Enhancements for thrulay

Google Summer of Code Project Report

Huadong Liu
hliu@cs.utk.edu

August 28, 2005

1 Introduction

Thrulay stands for THRUput and deLAY. It is a software package initially developed by Stanislav Shalunov at Internet2 to measure RTT for FAST TCP tests. What distinguishes **thrulay** from other network performance tester (e.g. Iperf [4], Netperf [5]) is that **thrulay** reports not only throughput but also delay information of a network. Delay measurement in performance tests is useful, especially for delay-based TCP flavors (e.g. FAST, Vegas). **Thrulay** measures network throughput and round trip delay by sending a bulk TCP stream over it. Starting from version 0.5, **thrulay** supports UDP tests. It measures one-way delay by sending a Poisson stream of very precisely positioned UDP packets [6].

The project (`http://thrulay-hd.sourceforge.net/`) is based on thrulay version 0.6. It aims to improve the original **thrulay** program. The remainder of the report describes enhancements to **thrulay** that have been accomplished.

2 Accomplishments

The most important enhancements to **thrulay** are API for programmatic execution of tests, testing with multiple streams, and online calculation of statistics in UDP tests. These enhancements either need significant change in software architecture or require carefully designed algorithms. Other enhancements involve more or less programming, including IPv6 support, porting to popular platforms, client authentication and others. We will detail the major ones and have a brief introduction of the others in this section.

2.1 API for Programmatic Execution

Existing testers usually provide command line tools for running performance tests. Although the console outputs provide sufficient information for later performance tuning, it is sometimes more desirable to provide an API for programmatic execution of tests so that the timing can be better controlled by a wrapper application such as BWCTL [1]. We implement the **thrulay** API that falls into three categories as shown in Table 1. For detailed information on arguments and return values of the API, please refer to `man(3)`.

Table 1: Thrulay API

Library Management	`thrulay_init`	`thrulay_exit`		
Test Management	`thrulay_open`	`thrulay_start`	`thrulay_wait`	`thrulay_close`
Test Information	`thrulay_get_setup`	`thrulay_get_report`	`thrulay_err_msg`	

The library management functions initialize or destroy a library context. The library is designed to be thread-safe and can be used in a multi-threaded environment. Each thread has a copy of the library context and the following test management calls do not affect library contexts of other threads. Once the **thrulay** library is initialized, the program can use `thrulay_open` to setup a test, `thrulay_start` to run a test, `thrulay_wait` to poll status of a test, wait for a test to finish or stop a test, and `thrulay_close` to clean up states of a test. Information of a test setup is available through `thrulay_get_setup`. Both intermediate and final test reports can be obtained from `thrulay_get_report`.

The `thrulay` library is asynchronous in a sense that `thrulay_start` does not block. It forks off a test process and returns immediately. The calling (control) process and the test process communicate through pipes and shared memory as illustrated in Figure 1.

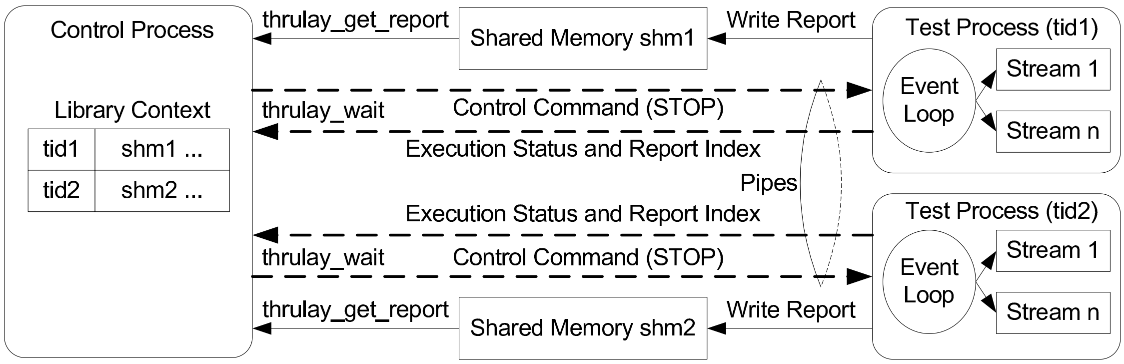

Figure 1: Communication between the control process and test processes in **libthrulay**.

2.2 Parallel TCP and UDP Streams

Running multiple TCP or UDP streams are useful in tuning network performance. For example, if the total aggregate bandwidth is more than what an individual stream gets, something is wrong. Either the TCP window size is too small, or the OS's TCP implementation has bugs, or the network itself has deficiencies [4]. In order to make multiple simultaneous connections between the client and server, most network performance testers choose a multi-process or multi-threaded architecture. This is natural because concurrent streams are independent. However, by choosing a multi-process architecture, you ask for more context switches that are expensive. By choosing a multi-threaded architecture, scheduling of concurrent threads are handed to the thread library. Unfortunately, there is no well-performing thread implementation available in practice. For example, the pthread library on Solaris cannot schedule parallel streams fairly. There are streams not scheduled to send any packet in a report interval at all.

We employ the single-process event-driven (SPED) [9] architecture to perform concurrent processing of multiple streams. SPED uses the `select` loop and non-blocking sockets to perform asynchronous `read` and `write`. The scheduling routine governs the execution of multiple state machines, each corresponding to a testing stream. Figure 2 depicts the SPED architecture in a multi-stream UDP test.

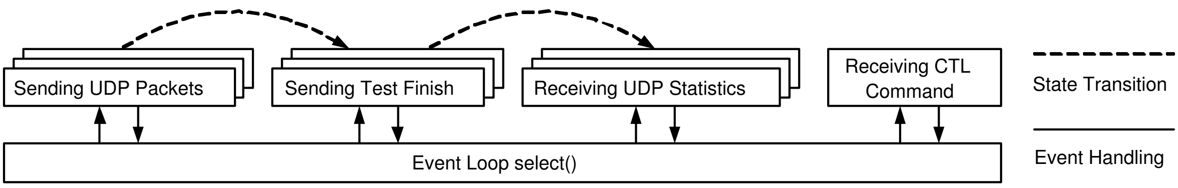

Figure 2: The SPED architecture in a multi-stream UDP test.

In each iteration, the scheduling routine performs a `select` to check for ready I/O events. If a `read` or `write` in a particular stream is ready, it completes the corresponding operation and initiates the next step, if appropriate. To ensure fairness, Round Robin scheduling is implemented for selecting one stream from multiple outstanding streams.

2.3 Online Calculation of Statistics for UDP Tests

Statistics in UDP tests such as n-reordering [2], quantiles of one way delay, number of duplicated packets and number of loss periods [7] must be calculated online, i.e. they must be computed by single-pass algorithms with $\mathcal{O}(1)$ memory requirements. Otherwise, you have to store information (e.g. delay, sequence number) for all packets and process the complete data set after the test is over. Although offline processing is accurate, it takes a large amount of memory and the client has to wait a long time for the results.

For n-reordering, we adapt the code from [2] and report up to 100-reordering. For 0-quantile (minimum), 0.5-quantile (median) and 0.95-quantile of one-way delay, we implement the framework presented in [8], using "Munro and Paterson" as the COLLAPSE policy. Approximation ratio in our implementation is 0.5%. A circular queue is devised for counting the number of duplicated packets and the number of loss periods online. Figure 3 illustrates the circular queue scheme.

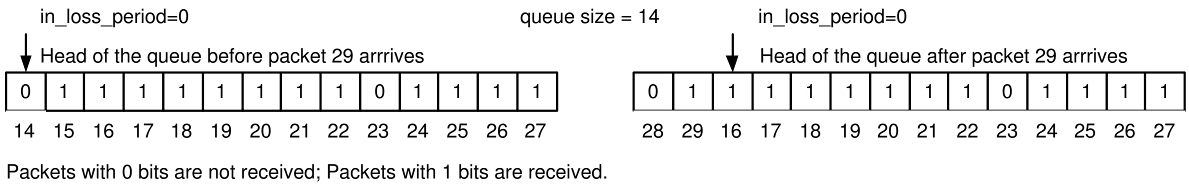

Figure 3: A circular queue scheme for duplicates and loss periods.

In the above example, when packet 29 arrives, head of the queue has to move forward by 2 bits to accommodate packet 29. As a consequence, packet 14 and 15 will be outdated. Since packet 14 is not received yet and we are not in a loss period, number of loss periods is incremented by 1 and we enter a loss period. However, packet 15 has been received, we go back to non-loss period again. When a packet arrives whose corresponding bit is set in the circular queue, number of duplicates is incremented by 1. When a packet arrives with a sequence number less than that at the head of the queue, either this is a duplicated packet or we have miss-calculated it as a loss when moving forward the head of the queue previously. Because of the $\mathcal{O}(1)$ memory requirements, we have no way to distinguish between them. In our implementation, the queue size is 1024×32 (number of bits in a `u_int32_t`), we hope this situation does not occur. If it does, the trouble packet is simply thrown away.

2.4 Other Enhancements

Compared with those "big" enhancements described in previous sections, features that will be introduced in this section do not require significant changes to the **thrulay** architecture, but they enable **thrulay** to play in a broader stage.

1. Porting from Linux to other popular platforms such as Solaris, FreeBSD and Mac OS X. GNU auto{conf, make, header} tools are used to automate the build process;

2. Integration with the **fasttime** project. Users can make a choice between the system `gettimeofday` and `fasttime_gettimeofday` simply by providing a configuration option. `fasttime_gettimeofday` makes use of the TSC register to obtain the current time of day. The primary advantage of using the **fasttime** library is to avoid making a system call, which involves two context switches.

3. Adaption of the socket code for supporting IPv6, reporting MSS/MTU and setting the TOS byte;
4. Authentication of client IP address in CIDR syntax. The **thrulay** server can setup an access control list so that only clients with desired IP addresses can have the connectivity;

3 Acknowledgements

This project is supported by the Google Summer of Code program [3]. The author greatly acknowledges Jeff W. Boote and Stanislav Shalunov for their outstanding technical leadership and valuable feedbacks.

References

[1] Bandwidth Control (BWCTL). `http://e2epi.internet2.edu/bwctl/`.

[2] Definition of IP Packet Reordering Metric. `http://www.internet2.edu/~shalunov/ippm/draft-shalunov-reordering-definition-02.txt`.

[3] Google Summer of Code Program. `http://code.google.com/summerofcode.html`.

[4] Iperf. `http://dast.nlanr.net/Projects/Iperf/`.

[5] Netperf. `http://www.netperf.org/netperf/NetperfPage.html`.

[6] Thrulay: network capacity tester. `http://www.internet2.edu/~shalunov/thrulay/`.

[7] R. Koodli and R. Ravikanth. One-way Loss Pattern Sample Metrics. RFC 3357, Aug. 2002.

[8] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *SIGMOD Rec.*, 27(2):426–435, 1998.

[9] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.