FreeBSD Implementation of PASTE

Michio Honda and Giuseppe Lettieri

1 Overview

PASTE is a network stack architecture that enables handling high I/O rates and efficient persistent memory integration. FreeBSD implementation of PASTE is an extension to `netmap(4)` in two ways. First, it integrates `netmap(4)` with the feature-rich kernel TCP/IP implementation. The netmap API enables a fast datapath that batches system calls and NIC I/O operations, and the use of the kernel TCP/IP implementation allows applications to use familiar socket APIs for control. Second, PASTE provides applications abstractions where they can move data between networks and persistent memory without data copy. This is an important property to efficiently support persistent memory which is two-three orders of magnitude faster than solid-state or spinning disks.

```
1  main()
2      fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
3      bind(fd, INADDR_ANY);
4      listen(fd, 30);
5      nmd = nm_open(stack:0);
6      ioctl(nmd->fd,, stack:em0);
7      s = socket();bind(s);listen(s);
8      int fds[2] = {nmd, s};
9      for (;;) {
10         poll(fds, 2,);
11         if (fds[1] & POLLIN)
12             ioctl(nmd, NIOCCONFIG, accept(fds[1]));
13         if (fds[0] & POLLIN) {
14             for (slot in nmd->rxring) {
15                 int fd = slot->fd;
16                 char *p = NETMAP_BUF(slot) + slot->offset;
17         }
18     }
19 }
```

Figure 1: Pseudo code of PASTE application

2 Kernel TCP/IP support

Support for kernel TCP/IP relies on various in-kernel socket APIs, including `sosend(9)`, `sorecv(9)`, `soupcall_set(9)` and `soupcall_clear(9)`. No kernel modification outside netmap is required. We already merged `sodtor_set(9)` to the kernel whose callback is fired on the socket close event. To understand how the netmap and in-kernel socket APIs interact with each other, we begin by explaining the APIs.

Figure 1 shows pseudo code of a TCP server application with PASTE. The server application initiates the socket as usual using `socket(2)`, `bind(2)` and `listen(2)` (line 2–4). It also open a netmap descriptor whose port type is "stack" (line 5). This port type is analogous to a ephemeral `vale(4)` port. To move data or packets between the stack port and a physical NIC, the application associates a NIC with the stack port (line 6). This process internally instantiates a bridge and attaches a given NIC port to it, which is much like attaching a NIC to a VALE switch. When the application dies or explicitly closes the stack port, the NIC port is also released.

The application then monitor two file descriptors using `poll(2)`: the TCP socket and netmap descriptor (line 8 and 10). Arrival of a new TCP connection is indicated by `POLLIN` event of the listen socket (line 11). The application `accept(2)`s this connection, and register the new socket to the stack port (line 12). After that, the application can send and receive data on this socket using the netmap API, as explained next.

`poll()` backend of the netmap descriptor triggers receive packet I/O of the NIC port associated with the stack port, followed by processing received packets in the TCP/IP stack and moves the buffers with in-order TCP segments to the stack port RX ring. The `poll()` also triggers TX ring processing of the stack port, having the TCP/IP stack process application data and the NIC transmit packetized data.

The application consumes data on the RX ring of the stack port (line 14–16). Each ring slot contains two metadata in addition to `len` used by the regular netmap applications: `fd` that indicates the file descriptor to which data belong and `offset` that indicates the application data offset or the length of the TCP/IP/Ether header. The application can put response data in a TX ring slot with supplying the `fd` and `offset` fields. The latter can be taken from the offset of a RX ring slot, because usually the header length is the same (e.g., minimum headers length plus TCP Timestamp).

We now describe what happens on packets received by the NIC in more detail. For each packet, netmap allocates an `mbuf` whose `m_ext.ext_buf` points a netmap packet buffer. It then calls `ifp->if_input()`. To intercept the data that are *ready* to be passed to the application, such as an in-order TCP segment, the kernel has set a callback using `soupcall_set()` when registering the socket to the

dtor.	upcall	mbuf status	Example
✓	✓	app. readable	In-order segments
✓	✗	consumed	Ack segments
✗	✗	held by stack	Out-of-order segments

Table 1: Relationship between `mbuf` status and occurance of `mbuf` destructor and soupcall

stack port (Figure 1 line 12). Further, we set a destructor callback to the `mbuf` before `if_input()`, which *marks* the underlying netmap buffer on `mbuf` consumption. Using these mechanisms together, we can identify what has happened to the `mbuf`, as shown in Table 1.

Therefore, when returning from `if_input()`, the kernel can identify whether the packet can be passed to the application (app. readable in the table) or not. The netmap context processes all the receiving packets and sets the ready buffers to the stack port RX ring. The other packet buffers are either discarded (will be recycled) or held by the kernel (swapped out of the NIC RX ring).

Connections terminated are indicated by a zero-length buffers, similar to zero-length `read()`. Unlike Linux, FreeBSD does not attach `mbuf` that contains a FIN TCP segment without data to the receive queue. Therefore, the socket upcalls triggered by such an `mbuf` observe nothing at the queue. However, when returning from `if_input()`, we would like to know `mbuf` or its underlying buffer that has caused socket closure,

This prevents the netmap from identifying whether the `mbuf` has been just consumed (e.g., pure ack packets that do not need to deliver the corresponding buffers to the client) or has triggered important event (e.g., FIN packets that need to deliver the corresponding buffers to the client to indicate zero-length buffer. Thus, we leverage per-CPU binary flag. They are always cleared before `if_input()` and the socket upcall with no available buffer sets them. After returning from `if_input()`, netmap makes a decision to deliver the buffer to the client based on this flag.

Transmissions are processed in a similar way to receiving packets. The `poll` backend of the stack port `sosend()`s a buffer pointed by each slot of the TX ring. Netmap intercepts `if_transmit()`, and identifies `mbuf`s that come from the netmap `txsync` context (i.e., from that `sosend()`) and those that come from the other contexts (e.g., ARP and pure ack packets). For the former, netmap copies packet headers in the `mbuf` to the headroom of the associated netmap buffer (provided by the clinet with offset field of the slot) if the length of the headroom and total length of the packet headers match. Otherwise it re-constructs the packet in the netmap buffer causing payload data copy. For the latter, netmap puts the packet in the host ring of the NIC port. netmap identifies `mbuf`'s context without relying on `mbuf` destructor, because `mbuf` is allocated inside `sosend()`.

3 Persistent Memory Support

Netmap uses regions of memory, shared between the kernel and userspace applications, to contain both its abstract data structures (netmap rings) and packet buffers. Packet buffers are also shared with networking hardware, allowing for zero-copy TX and RX operations.

Several memory regions may be defined and netmap applications may bind many ports to the same region (to support, e.g, zero-copy bridging) or to different regions (e.g., to enforce isolation among containers/VMs attached to distinct ports of a VALE switch).

In the legacy implementation, the memory supporting each region is allocated by the netmap module inside the kernel (in small clusters, using `conrigmalloc()`) and later `mmap()`ed by applications into their own address space. Recently we have also introduced the possibility of reverting this process: netmap applications may do a generic `mmap()` and pass the resulting address and size to netmap. Netmap will then use `vm_map_*()` to obtain the underlying pages and use them to support the shared region. In this way, PM can be easily supported: applications should allocate a file in PM-backed file system, mmap() it and then tell netmap to use the corresponding physical memory. Using Linux's DAX-like feature that is expected to be implemented in FreeBSD, this will also bypass the buffer cache, giving a direct path from network cards to PM. Importantly, given the same file, netmap will deterministically allocate the same data structures and buffers in it, thus retrieving the persistent state after a crash/reboot.

4 Performance

We benchmark performance of PASTE using a pair of the machines connected back-to-back. The server machine is equipped with Intel Xeon Silver 4110 CPU clocked at 2.1 Ghz, whereas the client is equipped with Xeon E5-2690v4 clocked at 2.60 GHz. Figure2 shows throughput (bars) and latency (numbers on top of bars) with and without PASTE. The client runs `wrk` HTTP benchmark tool to generate RPC-like workload in which each request retrieves a 64B message over one or more persistent TCP connections. In a single connection case, the performance does not differ much, but we observe much higher throughput and lower latency in the presence of concurrent TCP connections.

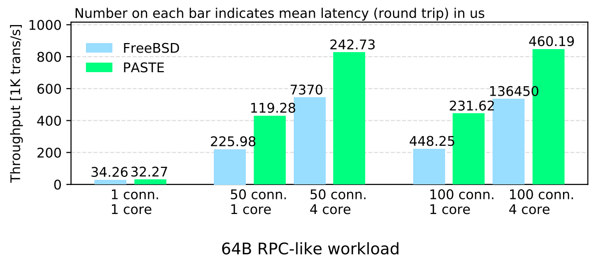

Figure 2: Throughput over concurrent TCP connections.

5 Ongoing Work

As shown in the graph, multi-core scalability is not high with PASTE in FreeBSD. Since the trend is same in the baseline, the reason seems to stem from FreeBSD TCP/IP and/or socket API implementation. Further, overall performance is lower than Linux at both baseline and PASTE (Linux performance can be found in the link at the end of this paper). We are currently working to address these issues.

6 Conclusion and Availability

PASTE is a network stack that integrates NVMM for efficient networked storage systems; [1] contains many more details and results. PASTE supports Linux (4.6 and higher) and FreeBSD (13.0-CURRENT) without kernel modifications. It is under active development; see https://micchie.github.io/paste/.

References

[1] M. Honda, G. Lettieri, L. Eggert, and D. Santry. "PASTE: A Network Programming Interface for Non-Volatile Main Memory". *Proc. USENIX NSDI*. 2018.