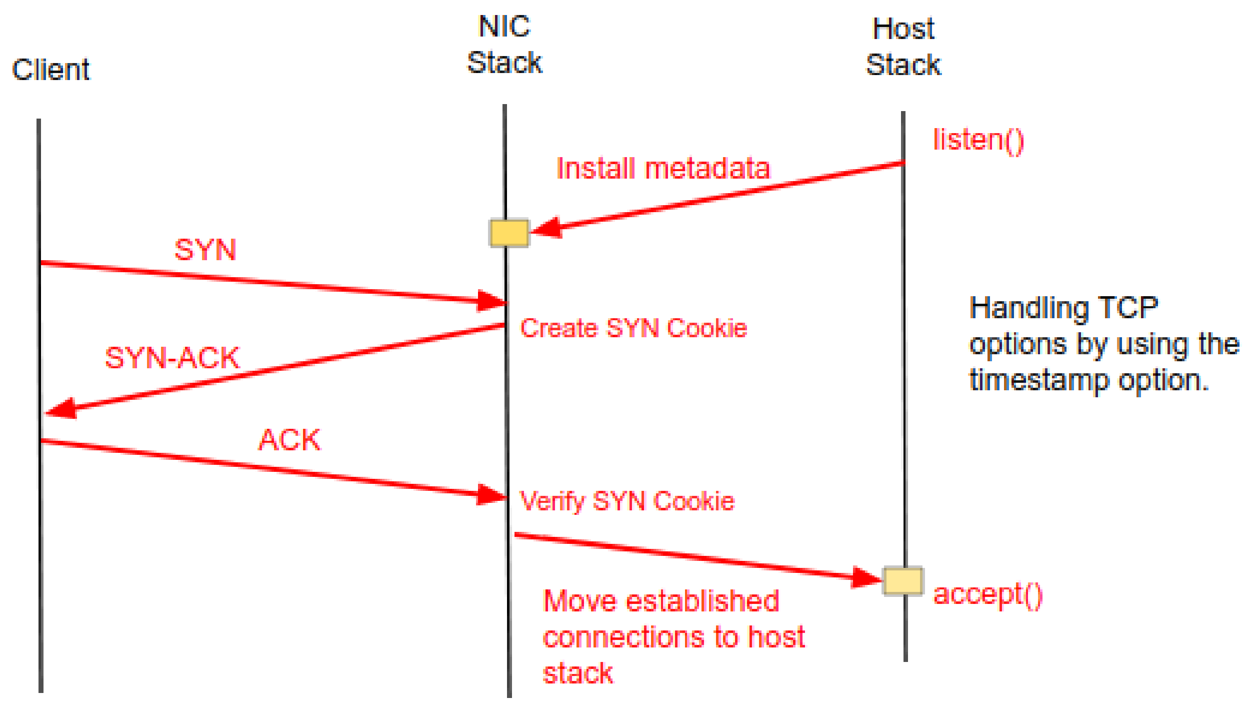


Figure 3.36: AccelTCP Connection Setup Offload

Let us look at how connection setup in AccelTCP works as shown in figure 3.36. The application running on the host stack would call listen() which would trigger offload of connection setup for that port to the NIC stack by installing some metadata like IP address and port listening for connections on the NIC stack. Then, a client sends a SYN, the NIC stack will create a SYN cookie in response to the SYN and it makes use of SYN cookies because the NIC stack would not like to maintain any intermediate state because of the limited amount of memory on NICs. The NIC stack will respond with a SYN-ACK. Next, the client will respond back with an ACK, on which the NIC stack will verify the SYN cookie, and on verification will pass the established connection to the host stack. The application at some point will call accept to get access to this established connection. The SYN packet might contain some TCP options, and to keep track of negotiated TCP options, the NIC stack sends the negotiated TCP options encoded inside the TCP timestamps option while sending the SYN-ACK packet, which the client relays back with the ACK packet, and so TCP options in SYN packet are handled in this way without maintaining any intermediate state at the NIC stack.

Connection Teardown Offload

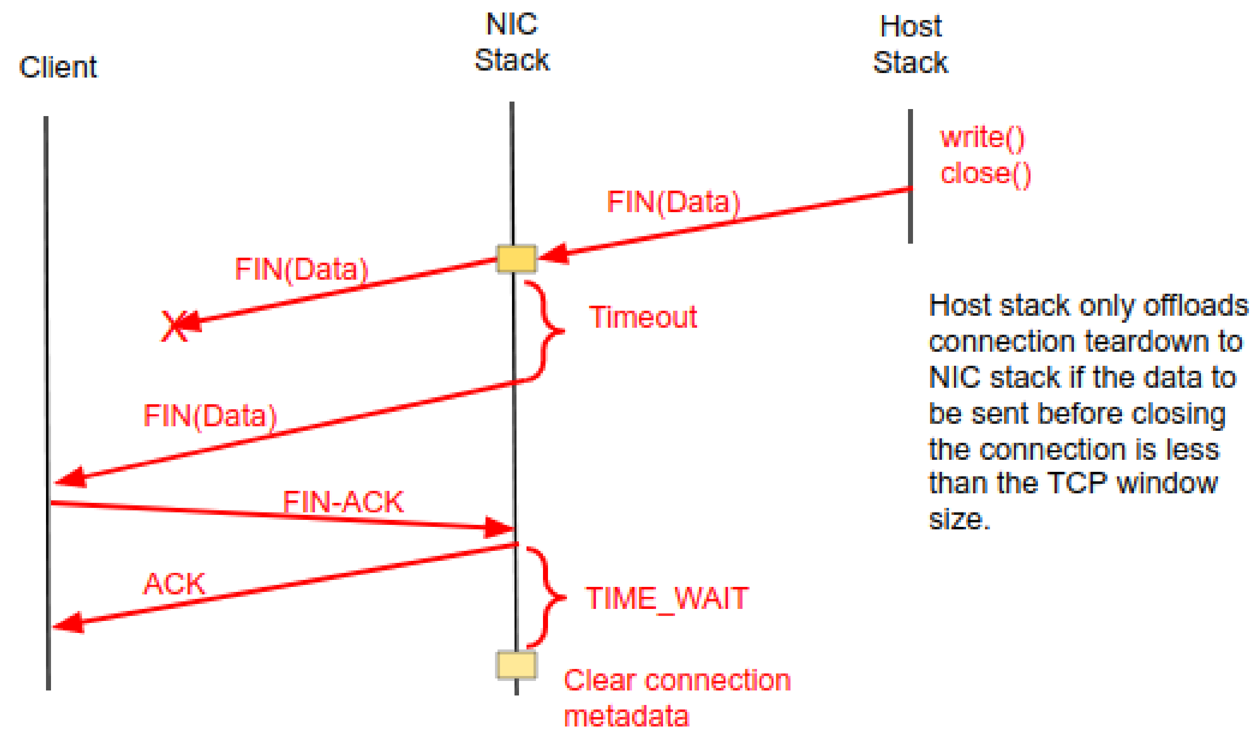

Client
NIC Stack
Host Stack
write()
close()
FIN(Data)
FIN(Data)
Timeout
FIN(Data)
FIN-ACK
ACK
TIME_WAIT
Clear connection metadata
Host stack only offloads connection teardown to NIC stack if the data to be sent before closing the connection is less than the TCP window size.

Figure 3.37: AccelTCP Connection Teardown Offload

Let us look at how connection teardown works in AccelTCP as shown in figure 3.37. An application would call write() and then call close(), so there is some data to be sent out on the connection before closing it. The host stack will only offload connection teardown if the data to be sent out is less than the window size to avoid any congestion control on the NIC stack. Let's say that the data to be sent is less than the window size, then the connection TCB is offloaded onto the NIC stack for connection teardown and the TCB is removed from the host stack. Note that they offload a very lightweight TCB onto the NIC stack, so it does not pose a burden on the limited memory available at NIC. Them, the host stack would send out a FIN packet piggybacked with the remaining data to be sent, and this packet could be lost, so there has to be some kind of retransmission mechanism available at the NIC stack which is mostly timeouts because in case of short connections there aren't enough packets to get 3 duplicate ACKs from the other side. So, on timeout it retransmits the packet, and receives FIN-ACK from the other end, and NIC stack responds with an ACK, and this would put the connection in time_wait state, after which it'll clear the metadata hence completing connection termination.

Connection Splicing Offload

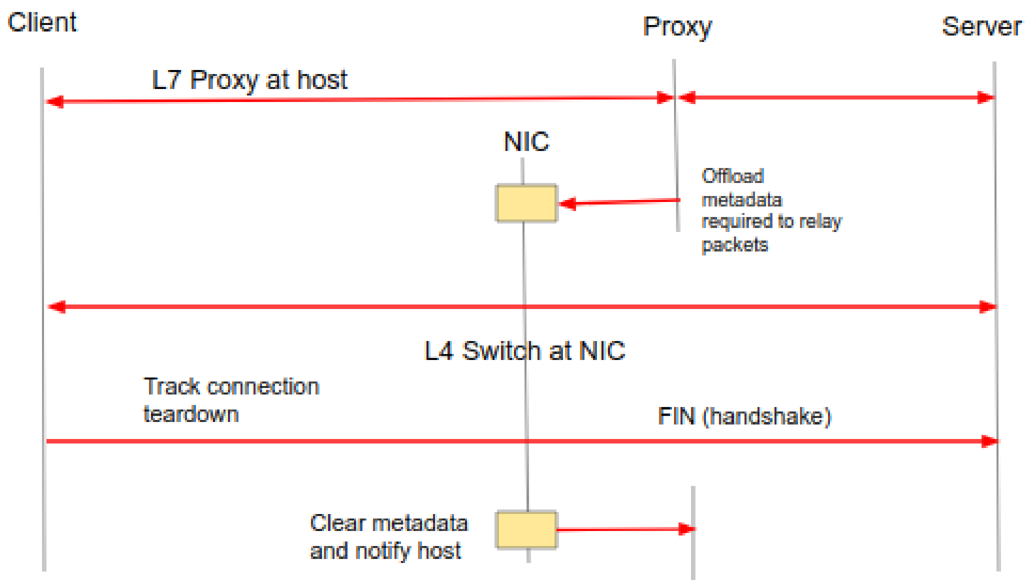

Client
Proxy
Server
L7 Proxy at host
NIC
Offload metadata required to relay packets
L4 Switch at NIC
Track connection teardown
FIN (handshake)
Clear metadata and notify host

Figure 3.38: AccelTCP Connection Splicing Offload

Let us look at how connection splicing is offloaded as shown in figure 3.38. The client would connect to a proxy running at the application layer which would find a server and then relay packets from client to the server without changing the contents of the packet. The proxy would offload the connection splicing for a specific client to server connection to the NIC stack by passing in some metadata. The metadata would include TCP/IP 4 tuple, sequence number and ACK number offset between the client to proxy and proxy to server connection, and the checksum would also change but instead of calculating it again they pass the checksum offset as well. The NIC on receipt of a packet from the client just has to switch the 4 tuple, add sequence number, ACK number and checksum offset and send the packet out. Finally on receipt of FIN, it would close both connections with client and server and after removing the metadata the NIC stack will host of the freed ports which can be used again now.

3.3.2 Tonic

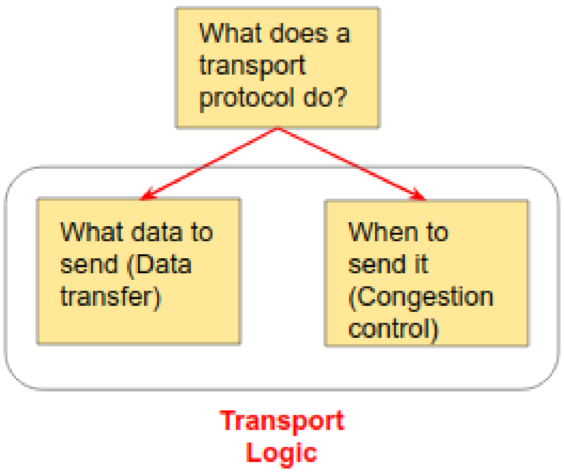

What does a transport protocol do?
What data to send (Data transfer)
When to send it (Congestion control)
Transport Logic

Figure 3.39: Transport Logic

The main idea of the paper is to provide a partly programmable hardware module which can be used to program a number of transport protocols which will be running on high speed NICs [3]. Let's begin by seeing what a transport protocol actually does which are two things basically:

1. The transport logic figures out what data to send across to the other side, and this is done by the data delivery algorithms in the transport protocol. In case of TCP, the algorithm would be to send out data starting with the byte which has sequence number equivalent to the acknowledgment number received.

2. it must also figure when to send this data out, and this is done by the congestion control algorithms of the transport protocol. So, for TCP this would be using ECN packets to control the rate of transmission or the sliding window protocol which places a limit on unacknowledged data.

There are two additional important things that transport protocols do, which is connection management and buffer management. But, Tonic relies on the host stack to perform these operations. They combine these two algorithms which we described above and call it the transport logic as shown in figure 3.39. They argue that there are common patterns in the transport logic across a number of transport protocols which they will hardcode into the NIC, such that these common pattern could be reused to build different transport protocols. So, this way a user of Tonic will get a transport protocol with very little programming running completely on NIC supporting close to 100Gbps data rates. Using some experiments they calculate that in order to support 100 Gbps data rate, their transport logic must not have a cycle period greater than 10ns. We will next look at the sequence of operations that the transport logic is expected to complete in this 10ns period.

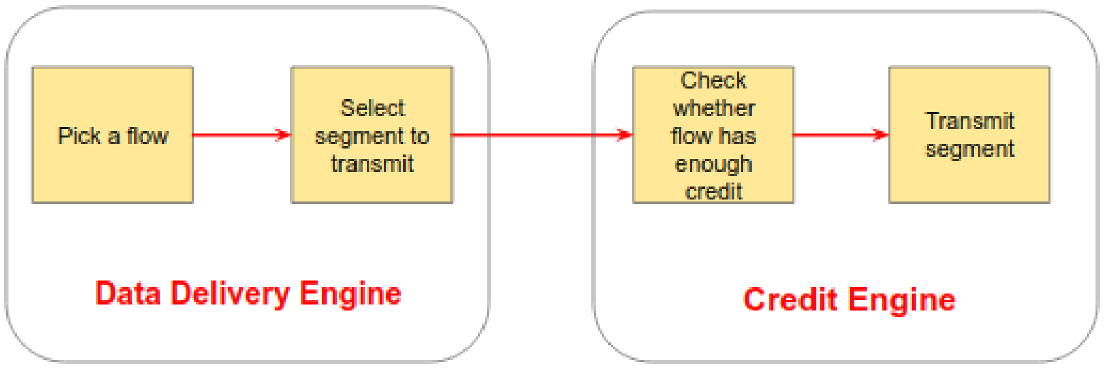

Pick a flow
Select segment to transmit
Check whether flow has enough credit
Transmit segment
Data Delivery Engine
Credit Engine

Figure 3.40: Steps involved in Transport Logic

The first thing the transport logic has to do is to pick a flow out of all the flows which have data to transmit as shown in figure 3.40. Then, you would have to select a segment from the socket buffer of that particular flow to transmit. After that, the transport logic would have to check whether the flow has enough credit to transfer data. Credit here means the number of bytes that the flow is allowed to transmit. Let's say that the flow has enough credit, then you transmit the segment. These four steps must be repeated every cycle and they argue that doing all of this in 10 nanoseconds is not feasible. So, they split these processes into two stages, namely the data delivery engine and the credit engine as shown in figure 3.40. Both of these engines run every cycle independent of each other, very similar to two stages in a pipeline. Essentially, now there is less work to be done in every cycle. Similarly, flow state is also divided and the part relevant to the data delivery engine is maintained there and the parts of the flow state relevant to the credit engine is maintained in the credit engine. Both engines obviously need to cooperate, and we'll see how that happens in the next section.

Design

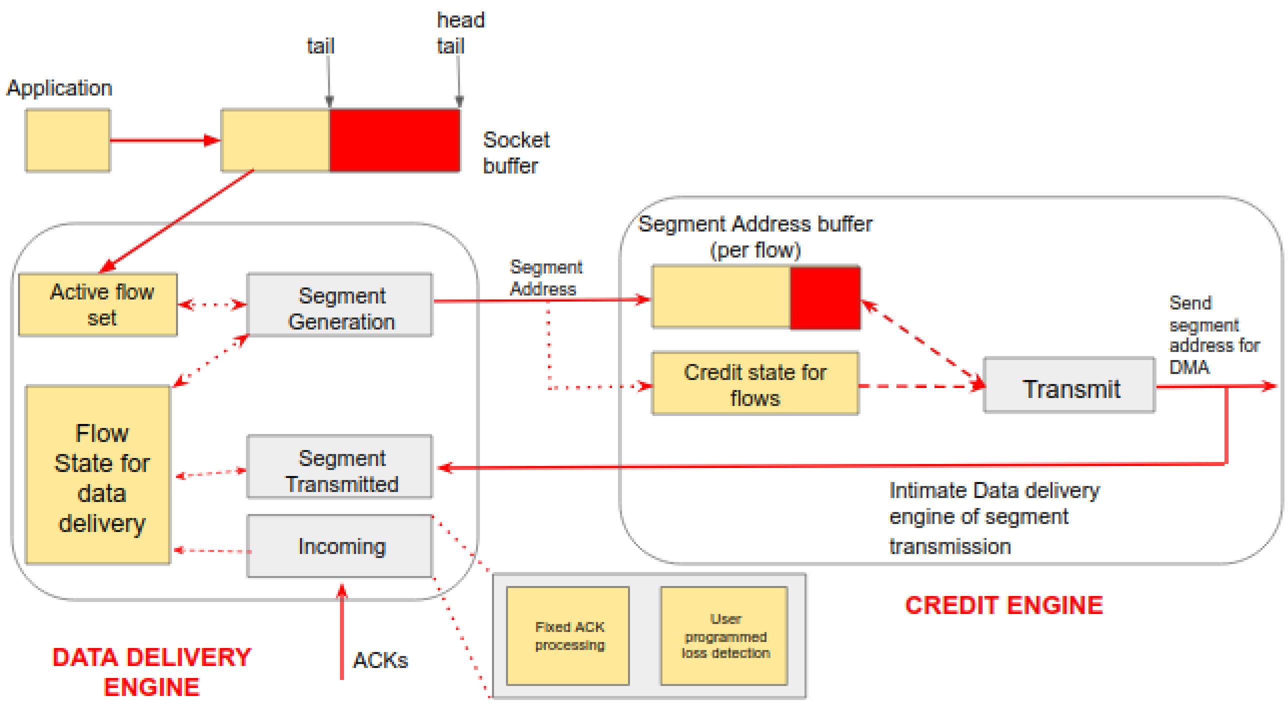


Figure 3.41: Tonic Design

Let's look at Tonic's design briefly as shown in figure 3.41. There are a lot of cases, but we'll only cover enough of them to explain the main idea behind the paper and let's start by looking at the tx flow. There are two rules which we need to keep in mind about their design before starting out:

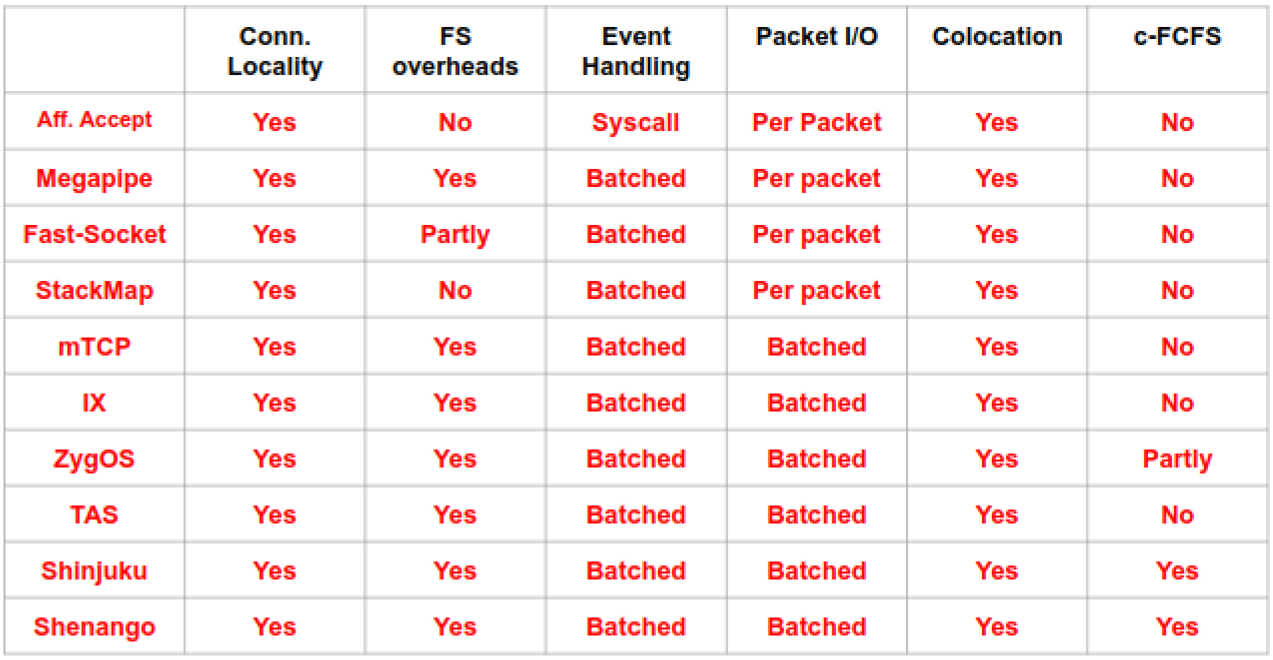
1. Tonic works with a pre configured fixed segment size.

2. At a time, a flow is only allowed to have N outstanding segments ready for transmission.

We start with the user application and the socket buffer. The application will write some data to the socket buffer hence updating the tail of the buffer, and this would place the flow in an active flow set maintained by the data delivery engine. The segment generation module of the data delivery engine will pick a flow from this set, and then will access the state of this flow to check whether it already has N outstanding segments or not, and let's say it does not, then the module increments the outstanding segment count and generates a segment address. A segment address because all segments are of equal size so a starting segment address is enough to identify a segment. After generating a segment address, this module passes off this segment address to be stored in a segment address buffer in the credit engine and all segments stored in this segment buffer would be counted as outstanding segments for this flow. Additionally the credit engine maintains current credit information for each flow which has segments to transmit, or in other words have segment addresses in their segment address buffer. Now in the credit engine the transmit module will pick a flow which has enough credit to send one segment out, after finalizing the flow it decrements the flow's credit and then grabs a segment address from that flow's segment address buffer and sends it out for DMA. It also signals the data delivery engine via the segment transmitted module to decrement outstanding segment count. The thing to look out for here is that the whole segment transmission process is fixed and nothing is programmable as they argue that this process is common in all transport protocols and there is no need for programmability. Now, if we look at the incoming module of the data delivery engine it is responsible for dealing with incoming packets and some of them will be acknowledgments. They say that handling of acknowledgments is also pretty similar across these protocols so on ack receival, it will update the segments which were acknowledged, but the thing which they notice to be different is packet loss detection. For example, in TCP if you receive 3 duplicate ACKs, that would qualify as packet loss. They allow for a user programmed sub module to run in the incoming module which would access variables of the flow state and detect loss and if loss is detected then act accordingly. So, the incoming module functions as a two stage pipeline, first a fixed function which updates some flow state, and second a user programmed function which detects segment loss based on the flow state updated in the previous cycle. This is how they intend their system to be used for building transport protocols running on hardware with the help of small and specific programmable modules.

3.4 Summary

In this section we summarise the state-of-the-art network stacks which we have surveyed in this chapter in terms of the features that each network stack support, and is shown in the table in figure 3.42.



	Conn. Locality	FS overheads	Event Handling	Packet I/O	Colocation	c-FCFS
Aff. Accept	Yes	No	Syscall	Per Packet	Yes	No
Megapipe	Yes	Yes	Batched	Per packet	Yes	No
Fast-Socket	Yes	Partly	Batched	Per packet	Yes	No
StackMap	Yes	No	Batched	Per packet	Yes	No
mTCP	Yes	Yes	Batched	Batched	Yes	No
IX	Yes	Yes	Batched	Batched	Yes	No
ZygOS	Yes	Yes	Batched	Batched	Yes	Partly
TAS	Yes	Yes	Batched	Batched	Yes	No
Shinjuku	Yes	Yes	Batched	Batched	Yes	Yes
Shenango	Yes	Yes	Batched	Batched	Yes	Yes

Figure 3.42: Summary of state-of-the-art network stacks

We have intentionally left out AccelTCP and Tonic from the table in figure 3.42 because of them make use of hardware offloads and are agnostic of the type of network stack running on top of them in software. Though it's important to note that both AccelTCP and Tonic are complementary. AccelTCP offloads connection setup and teardown on to the NIC hardware and does TCP processing in software, whereas Tonic offloads TCP processing on to the NIC hardware and does connection setup and teardown in software.

Chapter 4

Proposed PhD work

In the last chapter we looked at state-of-the-art in terms of network stack research, and we saw how different network stacks employed different techniques to achieve varying objectives. Although the common theme was not to compromise on the throughput and latency which the application running on top of these stacks was able to deliver. In this chapter we will lay out some of the work which we plan to carry out in the future.

It's important to note here that almost all of the network stacks surveyed in the last chapter focus on I/O intensive workloads, where the user level application does not take many CPU cycles in processing a single request. So, the focus of current research has been to maximise the output of I/O intensive applications by making the network stack efficient enough such that it is able to provide the application running on top of it with a high number of requests by using as less CPU cycles as possible. **We would like to explore the impact that these state-of-the art network stacks have on CPU intensive workloads**, where the user applications consume a lot of CPU cycles in processing a single request.

The 5G packet core is of particular interest to us. On some exploration we identify that some of the network functions in the 5G packet core are CPU intensive applications. Some example of such type of network functions would be the Access Mobility Function (AMF) and Authentication Server Function (AUSF). So, in our task of figuring out feasibility of state-of-the-art network functions surveyed in the last chapter on CPU intensive application, we would like to work with these network functions in the 5G packet core.

We also believe that network stacks could be customised to deliver optimal performance for a specific application.Along these lines we would also like to figure out some features which, when included in the network stack would elevate the network function's performance running on top of it. We think that each of the network functions which we end up working with will have such features.

Although our current future work only involves two network functions running on baremetal hardware, we would like to extend this to cover various kinds of network functions running in varied environments after we gain a better understanding based on the results of our initial exploits. **Our end goal is to come up with network stack designs which work well and facilitate peak performance of 5G Network functions, running in varied environments, for example, on baremetal hardware, inside VMs, or inside containers.** And, this goal covers both line of investigations which we would like to pursue as described earlier.

Bibliography

[1] *AccelTCP: Accelerating Network Applications with Stateful TCP Offloading.* `https://www.usenix.org/conference/nsdi20/presentation/moon`.

[2] *DPDK.* `https://www.dpdk.org/`.

[3] *Enabling Programmable Transport Protocols in High-Speed NICs.* `https://www.usenix.org/conference/nsdi20/presentation/arashloo`.

[4] *FlexSC: flexible system call scheduling with exception-less system calls.* `https://dl.acm.org/doi/10.5555/1924943.1924946`.

[5] *I'm Not Dead Yet!: The Role of the Operating System in a Kernel-Bypass Era.* `https://dl.acm.org/doi/10.1145/3317550.3321422`.

[6] *Improving network connection locality on multicore systems.* `https://pdos.csail.mit.edu/papers/affinity-accept:eurosys12.pdf`.

[7] *IX: A Protected Dataplane Operating System for High Throughput and Low Latency.* `https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-belay.pdf`.

[8] *Linux Network Stack.* `https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data`.

[9] *MegaPipe : A New Programming Interface for Scalable Network I/O.* `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han`.

[10] *mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.* `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong`.

[11] *NetMap: A novel framework for fast packet I/O.* `https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo`.

[12] *Protocol Stack.* `https://en.wikipedia.org/wiki/Protocol_stack`.

[13] *Scalable kernel TCP design and implementation for short-lived connections.* https://dl.acm.org/doi/10.1145/2980024.2872391.

[14] *Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.* https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[15] *Shinjuku: Preemptive Scheduling for microsecond-scale Tail Latency.* https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[16] *SO_REUSEPORT.* https://tech.flipkart.com/linux-tcp-so-reuseport-usage-and-implementation-6bfbf642885a.

[17] *StackMap: Low-latency networking with the OS stack and dedicated NICs.* https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata.

[18] *TAS: TCP acceleration as an OS service.* https://dl.acm.org/doi/10.1145/3302424.3303985.

[19] *ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks.* https://dl.acm.org/doi/10.1145/3132747.3132780.