Topic: Fast User-level TCP Stack on DPDK

Company: KAIST
Title: mTCP: A High-Speed User-Level TCP Stack on DPDK
Name: KyoungSoo Park
mTCP: A High-Speed User-Level TCP Stack on DPDK

KyoungSoo Park

In collaboration with
Eunyoung Jeong, Shinae Woo, Asim Jamshed,
Haewon Jeong, Sunghwan Ihm+, Dongsu Han
School of Electrical Engineering, KAIST &
+Department of Computer Science, Princeton University
Short TCP Flows Dominate the Internet

Middleboxes
- SSL proxies
- Network caches

End systems
- Web servers

* Commercial cellular traffic for 7 days
Comparison of Caching Strategies in Modern Cellular Backhaul Networks, MOBISYS 2013
Suboptimal Linux TCP Performance

- Large flows: Easy to fill up 10 Gbps
- Small flows: Hard to fill up 10 Gbps regardless of # CPU cores
  - Too many packets:
    14.88 Mpps for 64B packets in a 10 Gbps link
  - Kernel is not designed for multicore systems
Inefficient Kernel Code for TCP Transactions

CPU Usage Breakdown of Web Server
Web server (Lighttpd) Serving a 64 byte file
Linux-3.10

- Application: 17%
- TCP/IP: 34%
- Packet I/O: 4%
- Kernel (without TCP/IP): 45%

83% of CPU usage spent inside kernel!

Performance bottlenecks
1. Shared resources
2. Broken locality
3. Per packet processing

Bottleneck removed by mTCP

1) Efficient use of CPU cycles for TCP/IP processing
→ 2.35x more CPU cycles for app
2) 3x ~ 25x better performance
1. Shared resources
   - Shared listening queue
   - Shared file descriptor space
2. Broken locality

Interrupt handling core != accepting core

Interrupt handle
Core 0

Core 1

Core 2

Core 3

Per-core packet queue

Receive-Side Scaling (H/W)

accept()
read()
write()
Lack of Support for Batch Processing

3. Per packet, per system call processing

- Inefficient per packet processing
- Frequent mode switching
- Cache pollution
- Per packet memory allocation

- BSD socket
- Linux epoll

- Application thread

- User

- Kernel TCP

- Packet I/O

- accept(), read(), write()
### Previous Works on Reducing Kernel Bottleneck

<table>
<thead>
<tr>
<th></th>
<th>Listening queue</th>
<th>Connection locality</th>
<th>App &lt;-&gt; TCP comm.</th>
<th>Packet I/O</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux-2.6</td>
<td>Shared</td>
<td>No</td>
<td>Per system call</td>
<td>Per packet</td>
<td>BSD</td>
</tr>
<tr>
<td>Linux-3.9 SO_REUSEPORT</td>
<td>Per-core</td>
<td>No</td>
<td>Per system call</td>
<td>Per packet</td>
<td>BSD</td>
</tr>
<tr>
<td>Affinity-Accept</td>
<td>Per-core</td>
<td>Yes</td>
<td>Per system call</td>
<td>Per packet</td>
<td>BSD</td>
</tr>
<tr>
<td>MegaPipe</td>
<td>Per-core</td>
<td>Yes</td>
<td>Batched system call</td>
<td>Per packet</td>
<td>custom</td>
</tr>
</tbody>
</table>

Still, **78%** of CPU cycles are used in kernel!

How much **performance improvement** can we get if we implement a **user-level TCP stack** with all optimizations?
Clean-slate Design for Fast TCP Processing

- mTCP: A high-performance user-level TCP designed for multicore systems
- Clean-slate approach to divorce kernel’s complexity

Problems
1. Shared resources
2. Broken locality
3. Lack of support for batching

Our contributions
- Each core works independently
  - No shared resources
  - Resources affinity
- Batching from flow processing from packet I/O to user API
- Easily portable APIs for compatibility
Overview of mTCP Stack

1. Thread model: Pairwise, per-core threading
2. Batching from packet I/O to application
3. mTCP API: Easily portable API (BSD-like)
1. Thread Model: Pairwise, Per-core Threading

- **Application Thread 0**
- **mTCP socket**
- **mTCP thread 0**
- **mTCP epoll**
- **mTCP thread 1**

**User-level packet I/O (DPDK)**

**NIC Device**

Core 0

Core 1

**Per-core listening queue**

**Per-core file descriptor**

**Symmetric Receive-Side Scaling (H/W)**
System Calls to Context Switching?

System call

- BSD socket
- Linux epoll
- Kernel TCP
- Packet I/O

Context switching

- mTCP socket
- mTCP epoll
- mTCP thread
- User level packet I/O library

higher overhead

Batching to amortize context switch overhead

Linux TCP

Application thread

mTCP

Application Thread
2. Exploiting Batched Event/Packet Processing

Socket API
- accept()
- epoll_wait()
- connect()
- write()
- close()

Application thread

mTCP thread
- Rx manager
  - Accept queue
  - Event queue
  - Internal event queue
  - Payload handler

- TX manager
  - Control list
    - SYN
    - RST
    - FIN
  - ACK list
  - Data list
  - Data
  - ACK
  - SYN

- Rx queue
- Tx queue
3. BSD Socket-like API for mTCP

- Two goals: Easy porting + retaining popular event model
- Ease of porting
  - Just pre-append “mtcp_” to BSD socket API
  - `socket()` → `mtcp_socket()`, `accept()` → `mtcp_accept()`, etc.
- Event notification: Readiness model using `epoll()`
- Porting existing applications
  - Mostly less than 100 lines of code change

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Modified lines / Total lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lighttpd</td>
<td>An event-driven web server</td>
<td>65 / 40K</td>
</tr>
</tbody>
</table>
3. BSD Socket-like API for mTCP

- Two goals: Easy porting + retaining popular event model
- Ease of porting
  - Just pre-append “mtcp_” to BSD socket API
  - `socket()` → `mtcp_socket()`, `accept()` → `mtcp_accept()`, etc.
- Event notification: Readiness model using `epoll()`
- Porting existing applications
  - Mostly less than 100 lines of code change

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Modified lines / Total lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lighttpd</td>
<td>An event-driven web server</td>
<td>65 / 40K</td>
</tr>
<tr>
<td>ApacheBench</td>
<td>A webserver performance benchmark tool</td>
<td>29 / 66K</td>
</tr>
</tbody>
</table>
3. BSD Socket-like API for mTCP

- Two goals: Easy porting + retaining popular event model
- Ease of porting
  - Just pre-append “mtcp_” to BSD socket API
  - `socket() \rightarrow mtcp_socket()`, `accept() \rightarrow mtcp_accept()`, etc.
- Event notification: Readiness model using `epoll()`
- Porting existing applications
  - Mostly less than 100 lines of code change

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Modified lines / Total lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lighttpd</td>
<td>An event-driven web server</td>
<td>65 / 40K</td>
</tr>
<tr>
<td>ApacheBench</td>
<td>A webserver performance benchmark tool</td>
<td>29 / 66K</td>
</tr>
<tr>
<td>SSLShader</td>
<td>A GPU-accelerated SSL proxy [NSDI ’11]</td>
<td>43 / 6,618</td>
</tr>
</tbody>
</table>
3. BSD Socket-like API for mTCP

- Two goals: Easy porting + retaining popular event model
- Ease of porting
  - Just pre-append “mtcp_” to BSD socket API
  - `socket()` → `mtcp_socket()`, `accept()` → `mtcp_accept()`, etc.
- Event notification: Readiness model using `epoll()`
- Porting existing applications
  - Mostly less than 100 lines of code change

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Modified lines / Total lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lighttpd</td>
<td>An event-driven web server</td>
<td>65 / 40K</td>
</tr>
<tr>
<td>ApacheBench</td>
<td>A webserver performance benchmark tool</td>
<td>29 / 66K</td>
</tr>
<tr>
<td>SSLShader</td>
<td>A GPU-accelerated SSL proxy [NSDI ’11]</td>
<td>43 / 6,618</td>
</tr>
<tr>
<td>WebReplay</td>
<td>A web log replayer</td>
<td>81 / 3,366</td>
</tr>
</tbody>
</table>
/* socket creation, bind, listen functions */
int mtcp_socket(mctx_t mctx, int domain, int type, int protocol);
int mtcp_bind(mctx_t mctx, int sockid, const struct sockaddr *addr, socklen_t addrlen);
int mtcp_listen(mctx_t mctx, int sockid, int backlog);

/* accept and connect */
int mtcp_accept(mctx_t mctx, int sockid, struct sockaddr *addr, socklen_t *addrlen);
int mtcp_connect(mctx_t mctx, int sockid, const struct sockaddr *addr, socklen_t addrlen);

/* read functions */
int mtcp_read(mctx_t mctx, int sockid, char *buf, int len);
int mtcp_readv(mctx_t mctx, int sockid, struct iovec *iov, int numIOV);

/* write functions */
int mtcp_write(mctx_t mctx, int sockid, char *buf, int len);
int mtcp_writev(mctx_t mctx, int sockid, struct iovec *iov, int numIOV);

/* socket closure */
int mtcp_close(mctx_t mctx, int sockid);
int mtcp_abort(mctx_t mctx, int sockid);

/* rss queue mapping */
int mtcp_init_rss(mctx_t mctx, in_addr_t saddr_base, int num_addr, in_addr_t daddr, in_addr_t dport);
/* configuration file reading */
int mtcp_init(char *config_file);
void mtcp_destroy();

int mtcp_getconf(struct mtcp_conf *conf);
int mtcp_setconf(const struct mtcp_conf *conf);
int mtcp_core_affinitize(int cpu);

/* thread context manipulation */
mctx_t mtcp_create_context(int cpu);
void mtcp_destroy_context(mctx_t mctx);

typedef void (*mtcp_sighandler_t)(int);
mtcp_sighandler_t mtcp_register_signal(int signum, mtcp_sighandler_t handler);

/* pipe, getsock/setsockopt, set fd non-blocking mode */
int mtcp_pipe(mctx_t mctx, int pipeid[2]);
int mtcp_getsockopt(mctx_t mctx, int sockid, int level, int optname, void *optval, socklen_t *optlen);
int mtcp_setsockopt(mctx_t mctx, int sockid, int level, int optname, const void *optval, socklen_t optlen);
int mtcp_setsock_nonblock(mctx_t mctx, int sockid);

/* mtcp_socket_ioctl: similar to ioctl, but only FIONREAD is supported currently */
int mtcp_socket_ioctl(mctx_t mctx, int sockid, int request, void *argp);
static void thread_init(mctx_t mctx)
{
  int sock, lsock, ep, i; /* init declarations */
  struct sockaddr_in saddr; struct mtcp_epoll_event ev, events[MAX_EVENTS];

  /* create listening socket */
  lsock = mtcp_socket(mctx, AF_INET, SOCK_STREAM, 0);
  /* bind and listen to a specific port */
  saddr.sin_family = AF_INET; saddr.sin_addr = INADDR_ANY; saddr.sin_port = 80;
  mtcp_bind(mctx, lsock, (struct sockaddr *)&saddr, sizeof(struct sockaddr_in));
  mtcp_listen(mctx, lsock, 4096);
  /* create epoll queue & enlist listening port in epoll queue */
  ep = mtcp_epoll_create(mctx, MAX_EVENTS);
  ev.events = MTCP_EPOLLIN; ev.data.sockid = lsock;
  mtcp_epoll_ctl(mctx, ep, MTCP_EPOLL_CTL_ADD, lsock, &ev);

  while (1) {
    int nevents = mtcp_epoll_wait(mctx, ep, events, MAX_EVENTS, -1);
    for (i = 0; i < nevents; i++) {
      if (events[i].data.sockid == lsock) {
        sock = mtcp_accept(mctx, lsock, NULL, NULL);
        ...
      } else if (events[i].events == MTCP_EPOLLIN) {
        mtcp_read(mctx, ...);
        ...
      }
    }
  }
}
static void thread_init(mctx_t mctx)
{
    int sock, lsock, ep, i; /* init declarations */
    struct in_addr_t saddr, daddr; struct in_port_t sport, dport;
    struct mtcp_epoll_event ev, events[MAX_EVENTS];

    saddr = INADDR_ANY; daddr = inet_addr(DHOST_IP);
    dport = htons(80);
    /* initialize per-thread client-port RSS pool */
    mtcp_init_rss(mctx, saddr, 1, daddr, dport);
    ep = mtcp_epoll_create(mctx, MAX_EVENTS);

    while (1) {
        if (connection_count < conn_thresh)
            CreateConnection(mctx, ep); /* mtcp_connect() */
        int nevents = mtcp_epoll_wait(mctx, ep, events, MAX_EVENTS, -1);
        for (i = 0; i < nevents; i++) {
            if (events[i].events & MTCP_EPOLLIN)
                HandleReadEvent(...); /* mtcp_read() */
            ...
        }
    }
}
mTCP Implementation

• 12,727 lines in C code
  • Packet I/O, TCP flow management, user-level socket API, event system library

• Supports Intel DPDK
  • Fast packet I/O library + event-driven packet I/O
  • Originally based on PacketShader IOEngine [SIGCOMM’10]

• TCP protocol conformance
  • Follows RFC793
  • Congestion control algorithm: NewReno

• Passing correctness test and stress test with Linux TCP stack
Evaluation

• Does performance scale over CPU cores?
  • Performance comparison with previous solutions

• Does it improve the performance of real applications?
  • Web server (Lighttpd)
    • Performance under the real workload
  • SSL proxy (SSL Shader, NSDI 11)
    • GPU acceleration on crypto algorithms (RSA, AES, SHA1)
    • Bottlenecked at TCP stack

• Third party evaluation
  • HAProxy port to mTCP
  • nginx port to mTCP
Evaluation Setup

• Client – server HTTP transactions
• Server specification
  • One Xeon E5-2690 CPU (8 cores), 2.90 GHz
  • 32 GB RAM, 1 x 10G NIC (Intel 82599 chipset)
• Clients: 5 x machines with the same spec with server
Multicore Scalability

- 64B ping/pong messages per connection (Connections/sec)
- Heavy connection overhead, small packet processing overhead
- 25x Linux, 5x SO_REUSEPORT*[^LINUX3.9], 3x MegaPipe*[^OSDI’12]

![Graph showing scalability with number of CPU cores](image)

- Inefficient small packet processing in Kernel
- Shared fd in process
- Shared listen socket

*[^LINUX3.9] [https://lwn.net/Articles/542629/](https://lwn.net/Articles/542629/)
Performance Improvement on Real Applications

**Web Server (Lighttpd)**
- Real traffic workload: Static file workload from SpecWeb2009 set
- **3.2x** faster than Linux
- **1.5x** faster than MegaPipe

**SSL Proxy (SSLShader)**
- Performance Bottleneck in TCP
- Cipher suite
  - 1024-bit RSA, 128-bit AES, HMAC-SHA1
- Download 1-byte object via HTTPS

### Throughput (Gbps)

<table>
<thead>
<tr>
<th>System</th>
<th>Throughput (Gbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>1.24</td>
</tr>
<tr>
<td>REUSEPORT</td>
<td>1.79</td>
</tr>
<tr>
<td>MegaPipe</td>
<td>2.69</td>
</tr>
<tr>
<td>mTCP</td>
<td>4.02</td>
</tr>
</tbody>
</table>

### Transactions/sec (x 10^3)

<table>
<thead>
<tr>
<th>System</th>
<th>Transactions/sec (x 10^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>26,762</td>
</tr>
<tr>
<td>REUSEPORT</td>
<td>31,710</td>
</tr>
<tr>
<td>MegaPipe</td>
<td>36,505</td>
</tr>
<tr>
<td>mTCP</td>
<td>37,739</td>
</tr>
</tbody>
</table>

# Concurrent Flows

- **4K**: 28,208
- **8K**: 27,725
Third Party Evaluation

- Third-party company 1
  - Ported HAProxy to mTCP
  - Saw 2x performance improvement for small-file transactions

- Third-party company 2
  - Ported nginx to mTCP
  - Experiment setup
    - Server: 2 x Xeon E5-2699 v3 (36 cores in total)
      - 32 GB memory
      - 2 x 40G Intel NICs
      - DPDK: v2.2RC1
      - nginx release 1.9.6
    - Client: same spec with the server
  - We plan to merge the patches into mTCP github
nginx HTTP Connections per Second (CPS)

1.4M CPS @1KB
Throughput 14Gbps
> x3 improvement

- Use up 2 Client
- CPU hasn’t reach 100% on server side

Perf. gain from multiple tcp port is limited, not scale

Response size: 1KB

2 * 40G NIC connected with two clients

mTCP
kernel two TCP port
kernel multi-TCP port

# of workers

CPS million

0.00 0.20 0.40 0.60 0.80 1.00 1.20 1.40 1.60 1.80 2.00
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36
nginx HTTP Plaintext Responses per sec (RPS)

<table>
<thead>
<tr>
<th># of workers</th>
<th>RPS million</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>2.00</td>
</tr>
<tr>
<td>8</td>
<td>3.00</td>
</tr>
<tr>
<td>10</td>
<td>4.00</td>
</tr>
<tr>
<td>12</td>
<td>5.00</td>
</tr>
<tr>
<td>14</td>
<td>6.00</td>
</tr>
<tr>
<td>16</td>
<td>7.00</td>
</tr>
<tr>
<td>18</td>
<td>8.00</td>
</tr>
<tr>
<td>20</td>
<td>9.00</td>
</tr>
<tr>
<td>22</td>
<td>10.0</td>
</tr>
<tr>
<td>24</td>
<td>11.0</td>
</tr>
<tr>
<td>26</td>
<td>12.0</td>
</tr>
<tr>
<td>28</td>
<td>13.0</td>
</tr>
<tr>
<td>30</td>
<td>14.0</td>
</tr>
<tr>
<td>32</td>
<td>15.0</td>
</tr>
<tr>
<td>34</td>
<td>16.0</td>
</tr>
<tr>
<td>36</td>
<td>17.0</td>
</tr>
</tbody>
</table>

2 x 40G NIC connected with two clients

Response size: 1KB

3.5M RPS @1KB
Throughput 33Gbps
~50% improvement

2MRPS, throughput 20~30Gbps?
mTCP features Under Development

- Now supports multi processes
  - Easily port single-threaded network applications
  - Global variables => possible race conditions
  - HAProxy, Lighttpd etc.
- TCP Segmentation Offlaoding (TSO) patch ready
  - Third-party contribution
- Virtualization support
  - virtio work for container
- Zero copy I/O
  - DPDK buffer visible to stack
- Beyond end-host TCP stack?
  - mOS networking stack
The need for Resusable Middlebox Networking Stack

- Developing stateful middlebox is non-trivial
  - Typical IDS code ~= 135K SLOC
  - TCP flow management is complex and error prone

- Typical TCP applications
  - TCP application
  - Berkeley Socket API
  - TCP/IP stack
  - User level
  - Kernel level

- Typical middleboxes?
  - Middlebox logic
  - Packet processing
  - Flow management
  - Spaghetti code?
  - No layered abstraction!

- Why not Berkeley-socket API for middleboxes?
  - Nice abstraction: separates flow management from application
  - Knowledge of internal TCP implementation not needed
mOS Middlebox Networking stack

• Networking stack specialization for middleboxes
  • Abstraction for sub-TCP layer middlebox operations

• Key concepts
  • Separation of flow management from custom middlebox app logic
  • User-defined event definition and generation

• Benefits
  • Clean, modular development of stateful middleboxes
  • Developers focus on core logic only
    • Reuse a networking stack for TCP flow management
  • High performance from mTCP implementation
    • Optimized for multicore systems
    • Fast packet/TCP processing on DPDK
Middlebox Stack Monitors Both Ends

- Dual mTCP stack management
- *Infer* the states of both client and server TCP stacks

![Diagram of TCP state transition](image)

- Develop middlebox logic within event handlers

Generate client-side registered event handlers

Generate server-side registered event handlers
Conclusion

• mTCP: A high-performing user-level TCP stack for multicore systems
  – Clean-slate user-level design to overcome inefficiency in kernel

• Make full use of extreme parallelism & batch processing
  – Per-core resource management
  – Lock-free data structures & cache-aware threading
  – Eliminate system call overhead
  – Reduce context switch cost by event batching

• Achieve high performance scalability
  – Small message transactions: 3x to 25x better
  – Existing applications: 33% (SSLSHader) to 320% (lighttpd)
Thank You

Source code is available @
http://shader.kaist.edu/mtcp/
https://github.com/eunyoung14/mtcp