NS-2 TCP-Linux: An NS-2 TCP Implementation with Congestion Control Algorithms from Linux

David X. Wei
Department of Computer Science
California Institute of Technology
davidwei@acm.org

Pei Cao
Department of Computer Science
Stanford University
cao@theory.cs.stanford.edu

ABSTRACT

This paper introduces *NS-2 TCP-Linux*, a new NS-2 TCP implementation that embeds the source code of TCP congestion control modules from Linux kernels. Compared to existing NS-2 TCP implementations, *NS-2 TCP-Linux* has three improvements: 1) a standard interface for congestion control algorithms similar to that in Linux 2.6, ensuring better extensibility for emerging congestion control algorithms; 2) a redesigned loss detection module (i.e. Scoreboard) that is more accurate; and 3) a new event queue scheduler that increases the simulation speed. As a result, *NS-2 TCP-Linux* is more extensible, runs faster and produces simulation results that are much closer to the actual TCP behavior of Linux. In addition to helping the network research community, *NS-2 TCP-Linux* will also help the Linux kernel community to debug and test their new congestion control algorithms.

In this paper, we explain the design of *NS-2 TCP-Linux*. We also present a preliminary evaluation of three aspects of *NS-2 TCP-Linux*: extensibility to new congestion control algorithms, accuracy of the simulation results and simulation performance in terms of simulation speed and memory usage.

Based on these results, we strongly believe that *NS-2 TCP-Linux* is a promising alternative or even a replacement for existing TCP implementations in NS-2. We call for participation to test and improve this new TCP implementation.

Categories and Subject Descriptors

C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks; I.6.3 [**Simulation and Modeling**]: Application

General Terms

Simulation, Performance

WNS2'06, October 10, 2006, Pisa, Italy

Keywords

NS-2, TCP, Congestion Control, Linux

1. MOTIVATION

The TCP modules in NS-2 [7] were originally based on source code of the BSD kernel in the 1990s. Over the years, NS-2 TCP modules have tremendously helped the research community to analyze and understand TCP behavior, and led to the development of several new congestion control algorithms.

However, as major operating systems have evolved during the past decade, the TCP modules in NS-2 have deviated significantly from the mainstream operating systems such as FreeBSD and Linux. This leads to many difficulties in using NS-2:

- Extensibility: As NS-2's code structure deviates from that of mainstream operating systems, it becomes harder and harder to implement the NS-2 counterpart of a Linux algorithm. For new algorithm designers, it is a huge burden to implement the same algorithms in both NS-2 and a real system such as Linux.
- Validity of NS-2 results: As many improvements have been implemented in Linux but not in NS-2, the performance predicted by NS-2 simulation has become significantly different from the Linux performance. This problem has recently been noted in the literature [21].
- Simulation speed: NS-2 users often suffer from long simulation time when they try to simulate scenarios with LDFP (long-distance fat-pipe) networks. [6] shows some examples in which the simulator might take up to 20 hours to finish a 200-second simulation.

As a result, researchers these days often find it easier to use the Linux kernel and Dummynet [28] as tools for protocol research than to use NS-2. However we argue that, as a simulator, NS-2's utility in supporting network research is simply irreplaceable, especially with its flexibilities in topologies and scales.

Hence, we design *NS-2 TCP-Linux* in the spirit of closing the gap between the implementation community which works on Linux system and the analysis community which currently uses NS-2 as a tool. *NS-2 TCP-Linux* has three design goals:

- Enhance extensibility by allowing users to import congestion control algorithms directly from the Linux source code;

Name	Meaning	Equivalent in NS-2 *TCPAgent*
snd_ ssthresh	slow-start threshold	ssthresh_
snd_ cwnd	integer part of the congestion window	trunc(cwnd_)
snd_ cwnd_cnt	fraction of congestion window	trunc(cwnd_^2) %trunc(cwnd_)
icsk_ca_ priv	a 512-bit array to hold per-flow state for a congestion control algorithm	n/a
icsk_ca_ ops	a pointer to the congestion control algorithm interface	n/a

Table 1: Important variables in *tcp_sk*

- Provide simulation results that are close to Linux performance;
- Improve the simulation speed for LDFP networks.

Our preliminary results show that, by carefully redesigning the NS-2 TCP module, these goals can be achieved.

2. TCP CONGESTION CONTROL MODULE IN LINUX 2.6

This section gives a brief introduction to the TCP implementation in Linux 2.6. We focus on interfaces and performance-related features, which strongly influence the design of *NS-2 TCP-Linux*.

The Linux kernel introduced the concept of *congestion control modules* in version 2.6.13 [5, 20]. A common interface is defined for congestion control algorithms. Algorithm designers can implement their own congestion control algorithms as Linux modules easily. As of version 2.6.16-3, Linux has incorporated nine congestion control algorithms in the official release version and there are new implementations coming up as well.

2.1 Interface for Congestion Control Modules

In Linux, all the state variables for a TCP connection are stored in a structure called *tcp_sk*. Table 1 lists the most important state variables for congestion control.

When a congestion control decision is to be made (e.g., upon the arrival of a new acknowledgment or when a loss is detected), the Linux kernel calls the corresponding functions in the congestion control module, via pointers in the *icsk_ca_ops* structure, as shown in Figure 1(a). All functions in the module take the address of *tcp_sk* as a parameter so that the functions have the flexibility to change the TCP connection's state variables.

Three functions in the *icsk_ca_ops* structure are required to be implemented to ensure the basic functionalities of congestion control. They are:

- *cong_avoid*: This function is called when an ACK is received. This function is expected to implement the changes in the congestion window for the normal case, without loss recovery. In TCP-Reno (Reno) [18, 24], this means slow-start and congestion avoidance. Figure 1(b) is an example of this function.
- *ssthresh*: This function is called when a loss event occurs. It is expected to return the slow-start threshold after a loss event. For example, the return value is half of *snd_cwnd* in Reno, as shown in Figure 1(c).
- *min_cwnd*: This function is called when a fast retransmission occurs, after the *ssthresh* function. It is expected to return the value of congestion window after a loss event. In Reno, the return value is *snd_ssthresh*, as shown in Figure 1(d).[1]

As an example, Figure 1 is a very simple implementation (Reno) for this interface.[2]

More complicated congestion control algorithms might require more operations such as obtaining high resolution RTT samples. These advanced functions are introduced in [5, 2].

2.2 Differences in TCP Implementations between Linux and NS-2

The Linux TCP implementation [29] closely follows the relevant RFCs. However, there are a few major differences between the existing NS-2 implementation and Linux implementation, leading to a discrepancy between the NS-2 simulation results and Linux performance. Here we list three major differences that might result in significant performance discrepancy:

1. SACK processing [25]: current NS-2 TCP (e.g. *NS-2 TCP-Sack1* and *NS-2 TCP-Fack*) times out when a retransmitted packet is lost again. Linux SACK processing may still recover if a retransmitted packet is lost. This difference leads to better performance by the Linux kernel in scenarios with high loss rates.
2. Delayed ACK: the Linux receiver disables delayed ACKs in the first few packets. This difference leads to faster congestion window growth when the window size is small.
3. D-SACK [19]: current NS-2 TCP implementations do not process D-SACK information; Linux uses D-SACK information to infer the degree of packet reordering in the path and adjusts the duplicated ACK threshold so that Linux has better performance in scenarios with severe packet reordering.

The design of *NS-2 TCP-Linux* tries to eliminate these differences to achieve better accuracy.

3. DESIGN OF *NS-2 TCP-LINUX*

We believe that NS-2 will benefit from a new TCP implementation which conforms to Linux *congestion control module* interface. The benefits are two-fold. First, the research community can use NS-2 to analyze Linux algorithms, without implementing the NS-2 version of a Linux

[1]Linux has a complicated rate-halving process and *min_cwnd* is used as the lower bound of the congestion window in a rate-halving process after a loss event. In NS-2, *NS-2 TCP-Linux* has a simplified version of rate-halving, and the congestion window can be set to *min_cwnd* directly.

[2]A complete congestion control module in Linux also requires routine module-support functions such as *module register function*. These are not necessary for a TCP implementation in *NS-2 TCP-Linux*.

```
/* Create a constant record for the
 * interface function pointers of this
 * congestion control algorithm */
struct tcp_congestion_ops simple_reno = {
       .name       = "simple_reno",
       .ssthresh   = nr_ssthresh,
       .cong_avoid = nr_cong_avoid,
       .min_cwnd   = nr_min_cwnd
};
```

(a) Data structure declaration

```
/* This function increases congestion window for
 * each acknowledgment
 */
void nr_cong_avoid (struct tcp_sk *tp, ...)
{
   if (tp->snd_cwnd < tp->snd_ssthresh)
   {
      //slow start
      tp->snd_cwnd++;
   } else {
      //congestion avoidance
      if (tp->snd_cwnd_cnt < tp->snd_cwnd)
      {
         // the increment is not enough for 1 pkt
         // we increase the fraction of cwnd
         tp->snd_cwnd_cnt++;
       } else {
         // we can increase cwnd by 1 pkt now.
         tp->snd_cwnd++;
         tp->snd_cwnd_cnt = 0;
       }
   }
}
```

(b) Window increment function

```
/* This function returns the slow-start threshold
 * after a congestion event.
 */
u32 nr_ssthresh(struct tcp_sk *tp)
{
   return max(tp->snd_cwnd/2,2);
}
```

(c) Slow-start threshold adjustment function

```
/* This function returns the congestion window
 * after a congestion event -- it is called AFTER
 * the function nr_ssthresh in Figure 1 (c).
 */
u32 nr_min_cwnd(struct tcp_sk *tp)
{
   return    tp->snd_ssthresh;
}
```

(d) Window reduction function

Figure 1: A sample implementation (simplified Reno) of the Linux congestion control module interface. *NS-2 TCP-Linux* uses the same interface.

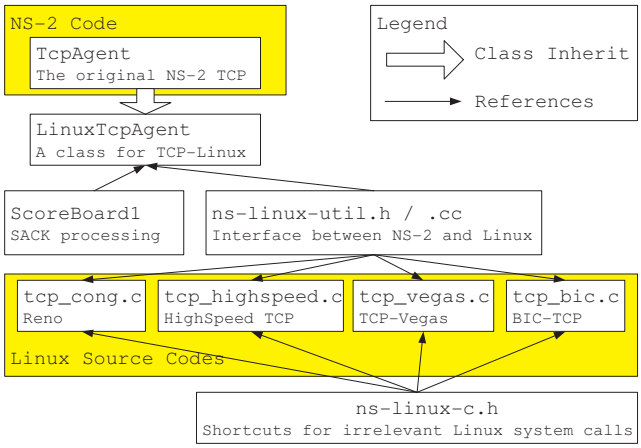

Figure 2: Code structure of *NS-2 TCP-Linux*
The boxes in yellow shades are components from existing NS-2 source code or the Linux kernel. The four boxes outside the shades are *NS-2 TCP-Linux* components.

algorithm. This leads to an improvement in both productivity and accuracy. Second, NS-2 can also be a tool for the Linux community to debug and test their new congestion control algorithms. This leads to more reliable and better understood implementations.

3.1 NS-2 interface for Linux congestion control modules

NS-2 TCP-Linux introduces a new TCPAgent, *LinuxTCPAgent*, that implements the Linux congestion control module interface. *LinuxTCPAgent* loosely follows the design of Linux ACK processing (the *tcp_ack* function in Linux), including routines for RTT sampling, SACK processing, fast retransmission and transmission timeout. It allows the actual Linux kernel source code to be used as implementations of different congestion control algorithms.

LinuxTCPAgent uses two glue mechanisms to facilitate whole-file imports from Linux kernel source code:

- ns-linux-util.h and ns-linux-util.cc: a set of data structure declarations that allow the C++ code in NS-2 to interact with the C code in Linux;
- ns-linux-c.h and ns-linux-c.c: a set of macros that redefine many Linux system calls that are not relevant to congestion control.

Together, they serve as a highly simplified environment for embedding Linux TCP source code. The code structure is shown in Figure 2.

For users, *LinuxTCPAgent* supports a command line option to select congestion control modules. Thus, users of *NS-2 TCP-Linux* can easily include new congestion control algorithms from Linux source code and run simulation with existing NS-2 scripts after minimal changes. This process is described in detail in [2, 1].

3.2 *Scoreboard1*: More Accurate Loss Detection

Scoreboard1 is a new scoreboard implementation that combines the advantage of *Scoreboard-RQ* in NS-2 and the Linux SACK processing routine (sacktag_write_queue).

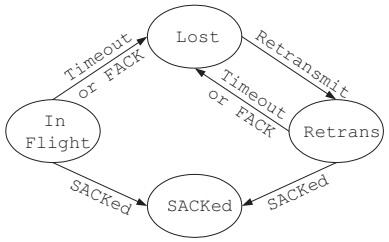

Figure 3: State machine of each packet

As in Linux SACK processing, each packet in the retransmission queue is in one of the four possible states: InFlight, Lost, Retrans or SACKed. The state transition diagram is shown in Figure 3.

A packet that is sent for the first time, but not acknowledged and not considered lost, is in InFlight state. It enters the SACKed state when the packet is selectively acknowledged by SACK, or enters the Lost state when either a retransmission timeout occurs, or the furthest SACKed packet is more than 3 packets ahead of it (FACK policy). When a lost packet is retransmitted, it enters the Retrans state.

Finally, the most challenging part is the transition from the Retrans state to the Lost state, which happens when a retransmitted packet is lost again. We approximate the Linux implementation: When a packet is retransmitted, it is assigned a *snd_nxt* (similar to *Scoreboard* in *NS-2 TCP-Fack*) which records the packet sequence number of the next data packet to be sent. Additionally, it is assigned a *retrx_id* which records the number of packets that are retransmitted in this loss recovery phase, as shown in the first two nodes in Figure 4. The (*snd_nxt*,*retrx_id*) pair helps to detect the loss of a retransmitted packet in the following ways:

1. When another InFlight packet is SACKed or acknowledged with its sequence number higher than *3+snd_nxt* of the retransmitted packet, the retransmitted packet is considered lost; or

2. When another Retrans packet is SACKed or acknowledged with its own *retrx_id* higher than *retrx_id+3*, the retransmitted packet is considered lost.[3]

With the definition of per-packet state, *Scoreboard1* has a clear image of the number of packets in the network (which is the sum of the numbers of packets in the InFlight and Retrans states). This number is compared against the congestion window to enforce congestion control.

To improve the speed of SACK processing, *Scoreboard1* incorporates a one-pass traversing scheme extended from *Scoreboard-RQ*. *Scoreboard1* organizes all the packets into a linked list. Each node of the linked list is either a single packet in the Retrans state, or a block of packets in other states, as shown in Figure 4.

[3]Strictly speaking, the Linux implementation uses a slightly different scheme for the this case: when a packet is acknowledged, and if the current timestamp is higher than an unacknowledged packet's transmission timestamp plus RTO, the unacknowledged packet is considered to be lost (head_timeout). This scheme has more overhead in both simulation speed and memory usage. The *retrx_id* scheme in *NS-2 TCP-Linux* is an approximate version of the head_timeout scheme. This difference might affect the throughput prediction when the number of packets in flight is smaller than 3.

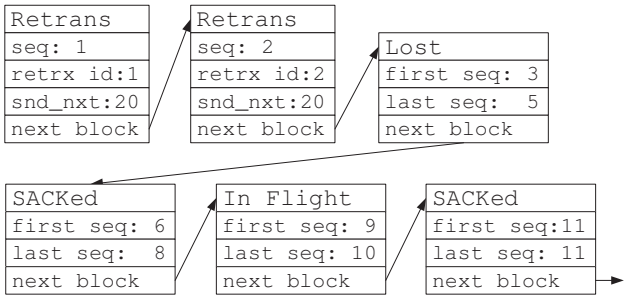

Figure 4: SACK queue data structure

The linked list allows *Scoreboard1* to traverse the retransmission queue only once every acknowledgment, regardless of the number of SACK blocks in the acknowledgment.

3.3 *SNOOPy*: a Faster Event Queue

The current NS-2 (Version 2.29) scheduler uses a calendar queue [13], also called a timing wheel, to store simulation events. A calendar queue is essentially a hash table of simulation events, with the events' time as the keys; each bucket has a sorted linked list to store events whose times fall on the same "second" (which is the hash value) but possibly in different "minutes". To insert a new event, the queue calculates the event's destination bucket via the hash value of its time, and the event is inserted into the in-order position in the destination bucket via linear search along the linked list. To dequeue the next event, the queue traverses all the buckets to find the bucket with the earliest event. The calendar queue can achieve an averaged complexity of $O(1)$ in both dequeue and enqueue operations, if the bucket width (i.e., the length of a "second") is properly set [13].

Clearly, the efficiency of the calendar queue is determined by the width of each bucket and the number of buckets in the hash table, and demands a trade-off between the two: If the width of a bucket is too large, inserting an event takes too long. If the number of buckets is too high, dequeuing an event takes too long. It has been suggested [10] that the bucket size should be dynamically set to the average interval in the fullest bucket. The NS-2 calendar queue takes this suggestion in setting the bucket width.[4]

However, this technique only works if the events are evenly distributed in the calendar queue. The simulation performance may degrade significantly with uneven event distribution. For example, it is very common for users to set an "end time" before a simulation starts. This end time corresponds to an event many hours ahead in the far future, while most of the other events in the simulation are clustered within several *ms* in the near future. If this end time event happens to be in the fullest bucket, the average interval in the fullest bucket will be in the order of hours or minutes, and NS-2 will set the bucket width to be very large. In this case, most of the events (clustered within *ms*) will go into a few buckets. The calendar queue hence degrades into a few

[4]Unlike [10], NS-2 does not update the bucket width until the number of events is too large or too small. This difference further degrades performance when the bucket width is set to an inefficient value while the number of events in the queue remains at a similar level for a long time.

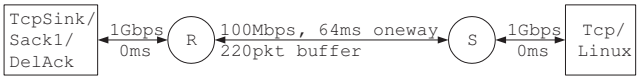

Figure 5: Setup of NS-2 Simulation

linked lists and long linear search happens in event insertion. Similar problems happen in simulations with LDFP networks where TCP burstiness usually generates uneven event distribution.

To fix this problem, we add an average interval estimation into the calendar queue scheduler. We use the average interval of each pair of *dequeued* events, averaged over a whole queue size of dequeued events, as an estimation of the bucket width. If the event departure pattern is unchanged over time, this width results in the complexity of $O(1)$ in both dequeue and enqueue operation.

To address the possible change of event departure patterns, we also implement a SNOOPy Calendar Queue [30], which dynamically adjusts the bucket width by balancing the linear search cost in the event insertion operations and the event dequeuing operation.

With these improvements, the scheduler performs consistently in a variety of simulation setups.

4. SIMULATION RESULTS

We evaluate simulation results from *NS-2 TCP-Linux* according to three goals: extensibility, accuracy and performance. To demonstrate extensibility, we import six new TCP congestion control algorithms from Linux to NS-2. To validate accuracy, we compare *NS-2 TCP-Linux* simulation results with results from a Dummynet testbed [28], part of the WAN-in-Lab infrastructure [8]. To evaluate performance, we compare the simulation time and memory usage of *NS-2 TCP-Linux* and *NS-2 TCP-Sack1*, the best TCP implementation in NS-2.[5] Finally, we present a real example on how *NS-2 TCP-Linux* helps the Linux community to debug and test congestion control implementations.

The setup of our NS-2 scenario is shown in Figure 5. There is one FTP flow running from the sender to the receiver for 900 seconds. We record the congestion window every 0.5 second.

The setup of our Dummynet experiment is shown in Figure 6. In the experiments, the application is Iperf with a large enough buffer. We read the /proc/net/tcp file every 0.5 second to get the congestion window value of the Iperf flow and compare the congestion window trajectories with the simulation results.

4.1 Extensibility

We have imported all nine congestion control algorithms from Linux 2.6.16-3 to *NS-2 TCP-Linux*. Six of them are not in the current NS-2 release (2.29): Scalable-TCP [22], BIC-TCP [33], CUBIC [27], H-TCP [23], TCP-Westwood [16], and TCP-Hybla [14]. Table 2 shows congestion window trajectories with these six algorithms. To make the figures readable, we rescale the time axes to include only six congestion epochs in each figure.

Table 2 shows that the congestion window trajectories predicted by *NS-2 TCP-Linux* are very similar to the results in Linux. The two cases which have most significant differences are TCP-Hybla, and TCP-Cubic. TCP-Hybla measures the round trip delay to set its additive increment (AI) parameter. Due to jitter in the Dummynet router, the Linux host measures higher delay in the testbed and calculates a higher AI parameter for TCP-Hybla, leading to a shorter length of congestion epoch than that in the simulation results. TCP-Hybla also sets a large congestion window in a flow's start-up phase. Both *NS-2 TCP-Linux* and Linux experience severe packet losses. But Linux receives a timeout earlier than *NS-2 TCP-Linux*. This results in differences of congestion window value at the start-up phase (though the rates predicted by *NS-2 TCP-Linux* are very similar to the Linux results). For TCP-Cubic, there are differences in both the congestion widow trajectory and the length of congestion epoch, which are still under investigation.

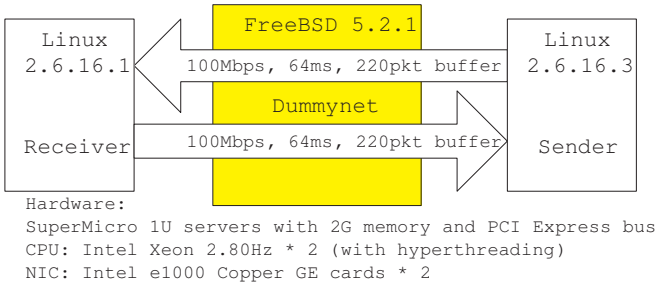

Figure 6: Setup of Dummynet Experiments

4.2 Accuracy

To evaluate accuracy, we compare the results of Linux, the simulation results of *NS-2 TCP-Linux* and the simulation results of *NS-2 TCP-Sack1* with Reno and HighSpeed-TCP [17] or *NS-2 TCP-Vegas* [12, 15], as shown in Table 3.

In general, the simulation results of *NS-2 TCP-Linux* are much closer to the Linux results, especially for Reno and Vegas. In the case of Reno, the difference between *NS-2 TCP-Sack1* and Linux is mainly due to the "appropriate byte counting" [11] implementation in Linux. The Vegas case is even more interesting. Both *NS-2 TCP-Linux* and Linux results have smaller congestion windows than the original NS-2 implementation *NS-2 TCP-Vegas*. We found that the combination of delayed ACKs and integer operations is the source of the problem. With delayed ACKs, two packets are sent into the network in a burst. Vegas can only measure the second packet's RTT due to the delayed ACK. Unfortunately, this RTT measurement includes most of the queuing delay introduced by the first packet in the burst.[6] Such a queuing delay is equivalent to *almost* one packet in the bottleneck queue. With integer operations in the Linux implementation and *NS-2 TCP-Linux*, Vegas sees one packet's worth of delay and stops increasing its congestion window because the α parameter is set to 1. However, *NS-2 TCP-Vegas* uses a high resolution "float" to calculate the available

[5]We also tried other existing implementations in NS-2. *NS-2 TCP-Reno* and *NS-2 TCP-NewReno* have many more timeouts than *NS-2 TCP-Sack1*, leading to poorer accuracy than *NS-2 TCP-Sack1*. *NS-2 TCP-Fack* and *NS-2 TCP-Sack-RH* have similar results as *NS-2 TCP-Sack1* but run much slower due to the inefficient implementation of Scoreboard.

[6]There is still a small gap between the two packets in a burst. The gap depends on the edge link capacity, which is 10 times the bottleneck link in our simulations and experiments.

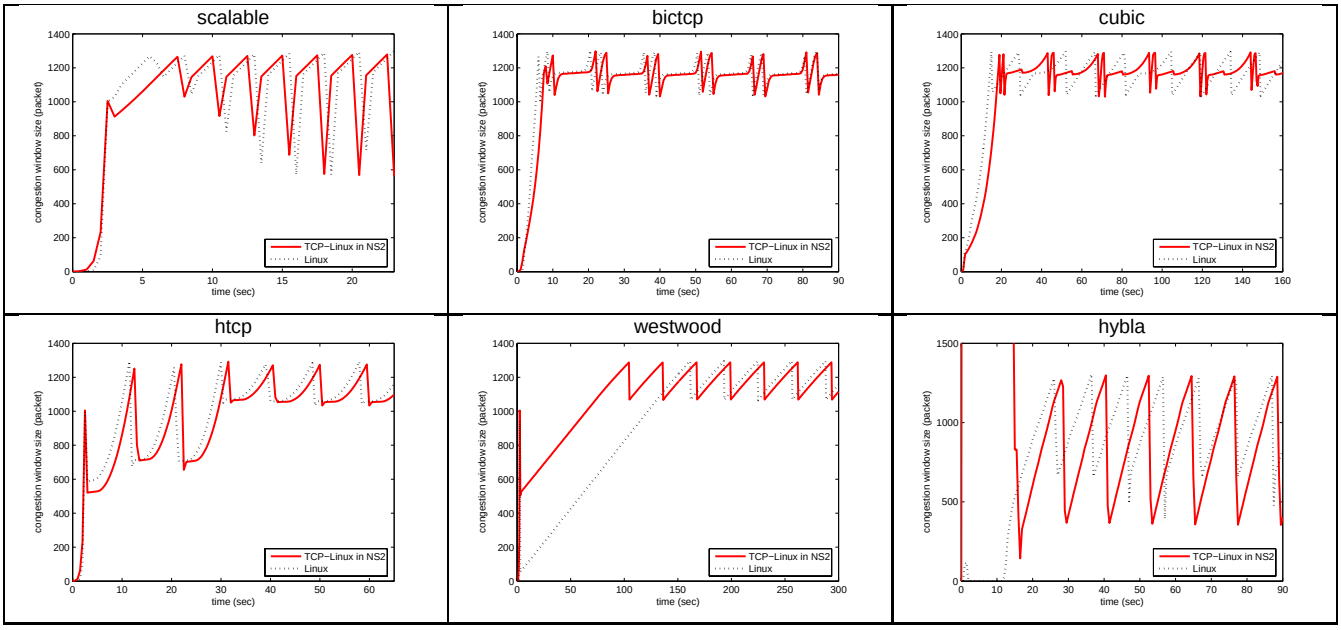

Table 2: Extensibility: congestion window trajectories of new congestion control algorithms: Scalable-TCP, BIC-TCP, CUBIC, H-TCP, TCP-Westwood, and TCP-Hybla

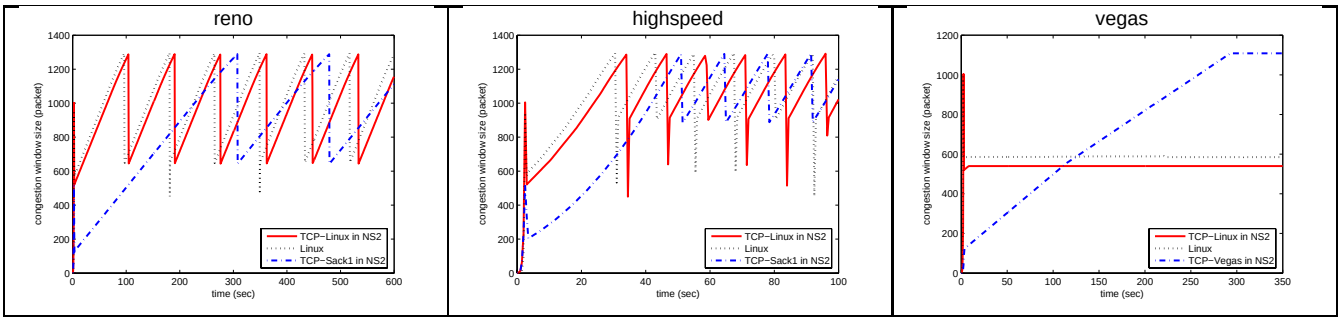

Table 3: Accuracy: congestion window trajectories of Reno, HighSpeed-TCP and TCP-Vegas

bandwidth and expected bandwidth. The results are converted to an integer only at the last step of the comparison, avoiding this problem.

We also ran simulations with different per-packet loss rates in the bottleneck and compare the throughputs of simulations and experiments. Figure 7 shows the throughputs of a single TCP Reno flow (running for 600 seconds) with different per-packet loss rates in the link. Each experiment or simulation is repeated 10 times and we present both the average and error-bar in the figure.

The results with *NS-2 TCP-Sack1* in NS-2 have a constant gap from the Linux performance. This gap, in log scale, implies a constant ratio in throughputs. The discrepancy is due to the implementation differences explained in Section 2.2.

The simulation results for *NS-2 TCP-Linux* are very similar to the experiment results of Linux, except for the case when the per-packet loss rate is 10%. In this case, the Linux receiver almost disabled delayed ACKs, achieving a better performance than the simulation, whereas the delayed ACK function in *NS-2 TCPSink* is not adaptive.

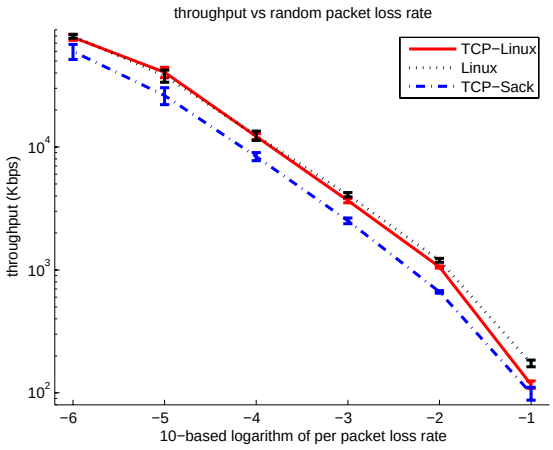

Figure 7: Accuracy: throughputs with different random loss rates (log-log scale)

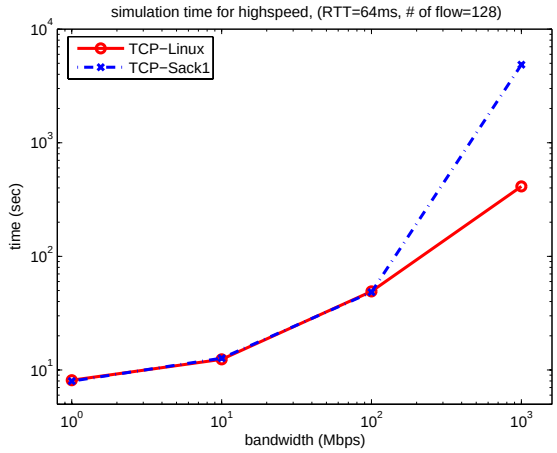

Figure 8: Performance: Simulation time of different bottleneck bandwidths (log-log scale)

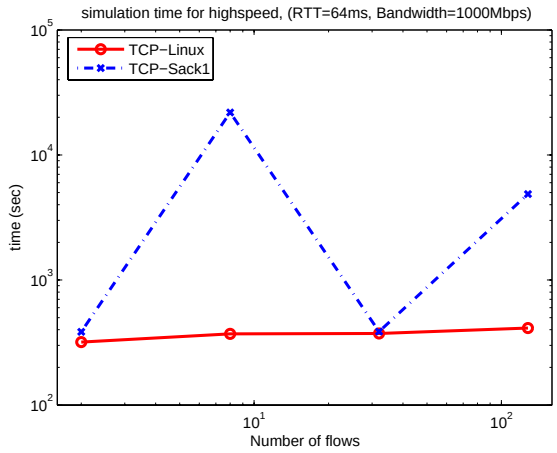

Figure 9: Performance: Simulation time of different number of flows (log-log scale)

4.3 Performance

We ran simulations with Reno and HighSpeed-TCP, with different numbers of flows (from 2 to 128), different round trip propagation delays (from 4ms to 256ms) and different capacities (from 1Mbps to 1Gbps) to test the speed and memory usage. We compare the simulation performance of *NS-2 TCP-Linux* with the performance of *NS-2 TCP-Sack1*.

Each case simulates the scenario for 200 seconds. All the simulations are run on a server with an Intel Xeon 2.66GHz CPU and a 512KB cache.

In most of the scenarios, *NS-2 TCP-Linux* and *NS-2 TCP-Sack1* have very similar simulation performance. Here we present the figures with the most differences.

Figures 8 and 9 show the simulation time of HighSpeed TCP simulations with different bottleneck capacities and different numbers of flows respectively.

Both figures show that *NS-2 TCP-Linux* usually has a very similar simulation speed to *NS-2 TCP-Sack1*. However, *NS-2 TCP-Sack1* does not perform consistently well and has much longer simulation time when the capacity is high, or the number of flows is large, due to the problem in the calendar queue scheduler, discussed in Section 3.3. [7]

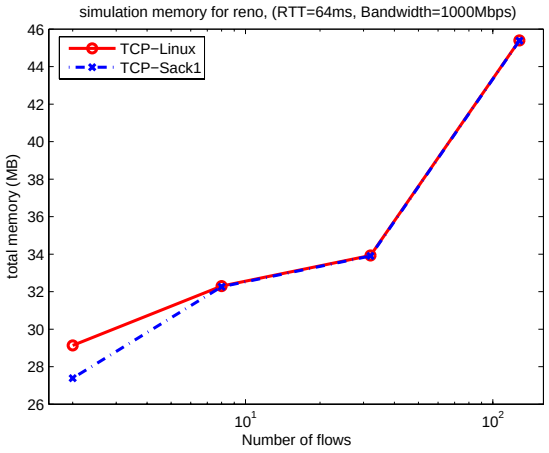

Figure 10: Performance: Memory usage of different number of flows (x-axle in log scale)

To quantify the memory usage, we measure the memory usage of the entire simulator process in the middle point of the simulation. *NS-2 TCP-Linux* uses almost the same amount of memory as *NS-2 TCP-Sack1* in most of the scenarios. The only case with observable difference is with Reno and with two flows, as shown in Figure 10. In this case, *NS-2 TCP-Linux* uses about 1.7 MBytes (6% of 27.3 MBytes) more than *NS-2 TCP-Sack1*.

Based on these simulation results, we believe that *NS-2 TCP-Linux* can be a good alternative or even a replacement for the existing NS-2 TCP modules, given its similar performance in terms of speed and memory usage and its advantages in terms of extensibility and accuracy.

4.4 An example: Fixing bugs in the Linux HighSpeed TCP implementation

Finally, Figure 11 shows an example of how *NS-2 TCP-Linux* can help the Linux community to test and debug the implementations of new congestion control algorithms.

This figure illustrates three bugs in the HighSpeed TCP implementation in Linux 2.6.16-3, found in our testings with *NS-2 TCP-Linux*.[8] The scenario is the same as in Figure 5, except that 32 *NS-2 TCP-Linux* flows are running with HighSpeed TCP.

Three bugs are revealed in this scenario:

1. The effective value of congestion window (combining *snd_cwnd* and *snd_cwnd_cnt*) may be infinitely large, as shown in the spikes in Figure 11. This is due to a variable overflow of *snd_cwnd_cnt*. In the implementation, when *snd_cwnd_cnt* is larger than or equal to *snd_cwnd*, *snd_cwnd* is increased by one, *BEFORE* *snd_cwnd_cnt* is decreased by *snd_cwnd*. In the rare situation when *snd_cwnd_cnt*== *snd_cwnd*, this oper-

[7] We also ran *NS-2 TCP-Sack1* with our improved scheduler. With the improved scheduler, *NS-2 TCP-Sack1* consistently runs in a very similar simulation speed to *NS-2 TCP-Linux*.

[8] In our testing process, we found different bugs in different scenarios. Figure 11 is a carefully constructed example to illustrate all three bugs, after we understood the details of these bugs.

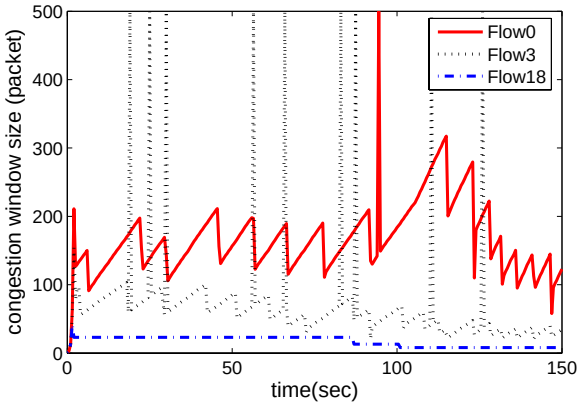

Figure 11: Example: Bugs in Linux 2.6.16-3 implementation of HighSpeed TCP

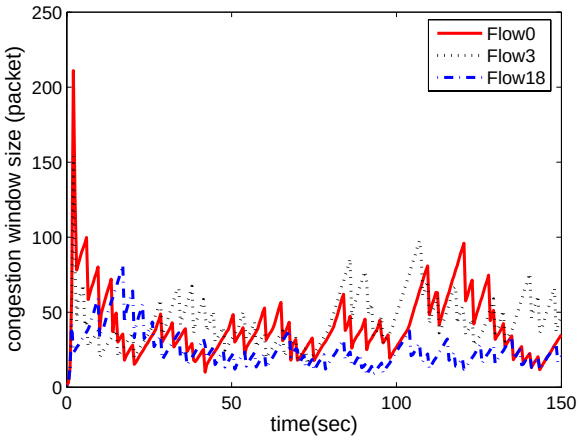

Figure 12: Example: Corrected implementation of HighSpeed TCP in Linux 2.6.16-3

ation leads to a value of -1 for *snd_cwnd_cnt*. Since *snd_cwnd_cnt* is an unsigned variable, the negative value causes overflow and results in an almost infinitely large *snd_cwnd_cnt*. Rarely happens, this bug is spotted by the variable tracing function of *NS-2 TCP-Linux*.

2. There are some flows (e.g. Flow 18 in Figure 11) that never increase their congestion windows. This is due to a bug that a flow's AI parameter is zero if it quits slow-start with a congestion window smaller than 38. Flow 18 in Figure 11 is of this case.

3. Some flows never converge to fairness (e.g. Flow 0 and Flow 3 in Figure 11). This is due to a bug that a flow's AIMD parameter is not changed when the flow's congestion window is decreased to a lower value. Once a lucky flow gets a large congestion window, it always keeps the AIMD parameter for the large window and grows its congestion window faster than other flows. Hard to be seen in simple cases, this bug is common in scenarios with large number of flows and long simulation time.

We fixed these bugs and Figure 12 presents the results from the same scenario with the bugfix patch. The congestion window trajectories of these three flows are much fairer. We further confirm from the detailed results that each of the 32 flows has an average throughput within $\pm 18\%$ range of the fair-share rate in the 900 second simulation. We have reported all these bugs to the Linux "netdev" mailing list, and the patch will be applied in the future release.

This example demonstrates two unique and powerful functions of *NS-2 TCP-Linux* in helping the implementation community: First, as an full-functioned NS-2 module, *NS-2 TCP-Linux* provides the variable tracing function which can capture rare events in the testing process. Second, with fast simulation speed, *NS-2 TCP-Linux* provides a controlled environment to conduct repeatable simulations with complex scenarios.

Seeing this example, we strongly believe that *NS-2 TCP-Linux* can help the implementation community to debug, test and understand new congestion control algorithms, and it can close the gap between the implementation community and the analysis community.

5. RELATED WORK

One view of *NS-2 TCP-Linux* is as an NS-2 TCP implementation that simultaneously improves extensibility, accuracy and simulation speed. We are not aware of similar efforts on improving NS-2 TCP extensibility. Related efforts in improving NS-2 TCP speed and accuracy include *Scoreboard-RQ*, and *Scoreboard-RH*. *Scoreboard-RQ* traverses the retransmission queue once every acknowledgment, similar to *NS-2 TCP-Linux*. However, *Scoreboard-RQ* cannot detect loss of retransmitted packets and generates many more timeouts in lossy environments. On the other hand, *Scoreboard-RH*, another scoreboard implementation, has to traverse the retransmission queue for each SACK block, leading to low simulation speed.

More broadly speaking, *NS-2 TCP-Linux* is an effort to utilize source code from real operating systems for network simulation. In this regard, there are many related efforts such as Simulation Cradle [21] and Linux UML Simulator [9]. These efforts create an artificial environment around the actual kernel and run the kernel as is; as a result, they achieve better accuracy, in the sense that they can produce results comparable to real systems in packet trace level. These tools are greatly helpful in studying detailed protocol behavior (such as three-way handshakes in TCP). However, the associated costs are slower simulations and larger memory usage, making them difficult to perform long-running complex scenario simulations. In contrast, *NS-2 TCP-Linux* aims to improve NS-2 TCP only, and is meant for the TCP congestion control community. It retains all the benefits of the NS-2, and achieves accuracy in TCP behavior, in congestion window trajectory level, without paying the costs in simulation performance. As demonstrated in Section 4.4, *NS-2 TCP-Linux* has its unique advantages in identifying performance problems that happen in rare events or with complex scenarios.

From the most general perspective, *NS-2 TCP-Linux* is part of the spectrum of tools for TCP performance analysis. As shown in Figure 13, this spectrum of tools spans from abstract mathematic models such as [26, 31], to more realistic emulation systems such as Emulab [3] (emulated with multiple Dummynet [28] machines), NetEm [20] and real

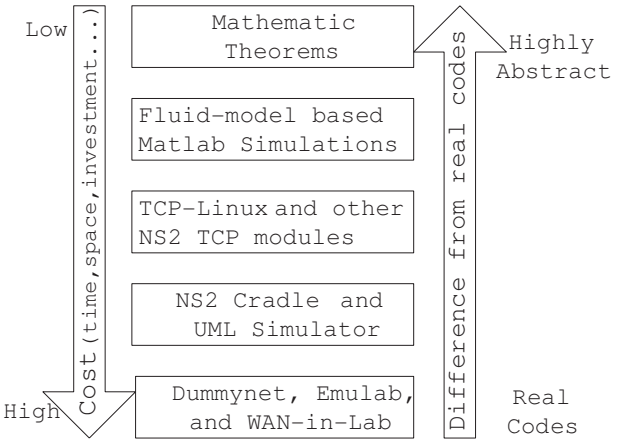

Figure 13: A spectrum of tools for TCP performance analysis

systems in both hosts and links such as WAN-in-Lab [8]. The more accurate and realistic the results, the higher the simulation cost in terms of time, space or investment, and the more complex the system. We believe that all the tools in this spectrum are critical for TCP performance analysis, each offering a unique perspective and insight, and none can be replaced by others. In this sense, *NS-2 TCP-Linux* sits right in the place with other existing NS-2 TCP modules, as a packet level simulator.

6. CONCLUSION AND FUTURE WORK

As a new NS-2 module, *NS-2 TCP-Linux* provides more accurate simulation results for Linux TCP, with similar simulation performance and much better extensibility to new congestion control algorithms.

We expect that the new module can be widely used as an alternative or even a replacement for the current NS-2 TCP modules.

We also expect that the new module can help the Linux community to use NS-2 to test and analyze new TCP algorithms in the future. This can close the gap between the analysis community and implementation community and improve the research productivity.

Currently, this NS-2 module has its limitations. It might not be able to simulate the Linux performance well in the case where the packet reordering in the path is severe or packet loss rate is extremely high. For the future work, we plan to include D-SACK [19] processing and the congestion window reduction undo function from Linux. We are also considering developing a delayed ACK module for NS-2 that performs similarly to Linux.

Since *NS-2 TCP-Linux* provides a very convenient platform for testing different TCP congestion control protocols in many scenarios, it is a good foundation towards a benchmark suite implementation for TCP congestion control algorithms. We plan to enhance our benchmark suites [32, 4] and summarize a set of NS-2 scenarios for the benchmark.

7. ACKNOWLEDGMENT

The authors acknowledge the use of Caltech's WAN in Lab facility funded by NSF (through grant EIA-0303620), Cisco ARTI, ARO (through grant W911NF-04-1-0095), and Corning. The work described in this paper is strongly supported by Dr. Lachlan Andrew of NetLab in Caltech. We are very grateful for his suggestions on the early drafts and his help in the experiments. We are grateful for the anonymous reviewers for their insightful comments and the feedbacks from users of *NS-2 TCP-Linux*. In particular, feedbacks from Salman Abdul Baset (Columbia University), Luiz Antonio F. da Silva (GTA/COPPE/UFRJ, Brazil) and Eryk Schiller (UST Cracow, Poland) have led to improvements on *NS-2 TCP-Linux*.

8. REFERENCES

[1] A Linux TCP implementation for NS-2. URL: http://www.cs.caltech.edu/~weixl/ns2.html.

[2] A mini-tutorial for NS-2 TCP-Linux. URL: http://www.cs.caltech.edu/~weixl/ns2.html.

[3] Emulab - Network Emulation Testbed. URL: http://www.emulab.net/.

[4] Framework for TCP benchmarking. URL: http://www.cs.caltech.edu/~hegdesan/sigcomtest/.

[5] Linux Kernel Documents: TCP protocol. linux-2.6.16.13/Documentation/networking/tcp.txt.

[6] Speeding up NS-2 scheduler. URL: http://www.cs.caltech.edu/~weixl/ns2.html.

[7] The Network Simulator - NS-2. URL: http://www.isi.edu/nsnam/ns/index.html.

[8] The WAN in Lab (WiL) Project. URL: http://wil.cs.caltech.edu/.

[9] UML (User-mode-Linux) simulator. URL: http://umlsim.sourceforge.net/.

[10] J. Ahn and S. Oh. Dynamic Calendar Queue. *Thirty-Second*, 00:20, 1999.

[11] M. Allman. RFC 3465 - TCP Congestion Control with Appropriate Byte Counting (ABC), Feb 2003.

[12] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[13] R. Brown. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[14] C. Caini and R. Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *INTERNATIONAL JOURNAL OF SATELLITE COMMUNICATIONS AND NETWORKING*, 22:547–566, 2004.

[15] N. Cardwell and B. Bak. A TCP Vegas Implementation for Linux 2.2.x and 2.3.x. URL: http://flophouse.com/ neal/uw/linux-vegas/.

[16] C. Casetti, M. Gerla, S. Mascolo, M. Sansadidi, and R. Wang. TCP Westwood: end-to-end congestion control for wired/wireless Networks. *Wireless Networks Journal*, 8:467–479, 2002.

[17] S. Floyd. RFC 3649 - HighSpeed TCP for Large Congestion Windows, Dec 2003.

[18] S. Floyd and T. Henderson. RFC 2582: The New Reno Modification to TCP's Fast Recovery Algorithm, April 1999.

[19] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. RFC 2883: An Extension to the Selective

Acknowledgement (SACK) Option for TCP, Jul 2000.

[20] S. Hemminger. Network Emulation with NetEm. In *Proceedings of Linux Conference AU*, April 2005.

[21] S. Jansen and A. McGregor. Simulation with Real World Network Stacks. In *Proceedings of Winter Simulation Conference*, pages 2454– 2463, Dec 2005.

[22] T. Kelly. Scalable TCP: Improving Performance in HighSpeed Wide Area Networks, 2003.

[23] D. Leith and R. N. Shorten. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of PFLDnet 2004*, 2004.

[24] M. Mathis and J. Mahdavi. Forward acknowledgement: refining TCP congestion control. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 281–291. ACM Press, 1996.

[25] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP Selective Acknowledgement Options, Oct 1996.

[26] V. Misra, W.-B. Gong, and D. Towsley. Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 151–160. ACM Press, 2000.

[27] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. In *Proceedings of PFLDNet 2005*, 2005.

[28] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[29] P. Sarolahti and A. Kuznetsov. Congestion Control in Linux TCP. *USENIX Annual Technical Conference*, pages 49–62, 2002.

[30] K. L. Tan and L.-J. Thng. SNOOPy Calendar Queue. In *Proceedings of the 32nd Winter Simulation Conference, Orlando, Florida*, pages 487–495, 2000.

[31] J. Wang, D. X. Wei, and S. H. Low. Modeling and Stability of FAST TCP. In *Proceedings of IEEE Infocom*, March 2005.

[32] D. X. Wei, P. Cao, and S. H. Low. Time for a TCP Benchmark Suite? URL: http://www.cs.caltech.edu/~weixl /research/technicals/benchmark/summary.ps.

[33] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast Long-Distance Networks. In *INFOCOM*, 2004.