OpenInfra Community
Days Korea

NVMe/TCP implementation on F-stack

Shinik Park
ECE, Seoul National University

Network Becomes The Bottleneck of NVMeoF

SATA with AHCI
(Advanced Host Controller Interface)

	SATA I	SATA II	SATA III
Max Bandwidth	1.5 Gbps	3 Gbps	6 Gbps

PCIe with NVMe
(Non-Volatile Memory Express)

PCIe 3.0 x4	PCIe 4.0 x4	PCIe 5.0 x4
32 Gbps	**64 Gbps**	**128 Gbps**

Fortunately, DAS (Directed Attached Storage) could be a feasible option at the datacenter.
However, DAS has a problem with low scalability.

NVMeoF (NVMe over Fabric) is becoming a trend which enables end nodes to access remote NVMe storage pool.

with combination of 100G Ethernet card (e.g., Intel E810, uses PCIe 4.0 x16)

NVMeoF: Techniques to access remote NVMe Storage

- NVMeoF could be supported by multiple options.
  - FC (Fibre Channel)
  - Infiniband
  - iWARP
  - RoCE (RDMA over Converged Ethernet)

Pros: Low latency, high bandwidth

Cons: Custom host bus adapters and drivers are required, difficult to deploy, maintain & cost inefficient

- TCP

NVMe/TCP (NVMe over TCP) is compatible with standard ethernet adapters.
→ Cost effective & Easy to deploy and maintain!

Possible Design Strategy for NVMe/TCP

□ Kernel level TCP implementation (NVMe/TCP)

- Directly modifying the raw kernel source code (i.e. kernel module implementation)

→ Due to the hard deployment, it would not be handled.

□ User-level implementation (NVMe/TCP)

① DPDK (Data Plane Development Kit) based implementation (i.e. mTCP , F-stack)

- Frameworks that implements TCP-stack into user-level.

② eBPF (extended Berkeley Packet Filter) based implementation

- Pushing packet processing code black into kernel hooking point.

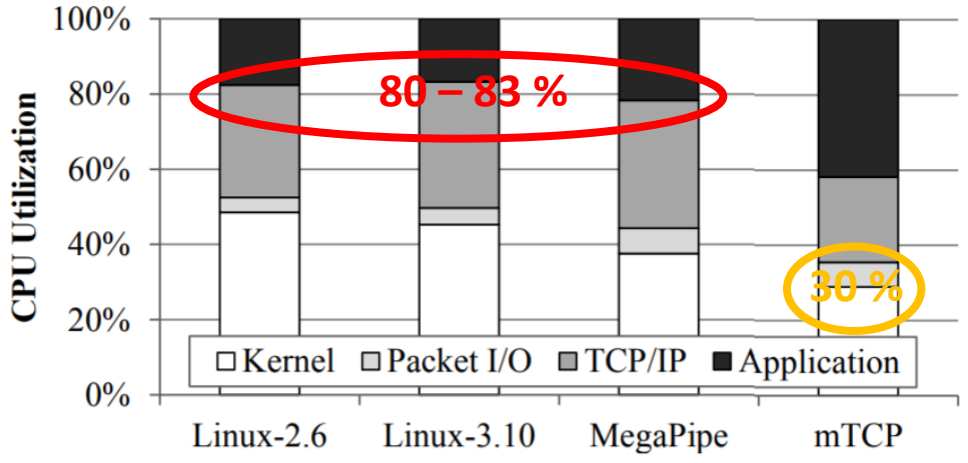
User-level TCP is cost effective option to support high performance NVMe/TCP.

User-level TCP Can Mitigate Kernel Overhead

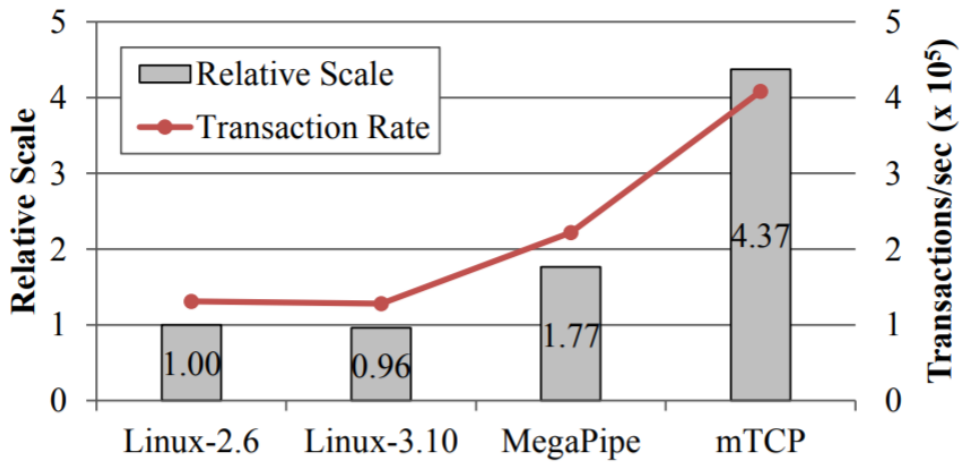
- mTCP [NSDI `14] conducts experiment to see Linux kernel overhead.

Experiment setup

- 8-core Intel Xeon CPU (2.9 GHz, E5-2690) / 32 GB memory
- 10 Gbps NIC (Intel 82599 chipsets)
- *Lighttpd* v1.4.32 web server
- Multiple clients repeatedly download a 64 KB file per connection.



[CPU usage breakdown]

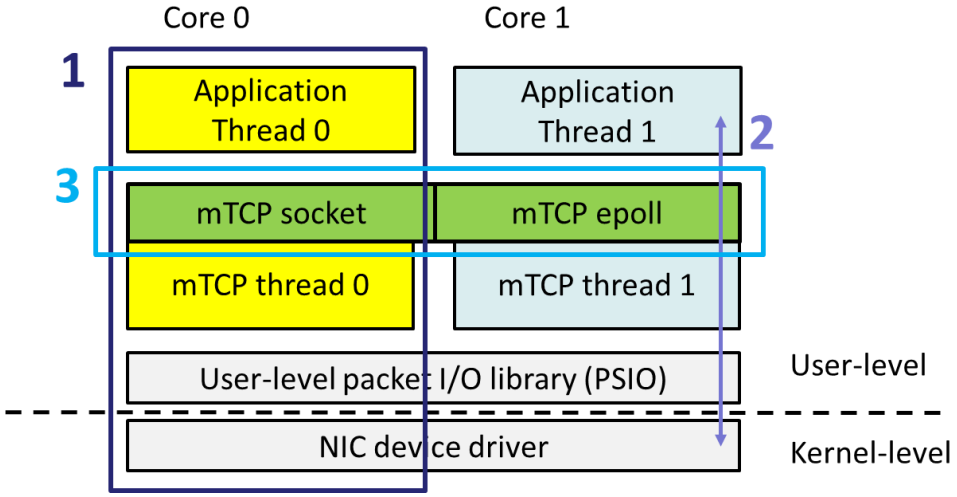


[Relative scale of # transactions processed per CPU cycle in the kernel]

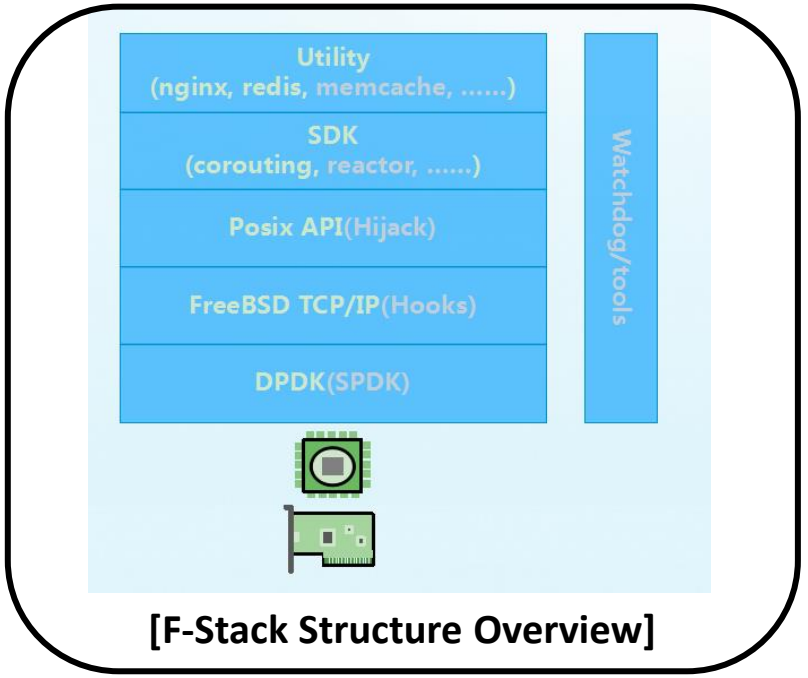
mTCP reduce CPU utilization of kernel from 80 % to 30 %.
→ mTCP can 4.3 times more effectively than linux.

User-level TCP Varients

- mTCP [NSDI `14]: One of the famous user-level TCP implementation. mTCP is basically framework for research so that has no compatibility to mature application.
- F-stack: Open source network framework developed by Tencent. F-stack include an user space TCP/IP stack(port FreeBSD 11.0 stable). It support Nginx, Redis and other mature applications.



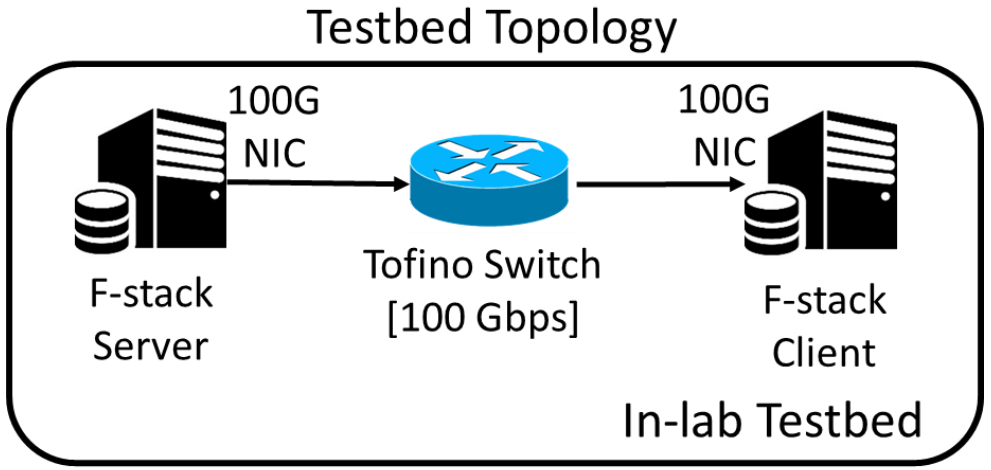
[mTCP Structure Overview]



[F-Stack Structure Overview]

Testbed Setup: Kernel TCP Stack Performance in 100G Environment

- Simple one-hop testbed is organized to test the performance of F-stack and legacy TCP.



- First, we test **default kernel TCP** to check the effect of kernel overhead.
  - Simple data transmission from server to client by *iperf* tool

→ Records around 30-35 Gbps when NIC and Switch can support up to 100 Gbps.

```
intel@intel-S2600WFT:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  4] local 10.10.10.1 port 5001 connected with 10.10.10.2 port 35676
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  37.4 GBytes  32.1 Gbits/sec
```

[Snapshot of server-side log]

```
intel@intel-S2600WFT:~$ iperf -c 10.10.10.1
------------------------------------------------------------
Client connecting to 10.10.10.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.10.10.2 port 35676 connected with 10.10.10.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  37.4 GBytes  32.1 Gbits/sec
```

[Snapshot of client-side log]

DPDK Speed Test with *testpmd* in 100G Environment

- *testpmd*: TestPMD is the reference application distributed with the DPDK.
  - Can measure performance by forwarding packets between Ethernet ports on a network interface.
- Results
  - ~**98 Gbps** is recorded → DPDK itself can fully support 100G hardware with appropriate parameters!

```
./dpdk-testpmd \
  -l 0-1 \
  -n 4 \
  -a 18:00.1 \
  -- --port-topology=chained \
  --nb-cores 1 \
  --forward-mode=txonly \
  --eth-peer=0,40:A6:B7:19:00:01 \
  --stats-period 1 \
  --disable-crc-strip \
  --burst=64 \
  --mbuf-size=4096 \
  --cmdline-file=/home/intel/f-stack/dpdk/build/app/cmdline-testpmd.txt
```

```
Port statistics ====================================
  ######################## NIC statistics for port 0  ########################
  RX-packets: 0          RX-missed: 0          RX-bytes:  0
  RX-errors: 0
  RX-nombuf:  0
  TX-packets: 73824511   TX-errors: 0          TX-bytes:  110736755972

  Throughput (since last show)
  Rx-pps:            0          Rx-bps:            0
  Tx-pps:      8202894          Tx-bps:  98434742024
  ############################################################################
```

[Server-side *testpmd* setup and statistics]

```
./dpdk-testpmd \
  -l 0-1 \
  -n 4 \
  -a 18:00.1 \
  --vdev="net_vdev_netvsc0,iface=enp24s0f1" \
  -- --port-topology=chained \
  --nb-cores 1 \
  --forward-mode=rxonly \
  --eth-peer=0,40:A6:B7:18:FD:A1 \
  --stats-period 1
```

```
Port statistics ====================================
  ######################## NIC statistics for port 0  ########################
  RX-packets: 65621411   RX-missed: 5518       RX-bytes:  98440390492
  RX-errors: 0
  RX-nombuf:  0
  TX-packets: 0          TX-errors: 0          TX-bytes:  0

  Throughput (since last show)
  Rx-pps:      8202807          Rx-bps:  98433687864
  Tx-pps:            0          Tx-bps:            0
  ############################################################################
```

[Client-side *testpmd* setup and statistics]

Does F-stack can fully utilize the great performance of DPDK?

Performance Test Tool for F-stack

- Performance comparison with kernel level TCP-stack
  - SpeedTest source code should be implemented on F-stack.
  - Since F-stack does not provide iperf-like application.

- Implementation of SpeedTest application
  - Done by using *kqueue.* (event driven I/O handling system call, similar with *epoll* in Linux)
  - Send Large file from F-stack server to F-stack Client to evaluate maximum throughput under the given environment.
  - Goal:

Max throughput comparison & improving **F-stack performance**

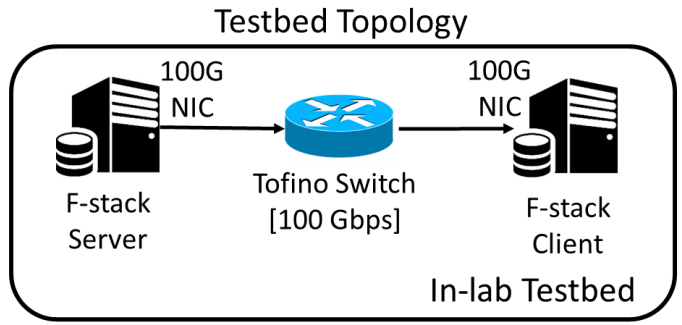
[Core code block of SpeedTest]

```
        } else if (event.filter == EVFILT_READ) {
                char buf[256];
                size_t readlen = ff_read(clientfd, buf, sizeof(buf));
                printf("readlen: %d\n", (int)readlen);
                if( readlen > 0 ){
                    for(j=0; j < send_once; j++){
                            ff_write(clientfd, dummy_bytes, packet_length);
                            pkt_cnt++;
                            printf("%d packet is sent.\n",pkt_cnt);
                    }
                }

                EV_SET(&kevSet, event.ident, EVFILT_READ, EV_DELETE, 0, 0, NULL);
                if( ff_kevent(kq, &kevSet, 1, NULL, 0, NULL) != 0 ){
                        printf("ff_kevent failed, errno:%d %s\n", errno, strerror(errno));
                }

                EV_SET(&kevSet, event.ident, EVFILT_WRITE, EV_ADD, 0, MAX_EVENTS, NULL);
                if( ff_kevent(kq, &kevSet, 1, NULL, 0, NULL) != 0 ){
                        printf("ff_kevent failed, errno:%d %s\n", errno, strerror(errno));
                }

        } else if (event.filter == EVFILT_WRITE) {
                for(j=0; j < send_once; j++){
                        ff_write(clientfd, dummy_bytes, packet_length);
                        pkt_cnt++;
                        printf("%d packet is sent.\n",pkt_cnt);
                }

                if( pkt_cnt >= num_packets ){
                        EV_SET(&kevSet, event.ident, EVFILT_WRITE, EV_DELETE, 0, 0, NULL);
                        ff_kevent(kq, &kevSet, 1, NULL, 0, NULL);
                        ff_close(event.ident);

                        break;
                }

        } else {
                printf("unknown event: %8.8X\n", event.flags);
        }
    }
}
```

Poor Performance of F-stack in 100G Environment

- On the same one hop 100G testbed, **default F-stack**

```
Socket 2048 connected
---------- SpeedTest Result ----------
Goodput: 1351.981 Mbps, Total Bytes: 100052700, Total elapsed time: 0.592036
-------------------------------------------------------------
```

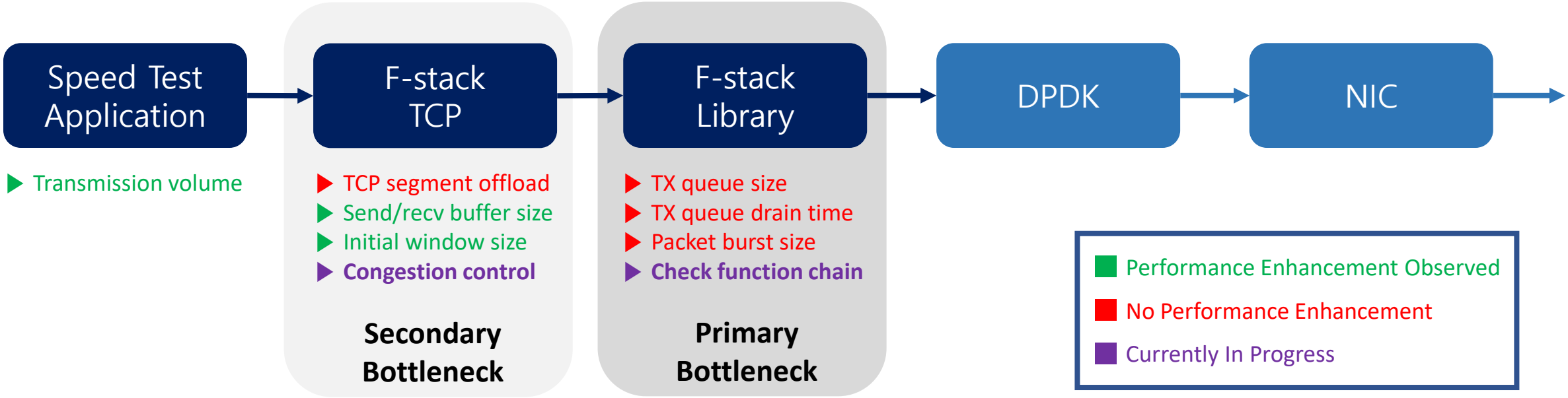


  - Records only ~**1 Gbps**… Poor performance compared to DPDK performance test.

→ This implies that performance bottleneck placed between DPDK and user-level TCP stack!

Where could be the bottleneck in F-stack…?

F-stack Break Down for Finding the Bottleneck



□ **3X Performance enhancement** compared to the default F-stack.

```
Socket 2048 connected
---------------------------- SpeedTest Result ----------------------------
Goodput: 4433.560 Mbps, Total Bytes: 300017360, Total elapsed time: 0.541357
--------------------------------------------------------------------------
```

① Main bottleneck is come from F-stack library to use DPDK functions.

② Development of congestion control algorithm for supporting NVMe/TCP

Congestion Control: Available Options in F-stack

- F-stack supports multiple congestion control algorithms.
  - I picked up some of algorithms that which could support NVMe/TCP.
    - TCP CUBIC [SIGOPS `08]
      - Loss-based algorithms used by default TCP of Linux(`06), MacOS(`14), Windows(`17).
    - DCTCP [SIGCOMM `10]
      - ECN (Explicit Congestion Notification)-based algorithms to support high bandwidth datacenter networks.
    - BBR [Queue `16]
      - Model-based congestion control developed by Google.
  - The performance of F-stack is highly dependent on the type of congestion control.
    ① Limitations of these methods to make hard to support NVMe/TCP with the best performance.
    ② Finding a suitable congestion control candidates to support NVMe/TCP.

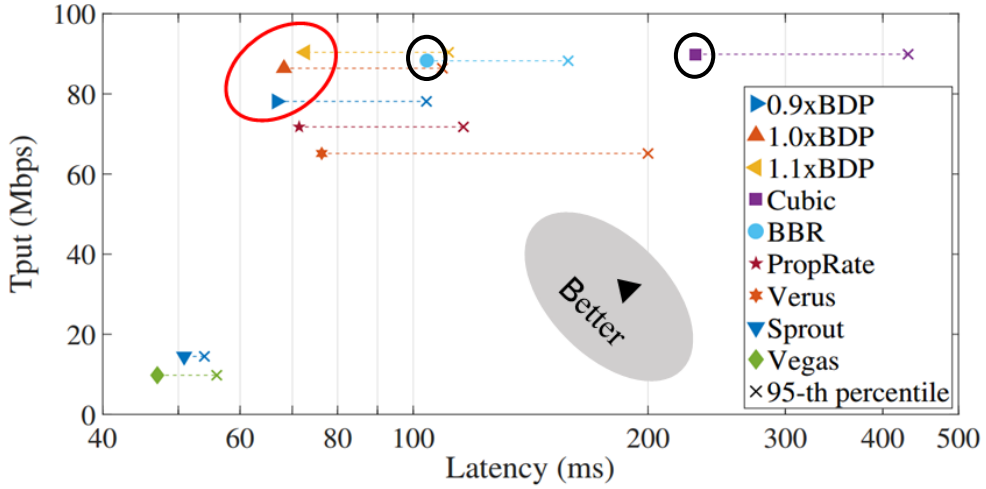
Congestion Control: Limitation of introduced CC

- The problem of TCP CUBIC and DCTCP which is implemented in F-stack
  - These algorithm can only check maximum available bandwidth by “Max probing”

  → Max probing implies that these algorithms **MUST accompany queueing** at the bottleneck link.

- The problem of BBR
  - Model-based congestion control algorithm has its limitation on slow adaptation.

  → Because it can respond to the network change **after satisfying the state transition condition**.



Mean and 95-th percentile RTT against the avg. throughput over a real network.

In actual performance evaluation, The performance of these CCs are far from BDP.

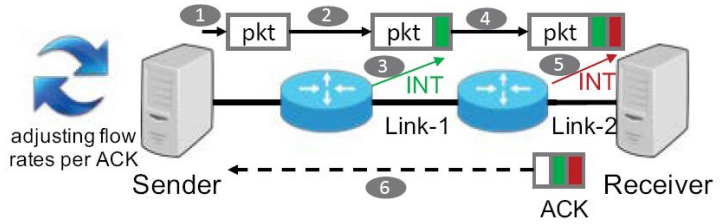
We decide to bridge the gap by using the core idea from HPCC [SIGCOMM `19].

HPCC: Direct Access to Queue Information via INT

Goal: To directly access the queue-related information of the bottleneck switch at the end node.

Key Idea: Using In-Network Telemetry (INT), delivering queue information to control cwnd and sending rate

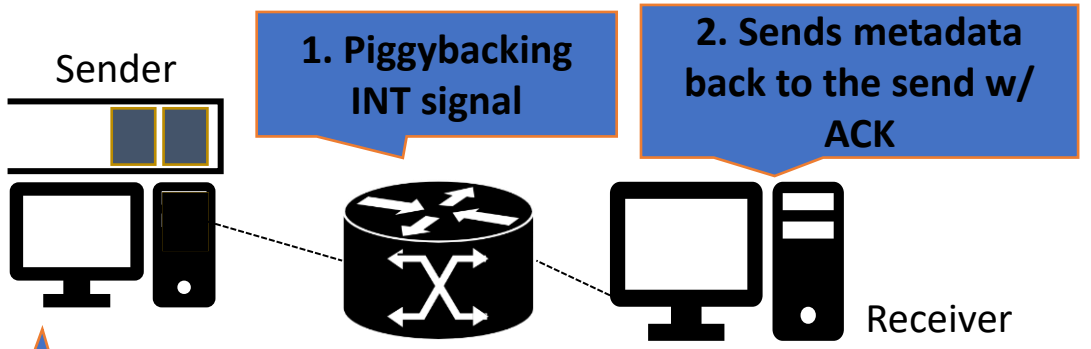
INT (In-Network Telemetry) Overview



INT Overhead (42 bytes for 5 hops)

nHop (4 bits)	pathID (12 bits)	1st Hop(64 bits)				2nd Hop (64 bits)
		B	TS	txBytes	qLen	

- **B**: The type of speed of the egress port
- **TS**: The timestamp when the packet is emitted from its egress port
- **txBytes**: the accumulative total bytes sent from the egress port
- **qLen**: the current queue length of the egress port



3. Control sending rate and inflight packets

1. Initialize sending window

$$W_{init} = B_{NIC} \times T$$ **T**: RTT; **BNIC**: NIC Bandwidth

2. Pace packet sending rate based on window

$$R = \frac{W}{T}$$ **T**: RTT; **W**: sending window

3. Estimate inflight bytes and update window

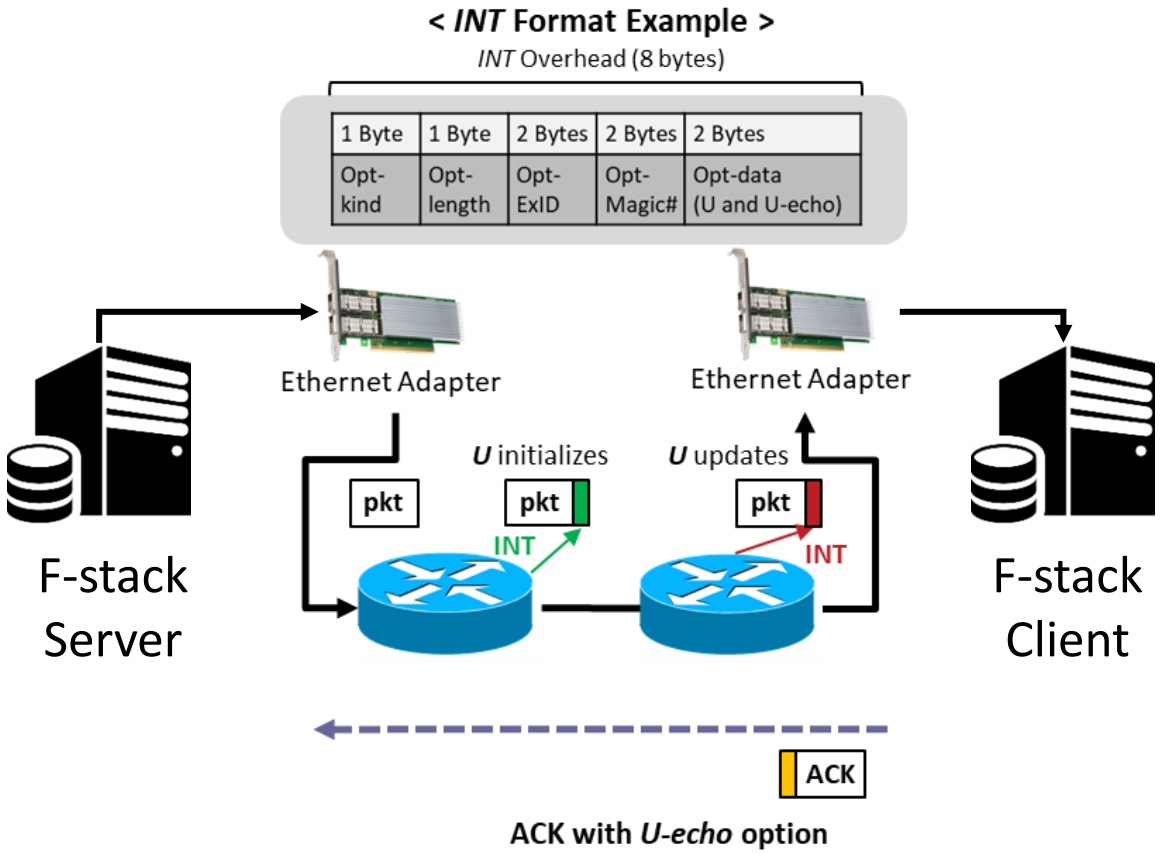
$$(1)\ U_j = \frac{qlen_j + txRate_j \times T}{B_j \times T} \qquad (2)\ W_i = \frac{W_i}{max_j(U_j)/\eta} + W_{AI}$$

(Prototype) NVMe CC: HPCC with less overhead INT

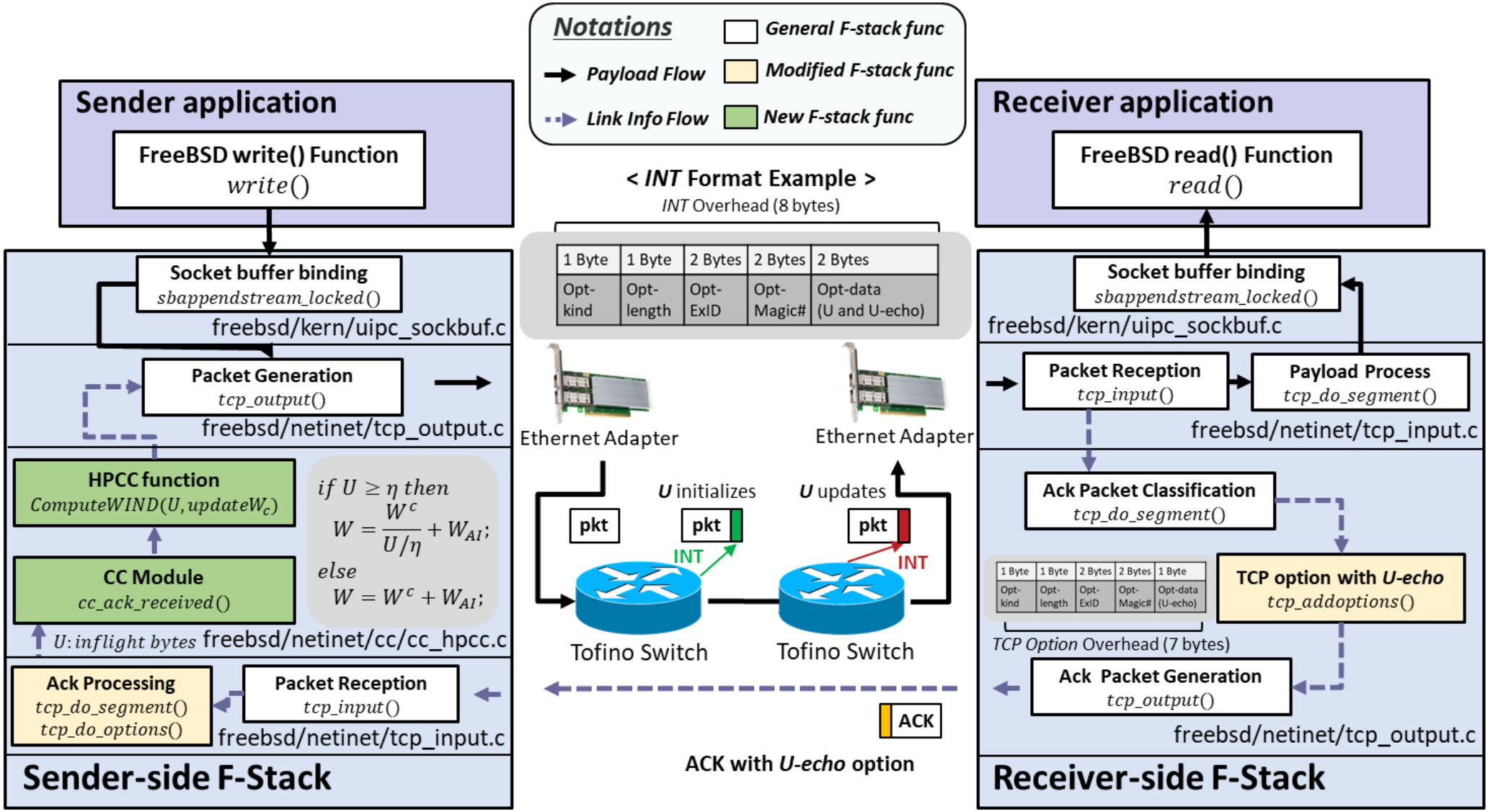
- NVMe CC inherits the core idea of HPCC.
  - By using INT at the programmable switch.

- However, prototype of NVMe CC has multiple advantages with HPCC.
  - ① U calculation is done at the switch side.
  - ② INT overhead becomes 7 or 8 bytes. (HPCC: $2 + 8 \times n$ bytes, n: # of hops)

- Prototype implementation is now in progress.



NVMe CC Prototype Implementation on F-stack



TCP Header Option Implementation: addoptions

- HPCC Option : Option length [server: 8 bytes, client: 7 bytes]

1 Byte	1 Byte	2 Bytes	2 Bytes	2 Bytes
Opt-kind	Opt-length	Opt-ExID	Opt-Magic#	Opt-data (U and U-echo)

[Server-side option field reservation]

1 Byte	1 Byte	2 Bytes	2 Bytes	1 Byte
Opt-kind	Opt-length	Opt-ExID	Opt-Magic#	Opt-data (U-echo)

[Client-side U-echo relaying]

- In freebsd/netinet/tcp_output.c, **tcp_addoptions()** modification.
  - with variable definition at **tcpopt structure** in freebsd/netinet/tcp_var.h

```
case TOF_INT:
        /* Receiver-side U-echo */
        /* U-echo option generation (7 bytes) in ACK packet at the receiver */
        /* To do: Have to relay the data from the data packet to the U-echo field! */
        if( to->uflags != 0 ) {
                while (optlen % 4) {
                        optlen += TCPOLEN_NOP;
                        *optp++ = TCPOPT_NOP;
                }

                optlen +=  TCPOLEN_RINT;
                *optp++ = TCPOPT_RINT;

                *optp++ = TCPOLEN_RINT;

                to->to_ExID = htons(to->to_ExID);
                bcopy((u_char *)&to->to_ExID, optp, sizeof(to->to_ExID));
                optp += sizeof(to->to_ExID);

                to->to_magicnum = htons(to->to_magicnum);
                bcopy((u_char *)&to->to_magicnum, optp, sizeof(to->to_magicnum));
                optp += sizeof(to->to_magicnum);

                /* U-echo with received U value */
                *optp++ = to->to_U;
        }
```

```
        /* Option field reservation in data packet at the sender.
         * Sender returns received U as the U-echo to the switch */
        else {
                while (optlen % 4) {
                        optlen += TCPOLEN_NOP;
                        *optp++ = TCPOPT_NOP;
                }

                optlen += TCPOLEN_SINT;
                *optp++ = TCPOPT_SINT;

                /* Field Reservation */
                *optp++ = TCPOLEN_SINT;

                to->to_ExID = 2;
                to->to_ExID = htons(to->to_ExID);
                bcopy((u_char *)&to->to_ExID, optp, sizeof(to->to_ExID));
                optp += sizeof(to->to_ExID);

                to->to_magicnum = 1;
                to->to_magicnum = htons(to->to_magicnum);
                bcopy((u_char *)&to->to_magicnum, optp, sizeof(to->to_magicnum));
                optp += sizeof(to->to_magicnum);

                /* U value reservation with 0 value */
                *optp++ = 0;

                /* Actual Uecho value return to the switch */
                to->to_Uecho = to->to_U;
                *optp++ = to->to_Uecho;
        }

        break;
```

TCP Header Option Implementation: doptions

- Packet Reception : Option length [server: 7 bytes, client: 8 bytes]
  - Server receives 7 bytes option from **ACK** / Client receives 8 bytes option from **the switch.**

1 Byte	1 Byte	2 Bytes	2 Bytes	1 Byte
Opt-kind	Opt-length	Opt-ExID	Opt-Magic#	Opt-data (U-echo)

[Server-side option reception from ack]

1 Byte	1 Byte	2 Bytes	2 Bytes	2 Bytes
Opt-kind	Opt-length	Opt-ExID	Opt-Magic#	Opt-data (U and U-echo)

[Client-side option reception from the switch]

- In freebsd/netinet/tcp_input.c, **tcp_dooptions()** modification.
  - with variable definition at **tcpopt structure** in freebsd/netinet/tcp_var.h

```
/* Sender interpret option header field (7 bytes) with this case  */
case TCPOPT_RINT:
        if (optlen != TCPOLEN_RINT)
                continue;
        to->to_flags |= TOF_INT;
        bcopy((char *)cp + 2, (char *)&to->to_ExID, sizeof(to->to_ExID));
        to->to_ExID = ntohs(to->to_ExID);

        bcopy((char *)cp + 4, (char *)&to->to_magicnum, sizeof(to->to_magicnum));
        to->to_magicnum = ntohs(to->to_magicnum);

        bcopy((char *)cp + 5, (char *)&to->to_U, sizeof(to->to_U));
        break;
```

```
/* Receiver interpret option header field (8 bytes) with this case  */
case TCPOPT_SINT:
        if (optlen != TCPOLEN_SINT)
                continue;

        to->to_flags |= TOF_INT;
        bcopy((char *)cp + 2, (char *)&to->to_ExID, sizeof(to->to_ExID));
        to->to_ExID = ntohs(to->to_ExID);

        bcopy((char *)cp + 4, (char *)&to->to_magicnum, sizeof(to->to_magicnum));
        to->to_magicnum = ntohs(to->to_magicnum);

        bcopy((char *)cp + 5, (char *)&to->to_U, sizeof(to->to_U));
        bcopy((char *)cp + 6, (char *)&to->to_Uecho, sizeof(to->to_Uecho));
        break;
```

  - Server have to save U-echo **from the ack** and relay **to the HPCC algorithm.**
  - Client have to save U value **from the switch** and feedback to the server **through the ACK.**

NVMe CC Core Algorithm Implementation

- HPCC module implementation is finished (Verified by unittest)
  - By relaying U-echo from ack, implements whole HPCC algorithm on the f-stack.
  - $cc\_ack\_received()$ roles the main control loop for the HPCC algorithm.

[HPCC algorithm from the paper]

procedure NEWACK(ack)
  if $ack.seq > lastUpdateSeq$ **then**
    W = COMPUTEWIND(MEASUREINFLIGHT(ack), $True$);
    $lastUpdateSeq = snd\_nxt$;
  else
    W = COMPUTEWIND(MEASUREINFLIGHT(ack), $False$);
  $R = \frac{W}{T}$; $L = ack.L$;

function COMPUTEWIND(U, $updateWc$)
  if $U >= \eta$ or $incStage >= maxStage$ **then**
    $W = \frac{W^c}{U/\eta} + W_{AI}$;
    if $updateWc$ **then**
      $incStage = 0$; $W^c = W$;
  else
    $W = W^c + W_{AI}$;
    if $updateWc$ **then**
      $incStage++$; $W^c = W$;
  return W;

[Core code block of HPCC algorithm]

```
static void
hpcc_ack_received(struct cc_var *ccv, uint16_t type)
{
        struct hpcc *hpcc_data;

        hpcc_data = ccv->cc_data;

        if( type == CC_ACK ) {
                if( ccv->curack > hpcc_data->lastUpdate_seq ) {
                        hpcc_compute_wind(CCV(ccv, downlink_U), true);
                        hpcc_data->lastUpdate_seq = CCV(ccv, snd_nxt);
                }
                else {
                        hpcc_compute_wind(CCV(ccv, downlink_U, false);
                }

        }

        CCV(ccv, snd_cwnd) = hpcc_data->wnd;
}


static void
hpcc_compute_wind(struct cc_var *ccv, uint8_t U, bool updateWc)
{
        struct hpcc *hpcc_data;

        hpcc_data = ccv->cc_data;

        /*To Do: have to check actual value is precisely reflect intened value */
        if( U >= (uint8_t)eta || hpcc_data->incStage >= hpcc_data->maxStage) {
                hpcc_data->wnd = hpcc_data->wnd_c * (u_long)(eta / U) + hpcc_data->wnd_ai;
                if( updataWc ) {
                        hpcc_data->incStage = 0;
                        hpcc_data->wnd_c = hpcc_data->wnd;
                }
        }
        else {
                hpcc_data->wnd = hpcc_data->wnd_c + hpcc_data->wnd_ai;
                if( updataWc ) {
                        hpcc_data->incStage++;
                        hpcc_data->wnd_c = hpcc_data->wnd;
                }

        }
}
```

Conclusion and Future Works

□ Conclusion

① NVMe/TCP is new trend to support fast access to the remote storage pool.
- With low cost & easy deployment

② User-level TCP is one of the best options for NVMe/TCP.

③ Performance bottleneck of user-level TCP is coming from F-stack library.

④ Congestion control algorithm is the key factor for the performance of NVMe/TCP.

□ Future works

① Main performance bottleneck is come from F-stack library.
- We have to focus on F-stack library optimization to dramatically increase F-stack performance.
- F-stack developers are also aware of this issue.

② Upgrade NVMe CC algorithm to utilize “the number of flows” which could be observed at the programmable switch.
- By doing so, fairness performance is expected to be definitely improved.