CSE 692, Spring 2015
Kelong Wang, 109600226

Project Report

Modifying Kernel TCP for Better Congestion Control with SPDY

1. Introduction

SPDY[1] is a TCP-based network protocol to manipulate HTTP traffic. It aims to reduce the load latency of web page. The main feature to achieve the goal is multiplexing, which allows multiple concurrent transportation streams over a single TCP connection. Traditional HTTP/1.1 clients typically open multiple persistent connections in parallel to improve the throughput because each connection allows only one outstanding request at a time.

While SPDY with a single connection helps on small objects, which is studied in SPDY paper[2], it degrades under high loss because its congestion window is reduced aggressively compared to HTTP which reduces the window on only one of the parallel connections. To mitigate the impact of congestion control algorithm for SPDY connection, the paper proposes a simple TCP modification in Linux which reduces the window less aggressively than default rate.

In this project, we implement the proposed modification as a loadable Linux kernel module which is configurable in command line and we set up the SPDY test bed to evaluate the throughput under different packet loss and congestion window backoff rate.

The rest of this report is organized as follows. Section 2 reviews the necessary background and the TCP modification. Section 3 describes the implementation detail. Section 4 presents the evaluation and a brief discussion. We conclude in section 5. And Section 6 provides a documentation of the module.

2. Approach

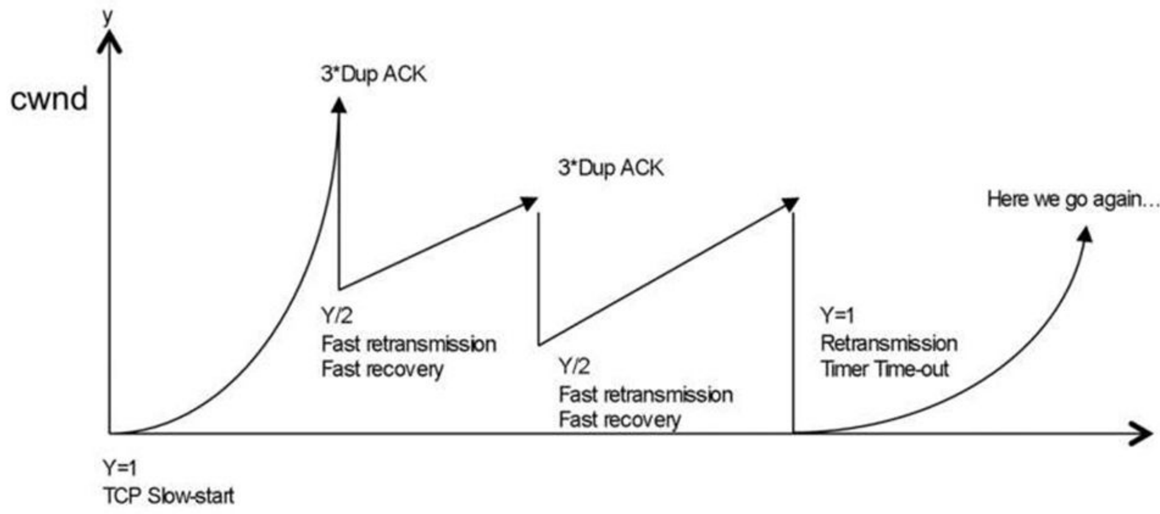
In this section, we first introduce the TCP congestion control standard algorithm followed by CUBIC variation which is the default algorithm in Linux. Then based on CUBIC we describe the modification originally proposed in SPDY paper. We also talk about Linux TCP congestion control framework.

2.1 TCP Congestion Control Background

For a TCP connection, we usually say congestion window (*cwnd*) is halved on packet loss. Specifically, as per RFC 5681 TCP Congestion Control[3] algorithm, slow start threshold (ssthresh) is set to 50% of the current congestion window under packet loss; congestion window, however, is not always halved. Upon a retransmission timeout (RTO) that indicates a

[1] SPDY: An experimental protocol for a faster web http://dev.chromium.org/spdy/spdy-whitepaper
[2] How Speedy is SPDY? NSDI'14 http://www3.cs.stonybrook.edu/~arunab/cs692/spdy.pdf
[3] RFC 5681 https://tools.ietf.org/html/rfc5681

severe congestion, *cwnd* is reset to one segment such that TCP will perform slow start regardless the value of initial congestion window size (*initcwnd*). Upon duplicate ACKs which suggests a slight congestion, TCP performs fast retransmit and fast recover, and *cwnd* is set to the value of ssthresh so it is halved as well. In this project the packet loss rate injected is up to 2% and supposedly most losses are from duplicate ACKs, resulting in halving the window.



Congestion window on duplicate ACKs and RTO[4]

2.2 CUBIC

CUBIC[5] is the default congestion control algorithm in Linux. The window growth function of CUBIC is a cubic function which is designed for high speed network. When congestion window is small, it still behaves the same as the original Reno algorithm. On packet loss, the window reduction calculation is simple: $cwnd = \beta \times cwnd$. In this project, we call β the backoff rate and $1 - \beta$ the reduction rate. Unlike standard TCP, the default CUBIC backoff (reduction) rate is 70% (30%) instead of 50% (50%).

2.3 Modification

Before we describe the modification, we first take a look at window reduction for SPDY connection and parallel HTTP connections on packet loss.
For SPDY:

$$cwnd \rightarrow \beta \times cwnd$$
$$reduction\% = 1 - \beta$$

For n HTTP connections:

$$n \times cwnd' \rightarrow (n-1) \times cwnd' + \beta \times cwnd'$$
$$reduction'\% = (1 - \beta) / n$$

[4] Figure from http://what-when-how.com/qos-enabled-networks/traffic-types-qos-enabled-networks-part-2/
[5] CUBIC: A New TCP-Friendly High-Speed TCP Variant
http://netsrv.csc.ncsu.edu/export/cubic_a_new_tcp_2008.pdf

As we can see, the reduction of SPDY is n times of parallel HTTP. In practice, the browsers usually open 6 connections for a single domain to fetch the objects in parallel. As a mitigation, the SPDY paper proposes to modify the CUBIC reduction rate from $1-\beta$ to $(1-\beta)/n$ to make it less aggressive. The backoff rate is therefore increased to $1-(1-\beta)/n$ from β. While the paper set $n=6$ for the experiment, one practical goal of this project is to make it configurable via command line tool without re-compiling the kernel or reloading the module. Note that the value representation in CUBIC is not a percent number directly. We detail the implementation in Section 3.

2.4 Linux Kernel Module

Many components in Linux kernel can be built as pluggable module which is loaded dynamically on demand. Pluggable congestion module[6] introduced in Linux 2.6 allows user to switch between different congestion control algorithms on the fly. Originally, the backoff rate in CUBIC can be configured during load time. We add support to configure the backoff rate during runtime. Sysctl[7] is a suitable mechanism to achieve this.

3. Methodology

We make a copy of existing *cubic* code in Linux which is called *cubic-spdy*. With the Linux kernel header files, we build it as a pluggable congestion module (*tcp_cubic-spdy.ko*). Then we change the backoff rate by adding a factor n which is called number of connections. At last we put n and initial backoff rate β (in its integer form) into Sysctl table so that they can be modified with *sysctl* command. To confirm the modification, congestion window upon each ACK is output into kernel ring buffer which can be observed with *dmesg*[8] command.

3.1 Parameters

Initially, *cubic* code uses *beta* and *BETA_SCALE* to compute the backoff rate β. By default *beta* is 717 and *BETA_SCALE* constant is 1024, so $\beta = 717/1024 = 70\%$. We add a new parameter *beta_n_conn* to represent n discussed previously. Then we introduce a new internal variable *beta_effective* to replace *beta*. But *beta* variable still exists, acting as the initial backoff rate (calculated together with *BETA_SCALE* constant). Consequently, *beta_effective* and β are computed as below:

$$beta\_effective = 1024 - (1024 - beta)/beta\_n\_conn$$

$$\beta = beta\_effective / BETA\_SCALE = beta\_effective / 1024$$

The table lists produced backoff rate by different combinations of n and *beta*. Notice that default *cubic* is a special case of *cubic-spdy* when $n = 1$.

[6] Pluggable congestion avoidance modules http://lwn.net/Articles/128681/

[7] https://www.kernel.org/doc/Documentation/sysctl/

[8] in released code I comment out this debug message. Section 6 has instruction to bring it back.

n *(beta_n_conn)*	*beta*	*beta_effective*	backoff rate β	reduction rate
1	717	717	70%	30%
2	717	871	85%	15%
3	717	922	90%	10%
4	717	948	92%	8%
5	717	963	94%	6%
6	717	973	95%	5%
1	512	512	50%	50%
2	512	768	75%	25%
3	512	854	83%	17%
4	512	896	87%	13%
5	512	922	90%	10%
6	512	939	91%	9%

3.2 Packet Loss Injection
When injecting packet loss, we choose uplink of the server (or downlink of the client) because it will cause the gap of receiver window and thereby duplicate ACKs. Otherwise if we drop packets for downlink of the server (or uplink of client), the subsequent ACK will achieve cumulative acknowledgement so the server will not detect such loss.

4. Evaluation

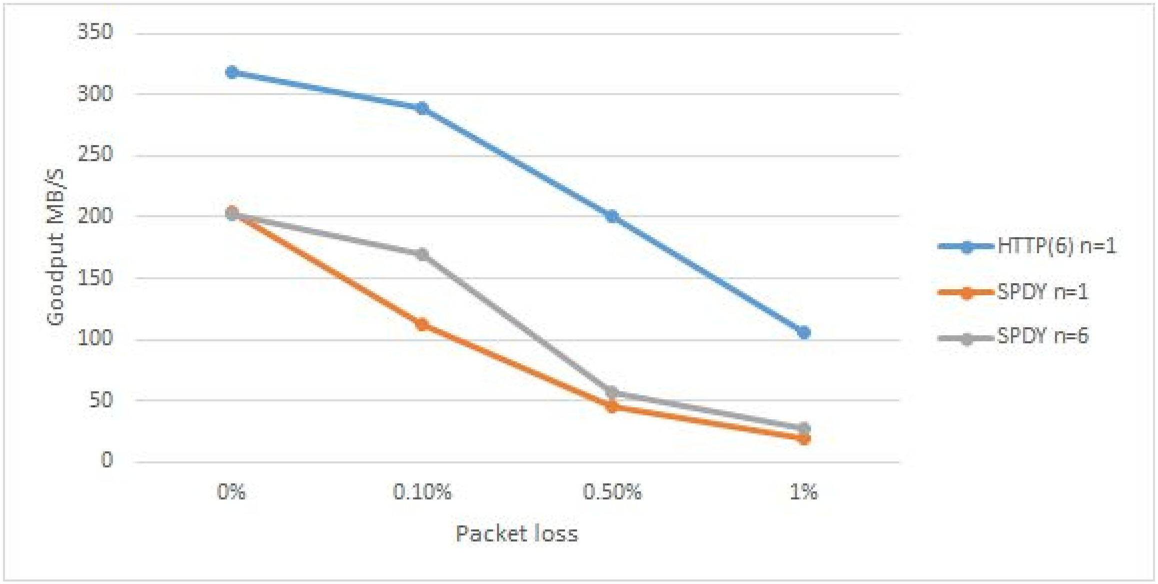
4.1 Setup
We create two virtual machines as SPDY / HTTP client and server. Both nodes have one dual-core processor and 4GB memory, running Ubuntu 14.04.2 LTS with 3.16 kernel. We install Apache 2.4.7 with mod_spdy[9] (SPDY/3) on server with SSL disabled. On client we use *h2load*[10] as SPDY client and *axel* as multi-thread HTTP client. *tc*[11] is used to inject packet loss on server side. To make sure we do not have network bottleneck in virtual machine environment, we use *iperf* to measure the maximum bandwidth between two nodes and the result is about 5Gb/sec. During the benchmark, we create ramdisk to host the files and therefore the hard drive will not

[9] mod_spdy ported to Apache 2.4 https://github.com/eousphoros/mod-spdy
[10] https://nghttp2.org/documentation/h2load-howto.html
[11] http://linux.die.net/man/8/tc

be the bottleneck either. We try to modify initial congestion window size (*initcwnd*) with *ip*[12] command but it does not take effect for unknown reason. So when we perform the benchmark, we run the experiment with large file for long enough time to minimize the impact of *initcwnd*. For SPDY client, we do increase the receiver window by 6 times so that the single connection has equal window as parallel HTTP connections. We do not impose latency or bandwidth limitation. *cubic-spdy* module is enabled on server side.

4.2 HTTP vs. SPDY

First we show the throughput decrease for HTTP with 6 parallel connections and single SPDY connection after injecting packet loss. In the figure, when $n = 1$, server defaults to original cubic. When $n = 6$, the backoff rate is only 95%. The client downloads a 1GB file from server.

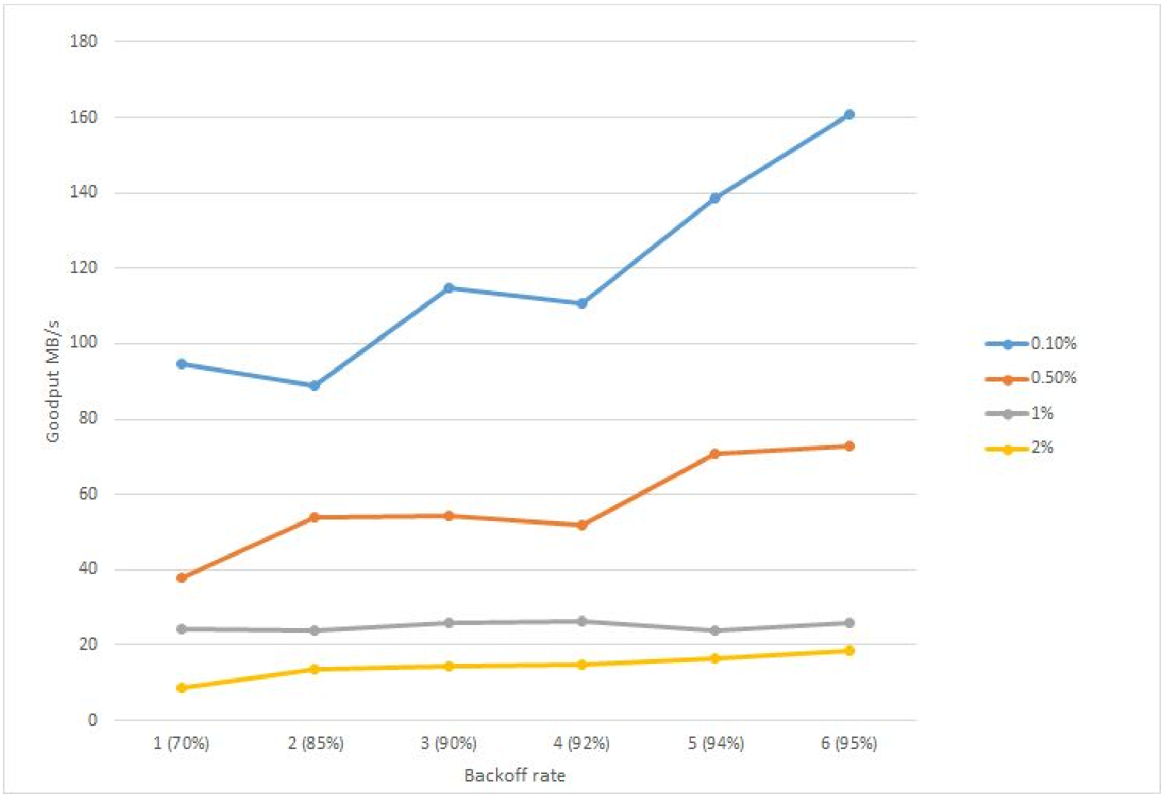


The figure indicates that HTTP performance does not suffer under low (0.1%) packet loss. But SPDY connection with default cubic suffers a lot (50%). When we increase the backoff rate from default 70% ($n = 1$) to 95% ($n = 6$), we observe the throughput increase under low packet loss. While under high packet loss ($\geq 0.5\%$), the throughput increase is not considerable.

[12] http://www.cdnplanet.com/blog/tune-tcp-initcwnd-for-optimum-performance/

4.3 Different Backoff Rate

We also perform similar benchmark for single SPDY connection to download a large file under different backoff rate (n from 1 to 6) and packet loss rate (from 0.1% to 2%).



Again under low packet loss, the throughput increase is obvious. Under high packet loss, the increase is negligible. We have two hypotheses which need further study. First the retransmission overhead increases and possibly makes the channel more congested. Second the possibility of RTO becomes perceptible and therefore the slow start is more likely to happen.

5 Conclusion

In this project, we add some configurable parameters to the *cubic* TCP congestion control module in Linux and present its impact for large file download with SPDY under packet loss environment. The module is designed as a practical tool to conduct TCP experiments. It is also a workaround or small optimization for SPDY under particular condition and could be a start point to seek SPDY-friendly algorithm. As mentioned in SPDY paper, a solution for real networks requires further consideration and extensive evaluation.

6. Usage

We demonstrate the usage of *cubic-spdy* on Ubuntu 14.04 LTS.

6.1 Compile

Prepare Linux headers:

```
$ sudo apt-get install linux-headers
```

Checkout the code:

```
$ git clone https://github.com/wkl/cubic-spdy.git
```

Build kernel module:

```
$ cd cubic-spdy
$ make
```

6.2 Enable

Load module:

```
$ sudo insmod tcp_cubic-spdy.ko
```

Switch to cubic-spdy:

```
$ sudo sysctl -w net.ipv4.tcp_congestion_control=cubic-spdy
```

You can use `enable.sh` instead with single command:

```
$ ./enable.sh
```

6.3 Disable

Switch back to default *cubic*:

```
$ sudo sysctl -w net.ipv4.tcp_congestion_control=cubic
```

Unload module if you want to compile the code and load a new version:

```
$ sudo rmmod tcp_cubic-spdy
```

You can use `disable.sh` instead:

```
$ ./disable.sh
```

Caution

- You cannot unload this module if *cubic-spdy* is the active algorithm. Use *sysctl* to switch to other modules (e.g., *cubic*) first.
- You cannot unload this module if there exists established TCP connection that uses *cubic-spdy*, **including your *ssh* connection** if you login after algorithm is enabled. You may need to logout from your *ssh* session and use your virtual machine console to switch and unload module.
- The recommended way for module development is to login with *cubic-spdy* disabled so that you have a normal *ssh* session. Then you can enable and disable module without limitation.

6.4 Configure

Change initial backoff rate if you like $(1 \le beta \le 1023)$:

```
$ sudo sysctl -w net.ipv4.cubic_spdy_beta=512  # default is 717
```

Change n, “number of parallel HTTP connections” $(1 \le n \le 10)$:

```
$ sudo sysctl -w net.ipv4.cubic_spdy_beta_n_conn=6
```

After you enable the module or change above two parameters, you can view current configuration with:

```
$ dmesg
```

or

```
$ tail /var/log/syslog
```

The output will look like:

```
beta/n: 512/6, effective_beta: 939, backoff rate: 91%
```

Note that the change will be applied to existing connections as well.

6.5 Debug

You may want to observe or confirm the change of congestion window in real time. Currently, you can remove the comment symbol starting with `INFO` in function `bictcp_cong_avoid()` and `bictcp_recalc_ssthresh()`. Then compile the code and watch *syslog* when enabled. Refer to CUBIC paper[13] for detailed explanation of those functions.

6.6 Practical Usage

Because *sysctl* changes the algorithm for all connections, it is probably more practical to use *cubic-spdy* on a per-connection basis, for example only for SPDY connection. *setsockopt(2)* can be used to set the congestion control algorithm for a particular socket[14].

[13] CUBIC: A New TCP-Friendly High-Speed TCP Variant
http://netsrv.csc.ncsu.edu/export/cubic_a_new_tcp_2008.pdf

[14] http://sgros.blogspot.com/2012/12/controlling-which-congestion-control.html