## Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Who</th>
<th>Revision</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>DM</td>
<td>Document created</td>
<td>5/13/03</td>
</tr>
<tr>
<td>1.0</td>
<td>Asgeir</td>
<td>Cleanup edits: defined a few more terms such as TID and PTID, made</td>
<td>7/22/05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>description more focused on the current TOM implementation in a couple of</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>places.</td>
<td></td>
</tr>
</tbody>
</table>
Table of Contents

1. **Introduction**........................................................................................................ 4
   1.1. Definitions ......................................................................................................... 5
2. **Architecture Overview**........................................................................................ 5
   2.1. Support for Offloading Other Protocols......................................................... 8
3. **The toedev Structure** ....................................................................................... 8
4. **The Offload Device Layer**.................................................................................. 9
5. **The Offload Driver**........................................................................................... 10
6. **The TCP Offload Module** ................................................................................ 11
   6.1. Data Structures.................................................................................................. 11
   6.2. Interfaces........................................................................................................... 12
   6.3. Neighbor (ARP) Management......................................................................... 13
   6.4. Route Management ......................................................................................... 13
   6.5. Path MTU Changes.......................................................................................... 13
   6.6. toedev Fields Set by TOM............................................................................... 14
   6.7. Listen Offload.................................................................................................. 14
7. **The Offload Switch** .......................................................................................... 14
   7.1. Offloading listening sockets............................................................................. 15
   7.2. Active open offload ....................................................................................... 15
   7.3. Passive open offload ..................................................................................... 16
8. **Dealing with Virtual Network Devices** ............................................................ 16
1. Introduction

A Protocol Offload Engine (POE) is a HW device that accelerates a network protocol by performing some or all of the protocol processing, thus freeing up the host OS and CPU to perform other tasks. This document describes Chelsio’s implementation for supporting TCP Offload Engines (TOE) in Linux kernels 2.4 and above. It integrates into the standard SW network stack to offload the TCP/IP processing that is implemented by the underlying HW devices. The goal is to work with the Linux community to extend the current Chelsio implementation to a fully fledged generalized Linux TCP offload architecture.

Goals of the proposed design include:

- Modular design. The architecture subdivides TOE support into a number of fairly loosely coupled modules and allows the easy addition of new module implementations to support new HW devices.
- Multi-vendor support. Although the architecture should provide robust support for Chelsio cards, every effort has been made for it to be able to accommodate alternate HW designs so as to be usable for other vendors’ products. This is considered a fundamental requirement for the architecture to gain community acceptance.
- Minimal changes to Linux. Although changes to the existing Linux networking code can’t be avoided altogether, it is our intent to keep them to a minimum, but without compromising performance or quality.
- Eventual integration into Linux. It is very desirable that TOE support be integrated into mainline Linux in order to provide solid infrastructure for TOE drivers and lower code maintenance costs. As part of this goal the design is a native Linux design, forgoing abstraction layers.
- High performance. The architecture offers lightweight services and enables the providers of the TOE support for each HW device to implement a high performance solution. In particular, it is possible to tailor the data path almost fully in order to leverage the strengths of each TOE device.
- Rich configuration and reporting. The architecture supports selective policy-based offloading both for TCP connections and for listening sockets, per-TOE and per-connection tunable parameters, as well as reporting and configuration through customary Linux mechanisms or extensions (e.g., netlink extensions to report offload-specific connection parameters). Whenever possible we support existing configuration and reporting tools (netstat, ifconfig, ip, ss, netlink, /sysfs, /proc, etc).
- Ability to use existing security mechanisms, such as firewalls or listen backlog limits. Offloading TCP does not imply reduced security compared to the host TCP. Although some offload drivers may not be able to support all such features, it is the intent of the architecture that TOE connections go through the security checks that the host TCP would normally employ.
The next section defines the TOE framework and gives a quick description of the building blocks. Subsequent sections delve into the details and describe changes made to the Linux kernel and the interfaces of the various architecture modules.

1.1. Definitions

- Link layer driver: the SW driver that controls the link layer (usually Ethernet) portion of a TOE device.
- NIC driver: a synonym for a link layer driver.
- POM: Protocol Offload Module, a SW component that interacts with a host network stack to offload one or more network protocols as supported by an underlying network protocol acceleration device.
- TOE: TCP Offload Engine, a HW device that offloads TCP processing.
- TOE driver: the SW driver that controls the offload portion of a HW TOE device.
- TOM: TCP Offload Module, a SW component that interacts with the host network stack to implement the higher level functions needed to utilize a TOE device. A TOM is a POM for TCP.
- ULP: Upper Layer Protocol, a protocol on top of a TCP, such as iSCSI.
- TID: a unique numeric identifier for a TCP connection or listening socket server (also referred to as PTID), or an active open request (also referred to as ATID).

2. Architecture Overview

A block diagram of the network stack architecture with the TOE additions is shown in Figure 1. Most of the TOE functionality is provided by two new components, the TCP Offload Module (TOM), and the offload driver. These two collaborate with the host TCP to provide the offloaded TCP functionality. A TCP offload module implements the upper, protocol-aware half of a TOE stack and provides the following functionality:

- Implements the portions of TCP processing not supported directly by the underlying HW. Examples include managing receive windows, etc.
- Maintains SW state for offloaded connections.
- Implements any of the API calls provided by the transport layer to its clients (e.g., socket APIs) that require different behavior from that of the host TCP. A prime example is the data path APIs, where a TOE will typically want to implement its own optimized method of sending/receiving data to/from the TOE HW (e.g., there’s no need for TCP headers, etc.).
- Implements any additional APIs not provided by the existing host TCP that may be of use to clients of the offloaded stack (e.g., special APIs to support upper layer protocols (ULPs) such as iSCSI or RDMA).
- Communicates with the underlying TOE HW to carry out TCP duties (sending/receiving data, shutting-down/closing connections after intercepting application shutdown/close, etc) or for control purposes (e.g., programming HW TCP parameters). An offload module does not directly manipulate the underlying HW, instead relying on messages and APIs provided by the TOE driver.
• Interfaces with the host network stack to set up and maintain any HW routing and neighbor (e.g., ARP/ND) resources necessary.

The second important TOE component is the offload (TOE) driver, which can be thought of as the low-level portion of the TOE stack and which is responsible for manipulating the HW TOE resources. Note that in general there are additional drivers associated with the same HW device, namely standard link-layer drivers, since most TOE cards are also usable as plain Ethernet devices. We have chosen to separate the Ethernet drivers from the offload driver to highlight the multiple roles of such HW, although, when it comes to implementation, the two functionalities may be provided by the same device driver. We give more details on the offload driver in a later section.

![Figure 1 - TOE Architecture](image-url)
As shown in Figure 1 a host may be equipped with multiple TOE devices and there can be more than one TCP offload module simultaneously available. This provides explicit support for setups with multiple TOEs, each implementing TCP offload differently. Each TOE can have at most one offload module associated with it at any time, but each TOM can handle any number of TOEs, including none. A TOE may be able to work with more than one of the available TOMs, but eventually only one of the eligible TOMs will be associated with the device. Note also that there is no relationship between the TOM and the offload driver associated with a TOE other than that the two must be able to work together.

The coupling of a TOM to a TOE is accomplished through the offload device layer, which provides registration, generic configuration and reporting, and generic data path services for protocol offloading. Both TOMs and offload devices register with the offload device layer, which uses information provided by the devices to match up TOEs with offload modules that can handle them. This loose coupling allows flexibility in loading order of offload modules and TOE drivers and enables support for non 1-1 relationships between offload drivers and TOMs.

Despite the presence of potentially several different TCP offload modules there is a single TCP that is visible to applications, namely the standard TCP of the host’s protocol stack. The offload modules do not need to register with IP or IPv6 address families and sockets are not associated directly with a TOM. Connections and listening sockets are switched between the host TCP and one or more TOMs through a switch component that offloads active connection open, and listen based on offload policies configured by the administrator. This switch is driven by a set of rules that instruct the switch which listening sockets should be passed to TOMs for HW connection setup and which active connection open requests can be offloaded. The switch then collaborates with the TOMs to determine if there are adequate TOE resources for the offload and if so it offloads the connection to the TOE. The offload policy rule database is built on the Linux netfilter framework with a few modifications. We describe how offloading of listening sockets and TCP connections works separately as the two use different mechanisms.

A listening socket that is not bound to a particular device needs to be offloaded to all TOEs to offload connection setup. To minimize changes to the existing stack we use a publish/subscribe scheme. TOMs with listen offload capability subscribe to receive events triggered by the listen and close system calls on listening sockets. When a socket starts listening the offload switch checks whether the socket should be offloaded and if so it publishes an appropriate event. TOMs that have subscribed are then free to do whatever they need to inform their associated TOEs of the listening socket. The same mechanism works when a listening socket is closed. This provides a rather non-intrusive mechanism for listen offload as well as offers each TOM the ability to implement listen offload as appropriate for its underlying device.

Active socket opens check at connect time whether the connection should be offloaded. If the offload policy allows it the switch again works with the TOM to determine if adequate resources exist and if so the connection is offloaded. For passive opens there are two cases: a) if the listening socket is offloaded and HW handles the TCP 3-way handshake without SW intervention the new connection is handled by the HW TCP, and b) if the listening socket is offloaded but connection setup is SW assisted the switch,
under the direction of the offload policy, may decide to have the SW TCP or let the HW handle it, so that when the connection is established it could be handled by either SW or HW.

### 2.1. Support for Offloading Other Protocols

Although we are concentrating on TCP offload recall that a TOM is just a special case of a POM. The architecture described above is not dependent on TCP and could be used to support offloading of other network protocols by replacing TOMs with POMs for those protocols. Additionally, a HW device may be able to offload multiple network protocols. In this case there would be additional offload drivers for the device to support the extra protocols, and each such offload driver would be associated with a corresponding POM giving rise to a setup where multiple POMs could be associated with the same underlying HW device (e.g., a TCP offload module and an RDMA offload module could be associated with a HW device that is both a TOE and an RNIC, or a TOE with iSCSI and IPsec support could be associated with TOE, iSCSI, and IPsec POMs). As before a single POM could manage multiple devices and each offload driver would be associated with at most one POM at a time.

Note that although architecturally we identify a separate offload driver and POM for each protocol that can be offloaded by a HW device, an implementation may combine several or all of the offload drivers into a single SW module, and likewise for POMs supporting different protocols. It is even possible for an implementation to combine one or more offload drivers and one or more associated POMs into the same SW module. A SW module that is integrated enough to include both the offload driver(s) and the associated POM(s) may forgo registering with the offload device layer.

### 3. The toedev Structure

The central data structure for each TOE that is shared by the various components of the TOE architecture is a `struct toedev`. These structures contain various parameters of the underlying HW as well as a number of device methods (function pointers), and they should be viewed as the TOE analog of Linux’s `net_device` structures for link-layer devices. See `../include/linux/toedev.h` for the complete details of this structure. Its most important members are:

- **name** – This is the name of the TOE device, e.g., “toe0”. Assigned by the offload driver and the TOE device layer.
- **ttid** – The TOE type id, assigned by the driver. The TOE device layer uses this to find appropriate TOMs for the new device.
- **mtu** – The maximum size of an offload packet, including headers, the device can receive from the host.
- **nconn** – The maximum number of offloaded connections.
- **lldev** – The link layer device associated with the TOE device. Packets exchanged between the toe driver and the upper layers appear to be coming from this device for purposes of packet capture applications (e.g., ethereal or tcpdump).
- **open** – This function is called to activate the TOE, typically when an appropriate TOM is found. It is implemented by the offload driver and should initialize any TOE related HW resources and advertise TCP offload capability on all attached ports.
- **close** – Deactivates TOE functionality on a device. It should disable the TCP offload capability on attached ports.
- **can_offload** – This is called by the switch when it has decided that a connection should be offloaded and returns a boolean indication of whether the offload is possible (usually due to resource availability). It is implemented by the TOM.
- **connect** – This is called by the offload switch when it has decided to offload an active open. It is implemented by the offload module and should perform any steps needed for the TOM to take over the new connection.
- **neigh_update** – Called when the host’s neighbor subsystem (e.g., ARP) changes the state of some neighbor entry. It is provided by the offload module and should update the HW state for the modified neighbor, if applicable.
- **route_update** – Called when the host’s routing subsystem wants to communicate a change to a connection’s routing information.
- **pmtu_update** – Called by the host stack when a path MTU change has been detected.
- **send** – Used to send a message to the TOE. Implemented by the offload driver.
- **receive** – Called by the offload driver to deliver one or more received offload messages. Specified by the TOM.
- **recv_stray_packet** – Called by the host stack when it has received a non-offloaded packet for an offloaded connection. This can happen due to routing mishaps. When the host stack detects such a packet it passes it to the TOE driver through this call. It is up to the TOE driver to deal with the packet according to the capabilities of the TOE device.
- **ctl** – An ioctl-like function provided by the offload driver and used by the other components to control the TOE (e.g., program various HW tables when it cannot be done through messages, set HW TCP parameters, etc). Also used to query various parameters of the TOE device.
- **l2opt, l3opt, l4opt, ulp** – These are pointers to non-standard optional structures used by the L2 (ARP), L3 (routing), L4 (TOE), and ULP code. The structures are created by the appropriate subsystem and attached to the toedev.

A toedev structure is created by the offload driver and registered with the offload device layer. The information in a toedev is initialized either by the offload driver or by the associated TOM.

### 4. The Offload Device Layer

This subsystem provides for offload devices functionality similar to that offered by `net/core/dev.c` for Linux’s network devices (`net_device`). It provides registration services for offload drivers and TOMs (or other POMs) and performs the association between TOMs and compatible TOE devices. The main data structures used for registration are the following:

- `toedev` structures described previously, that are registered by the TOE drivers.
• **tom_info** structures registered by TOMs. Each such structure contains the TOM’s name, a table of TOE ids listing the TOE devices a TOM is compatible with, and two methods, attach and detach.

The offload device layer maintains two doubly linked lists, one for each of the two structures mentioned above. In addition to the registration and deregistration methods for these structures, the offload device layer provides methods to activate and deactivate an offload device. When a TOE device is activated the offload device layer looks for a compatible TOM and attempts to associate it with the TOE device by calling its attach method. The compatibility test entails comparing a toedev’s ttid against the ids listed in a TOM’s tom_info. If a compatible TOM is not readily available the offload device layer will repeat the attempt as each subsequent TOM is registered. If the attach is successful the toedev is associated with the offload module, the rest of the toedev fields are filled in by the offload module, and the TOE is fully activated by calling the open method listed in its toedev.

When an offload module is registered the offload device layer looks for any unclaimed toedevs with matching ttids and then follows the above procedure. This scheme allows offload modules and drivers to be loaded in any order, even allowing hot-plug TOEs.

Besides managing toedev/TOM registration and activation this subsystem also provides the glue logic to send and receive offloaded packets, which also takes care of making the packets available to any active network taps. This is accomplished through two functions:

- **toe_send** – sends offload packets from a TOM to its TOE driver by calling the toedev’s send method
- **toe_receive_skb** – delivers packets from the TOE driver to the associated TOM by calling the toedev’s receive method.

These calls provide for batch exchange of packets between a TOM and its TOE driver for increased performance, as well as for recycling received buffers back to the TOE driver.

Unlike the Linux net_device layer, the offload device layer does not provide any receive queues or any services for scheduling packet delivery. In particular there are no special soft interrupts. TOE drivers are expected to use standard Linux facilities, such as NAPI, for managing their receive queues.

### 5. The Offload Driver

The offload driver is the low level half of the offloaded protocol stack and is responsible for managing the TOE functionality of the underlying HW. It is the only SW component that directly manipulates HW resources. It creates a toedev structure and sets up the methods through which every other piece of SW can interact with the HW.

When the offload driver is associated with a TOE HW device during PCI device/driver match up the driver will allocate a toedev structure to describe the TOE. The driver should not initialize the HW at this time, this is done when the open method is called later. The following toedev members should be initialized:

- **ttid** – The driver sets the TOE type id required by the HW. Only TOMs that support this type can attach to the TOE. This id is determined from the PCI ids of the HW.
• mtu – The maximum amount of data that can be sent to the TOE, exclusive of headers.

• nconn – The number of connections supported by the TOE, derived from the size of the appropriate TCAM region.

• open – This function is called when the TOE HW should be activated. The driver should initialize any HW resources (memory controllers, TCAM, device registers) and it should arrange for any Ethernet devices associated with the TOE to start advertising that they can offload TCP by setting the NETIF_F_TCPIP_OFFLOAD flag in the corresponding net_devices’ features word. The architecture also supports the NETIF_F_TCPIP6_OFFLOAD flag for TOEs that support TCP over IPv6. Additionally, a pointer will be set in each net_device associated with the TOE to point to the corresponding toedev. This will allow the offload switch to locate the TOE given the network interface. To avoid adding new members to the existing net_device structure we use one of the existing pointers in that structure that are normally unused in this situation. We have chosen to use the Econet pointer and hide this detail behind a pair of get/set macros. Note that these pointers may be set up once when the net_device and the toedev structures are created, they will not be looked at unless the NETIF_F_TCPIP_OFFLOAD flag is set. Upon return from the open function the offload driver must be able to send and receive offloaded traffic. The driver must be able to support any number of calls to the open method as long as each call besides the first is preceded by a call to close.

• close – This is called when the TOE functionality is no longer needed. The driver will turn off the NETIF_F_TCPIP_OFFLOAD flags on the associated net_devices and it may optionally remove the pointers from the net_devices to the toedev. After this function is called the driver can assume that there will be no more offloaded traffic.

• send – All offloaded data and control packets are submitted to the driver through this interface as sk_buffs.

• ctl – This is the interface through which other SW components can program the various tables (MTU table, HighSpeed TCP parameters, link aggregation tables, etc) and TCP related registers of the HW device. It is also used to query various parameters of the TOE, such as TX/RX page sizes, TCAM regions, etc. It is similar to an ioctl handler and the commands available are dependent on the driver.

Once the driver initializes the toedev structure it registers it with the offload device layer by calling register_toedev(). The companion function unregister_toedev() can be called to remove the toedev.

Offloaded packets received by the driver are delivered to upper layers through the toe_receive_skb call provided of the offload device layer while the driver receives packets to send through its toedev’s send method. The send method may also be used to send control messages to the offload driver that are handler by the driver itself and not forwarded to the HW device.

6. The TCP Offload Module

The responsibilities of TCP Offload Modules were detailed in Section 2. We concentrate on the data structures and main code paths of a TOM below.
6.1. **Data Structures**

Although a TOM is free to maintain whatever structures it needs to hold the state of offloaded connections (some of the details depend on the underlying HW device) it should also update the state stored in the SW stack’s TCP socket structure (`tcp_sock`). This is essential because a number of tools obtain the information they report to users from fields in these structures and keeping the structures up to date means these tools can handle offloaded connections without any changes.

Below we discuss how the Chelsio TOM maintains its connection state.

The Chelsio offload module does not use any per-connection data structures of its own to maintain the state of offloaded connections. Such state is substantially more compact than the state needed by the SW TCP so we use the same data structure that SW TCP uses (a `struct tcp_sock`) with some of its fields reinterpreted to satisfy the needs of the TOM. This is convenient because a `tcp_sock` exists anyway (it is created when the socket is created before any offloading decisions are made), so this sharing means we need no additional per-connection memory to support offloading. This implicitly keeps `tcp_sock` up to date as required for the standard TCP reporting tools to work unmodified.

This data structure sharing doesn’t quite work for listening sockets due to the fact that a listening socket that is not bound to a specific interface may need to be offloaded and tracked by several TOEs or ports. In particular, we cannot store TIDs in the shared structure. For this reason the TOM maintains a hash table for each TOE device it manages mapping listening ports to TIDs. This table is typically small as most systems have a very modest number of listening sockets and is used only when a listening socket is closed, a rather infrequent event.

Finally, TOM has three maps to convert the various types of TIDs to sockets. The most important map converts TIDs assigned by the TOE to sockets. It is implemented as an array of socket pointers whose size is determined to accommodate the largest number of offloaded connections possible given the TOE’s TCAM size. The advantage is that this is very scalable, allowing lookups without locking. The downside is that it uses the maximum amount of memory independent of the number of actual connections. The memory requirement isn’t bad though. For 64K connections on a 32-bit platform we’d need 256KB, and the worst case of 1 million connections on a 64-bit platform requires 8MB. This isn’t excessive for a server that can handle such numbers of connections so we opt for performance.

The other two maps are used for PTIDs used by listening sockets and ATIDs used by embryonic active opens. The sizes of both maps are determined by the administrator and are also implemented as arrays. For ATIDs we also impose a limit of half the TCAM TID range, since otherwise active opens could consume all TCAM resources.

6.2. **Interfaces**

TOM implements the APIs used by applications to interface to the network stack. In most cases these are the socket APIs, but a TOM may provide additional APIs for use by applications or upper layer protocols, as well as calls that need to be implemented differently by TOMs but that are not official APIs of the host stack. Whenever appropriate a TOM may
use the existing API implementations of the host stack if their existing behavior satisfies its needs. The remaining APIs can be implemented by the TOM in a way that is most appropriate for it and the underlying offload device.

In Linux most of the relevant APIs are encapsulated in the `struct proto` structure, and a TOM can reimplement as many of these as it needs in a customized way. The most important among them are the send and receive calls (`sendmsg`, `sendpage`, and `recvmsg`), and the ability to reimplement them enables a customized data path that is important for achieving superior performance.

In addition to the APIs in `struct proto` a number of additional calls are provided that are needed to support TOEs but that the existing host stack has not already designated as customizable APIs. This includes calls to set a connection’s keepalive timer (Linux TCP has a hardcoded implementation that is not appropriate for most TOEs), a call to specify the actions performed when data is consumed from a socket’s receive buffer (another Linux hardcoded behavior that is usually inappropriate for TOEs), and a call to return data for offloaded connections through `tcp_diag`.

There are two ways to provide these additional calls. One involves extending the `proto` structure with additional methods and determining if the extra methods are available by checking if a connection is offloaded (a method to check if a connection is offloaded is provided by the offload switch code). The second option adds the additional methods to the socket structure as TOE specific methods. Either way allows us to provide as many additional methods as needed to support a TOM’s extra functionality. The additional methods are provided through function pointers.

Finally, a TOM may implement additional methods to enhance the capabilities of applications or ULPs. As an example of such a method, the Chelsio TOM provides a method `send_skb()` that permits a caller to build a number of packets itself and then add them all to a socket’s send queue in a single call, instead of relying on the traditional `sendmsg` or `sendpage` Linux calls that would require multiple invocations. By making such an API available we get superior performance by a) allowing the caller to build the packets more optimally since it has better knowledge, and b) reducing the number of calls needed to transmit the data compared to the `sendmsg/sendpage` pair.

### 6.3. Neighbor (ARP) Management

The host stack’s ARP code is slightly modified (the changes amount to a handful of lines) to make a callout whenever some ARP entry is modified, whether due to a state change, or due to being associated with a different Ethernet address. The callout invokes a function provided by the offload device layer which checks if the device associated with the ARP entry is offload capable (by checking for `NETIF_F_TCPIP_OFFLOAD` in the device’s features), and if so it invokes the associated `toedev`’s `neigh_update` method to inform the TOM of the neighbor change. It is up to TOM and the underlying driver to perform any SW or HW state changes necessitated by the neighbor change.

The same mechanism applies to IPv6 Neighbor Discovery (ND) changes that are propagated to TOEs capable of IPv6 offloading (the only change being that we would now check for the `NETIF_F_TCPIP6_OFFLOAD` capability).
6.4. Route Management

Similar to ARP changes, when either an existing route changes or a connection is rerouted a function is invoked that propagates the route change to the appropriate TOM by invoking a toedev’s route_update method. It is up to the TOM and the associated driver to react to such changes as appropriate for the underlying HW.

6.5. Path MTU Changes

Path MTU discovery is handled by the host stack, which is responsible for ICMP processing. When a path MTU changes, the update is communicated to the appropriate TOM by calling a toedev’s pmtu_update method. The TOM and the associated offload driver coordinate to update the affected connection(s) as appropriate for the device.

6.6. toedev Fields Set by TOM

When a TCP offload module attaches to a toedev it sets the following fields:

- can_offload – This function determines whether the TOE can offload a new connection by checking for resource availability and the operating state of the offload module. If this succeeds the actual offload may still fail later. This exists to give an early indication of failure before the more expensive switch of protocol operation tables.
- connect – Called by the offload switch once it has decided that an active open should be offloaded. A lot of the connection state is setup by the host stack prior to calling into TOM through this interface. This includes choosing a local port for the connection if necessary to complete the 5-tuple, and routing. The TOM may refuse to offload a connection by returning an error code. The offload switch will then allow the host stack to maintain ownership of the connection and complete the active open.
- receive – This function is called by the offload driver to deliver packets to TOM. Multiple packets can be delivered at once for higher performance.
- neigh_update, route_update, pmtu_update – Called to communicate neighbor, route, or PMTU changes to TOM as described above.
- recv_stray_packet – Called by the offload switch to pass an unexpected packet for a connection to the TOM managing it. The packet may have been misdelivered for a variety of reasons, including routing mishaps, etc. A TOM handles such packets in accordance with the capabilities of the TOE device (e.g., it may discard such packets, or forward them to the device, etc).
- l2opt – This points to TOM’s data structures used for maintaining neighbor information (ARP table, etc).
- l3opt – Points to TOM’s routing state.
- l4opt – Points to TOM’s L4 data structures (TID maps, etc)
- ulp – Points to TOM’s state for ULP support (e.g., page pod state for Chelsio TOEs).

6.7. Listen Offload

If a TOE device can offload listening servers, its associated TOM may subscribe to receive events emitted whenever a listen(2) call is made or when a listening socket closes. Upon receiving such notifications it is up to the TOM to decide whether to offload the listen and how to coordinate with the TOE device to effect this. The notification uses standard Linux notifier chains. Although a TOM may sleep while processing such a notification it is
expected that any sleeping will be short as we allow offloading only one listen at a time for simplicity.

Generating these events is done by providing a function in the offload switch (described below) that is called by the host stack when a listen is started, or when a listening socket closes. The function calls through the notifier chain passing the affected socket and the type of event (start or close) to any TOMs that have subscribed to be notified.

7. The Offload Switch

The offload switch determines which connections or listening endpoints should be offloaded and when. Offloading decisions are guided by offload policies in the form of rules configured by the administrator. The offload policy database as well as the rule evaluation is built upon the Linux netfilter framework, which is also used by Linux’s firewall subsystem. This choice is made because a significant portion of the task’s complexity is storing and manipulating rules and netfilter already solves this problem for firewalls. In addition to deciding whether to offload, the rules may also specify connection-specific parameter values, such as what flavor of TCP congestion control to use or whether to enable receive coalescing.

As provided by Linux the existing netfilter framework is not immediately applicable to offload control because netfilter operates on packets, while offloading operates on connections (usually as sockets). To solve this problem without modifying netfilter the offload switch uses the following approach: when presented with a connection it extracts the 4-tuple information and inserts it into a synthetic TCP segment that comprises just IP and TCP headers; the information from the connection provides the local and remote IP addresses and ports that are inserted into these headers. The resulting packet is sent through netfilter, which applies the offload rules to it. If the packet is not filtered out the connection is accepted for offload. In addition, the rules may specify connection specific parameters that are stored in the packet control block and are then provided to the selected TOM.

Offload requests can happen for a variety of reasons:

- a connection can be considered for offload after predetermined events, e.g., when a socket starts listening, or when an active open is initiated.
- in response to requests from a TOM, often due to events it cannot handle itself, e.g., reception of urgent data or IP fragments. In this case a TOM may invoke a function to terminate the connection.
- in response to requests from other components, e.g., a request from link aggregation to move a connection to another port within the same NIC for load balancing or fail over, routing changes.

Switching a connection is done by modifying the protocol operations of a connection (struct proto) as well as any other methods provided by a TOM from the methods of the source TCP provider to those of the target provider. We discuss how the switch operates for listening sockets, active opens and passive opens separately as its operation is different in each case.
7.1. **Offloading listening sockets**

This entails a small (one line) addition to the host stack listen code. Once the listen is completed in the host stack the socket is passed to the offload switch through a new function. If the offload policies authorize the offload the switch posts an event on a standard Linux notifier. TCP offload modules that offload listens subscribe with the notifier when they are activated. Upon receiving a notification of a new listening socket the offload module creates its private state for tracking the listener and notifies the HW. This scheme allows the SW stack to remain completely unaware of any structures or operations needed to support listen offloading, it is only required to post events when listening sockets come and go.

7.2. **Active open offload**

When the route for the connection is determined we check the capabilities of the egress device to see if it advertises TCP offloading through its `features` field. If it does and the offload policy permits offloading we retrieve the associated toedev through the egress device and check if the TOE can accept the new connection by calling its `can_offload` method. If the TOE agrees the offload switch then invokes the toedev’s `connect` method to complete the offload. If the call fails the switch continues the active open in the host stack.

7.3. **Passive open offload**

Passive open offload works only for offloaded listening sockets and is automatic in that if the associated listening socket is offloaded the new passively open connection will start as offloaded. The associated TOM is responsible for performing the various tests the host stack would normally perform upon receiving a SYN, such as checking routing, firewall rules, and the listen backlog. As many of these tests as possible should be made to maintain as much as possible the behavior of the host stack.

8. **Dealing with Virtual Network Devices**

A number of services, such as link aggregation and VLANs, are provided in Linux through virtual network devices that sit atop physical link-layer devices or other virtual devices. To support such device stacks on top of offload-capable physical devices two changes are made to the virtual devices.

First, as described previously, a link-layer device indicates its offload capabilities by setting appropriate flags in its `net_device features` word, such as `NETIF_F_TCPIP_OFFLOAD` that indicates IPv4 TOE capability. Virtual devices propagate offload capabilities from the devices below them by setting appropriate flags in their own `features` word. The rules used to derive a parent device’s offload capabilities from those of the devices below it (its children) differ depending on the nature of the virtual device. For instance, a VLAN virtual device has only one subordinate device and thus sets its offload features directly from those of its child. A link aggregation device on the other hand in general has several children that may have different offload capabilities. Such a device could use the intersection of its children’s capabilities, so that it would offload if all the subordinate devices could do so, or it could use the union of its children’s capabilities, so that it could offload if any of the children could, or a scheme in between the two extremes.
If a virtual device does support offload, the second change needed is to associate it with a virtual `toedev` that will implement the toedev operations as appropriate for the semantics of the virtual device. These operations typically forward the calls to the children devices, perhaps after some processing of their own. As examples, a VLAN device would forward the calls to its sole child along with VLAN information, while a link aggregation device might select one of its children according to its distribution policies before forwarding the call to it.

We note here that although in Linux VLAN and link aggregation are implemented through virtual devices, in other OSs they may be integrated into the link-layer driver. In this case offloaded VLAN and link aggregation functionality can be implemented into the offload driver.