Daytona : A User-Level TCP Stack

Prashant Pradhan[y] Srikanth Kandula[z] Wen Xu[–] Anees Shaikh[y] Erich Nahum [y]

Enterprise Networking Dept.[y] IBM T.J. Watson Research Center
Dept. of Computer Science[z] University of Illinois
Dept. of Computer Science[–] Princeton University

Abstract

This paper presents Daytona, a user-level TCP stack for Linux. A user-level TCP stack can be an invaluable tool for TCP performance research, network performance diagnosis, rapid prototyping and testing of new optimizations and enhancements to the TCP protocol, and as a tool for creating adaptive application-level overlays. We present the design and implementation of Daytona, and also describe several projects that are using Daytona in a rich variety of contexts, indicating its suitability as an open-source project.

1 Introduction

TCP is the most widely used transport protocol in the Internet today, which has made it a subject of much research over the past two decades. Understanding various aspects of TCP performance and its behavior under various network conditions remains crucial to understanding the performance of networked applications, tuning network server performance, and designing novel networked applications.

Historically, most academic studies on TCP have involved simulation studies using tools such as ns [1]. Such tools can often provide a lot of useful information about TCP behavior in a simulated network. Unfortunately, tools such as ns can't be used to interact with actual network applications over a real network. In fact, a routinely criticized aspect of ns-based studies has been that the simulated network conditions often fail to capture traffic characteristics in the real Internet, which can be extremely hard to model.

The other extreme is to use a standard, in-kernel TCP implementation for such studies. While this approach clearly captures actual network behavior and interactions with actual applications, it is hard to extract useful protocol data from the kernel, which is spread over various state variables of the TCP protocol. For example, the congestion window, loss characteristics and round-trip time estimate maintained by TCP for each connection, capture important characteristics of the state of the network path used by that connection. There is no clean way to extract this information without introducing new APIs into the kernel. Some APIs, for example the TCP_INFO socket option in Linux 2.4 and the SO_DEBUG option in FreeBSD, allow one to extract TCP state variables from the kernel. However these typically provide aggregated information, which is not synchronously correlated with application actions.

A similar issue arises while studying network server performance. Overheads in the networking stack dominate the achievable performance of a high-volume network server. Profiling and instrumentation of the TCP stack can reveal sources of such overheads and help in tuning server performance. However, doing this in a flexible and easy manner becomes a challenge when the TCP implementation resides in the kernel.

Recently a lot of interest has been generated by application-level overlays and peer-to-peer systems such as Napster and Gnutella. While in principle, creating such overlays only requires socket applications, in practice it is

often useful to make such overlays adaptive with respect to the state of the network. Having TCP state information available at the application layer provides a complete and sound information base for guiding such adaptation. While it is possible for applications to develop their own mechanisms for probing network characteristics such as available bandwidth and loss rates, TCP's bandwidth estimation mechanisms are formally known to be stable and fair to competing flows in the network. Hence it is more desirable for adaptive network applications to reuse information gathered by TCP about the state of the network.

Interestingly, a user-level TCP stack is a "least common denominator" that can help us address these diverse challenges. If such a stack were to be available, it would be straight-forward to customize it to provide the specific information required by each problem, in a fairly flexible manner. Daytona is essentially inspired by this goal. Daytona is a library available to Linux applications, which works with any arbitrary network interface and is largely independent of the kernel version. Since Daytona finds applications in a rich variety of contexts, we believe it is well-suited to be an open-source project.

The rest of the paper is organized as follows. In section 2 we briefly discuss some background on how Daytona evolved from related projects, and improved upon them by addressing their key limitations. Section 3 describes the design of Daytona and section 4 fills in relevant implementation details. Section 6 describes several projects that are currently using Daytona. Section 7 concludes with some description of ongoing work.

2 Background

Once it was clear that a user-level TCP stack is the key tool to address the kinds of problems described above, our first attempt was to try and find an existing implementation. Indeed, other projects have realized the need for a user-level networking stack [11, 13, 14, 15, 16, 17]. However, these implementations were either available on non-Linux platforms, or provide only part of the complete TCP functionality at the user-level. Non-Linux implementations did not serve our purpose since we believe that a solution for Linux would have wider applicability by virtue of its larger user and developer base. Similarly, tools that did not provide complete stack functionality could not serve as a base user-level TCP implementation that would apply to a rich variety of problems.

Arsenic [12] was a Linux 2.3.29-based user-level TCP implementation developed by Pratt and Fraser at Cambridge University. The goal of their project was to give user-level applications better control for managing bandwidth on a network interface. Their implementation was designed to work with a specialized gigabit network interface from Alteon called ACEnic, which provided per-connection contexts in interface hardware. Their user-level stack communicated with the interface through a control API provided by the card device driver. This API was used to install filters for packet classification, buffers for posting data for various connections, and for specifying network QoS parameters for these connections.

Essentially, Arsenic's goal was to extract the best performance and QoS functionality from a specialized network interface. As a general rule of thumb, *specialization* is often the key to achieving the highest performance from a given system, and utilizing all features provided by its hardware. This is also true of the Arsenic implementation. Arsenic's design naturally involves a coupling with the interface hardware, and a dependence on the kernel version. For example, buffer management is one of the most specialized aspects of the Arsenic implementation. Providing zero-copy transfers and connection-specific buffers was a key goal of the project, both for the purposes of low overhead and QoS isolation. This involves co-mapping of buffers between the user-level TCP library and the cards kernel device driver. Arsenic uses kernel modules for providing this functionality, which leads to a dependency on the VM implementation of the 2.3.29 Linux kernel. The co-mapping facility also allowed some of the kernel's timing variables (e.g., jiffies) to be visible directly in user space. This indirectly leads to a dependence of the TCP timer management code upon the kernel version. Classification and demultiplexing of packets to various connections was handled by the specialized ACEnic hardware, and hence was essentially still a kernel function. Arsenic also uses kernel modules for extracting routing tables and ARP tables from the kernel, which requires porting to work with newer kernel versions. At first, we attempted to minimally change the implementation, and

adapt some of the functions implemented in kernel modules to work with newer kernels. However, it soon became apparent that for the tool to be useful and extensible, all dependence on kernel modules must be removed.

In contrast with Arsenic, our goal is to provide a very general user-level TCP stack that works with arbitrary network interfaces, with no kernel dependencies. Providing this functionality, rather than the highest performance, is our key goal. In the next section we outline the key design decisions in Daytona that allowed us to achieve this goal. A significant part of the Arsenic implementation could still be directly used. Arsenic pulled the Linux 2.3.29 networking stack code into user-level and packaged it into a library. By pulling the code directly from the Linux implementation, we get a faithful protocol implementation at the user-level. This code can be updated to incorporate enhancements introduced in later kernel versions. However, being user-space code, this code can run on an arbitrary kernel version. Arsenic used user-level threads (GNU pthreads) to provide per-connection context and synchronization features into the stack, replacing the traditional ones that the kernel uses. Hence it provided us a fairly strong foundation to start with.

3 Design Overview

Daytona is a user-level library that provides the sockets and TCP/IP functionality normally implemented in the kernel. Applications can link with this library and get the same networking functionality as provided by the kernel. The kernel is just used as a communication channel between the library and the network, through the use of raw IP sockets. Raw IP sockets can be used as long as the machine has network connectivity, hence Daytona has no dependency on any particular network interface. An important benefit of this choice is that neither IP routing nor MAC-layer information is needed from the kernel. The kernel itself performs routing table and ARP table lookup to create MAC layer headers for the packets sent over a raw IP socket. Thus the kernel modules needed for this purpose by Arsenic are no longer needed.

Normally, a copy of the packets received by the kernel networking stack are handed to a raw socket if certain criteria are met [3]. Another copy is processed by the kernel, in the standard manner. However, TCP and UDP packets on general ports do not satisfy these criteria. To trap such packets, a packet capturing library is needed. Such a library installs filters in the kernel expressed in a generalized interpreted language (e.g. BPF [4]). The overhead of filtering will typically be quite small if, for example, we are interested in only getting packets of some specific application port to the user-level stack. The networking code in the kernel checks packets for a match with such filters, and sends them up to a raw socket opened by the library. Care should be taken to also put firewall DENY rules for such packets, so that the kernel would drop its own copy of such packets and not attempt to do redundant processing for them.

We use the pcap [5] packet capturing library used by the well-known tcpdump tool. To integrate packet capturing, we reused some of the pcap code from the Sting project at the University of Washington [6]. One of the threads in Daytona acts as the “bottom half” thread and constantly polls for packets on the raw socket. Some peculiarities of the pcap library had to be taken into account in the design. For example, in pcap packet reception happens as a side-effect of polling for a packet on the raw socket. Another aspect is that the memory occupied by the captured packet is internally reused by the library for subsequent operations, and hence Daytona must make a copy of the packet provided by the pcap library before passing it up to an application thread. Once packets are captured by Daytona, they are classified and demultiplexed for TCP processing.

Daytona performs buffer and timer management completely in user-space. Buffer management is done in user space in a manner similar to that done inside the kernel during transmit and receive processing. For TCP timer management, a baseline 10 msec timer is implemented to emulate “jiffies”. The Linux kernel provides a higher granularity on top of jiffies when a timestamp is taken, but this can be easily emulated in user space by using the gettimeofday() system call at the timestamping instant, or, in case of the X86 architecture, by using simple macros to read hardware cycle counting registers.

Similar to the case when the TCP/IP stack is in the kernel, calls made by application threads into the Daytona

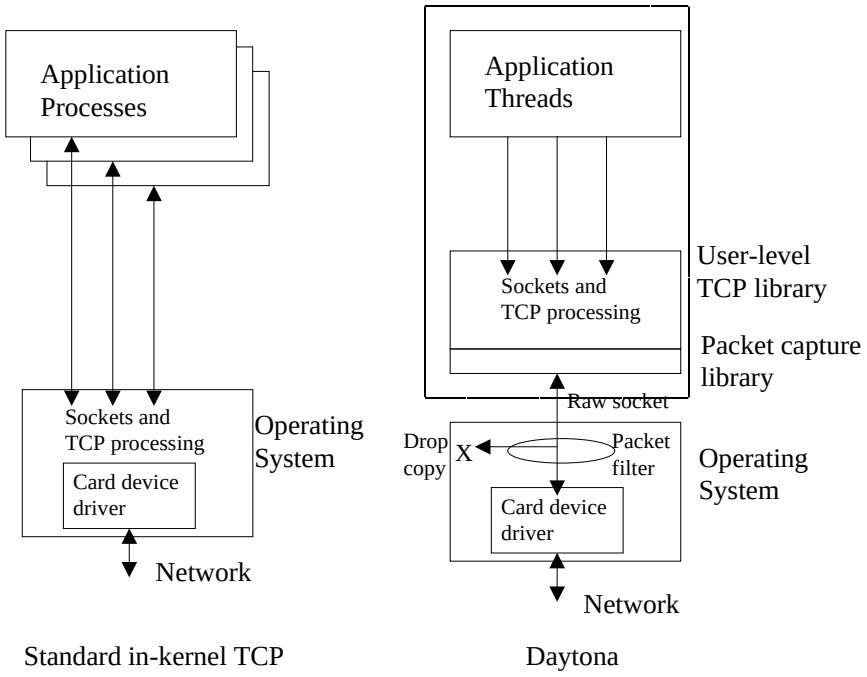

Figure 1. *The high-level design of Daytona compared to the stanrard in-kernel TCP stack.*

library carry their execution contexts with them. When TCP processing must block waiting for events (network events or timers) or resume in response to events, these threads are blocked and unblocked in the same manner as processes are blocked and unblocked inside the kernel.

Note that if multiple processes link to Daytona, each of them gets their own copy of the data maintained within the TCP/IP stack and the socket layer (though the code is shared). Thus, while multiple application threads using Daytona share data, multiple application processes using Daytona do not. This may need to be taken into account depending upon the intended use of Daytona. For example, to measure the packet classification overhead of the TCP stack with a large number of connections, a common classification table should be used. But if the connections span multiple processes, each process will have its own copy of a smaller TCP classification table. This would not capture the overheads correctly. To achieve sharing across processes, it is straight-forward to make Daytona a server process which is called by other processes through IPC calls.

Figure 1 illustrates the high-level multi-threaded design of Daytona, when compared to the standard, in-kernel TCP stack.

4 Code Structure

The intent of this section is to describe the high-level code structure that implements the various pieces of Daytona functionality. This should allow one to understand and potentially modify Daytona to suit their purpose.

4.1 Socket API

At the top level is the library interface to Daytona, which exposes the traditional sockets API, and acts as the entry point for application threads into the user-level networking code. In the spirit of the kernel "inet" functions, most of the functions here simply act as indirections into protocol-specific functions, the protocol being TCP in our case. An important part implemented by this code is the initialization of the library, which also initializes the buffer pool for the user-level networking stack. This buffer pool is essentially the user-level "skbuff" pool.

4.2 Network Bottom-half

The equivalent of the Linux network bottom half ("NetBH") is a thread which uses the pcap interface to poll for work arriving from the network. This thread initiates receive processing, which typically leads to a state change for some TCP connection, followed by a wakeup call on the associated application thread. The application threads and the bottom half thread are scheduled under the control of the pthreads scheduler.

4.3 Transmit Processing

The pcap library uses a raw socket for packet capture on the receive side. Similarly a raw socket interface is also needed on the transmit side. Completely formed IP packets are handed to this socket for transmission by the kernel. The kernel treats these as standard IP packets and performs routing table lookup and MAC processing on them, eventually queuing them to some device driver transmit queue.

4.4 Timers

A 10 msec user-level timer is setup using the standard setitimer() system call to emulate "jiffies". At the timestamping instant, more granularity can be added either by using gettimeofday(), or Pentium timestamp counters.

4.5 Protocol Processing

The code implementing the TCP/IP protocol processing exactly mimics the code and structure of the Linux 2.3.29 code. A tricky problem here is that while normally there is a natural independence between datatype declarations used in the kernel code and those used in applications, user-level TCP must have kernel code as well as code that interfaces with applications. Hence, some of the header files may cause conflicts. Arsenic avoided this problem by collecting all datatype declarations needed by the kernel code in a single header file, which was then included by the kernel-derived code. For code that interfaces with the applications and needs some of these data structures, relevant parts of these data structures were separately exposed in another header file.

Most of the code here mimics the kernel code, except that the kernel sleep and wakeup functions are replaced by thread suspension and wakeup functions. All kernel functions used by the TCP/IP code whose implementations lie outside the networking code are aggregated in a single file. This includes code for Ethernet interaction (now replaced by raw socket communication), synchronization primitives (replaced by thread primitives), and memory and buffer allocation/deallocation routines.

5 Using Daytona

The usage of Daytona is fairly simple. Daytona assumes that the interface eth0 on the host is used as the connection to the network, so no setup is required as long as eth0 exists in the system (this can be confirmed by invoking the command ifconfig on the shell). Daytona applications have to be run as root so that the library can open raw sockets.

Applications use Daytona in a manner similar to the standard C socket library, except that all the standard socket calls have to be prefixed with a string. If calls in the application code cannot be modified for some reason, another alternative is to modify the library linking order so that the Daytona library is linked before the standard C library.

6 Projects Using Daytona

As mentioned before, we expected Daytona to find applications in a rich set of contexts where the problem at hand requires TCP functionality and/or TCP state information. The following subsections give an overview of some projects that are using Daytona in various contexts.

6.1 Network Server Performance Tuning

At IBM Research, we are involved in a project investigating server designs that scale well with next-generation I/O and networking standards such as 10G Ethernet. Daytona proved to be a very convenient tool for us to study and tune network server performance. For example, we were able to use Daytona with the Flash [7] Web server to study TCP performance bottlenecks for Web serving workloads in detail. By having a user-level implementation, we were able to get extensive analysis of copying overheads by using user-level cache profiling tools such as cacheprof [8] on Daytona. Further, Daytona allowed us to extensively instrument and profile TCP overheads for various combinations of applications and workloads. While kernel profiling is also an option for obtaining such overhead information, it provides a break-up of overhead on a functional basis. What we needed is a measurement on an activity basis, for example, overheads of data copying, receive processing and transmit processing. The functional breakdown reported by kernel profiling does not isolate how the overheads of common functions used by these activities (e.g. skbuff functions), should be charged to each kind of activity. By instrumenting the code according to various activities of interest, we were able to easily extract performance data on an activity basis. This process would have been tedious and time-consuming if we had done it in the kernel.

Another significant advantage of Daytona was that it allowed us to create a performance model of TCP processing in a simulator. We have created a performance modeling framework for a network server by modeling various system components (e.g. processors, buses, caches) in a discrete-event simulator. We are interested in understanding the effect of off-loading some of the network processing onto intelligent interfaces with network processors on them. A unique aspect of our modeling framework is that it derives performance models of processors from the applications running on them. Clearly, in this case, we needed a TCP processing program that we could use to derive the network processor model. Daytona was used in our simulation framework to provide this model.

6.2 Adaptive Overlays

Various academic research groups are investigating the design of resilient and adaptive application-level overlays that adapt their routing behavior and function placement to network and server load conditions. Network state information, needed by these overlays to guide their adaptation, is cleanly abstracted out in TCP state variables. Large-scale academic testbeds such as Emulab [9] and PlanetLab [10] are being used to design and evaluate such application-level overlays. In such testbeds, having TCP state information at the application layer obviates the need to modify the kernels of the overlay nodes to provide APIs to extract such information. This approach also facilitates heterogeneity in the kind of machines used in the overlay since otherwise one has to worry about providing OS-specific APIs to extract network state from TCP. Some of these research groups are using variants of Daytona to design and study such overlays.

6.3 Integrated Network Performance Diagnostics

One of the most challenging parts of running a network-based service, for instance Web hosting, is monitoring and managing performance. End-to-end performance may be influenced by numerous factors, and problems are not easily attributable to their correct source. A common performance management approach involves monitoring the application (i.e., user-perceived) performance at a relatively coarse level, and then conducting further, more detailed, tests when a potential problem is detected. Such an approach requires the use of multiple techniques and tools, operating at multiple levels. Hence, the problem remains of how to correlate the information to construct a more complete view of low-level network events (e.g., packet retransmission) and the application actions that triggered them (e.g., HTTP request). An approach taken by some researchers at IBM is to provide application-level measurement tools with direct access to pertinent information from lower layers. This enables a diagnostician to detect specific packet-level events in various contexts of the application in an automated way, without having to unify multiple traces. This approach is being embodied in a measurement and monitoring tool for identifying the causes of performance problems in networked applications.

To avoid reliance on a particular kernel configuration, or the presence of specific kernel support, the architectural approach taken by this project is to make the bulk of the kernel networking stack available at the user level with Daytona. As Daytona provides access to the main protocols in the TCP/IP protocol suite, the measurement application can receive extensive diagnosis information. Changes in TCP state variables, bursty vs. isolated packet loss events, and information about the source and type of received ICMP messages, are examples of the type of additional details that are not otherwise available to an application-level measurement tool. The Daytona library is currently being instrumented under this project to deliver notifications about events of interest to the measurement application. The intent is to use the tool to explore the benefits of the integrated approach for enhanced diagnosis of Web performance.

7 Conclusions and Ongoing Work

In this paper, we have motivated the need for a user-level TCP stack and presented the design and implementation of a user-level TCP stack called Daytona. We have also described various projects that have found Daytona to be useful in a very rich variety of contexts, which we believe indicates its suitability as a widely available open-source tool.

We are currently in the process of improving Daytona in several ways, an important one being to incorporate some of the TCP implementation enhancements that appear in more recent Linux kernel versions. At IBM Research, we are also going through a formal open-sourcing process to make the tool widely available.

Questions, comments or bug reports related to Daytona may be sent to ppradhan@us.ibm.com.

References

[1] "The Network Simulator NS-2." http://www.isi.edu/nsnam/ns/.

[3] "Raw IP Networking FAQ." http://whitefang.com/rin/rawfaq.html.

[4] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In Proceedings of the USENIX Winter Conference, San Diego, California, January 1993.

[5] "The libpcap Packet Capture Library." ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[6] Stefan Savage. "Sting: A TCP Based Network Measurement Tool." In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, Colorado, USA, October 1999.

[7] Vivek Pai, Peter Druschel and Willy Zwaenepoel. “Flash : An Efficient and Portable Web Server.” In Proceedings of the USENIX Annual Technical Conference, Monterey, California, USA, June 1999.

[8] Julian Seward. “The Cacheprof Home Page.” http://www.cacheprof.org.

[9] The Emulab Project. http://www.emulab.net.

[10] The PlanetLab Project. http://www.planet-lab.org.

[11] David Ely, Stefan Savage and David Wetherall. “Alpine: A User-Level Infrastructure for Network Protocol Development.” In Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems, San Francisco, California, USA, March 2001.

[12] Ian Pratt and Keir Fraser. “Arsenic: A User-Accessible Gigabit Ethernet Interface.” In Proceedings of IEEE INFOCOM, Anchorage, Alaska, USA, April 2001.

[13] Peter A. Dinda. “The Minet TCP/IP Stack.” Technical Report NWU-CS-02-08, Department of Computer Science, Northwestern University, 2002.

[14] Torsten Braun, Cristophe Diot, Anna Hoglander and Vincent Roca. “An Experimental User Level Implementation of TCP.” Technical Report RR-2650, INRIA Sophia Antipolis, 1995.

[15] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy and Edward D. Lazowska. “Implementing Network Protocols At User Level.” IEEE/ACM Transactions on Networking, Vol. 1, No. 5, October 1993.

[16] Aled Edwards and Steve Muir. “Experiences Implementing a High-Performance TCP in User-Space.” In Proceedings of ACM SIGCOMM, Cambridge, MA, USA, August 1995.

[17] Chris Maeda and Brian Bershad. “Protocol Service Decomposition for High-Performance Networking.” In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP), Asheville, NC, USA, December 1993.