Сетевой стек Solarflare OpenOnload. В чем и почему он обыгрывает ядро Linux

Константин Ушаков
OKTET Labs.
Content

• Kernel bypass networking
• Onload architecture and motivation behind it
• Safety and security
• Implementation challenges
• What’s faster: examples
Kernel bypass networking

- Getting application closer to the network
Approaches: APIs

- **Socket API** (socket(), send(), poll(), epoll_wait() etc.)
  - OpenOnload

- **Special APIs**
  - Solarflare EF_VI
  - DPDK
  - netmap
  - Infiniband verbs
  - etc.
Architecture [1/5]

- VI (Virtual Interface ; v-nic): minimum set of resources required to send/receive traffic
  - TX
  - RX
  - notification queue (EVQ)

- Filtering
Architecture [2/5]

- Traditional interfaces
Architecture [3/5]

- Application
  - shared state
- Application
  - shared state
- Application
  - shared state

Kernel
- eth0
- eth1

Shared state
Kernel state

NIC Context
- VI
- VI
- VI
- VI
Architecture [4/5]
Architecture [5/5]
Architecture [5/5]

Application

Socket Protocol

Application

Socket Protocol

Application

Socket Protocol

shared state

kernel state

NIC Context

Kernel
Architecture [5/5]
API

• `socket()`, `listen()`, `connect()`, `accept()`, `recv()`, `send()`, `read()`, `write()`, `select()`, `poll()`, `epoll_wait()`, `fcntl()`, `dup()`, `accept4()`, `ioctl()`, `setsockopt()` etc. – full Socket API
  – you don’t know how many functions people use and in which ways...

• Access point of the API is `socket file descriptor`

• **LD_PRELOAD** loads the library (libonload.so)

• libonload.so provides Linux compatible Socket API
API: LD_PRELOAD

Application

Application

libonload.so

libc

shared state

JVM

shared state

Bash 😊

libonload.so

libc

shared state

Kernel

Protocol

shared state

Kernel state

NIC Context

VI VI VI VI

VI VI VI VI
API: recv() / poll()

1) if receive queue not empty -> return data;
2) if notification queue not empty -> handle the event -> return data;
3) <here we have neither data nor event>
4) spin?
   – spin (for some time) in userland waiting for an event
5) go to the kernel ← slow
6) wait for interrupt and “wake up the socket”
7) wake up in userland -> return data
→: copy to user buffer; Zero Copy API get rids of it
API: TCP send()

- copy user data -> packet buffer
- packet buffer is added into socket sendq
  - sendq is in shared state
- send window & congestion window OK
  - can send => send into the VI (NIC)
- otherwise send provoked by event handler (in userland OR kernel)
Why do we need Shared State

- `fork()` : duplicates everything, need to be in sync
- `exec()` : just wipes everything out
- Process can send fd/socket via UNIX domain socket
- Process exits (perhaps in fire): data should be delivered + socket should be shut down
Shared state: fork()
Shared state: fork()
Shared state: exec()
Shared state: exec() – wipes it all
Shared state: exec()

Socket (FD) survives exec() if no CLOEXEC
stat() knows this
Shared state: exec()
Shared state: internals

• What is in it:
  – sockets,
  – packet buffers,
  – VI state,
  – timers (retransmit, keepalive etc.),
  – free resources,
  – configuration,
  – demux table (selects socket).
Shared state: Addressing

• Mapped into multiple processes + kernel
  – pointers are indirect and kernel-managed,
  – sockets and packet buffers are identified by index,
  – other fields identified by offset.

• User-space code can corrupt the state
  – state sharing = trust

• Kernel code should check state is not corrupted by user-space code
Security: Kernel state

- Kernel state = Trusted state
- Pointers = offsets
  - kernel: verified and converted
  - userland: not verified and converted
- Lists: you can loop them even with valid pointers
  - traverse stack with a counter
More security!

- Packet buffers: place where HW writes packets and from which we copy data to the recv() etc. buffers
- NIC maps BufID -> Physical Addr
- You can’t read/write/spoil buffer that is not yours
Stack: basics

- Stack: entity that allows socket communication to the NIC
  - application only entrance point is socket = socket()
- Lifetime: independent of the application
- Tightly connected to the shared state
Stack: <-> processes

- Arbitrary mapping
- Can change over time
Stack: default stack for the process

- Default stack is where socket() creates a socket
- Default can be changed
- Default depends on how we got here
  - fork, exec, settings etc.
- If there is actually no stack -> socket() creates a stack
  - slow and logic avoids this
- close():
  - destroys stack if it’s the last socket
  - does not destroy default process stack
Default stack: socket()
Default stack: fork()
Default stack: socket() in Application#2: default behaviour
Stack for a socket

- **EF_NAME**: just tell the stack name to your process
- **Granular policies:**
  - different users -> different processes
  - different groups -> different processes
  - etc.
- **Move socket between stacks**: in some cases
Stack: locks

- Smart mix of:
  - stack lock
  - socket lock
  - atomics

- Atomics are expensive

- Receive path: HW should be able to queue packets while socket reads them

- Transmit path: sendq + pre-queue mechanism that allows socket to queue packets without taking the stack lock
Onload FD: onloadfs

- Socket is an FD
- Onload socket is also an FD
- `/proc/pid/fd/239 : special onloadfs`
  - similar to socketfs
- It’s an FD, so even **without Onload:**
  - read()
  - write()
  - poll(), epoll_wait(), select()
Onload FD: OS socket

- OS Socket: reserve ports
- No OS FD: don’t waste 2 FDs per socket
Onload FD: OS socket

- OS Socket: reserve ports
- No OS FD: don’t waste 2 FDs per socket
close(): what if OS closes the socket

- close() called via libc is a problem
- FD table in userland → unnoticed close will spoil the table
close(): return properly!

- close() called via libc is a problem
- FD table in userland → unnoticed close will spoil the table
What if I send via non-SF NIC

• 1: socket(, SOCK_DGRAM, ) -> s1
• 2: sendto(s1, ) \(\rightarrow\) accelerated interface
• 3: sendto(s1, ) \(\rightarrow\) non-accelerated interface

• Onload detects that you’re working with non-SF adapter and passes packet in (3) to the kernel

• 1: socket(, SOCK_STREAM, ) -> s1
• 2: bind(INADDR_ANY) + listen()
• 3: s2 = accept(): checks Onload connections and then “Linux” connections

• If s2 is Linux we’ll honor it
Control Plane

- ARP
- Route (no multi-table setup)
- Interface addresses
- ip rule (no source-based routing)
- iptables:
  - limited support,
  - SolarSecure provides improved support, cool statistics and Norse Darklists integration
- Control plane structures are RO for userland
Diff with Linux

- Automatic detection (>15000 testcases)
- Usually diversity is intentional and can be tweaked with env variable
- TCP protocol implementation is a bit different
Acceleration: some examples

• Latency
• Local communication:
  – TCP Loopback,
  – UDP Loopback & UDP Multicast,
  – PIPE
• Nginx
Latency (UDP)

- Linux raw data on 64B packets was: >10ms : terrible
  - Linux improves, but not fast
- Recent Linux has SO_BUSY_POLL:
  - works similar to SPIN mode of Onload
  - modify your application (Java?) or
  - enable it globally (CPU)
  - Onload spin happens in the application
## Latency (UDP)

<table>
<thead>
<tr>
<th>payload</th>
<th>RHEL7</th>
<th>RHEL7+busy_poll</th>
<th>onload</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6428</td>
<td>3940</td>
<td>1653</td>
</tr>
<tr>
<td>2</td>
<td>6432</td>
<td>3947</td>
<td>1652</td>
</tr>
<tr>
<td>4</td>
<td>6405</td>
<td>3921</td>
<td>1651</td>
</tr>
<tr>
<td>8</td>
<td>6856</td>
<td>3925</td>
<td>1653</td>
</tr>
<tr>
<td>16</td>
<td>6419</td>
<td>3954</td>
<td>1653</td>
</tr>
<tr>
<td>32</td>
<td>6413</td>
<td>3940</td>
<td>1681</td>
</tr>
<tr>
<td>64</td>
<td>6484</td>
<td>3997</td>
<td>1707</td>
</tr>
<tr>
<td>128</td>
<td>6602</td>
<td>4127</td>
<td>1823</td>
</tr>
<tr>
<td>256</td>
<td>7046</td>
<td>4616</td>
<td>1974</td>
</tr>
<tr>
<td>512</td>
<td>7138</td>
<td>4699</td>
<td>2146</td>
</tr>
<tr>
<td>1024</td>
<td>7688</td>
<td>5255</td>
<td>2786</td>
</tr>
<tr>
<td>1472</td>
<td></td>
<td></td>
<td>3324</td>
</tr>
</tbody>
</table>
Latency (UDP)

Latency (microseconds/pkt-length)

- linux + busy_poll
- onload
- linux
Loopbacks & PIPE

• TCP Loopback or UDP unicast
  – if sockets share the state, let’s communicate through it

• One stack, but we have helpers that can:
  – move both sockets to either listen()-er or connect()-er socket,
  – create new stack and move both socket to it.

• UDP loopback (multicast):
  – replay to listeners in the same stack
  – replay to listeners from other stacks (and virtual machines) with HW assistance

• PIPE
  – ends in the same stack
Nginx: connection rate

- Maximize number of requests Nginx can handle
- Intel Xeon E5-2620 v3 processors with HT running at 2.4 GHz, and 64 GB DDR4 RAM running at 1867 MHz
- SO_REUSEPORT is set
- Static content; 10000 in length stored on RAM filesystem

10Gbps Connections

![Graph showing 10Gbps Connections with different core counts and Nginx versions]
Nginx: connection rate

• How any of what I’ve told helps connection rate?
Nginx: reasons

- Reduced cost of networking calls
- Onload stack per worker means that almost nothing is shared: no lock contention and cache bouncing
- `epoll_wait()` scaling improved (latest release only!): $O(1)$
Nginx: socket caching

- We’re not going into the kernel
- listen()
- accept() → s1
- accept() → s2
- close(s1)
- → SYN received:
  - take s1
  - no need to go to the kernel!
Nginx: response bandwidth (10G)
Nginx: response bandwidth (40G)

40 Gbps Connections

Connections p/s

0 100,000 200,000 300,000 400,000 500,000

1 2 3 4 5 6 8 10 12

Number of Cores

SFN7142Q-OpenOnload
SFN7142Q-Kernel

40 Gbps Response Bandwidth

Gbps

0 10 20 30 40

1 2 3 4 5 6 8 10 12

Number of Cores

SFN7142Q-OpenOnload
SFN7142Q-Kernel
Nginx: VOD

- 1Mbps stream
- Watermark: 1MB buffer; next request if buffer is at 50%
- 20000 IP addresses
- WRK testbench
Nginx: VOD 10G

Figure 2. Maximum simultaneous connections vs. number of cores at 2x10GbE.
Nginx: VOD 40G

Nginx Performance 40GbE

Figure 3. Maximum simultaneous connections vs. number of cores at 40GbE.
Performance/behaviour tuning

• Very granular:
  – Per-application
  – Per-stack
  – Per-socket (setsockopt() implementation)

• Non Socket API functions:
  – Zero Copy
  – Ordered epoll (RX packets from many sockets in wire order)
Thank you.

Konstantin.Ushakov@oktetlabs.ru