acceleraTOR: Tor in the Fast Lane

Spring15 : 18-731 Network Security - Final Report

Antonio Rodrigues
antonior@andrew.cmu.edu

Saurabh Shintre
sshintre@andrew.cmu.edu

Soo-Jin Moon
soojinm@andrew.cmu.edu

Zijie Lin
zjlin@cmu.edu

***Abstract*—The abstract goes here.**

I. Introduction

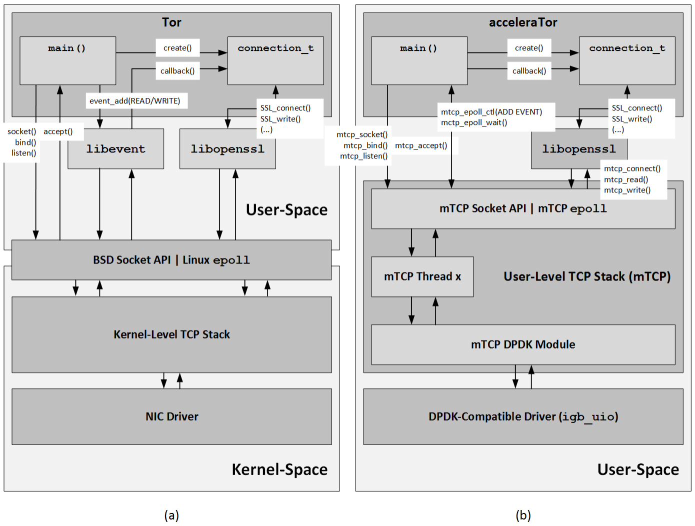
II. Background and Motivation

III. acceleraTor Design and Implementation

In this section we describe acceleraTor's design in some detail and briefly explain relevant implementation issues.

A. Design Overview

acceleraTor's overall design is depicted in Figure 1: we compare it to the design of 'vanilla' Tor to clearly emphasize their differences.



(a) (b)

Fig. 1: 'Vanilla' Tor (a) vs. acceleraTor's (b) design overview.

At the extremes of Figure 1 b) we have acceleraTor (top), a TCP-based network application which implements Tor OR logic; and DPDK's user-level packet I/O libraries (bottom), which read/write packets directly from DPDK-compatible Network Interface Cards (NICs). The main difference with respect to 'vanilla' Tor is that all levels in acceleraTor's design run in user space. In-between, we recur to a user-level TCP stack – mTCP [1] – to interconnect acceleraTor and DPDK, and ultimately reserve the 'services' of a single DPDK-compatible NIC to the application. Although not part of acceleraTor's design per se, this application-NIC 'exclusivity' led us to run acceleraTor in nodes equipped with two NICs, one to be handled by a DPDK-compatible driver and reserved for acceleraTor, another handled by a Linux kernel driver, which allows for the parallel operation of 'usual' networking applications.

In the next subsections, we discuss design and implementation aspects for each one of the levels in more detail.

B. Tor's Integration with mTCP

Some of the benefits brought about by DPDK come from a bypass of the kernel networking stack, allowing applications to process 'raw' packets at the user-level and thus overcome kernel-level inefficiencies such as heavy system call overhead [1], [2]. Nevertheless, such a 'bypass' also involves challenges. Without kernel-level support of TCP, we either:

1) Modify Tor to become independent of TCP, having it directly process packets 'fresh' out of the NICs;
2) Develop/use an user-level TCP stack, integrated with DPDK, on top of which Tor can be adapted.

Option 1 would heavily modify the way Tor works – e.g. Tor makes use of TLS, which depends on TCP's reliable transport – not only making acceleraTor ORs incompatible with 'vanilla' Tor ORs, but also making modifications to Tor unnecessarily intrusive and harder. Option 2 is more desirable: ideally, it should rely on an intermediate module which (1) interfaces with DPDK to read/write 'raw' packets from/to NICs, (2) implements "enough of" TCP to have TCP-based applications and TLS work on top of it; and (3) exposes a 'BSD-like' socket API, allowing for easy integration of TCP-based applications and libraries (e.g. OpenSSL[1] for TLS).

mTCP [1] is a user-level TCP stack which – while specifically designed to solve TCP performance issues in multicore systems – fulfills the points of the 'ideal' option 2: as of version 3[2], mTCP comes directly integrated with DPDK (v1.8 and v2.0). Other choices have been quickly surveyed – e.g. drv-netif-dpdk[3], a DPDK interface for Rump Kernels [3]; the solution proposed by Kantee et al. in [4]; or NUSE [5] – but dismissed since these could not beat mTCP's suitability to the aforementioned requirements.

Despite its convenience, the use mTCP to integrate Tor and DPDK poses the following set of implementation challenges, some of which are briefly addressed in the following subsections:

[1] https://openssl.org/
[2] Released in April 2, 2015, https://github.com/eunyoung14/mtcp.
[3] https://github.com/rumpkernel/drv-netif-dpdk

- mTCP's API slightly differs from the 'BSD' socket API used in Tor, and does not provide all its functionalities. acceleraTor must accommodate these limitations without major loss of functionality.
- In its 'vanilla' version, Tor makes use of libevent[4] to handle asynchronous input/output (IO). mTCP provides its own epoll-like event system to monitor mTCP sockets, not compatible with libevent[5]. Therefore, Tor's asynchronous IO logic must be changed.
- Tor makes heavy use of Transport Layer Security (TLS) connections, through OpenSSL, which must now be modified to comply with mTCP's API.

1) Porting Tor to mTCP: Since mTCP provides a 'BSD-like' socket API, most of the porting effort consisted in identifying the code using 'BSD-like' system calls and replacing these mTCP's equivalent calls. Such changes have been kept as contained and isolated as possible: we allow for the selective compilation of either 'vanilla' Tor or acceleraTor via a new Tor configuration option[6].

Besides 'syscall-translation', the porting effort also included the initialization of a single mTCP thread from within Tor's main thread, which ultimately 'binds' to an available DPDK-compatible NIC. Tor's main loop was changed to continuously wait for mTCP events (e.g. new connections, read/write events on existing connections) using mTCP's user-level event system : in Tor's original design, a similar approach is used to listen for new events, using libevent's API instead. More details are given in section III-B2.

2) Event Handling Logic: Tor uses asynchronous input/ output (IO) to monitor sockets associated with inter-OR TCP connections and invoke appropriate connection-handling callbacks whenever read/write events are triggered. mTCP provides its own epoll-like event system to monitor mTCP connections, not compatible with libevent, used by 'vanilla' Tor.

When adding creating a new OR/client connection, Tor registers new read/write events to monitor the file descriptors (i.e. sockets) associated with the connection. libevent's API allows for a direct association between the registered event and a specific connection descriptor. The same is not possible in mTCP: mTCP's API associates events with a socket file descriptor. We therefore need some 'external' way of mapping socket file descriptors to the respective connection descriptor objects, and then re-use already existing read/write callbacks. We accomplish this via a global hash table, implemented using uthash[7].

3) Integration with OpenSSL: Last – but definitely not least – there is the issue of TLS: Tor makes heavy use of TLS to secure OR/client connections, and in turn uses OpenSSL's API to get TLS support. OpenSSL must then be also ported to mTCP. As OpenSSL's implementation relies on 'BSD'-like system calls, the porting effort mostly consists in tracking down the code which uses such system calls and replace them with those exposed by mTCP API. Besides the 'translation' effort, we introduce the changes in OpenSSL's API:

a) Added a new attribute in SSL structures (i.e. representations of a TLS connection), mctx_t mctx, which represents the mTCP thread context to which the SSL structure should be associated with.

b) Added a new call to OpenSSL's API

```
int SSL_set_mtcp_ctx(SSL * s, mctx_t mctx)
```

which lets accelaraTor (or other mTCP-based applications willing to use OpenSSL) set the mTCP thread context attribute on some SSL structure s. These changes are convenient, as other calls exposed by OpenSSL's API – e.g. SSL_connect(SSL * s) – which originally only take a single SSL structure as argument, do not have to be changed to include an additional reference to the respective mTCP context as argument. This eases the integration effort on Tor's own source code ('wrapper' functions for OpenSSL's API, defined in tortls.c).

C. mTCP and DPDK

As of version 3, mTCP comes directly integrated with DPDK (v1.8 and v2.0), and so the relevant issues during development were implementation-related, mostly consisting in finding compromises with (OS- and hardware-level) constraints imposed by both mTCP and DPDK.

- DPDK-compatible mTCP (v3) could only work with the Linux-3.13.0 kernel (we have used 3.13.0-46-lowlatency, which comes with the uio kernel module installed by default);
- We have tested setups in both 32- and 64-bit architectures: in the latter case, DPDK's libraries had to be compiled as shared;
- DPDK is compatible with a limited set of NICs: we have successfully tested DPDK with Intel 82545 EM[8]and Intel 82599[9] Ethernet adapters.

IV. Evaluation

V. Related Work

VI. Conclusions and Future Work

References

[1] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[2] J. D. Brouer, H. F. Sowa, D. Borkmann, and F. Westphal, "Challenge 10Gbit/s," pp. 1–33, 2014. [Online]. Available: http://people.netfilter.org/hawk/presentations/nfws2014/dp-accel-10G-challenge.pdf

[3] R. K. Project, "Rump Kernels," p. 1, 2014. [Online]. Available: http://rumpkernel.org/

[4] http://libevent.org/

[5] I.e. libevent does not recognize the event-checking 'method' made available by mTCP's event system.

[6] Source code and setup scripts available at https://bitbucket.org/clockWatchers/accelerator

[7] http://troydhanson.github.io/uthash/

[8] Oracle Virtual Box: https://www.virtualbox.org/.

[9] AWS EC2 c4.large instances.

[4] A. Kantee, “Environmental Independence : BSD Kernel TCP / IP in Userspace,” in *Asia BSDCON 2009*, 2009, pp. 1–10. [Online]. Available: https://2009.asiabsdcon.org/papers/abc2009-P5A-paper.pdf

[5] S. Han, “NUSE: Networking Stack in Userspace?” p. 1, 2014. [Online]. Available: https://www.eecs.berkeley.edu/~sangjin/2013/01/14/NUSE.html