TCP and ULP Offload via AF_XDP Sockets

Tom Herbert

SiPanda
USA
tom@sipanda.io

Abstract

This paper presents a new approach to the old problem of TCP Offload. TCP Offload Engines (TOE) was a hot topic in the early 2000's. The premise was that TCP protocol processing could be offloaded to specialized hardware for a significant performance boost. While several solutions were developed and marketed, it quickly became clear that TOE had serious drawbacks. Except for a few niche use cases, TOE never gained much traction. While these early attempts at TCP Offload failed, the desire to speed up TCP processing never went away. But more than that, there's a new motivation for TCP offload: Cloud providers want to offload infrastructure compute from their server CPUs to free up cycles for running application code. We propose a modern take on TCP Offload that offloads infrastructure compute, including TCP and Upper Layer Protocol processing, from server CPUs. This design facilitates the use of hardware acceleration to speed up TCP processing, but avoids the pitfalls of TOE.

Keywords

TCP; TCP offload; XDP; eBPF; AF_XDP; sockets; ULP; ULP offload; SmartNIC; accelerators

Introduction

This paper presents a new design for TCP and Upper Layer Protocol (ULP) Offload.

There are two primary benefits of TCP and ULP Offload:

- Infrastructure compute can be offloaded from a server CPU to an offload device. This frees up cycles in server CPUs to run more application code, which in turn allows cloud providers to sell more CPU cycles to their customers
- Offloading to a specialized devices facilitates the use of accelerators to speed up processing which can result in lower latency, higher throughput, and lower power consumption

These two goals are not necessarily coupled. For instance, a simple approach to TCP Offload might be to offload connection processing from a server CPU to an Application CPU in a SmartNIC. This addresses the goal of offloading CPU cycles from the server, but without further acceleration, the net effect on performance could be actually *worse* than not offloading. In some use cases, saving server CPU cycles may be of value even at the cost of a small performance hit. Nevertheless, due to the Application CPUs proximity to hardware, there are plenty of opportunities to close that gap and ultimately improve performance with TCP Offload.

The basic idea of this design is to offload TCP protocol processing and the TCP state machine from a host CPU to an Application CPU (App CPU) in a SmartNIC. ULP processing, such as TLS encryption/decryption, may be offloaded as well. The TCP offload is orchestrated by the *XDP-Transport* interface. This interface uses AF_XDP sockets [1] with shared hardware queues to facilitate generic and extensible message based communications between applications and devices. The use of AF_XDP sockets for messaging is a novel approach that has applications outside of TCP Offload. For TCP Offload, this model is advantageous since no kernel changes are required, and we can leverage eBPF and XDP to optimize datapath operations.

The Application CPU in a SmartNIC is the target of TCP Offload. This can be a plain commodity CPU running Linux where TCP processing is done in the Linux stack, thereby avoiding the challenges of TOE in running the TCP state machine in hardware. A user space proxy in the App CPU, called the *offload proxy*, implements the connection offload. On one side of the proxy, XDP-Transport messages are received on AF_XDP sockets that map to peer AF_XDP sockets in the server; on the other side of the proxy are the sockets for the TCP connection. The server CPU sends XDP-Transport messages to establish TCP connections and send data, and the offload proxy converts messages received on the AF_XDP socket into socket functions calls. The offload proxy receives data on TCP sockets and sends XDP-Transport messages with the received data to the server.

The offloaded protocols in this design are not limited to TCP, but in fact other transport protocols could similarly be offloaded such as QUIC or DCCP. The design also allows various Upper Layer Protocols (ULPs) to be offloaded as well, including TLS, gRPC, RDMA/TCP, and HTTP2.0. When a ULP is offloaded, the application sends and receives Protocol Data Units (PDUs) for the ULP. This eliminates the need for the application to perform low level protocol processing, and instead the application can focus on processing the payload of application protocols.

This design facilitates optimizations in both the host to device interfaces and protocol accelerations. The XDP-Transport interface allows zero copy send and receive, fine grained PDU steering, and various forms of connection multiplexing. The Application CPU can use all the typical offloads available in the networking stack. Due to the Application CPUs proximity to hardware and the constrained environment of a SmartNIC, there are opportunities to seamlessly use advanced accelerations and offloads like TLS offload, packet header data split, CAM lookup, direct data placement, and domain specific CPU instructions.

The Unfortunate History of TCP Offload

TCP Offload Engines, or TOEs, have been a goal of networking vendors dating back to the late 1990's. There have been several vendor efforts, including startups such as Silverback Systems, that developed NICs with TCP offload. Microsoft's Chimney supported TCP offload in Windows starting in 2008.

As illustrated in Figure 1, the idea of TOE was straightforward. IP and TCP protocols are processed in hardware instead of the operating system which is the traditional approach. The hardware presents an interface for connection management and for sending and receiving data. The hardware interface can either be exposed to the operating system via a device driver or directly to an application library with kernel bypass. The premise of TOE was that hardware can process TCP much faster than software resulting in lower CPU utilization, lower latency, and higher throughput.

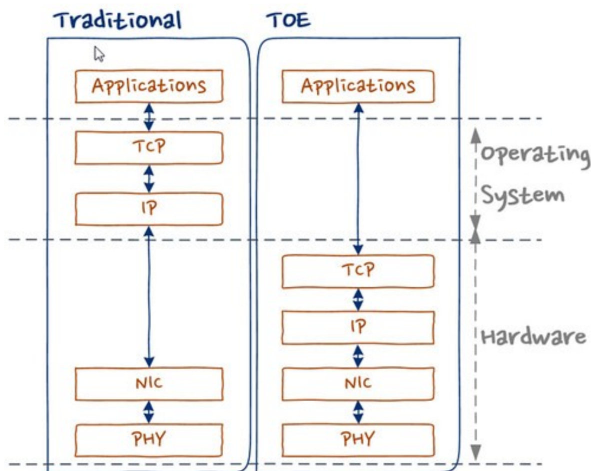

Figure 1. Processing in traditional networking stacks versus that in TCP Offload Engines (TOE)

While the development of TCP Offload Engines established the performance benefits of TCP offload, TOE is generally considered an abysmal failure [2]. In fact, Microsoft eventually deprecated Chimney in 2018 [3].

The reasons for TOE's demise can be summarized:

1. A TCP implementation in hardware is difficult to fix which becomes especially problematic when security issues with the TCP stack are found
2. Under edge conditions, a hardware implementation of TCP may actually yield worse performance than a TCP stack running in the host OS
3. The cost for all TCP connection offloading is fixed; there's no way for the operating system to optimize specific use cases
4. The NIC code wasn't written with TCP in mind; thus, not all TCP features are implemented
5. Transparent TCP offload required seamless integration into the host OS
6. Limited visibility into a TCP stack in hardware makes debugging and diagnostics difficult

Motivation and Requirements for a new solution

In our design, the problems of TOE are primarily addressed by offloading TCP to a CPU with a software networking stack instead of offloading to fixed function hardware. That point addresses problem #1 since if the offload device is running TCP in software then it's easy to fix problems or security issues. For #2, as we mentioned using an offload for performance gain isn't necessarily the only priority anymore-- even without performance gain, TCP Offload to another CPU would save host CPU cycles. Still, this design encourages the use of hardware accelerations to improve performance. #3 is no longer applicable, since the offload is being processed in a CPU, there are opportunities for programmable optimizations. Similarly, for problem #4, the network stack in the App CPU can use all the TCP features and accelerations available in software or hardware. For #5, the use of AF_XDP sockets and a user-space proxy in the App CPU avoids the need to change either the kernel in the server or the kernel in the App CPU. #6 is addressed by offloading to a CPU running a TCP stack in software, standard tools and techniques can be used for debugging and diagnostics.

This requirements for a generic TCP Offload solution are:

- Reduce infrastructure and network processing overhead in the server and hence reduce server CPU cycles spent on infrastructure processing
- Optimize performance of offloaded TCP connections by employing hardware accelerations and offloads available to the offload stack
- Offload Upper Layer Protocols over TCP. Parse the TCP stream and present ULP messages on application sockets. Note that nearly all TCP communications are message oriented (e.g. TLS, NVMe, RDMA, RPC, etc.)
- No changes to the server OS. We only require that server OS is a reasonably modern kernel version that supports AF_XDP sockets. This requirement is motivated by the fact that kernels run in the data center are typically two years behind
- No fancy hardware required for basic TCP Offload. The minimal requirements are for a device with an Application CPU and network interfaces, and lossless shared hardware queues between server CPUs and App CPUs
- Minimize application changes. Application interfaces for TCP offload can be provided by a library, and changes to the application should be straightforward. A zero application change solution might be possible using LD_PRELOAD to override system calls as is done in OpenOnload
- Assume the kernel running in the Application CPU may be changed if necessary. Presumably, the kernel running in the App CPU is provided by the vendor and can be upgraded for new features
- Support well known optimizations including polling, zero copy send and receive data operations, header/data split, and connection multiplexing similar to KCM.
- For higher levels of performance, advanced hardware accelerations in the device can be employed

Design Overview

This section presents the overview of TCP and ULP Offload via AF_XDP Sockets.

TCP offload is accomplished by offloading TCP processing from one CPU, the "Server CPU", to another CPU, the "Application CPU" or "App CPU". The server CPU resides in the host server, and an App CPU resides in a SmartNIC or other "smart device". App CPUs run in a constrained environment and are intended for low level datapath processing, and not for running general server applications as CPUs do. Communications between the Server CPU and the App CPU happens over shared hardware queues as shown in Figure 2.

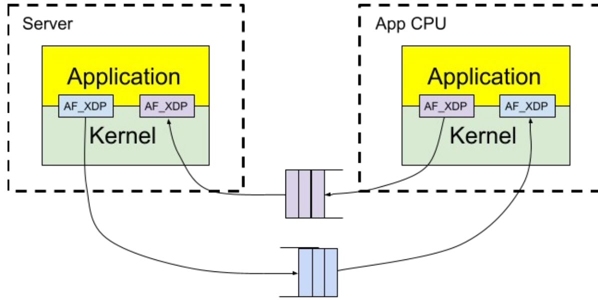

Figure 2. Communications between a server and an Application CPU via AF_XDP sockets and shared hardware queues

AF_XDP sockets are used to present a queue interface to applications running in the Server CPU and the App CPU. In the Server CPU, the application is a user application that interacts with the offload device using the *XDP-Transport library* (the *xdp_xport* library). The XDP-Transport library provides user space sockets to create offloaded connections and to send and receive data. In the App CPU, the application is a type of TCP proxy, called the *offload proxy*. The offload proxy maps messages received on AF_XDP sockets to socket calls in the App CPUs networking stack.

TCP offload

Figure 3 illustrates the components and their communications for TCP and ULP Offload via AF_XDP Sockets.

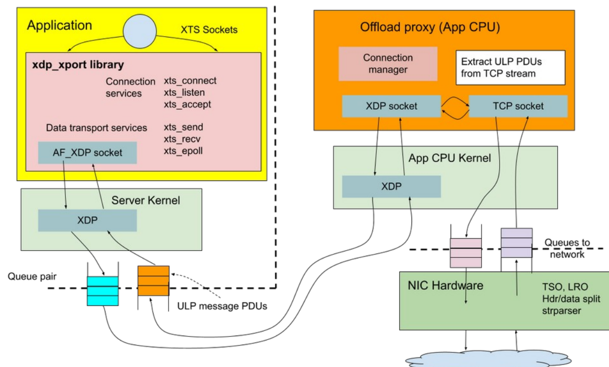

Figure 3. Component diagram of TCP Offload via AF_XDP Sockets. The application runs in the server CPU in the left of the diagram, the offload proxy runs in the App CPU at the right of the diagram. An application uses the XTS sockets interface to offload connections. XTS socket calls are mapped to messages sent to the offload proxy over AF_XDP sockets and shared hardware queues. The offload proxy maps these message to socket calls in the local networking stack

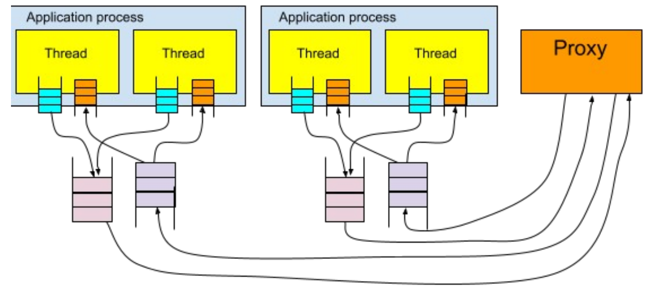

Figure 4. Server applications to offload proxy communications with multiplexing over shared hardware queues. Two application processes are depicted. Each process has two threads. Each thread has its own AF_XDP socket that is multiplexed over a hardware queue dedicated to the application processes

The user application, shown on the left in Figure 3, links with the *xdp_xport* library. Library functions provide proxy management and data operations over AF_XDP sockets. The AF_XDP sockets are bound to TX/RX device queues. Several AF_XDP sockets may be multiplexed over hardware queues as shown in Figure 4. Multiplexing AF_XDP sockets over hardware queues presents a trade off between the number of dedicated queues for threads and the synchronization requirements when multiple CPUs access the same hardware queues.

The messages sent on the AF_XDP sockets are called *XDP-Transport* messages and are encapsulated in Ethernet frames with an experimental EtherType. Messages may be variable length and are preceded with a common XDP-Transport header that indicates a message type. The message type determines the format of the message body following the header.

XDP-Transport sockets

The *xdp_xport* library provides various interfaces and functions to implement TCP offload. The XDP-Transport library provides user space sockets, called *XDP-Transport Sockets* (*XTS sockets*), as the application interface to TCP Offload. XTS sockets are based on BSD sockets and have similar semantics. For instance, *xts_sendmsg* is an analogue of *sendmsg*, and *xts_socket* is an analogue of *socket*. An XDP-Transport socket number refers to a user space XTS socket, it takes the place of the file descriptor argument of analogous BSD socket calls.

Offload proxy

The offload proxy runs in user space in the App CPU. Its function is to receive offload connection requests from the server, to establish and maintain TCP connections, and to proxy send and receive data. To send data on an offloaded connection, the server sends a "send data" message to the offload proxy that includes the data to send; the proxy writes the data on the TCP socket. When data is received on a TCP socket, the offload proxy sends it to the server in a "receive data" message. At initialization, the offload proxy creates AF_XDP sockets that are mapped to shared hardware queues. The XDP-Transport library creates peer AF_XDP sockets for sending commands and receiving replies. Commands are sent by the application via the XDP-Transport library and include requests to create connections, start listeners, close connections, and send data. Replies are sent by the offload proxy and include connection succeeded reports, new incoming connections, and received data.

Identifying connections and listeners

The XDP-Transport library and offload proxy keep identifiers to connections and listeners. In the XDP-Transport library, XTS socket numbers are the identifiers of offloaded connections and listeners. The offload proxy has its own numerical identifiers for offloaded connections and listeners that don't necessarily correspond to XTS socket numbers. When a message is sent, it includes the identifier of the connection or listener relative to the *target* of the message. The first message regarding a connection or listener sent by either side includes the local endpoint's identifier for the connection. When an endpoint receives the first message, it records the peer's identifier in its control block for the connection. The identifier is included in subsequent messages sent to the peer, and the receiver uses the identifier to find the control block for connection referred to in the messages.

Message Communications

A message based protocol, called *XDP-Transport Messages*, is used for communications between the XDP-Transport library running in the server and the offload proxy running in an App CPU. This section describes the message format and characteristics of communications.

Example processing flow

An example message flow for creating a connection is:

1. The XDP-Transport library in the server sends a "connect request" message. The message contains an XTS socket and parameters for the connection including IP address and port number
2. The offload proxy receives the "connect request" message, creates a local context, saves the received peer identifier in the context, and initiates a connection
3. When the connection is established, the offload proxy sends a "connection ack" reply message to the server. The message includes both the identifier received from the peer and the local identifier for the connection
4. When the server receives the "connection ack", it locates the control block from the XTS socket number in the message. It saves the identifier of the peer in the XTS socket control block context for the connection
5. At this point the connection is fully established and each side knows the identifier used by its peer. Any further messages sent by either side contain the peer's identifier for the connection. When either side receives a message, the identifier in the message is used to lookup the local context for the connection

Message Formats

XDP-Transport messages are encapsulated in Ethernet Frames. The general message format is illustrated in Figure 5.

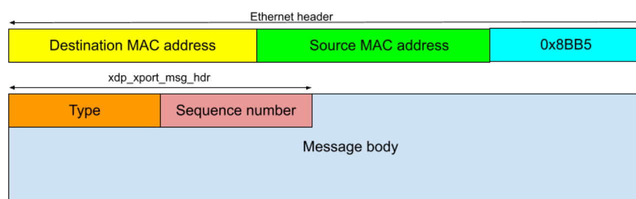

Figure 5. Format of an XDP-Transport message encapsulated in an Ethernet frame

The Destination MAC address and the Source MAC address should be set to a local MAC address of the NIC. A SmartNIC should internally loopback packets addressed to itself without sending the packet on the wire. This behavior could be implemented in a virtual switch in the NIC.

The server and Application CPU communicate using messages sent over the AF_XDP sockets and shared hardware queues. These messages are framed in Ethernet packets using an experimental Ethernet protocol (0x855B) to prevent misinterpretation.

Ethernet encapsulation of messages has some salient properties:

- The message interface between the Server and App CPU is generic and extensible. Messages could potentially have other use cases than just TCP or ULP offload
- Messages are properly formatted Ethernet frames. The messages are processed internal in a system, however it is conceivable that an Ethernet protocol could be developed to allow offloading to devices across an Ethernet network

The payload of the Ethernet Frame contains the XDP-Transport message. Each message is preceded by a message header:

```
struct xts_msg_hdr {
    __u8 type;
    __u8 seqno;
    __u8 data[0];
}
```

type indicates the message type. s*eqno* is used to enforce reliable and in order communications (the use of this field is described below). *data* contains the message body and is variable length where the format and length are determined by the message type.

Structure definitions for various messages can be defined and contain a *xts_msg_hdr* structure as the first element. For instance, "struct xts_msg_connect" might be defined which includes the various parameters for a "create connection" message. The structure could be cast as a *xts_msg_hdr* structure.

Reliable communications

Communication between a server and an offload proxy must be reliable, messages cannot be dropped and must be delivered in order. This means the shared hardware queue must be lossless and support in-order delivery. To verify correct operation, each message has a sequence number in the message header. Sequence numbers are maintained by each communicating peer as a simple counter for each AF_XDP socket. When a message is sent on an AF_XDP socket, the sender sets the current counter value as the sequence number in the message header and increments the counter. The first message must have sequence number zero. Message receivers track the sequence numbers to verify that messages are received in order and without loss. If an unexpected sequence number is received then the AF_XDP socket is in error and XTS sockets must be closed. This is a fatal error and the best course of action may be to terminate the application.

Flow Control

Flow control for sending and receiving data on an XTS socket is managed by a credit mechanism called *post credits*. Both the XDP-Transport library and the offload proxy provide credits to their peers for sending data. Post credits is the number of bytes that are allowed to be posted to the peer. This is either the amount of data that can be in a “send data” message sent by the server to the offload proxy, or the amount of data in a “receive data” message sent by an offload proxy to the server. Each byte consumes one credit. When data is posted, the number of credits must be greater than or equal to the number bytes in the operation, and the number of post credits is decremented by the number of bytes in the operation.

As the peer processes post data, credits are *returned* in credit advertisements. Credit advertisements can be sent in “pure credit advertisement” messages or can piggyback on other messages. Each side keeps a counter of “unadvertised credits” that can be advertised to its peer. Returned credits are additive. For instance, if the current post credits is 20 and a credit advertisement is received with a value of 10 then the new post credits is 30.

Offload proxy flow control

The offload proxy maintains its own per socket buffer, the *spill buffer*, in addition to the socket buffer for the TCP connection in the kernel. The spill buffer is used to manage post credits. The size of the spill buffer is assumed to be at least equal to the size of the send socket buffer. The sum of the sizes of the spill buffer and the send socket buffer is the maximum number of post credits given to the application.

When the offload proxy receives a “send data” message from the server it performs the following procedures. Note *num_to_send* is the number of bytes to send, *num_sent_bytes* is the number of bytes successfully written, $POST_CREDITS_{UNA}$ is the number of unadvertised credits, and POST_CREDITS_LIMIT is the threshold to send a pure credit advertisement message.

1. If the spill buffer is empty, write new data on the TCP socket and perform the following:
    a) $POST_CREDITS_{UNA}$+= *num_sent_bytes*
    b) If $POST_CREDITS_{UNA}$>POST_CREDITS_LIMIT then send a pure post credits advertisement and set $POST_CREDITS_{UNA}$=0
    c) If *num_sent_bytes* < *num_to_send* then copy the unsent bytes of data to the spill buffer
2. If the spill buffet is not empty copy *num_to_send* bytes to the tail of the spill buffer

When a TCP socket becomes readable perform the following:

1. Send data from spill buffer (*num_sent_bytes* is the number of bytes successfully sent)
2. $POST_CREDITS_{UNA}$+= *num_sent_bytes*
3. If $POST_CREDITS_{UNA}$>POST_CREDITS_LIMIT then send a pure post credits advertisement and set $POST_CREDITS_{UNA}$=0

Server flow control

The XDP-Transport library in the server maintains a receive buffer for each XTS socket. The maximum amount of outstanding data that the offload proxy can post to the application is the size of this buffer. The procedures for draining the buffer are:

1. The application calls *xts_recv_** and reads bytes from the receive buffer (*num_recv_bytes* is the number of bytes read by the application)
2. $POST_CREDITS_{UNA}$+= num_recv_bytes
3. If $POST_CREDITS_{UNA}$>POST_CREDITS_LIMIT then send a post credits advertisement and set $POST_CREDITS_{UNA}$=0

Message Types

Constants for message types have the form XTS_CMD_* for messages sent by XDP-Transport library in the server, and the form XTS_REPLY_* for messages sent by the offload proxy. The basic XDP-Transport message types are:

XTS_CMD_CONNECT
Start a proxy connection. Message parameters include the XTS socket number, IP protocol version (4 or 6), IP protocol (currently TCP), address and port for the connection

XTS_REPLY_CONNECT_ACK
Report success or failure of a connection request. Parameters include the XTS socket number that was received in the XTS_CMD_CONNECT message the connection identifier of the offload proxy

XTS_CMD_LISTEN
Request to create a listener. Message parameters include the XTS socket number for the listener and the port number

XTS_REPLY_LISTEN_ACK
Acknowledge an XTS_CMD_LISTEN message and indicate the listener started. Message parameters include the XTS socket number and offload proxy’s identifier for the listener

XTS_REPLY_NEWCONN
Report a new connection received on a listener. Message parameters include the XTS socket number of the listener, the offload proxy’s connection identifier, and the address and port number of the peer endpoint of the connection

XTS_CMD_NEWCONN_ACK
Acknowledge an XTS_REPLY_NEWCONN message. Message parameters include the connection identifier of the offload proxy, and the XTS socket number

XTS_CMD_SEND
Send data and/or close the connection (or listener). Message parameters include offload proxy’s connection identifier, the length of data, the data to send, and advertised post credits

XTS_REPLY_RECV
Post received data and/or report connection was closed. Message arguments include the XTS socket number, the length of data, the received data, and advertised post credits

Application API

The XDP-Transport socket API is an analogue of the standard BSD sockets API [4]. Function names have the form *xdp_xport_** where * is replaced by the name of the corresponding socket function. For instance, *xdp_xport_socket* is the analogue of the *socket* function, *xdp_xport_listen* is the analogue of the listen function, and so on. *xdp_xport_ioctl* and *xdp_xport_fcntl* are also defined as the analogue of the *ioctl* and *fcntl* functions which are not strictly BSD socket functions. The XDP-Transport socket functions are:

```
xts_socket          xts_accept
xts_close           xts_send
xts_ioctl           xts_sendto
xts_fcntl           xts_sendmsg
xts_getsockopt      xts_recv
xts_setsockopt      xts_recvfrom
xts_bind            xts_recvmsg
xts_connect         xts_getsockname
xts_listen          xts_getpeername
```

These functions take the same arguments as their counterparts in BSD sockets. The major difference is the number used to identify sockets. In BSD sockets, the file descriptors identify sockets-- the *socket* and *accept* functions return a file descriptor and a file descriptor is the input argument to various socket functions. In the XDP-Transport sockets API, XTS sockets are represented by XTS socket numbers that are identifiers in their own number space-- the *xdp_xport_socket* and *xdp_xport_accept* calls return an XTS socket number and an XTS socket number is the input argument to various XTS socket functions. Note that an XTS socket number must never be used as input to functions expecting a file descriptor.

The steps for an application to create an offloaded connection are:

1. Create an XTS socket by calling *xts_socket*. The arguments are similar to those of a *socket* call to specify IPv6 or IPv6 and TCP protocols
2. Connect to a destination by calling *xts_connect*. The arguments are similar to those of a *connect* socket call. The XDP-Transport library on the server sends an XTS_CMD_CONNECT message to the offload proxy over an AF_XDP socket. *xts_connect* returns an error code indicating whether connection establishment was successful
3. The offload proxy receives an XTS_CMD_CONNECT message. It creates a TCP socket by calling socket and calls *connect* to establish a connection
4. When the connection is established, the offload proxy sends a XDP_XPORT_CMD_CNX_REPLY message to the server on the appropriate AF_XDP socket
5. The XDP-Transport library on the server receives the connection reply message and returns to application from the *xts_connect* with an error code for success

The steps for an application to create an offloaded listener and to accept connections are:

1. Create an XTS socket by calling *xts_socket*. Arguments are similar to those of a *socket* call to specify IPv4 or IPv6 and TCP protocols
2. Start a listener by calling xts_listen. Arguments are similar to those of the *listen* socket call. The XDP-Transport library on the server sends an XTS_CMD_LISTEN message to the offload proxy
3. The offload proxy receives the XTS_CMD_LISTEN message. It creates a TCP socket by calling *socket* and calls *listen* to listen on the socket. It then calls *accept* on the socket in an event loop
4. The application calls *xts_accept* and waits for new connections. Polling, described below, may also be used for asynchronous listeners
5. New connections from the offload proxy are returned from *accept*. For each new connection, the offload proxy sends an XTS_REPLY_NEWCONN message to the server. The message parameters include the proxy's identifier for the new connections and the XTS socket number of the listener
6. When the XDP-Transport library on the server receives a XTS_REPLY_NEWCONN message, it creates a new XTS socket for the connection. The library sends an XTS_CMD_NEWCNX_ACK message to the offload proxy. The message parameters include the XTS socket number and the offload proxy's connection identifier that was received in the XTS_REPLY_NEWCONN message
7. The library returns to the application from *xts_accept* with the XTS socket number for the new connection

Send and receive operations

When an offloaded connection is established the application can begin sending and receiving data on the offloaded connection. The application calls *xts_send_** functions to send data, and *xts_recv_** functions to receive data.

When the application sends data on an XTS socket, the XDP-Transport library sends an XTS_CMD_SEND message to the offload proxy. The message contains the data length and the data itself. The message may also include advertised post credits and may indicate the connection is to be closed. When the offload proxy receives the XTS_CMD_SEND message, it writes the message on the TCP socket using *sendmsg* socket functions.

The offload proxy receives data on sockets by calling *recvmsg*. Received data is sent to the server in an XTS_REPLY_RECV message. The message contains the length of data and the data. The message may also include advertised post credits and may indicate that the connection was closed. When the XDP-Transport Library in the server receives the XTS_REPLY RECV message it buffers the data and provides it to the application when *xts_recv_** functions are called.

Polling

Similar to Linux file descriptors, XTS sockets can be polled. However, since they're not file descriptors they cannot be polled directly by *epoll* [5]. The strategy is to poll the underlying AF_XDP socket, and then call a secondary polling function on poll events for AF_XDP sockets. The secondary polling function checks the status of XTS sockets associated with an AF_XDP socket and returns events for those XTS sockets that are ready.

To enable polling of XTS sockets, an *XTS polling context* is first created by calling *xts_epoll_ctx_create*. This function allocates and initializes a data structure to hold the context for polling XTS sockets, and returns a pointer to the allocated structure. One XTS polling context is created for each AF_XDP socket associated with XTS sockets. XTS sockets are added to the XTS polling context by calling the *xts_epoll_ctl* function. This function is analogous to *epoll_ctl* and takes the same arguments except that the *epoll* file descriptor argument is replaced by a pointer to the XTS polling context. Polling is enabled for AF_XDP sockets by calling *epoll_ctl* where the event argument contains a pointer to the XTS context created for the AF_XDP socket.

Applications typically call *epoll_wait* in an event loop. When an AF_XDP socket with XTS sockets is ready, an associated event is returned by *epoll_wait*. The returned event structure includes a pointer to the XTS polling context associated with the AF_XDP socket. The program calls *xts_epoll* which takes the XTS polling context as an argument. *xts_epoll* returns a set of events for ready XTS sockets that were added to the XTS polling context. The XTS socket corresponding to each event can then be processed.

The API functions for polling are listed below:

```
void *xts_epoll_ctx_create();
```

Create an XTS polling context. The return value is either a pointer to the allocated context or NULL in case of failure

```
int xts_epoll_ctl(void *ectx, int op,
                  int xts_socket_num,
                  struct epoll_event *ev);
```

Add, modify, or remove entries in the interest list of an XTS polling context. *ectx* is an XTS polling context. *op* indicates the operation (EPOLL_CTL_ADD, EPOLL_CTL_DEL, or EPOLL_CTL_MOD). *xts_socket_num* indicates the XTS socket. *ev* is an *epoll* event structure that is returned by *xts_epoll* when the XTS socket is ready.

```
int xts_epoll(void *ectx,
              struct epoll_event *events,
              int maxevents);
```

Return the ready events for an XTS polling context. *ectx* is an XTS polling context. *events* is a buffer that holds information about returned events in *epoll_event* structures. *maxevents* is the maximum number of events that can be returned.

Example poll loop

The code snippet below provides an example poll loop with polling of XTS sockets. Modifying a typical application poll loop to support offloaded connections is expected to be a matter of inserting code to perform secondary polling of AF_XDP sockets. Note that an XTS polling context is returned in an event for an AF_XDP socket. If other file descriptors are also polled their events need to be distinguished from those of AF_XDP sockets. In this example that is done by including a pointer to an XTS polling context in all events; If the pointer is NULL then the event is not for an AF_XDP socket with XTS sockets.

```
/* Setup code for polling XTS sockets */
struct my_event {
  void *xts_ctx;
  /* Fields for non xdp_xport events */
};
struct epoll_event ev, *evp, evs[MAX_EVS];
int epfd, num, I, xnum, j;
struct my_event *myev;
void *xectx;

epfd = epoll_create(0);
xectx = xdp_xport_epoll_create();
myev = malloc(sizeof(*myev));
myev→xts_ctx = xectx;
ev.events = EPOLLIN;
ev.data.ptr = myev;
epoll_ctl(xectx, EPOLL_CTL_ADD, af_xdp_fd,
          myev);

ev.events = EPOLLIN;
ev.data.u64 = xts_sock_num;
xts_epoll_ctl(xctx, EPOLL_CTL_ADD,
              xts_sock_num, &ev);

/* Event loop for polling XTS sockets */
while (1) {
  num = epoll_wait(epfd, evs, MAX_EVS, -1);
  for (i = 0; i < num; i++) {
    myev = (struct my_event *)
                      evs[i].data.ptr;
    if (myev->xts_ctx) {
      struct epoll_event xevs[MAX_EVS];

      xnum = xts_epoll(myev→xts_ctx,
                       xevs, MAX_EVS);
      for (j = 0; j < xnum; j++) {
          /* XDP-Transport socket ready */
          process_xts_socket(
                     xevs[i].data.u64);
      }
    } else {
      /* Process application fd */
      process_ready_fd(event.data.ptr);
    }
  }
}
```

ULP Offloads

Offloading the processing of Upper Layer Protocols (ULPs) is done by creating an offloaded TCP connection and "pushing" ULP functions onto the TCP socket. Logically, this creates a protocol stack on the socket. An XTS socket with a pushed ULP stack is effectively a *ULP socket* that allows an application to send and receive Protocol Data Units (PDUs) corresponding to the protocol of the offloaded ULP. The protocol used with a ULP socket is programmable such that ULP sockets can be created for various protocols including RDMA/TCP, NVMe/TCP, and TLS.

In the example of Figure 6, ULP functions are pushed onto a TCP socket to create "HTTP/2-HTTPS sockets" [6]. An application would send and receive plain text HTTP messages on such a socket. As shown in this example, the send and receive ULP processing for a socket may be different. In the send path, the application sends HTTP/2 requests, and in the offload processing the message data is encrypted by TLS and the output data is then sent on the TCP socket. The receive path is a bit more involved. Data is received on the offloaded TCP socket, and an instance of strparser (described below) delineates the data stream into TLS records. The TLS records are decrypted by the TLS layer and the output is a data stream. The data stream is then processed by a second instance of strparser that delineates the stream into HTTP/2 messages. The HTTP/2 messages are received by the application on an XTS socket.

strparser

strparser ("stream parser") is a generic facility to parse a data stream into discrete ULP messages. strparser can parse any ULP protocol on a data stream that has a standard message format with a message header that indicates the length of the message. strparser has been implemented in Linux kernel where the parsing of the ULP header is done by an eBPF program. An eBPF program is attached to the strparser instance parsing a TCP connection and provides the instructions to parse the ULP header and return the length of the next message to strparser. Below is a snippet of eBPF code parse theHTTP/2 header:

```
int bpf_prog1(struct __sk_buff *skb)
{
   __u32 w;
   if (bfp_skb_load_bytes(skb, 0, &w, 3))
       return 0;
   return ntohl(w);
}
```

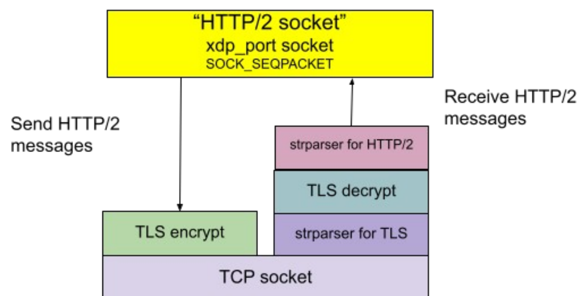

Figure 6. Example of making "HTTP/2 sockets" with TLS by pushing ULP programs

API

ULP protocols are pushed onto an XTS socket by a "ULP push" IOCTL. Two IOCTLs are defined: SIOCPUSHULPTX and SIOCPUSHULPRX for pushing ULPs on the transmit side and receive side respectively.

The argument of the IOCTL is a castable "push info" structure:

```
struct ulp_info {
    __u16 ulp_id;
    __u8 ulp_parameters[0];
};
```

ulp_id identifies the ULP being pushed. Identifiers for supported ULPs can be advertised by the offload device using an out-of-band mechanisms. *ulp_parameters* are variable length parameters specific to the ULP being pushed.

The pseudo code to create the pushed protocol stacks for HTTP/2 sockets illustrated in Figure 6 is:

```
s = xdp_xport_socket(AF_INET6,
             SOCK_SEQPACKET, IPPROTO_TCP);

/* Push ULPs for receive side */
xts_ioctl(s, SIOCPUSHULPRX,
          &strparser_tls_info);
xts_ioctl(s, SIOCPUSHULPRX, &tls_rx_info);
xts_ioctl(s, SIOCPUSHULPRX,
          &strparser_http2_info);

/* Push ULPs for transmit side */
xts_ioctl(s, SIOCPUSHULPTX, &tls_rx_info);
```

ULP protocol handshake

In some cases, a ULP protocol is used starting from the first byte of a TCP connection, however in other cases there may be a protocol handshake over the TCP connection that precedes a switch to using an Upper Layer Protocol on the connection. An example is when an HTTP connection on port 80 switches to use TLS encryption. Note that once the ULP switch-over happens, for the rest of the lifetime of the connection only ULP messages of the negotiated protocol are used. To support ULP protocol handshake, the offload proxy connection can be set to "peek mode" at the beginning of a connection. While in peek mode, the offload proxy peeks the data in the TCP socket buffer so that received data remains in the socket buffer (by using MSG_PEEK with *recvmsg*). The peeked data is sent to the application that can parse the data and determine the ULP functions to be pushed. Once the proper ULPs have been pushed, the application performs an XTS IOCTL "end peek mode" operation (SIOCENDPEEKMODE). The arguments to the IOCTL include the number of bytes to drop from the first received byte on the connection. When the proxy server receives the message, it reads and discards the number of bytes to drop. Subsequently, normal operations commence where data is read from the TCP socket and is processed by any pushed ULPs and PDUs are delivered to the application.

Implementation

When ULPs are offloaded to an App CPU, it is the discretion of the device how to implement the datapath for the protocols. In the simplest case, all of the ULP processing could be done in the user space offload proxy. Alternatively, kernel functions or hardware offload could be employed. A hybrid approach could be used where some ULPs are offloaded to the kernel or hardware, and others are implemented in the user space proxy. For instance, in the HTTP/2 example illustrated in Figure 6, the TLS processing could be offloaded to a NIC hardware that supports TLS offload, but the strparser for delineating HTTP/2 messages might run in the userspace offload proxy.

Security

Security is provided by mechanisms of AF_XDP sockets. This is supported by enforcement of permission to allow a program to create an AF_XDP socket as well as by eBPF/XDP programs that are attached to network interfaces and can implement arbitrary security policies. Note that scalable implementation with security is a topic for future work.

Performance Results

The primary goal of TCP offload via AF_XDP sockets is to offload CPU cycles from host CPUs. Figure 7 compares the host CPU utilization with and without offload. Note that this data is based on preliminary analysis and extrapolation of a “proof of concept” implementation. Future work includes a more complete implementation and detailed performance analysis.

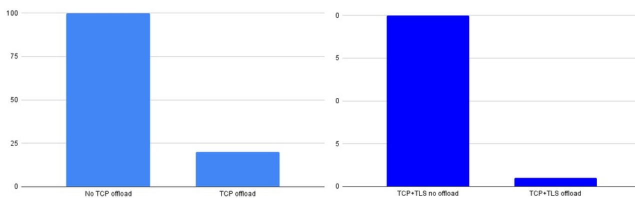

Figure 7. Performance of TCP offload via AF_XDP sockets. The graphs compare the relative host CPU utilization with and without offload. The graph on the left compare no TCP offload with TCP offload, and the graph on the right compares TCP and TLS with and without offload

The primary performance benefits of TCP offload come from eliminating the instructions in the execution path for processing protocol headers, and in the case of a ULP, such as TLS, there are also benefits in offloading the algorithm. AF_XDP sockets are expected to have inherently less overhead of typical protocol sockets like TCP and UDP sockets. Managing the offload and supporting an API does require some overhead in userspace. This is mostly a case of dereferencing user space sockets in library functions. Flow control is implemented as part of the user space sockets library, it’s not particularly more complex than managing socket buffers for flow control, and note that pure flow control messages sent to and from the offload device can be minimized.

Acknowledgments

The author would like to thank Mike Rubin, Michael Winsner, and Michae Davidson for their valuable feedback.

References

1. Topel, B., Karlsson, M., Duyck, A., Starovoitov, A., Borkmann, D., Brouer, J., Fastabend, J., Corbet, J., Tsirkin, M., Zhang, Q., and W. de Bruijn, "The Linux Kernel: AF_XDP", https://www.kernel.org/doc/html/next/networking/af_xdp.html.

2. Linux Foundation, L., "Wiki: TOE", https://wiki.linuxfoundation.org/networking/toe.

3. Wilson, B., "Microsoft: Why Are We Deprecating Network Performance Features", https://techcommunity.microsoft.com/t5/core-infrastructure-and-security/why-are-we-deprecating- network-performance-features-kb4014193/ba-p/259053.

4. Chukov, S. P., "Socket programming", https://wiki.netbsd.org/examples/socket_programming/.

5. Linux manual page, "epoll(7)", https://man7.org/linux/man-pages/man7/epoll.7.html>. https://man7.org/linux/man-pages/man7/epoll.7.html

6. Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, https://www.rfc-editor.org/info/rfc7540.

7. Herbert, T., "Stream Parser (strparser)", https://docs.kernel.org/networking/strparser.html.

Author Biography

Tom Herbert is Chief Technical Officer of SiPanda working on high performance programmable data paths, open source development , and protocol development.