TCP over UDP

A report submitted in partial fulfillment
of the requirements for the completion of course

CSL724 - Advanced Computer Networks

by

J Phani Mahesh
Entry No. 2010EE50549

Sri Harsha Prem Kumar Guggilam
Entry No. 2010EE50560

Cherukuri Sesha Satya Chaitanya
Entry No. 2010CS10214

Under the guidance of
VINAY JOSEPH RIBEIRO

INDIAN INSTITUTE OF TECHNOLOGY DELHI
JANUARY-APRIL 2014

***Abstract*—TCP over UDP is the practice of decoupling the control algorithms of TCP from the Operating System's Network stack which is a part of the kernel itself. Lightweight UDP protocol is used at the network layer and the actual control algorithms that were traditionally part of TCP will be implemented at the application Layer. Depending on the specific application requirements, the algorithms can be fine tuned for maximum performance. Decoupling the control from Kernel allows multiple applications with possibly conflicting requirements to run on the same machine, each using their own optimised versions of TCP algorithms.**

In our project, we will study high-speed TCP algorithms (TCP FAST, TCP CUBIC, TCP Scalable) in detail and implement them as a library at the application layer, over UDP. A File sharing application will be built to compare the protocols thus implemented in terms of performance, reliability, fairness to existing TCP variants.

***Index Terms*—TCP over UDP,High speed TCP**

I. Introduction

UDP stands for User Datagram Protocol. It is a very simple, lightweight protocol that does the bare minimum. It is not designed to handle flow control, congestion control, or reliable delivery.

TCP stands for Transmission Control Protocol. TCP ensures a best effort reliable delivery. Congestion control, Flow control, and reliable delivery are at the core of TCP design.

Most modern application layer protocols are built atop TCP since it provides reliable delivery. A lot of work has been done by the scientific community in designing better algorithms to handle various scenarios. This resulted in a multitude of variants, though only a handful survived the test of time.

II. Implementation

A. TCP over UDP

In the TCP/IP protocol suite, at the transport layer, UDP is used for communication. The UDP payload is crafted to include a header containing the following fields:

- $hdrlen$ - Header Length
- $seqno$ - Sequence Number
- $ackno$ - Acknowledgement Number
- $sack$ - Selective Acknowledgements (Array)
- $cwnd$ - Congestion Window
- $rwnd$ - Reciever Window
- $uptr$ - Urgent pointer
- $flags$ - SYN, RST, FIN
- $options$ - TCP Options

The TCP algorithms will be implemented as individual libraries.

B. Libraries

One library per variant of the TCP will be implemented. All such libraries will be able to inter-operate using the same UDP payload structure.

C. Application

A file sharing application will be implemented which can choose the variant of TCP from among the libraries.

D. Testing

Over a high-speed network link, a pair of nodes send a file back and forth, each running one variant of TCP will download a file of size 1 GB from its corresponding node at the other end of the link. Performance metrics such as cwnd and goodput were plotted.

E. Toolchain

We have used Python for implementing the libraries. We leveraged a asynchronous event loop system available in a module 'circuits' and built our TCP stack based on it. A couple of shell scripts and helper files were used to test the performances.

III. Framework

A. Guiding Principles

All versions of TCP have certain features in common. The headers follow a common structure, and so does the handshake behaviour. A modular structure allows for rapid development and easier debugging, and also allows code reuse. This is also in line with KISS and DRY principles.

B. Protocol Implementations

Each protocol version is made a separate module, which builds on the abstract class `TCP_over_UDP` by defining all properties and methods on itself from the base. Protocol specific implementations and event handling is done in these libraries.

C. Usage

An external application can create a TCP connection instance using any such TCP version and the actual translation to UDP packets and transmission as guided by the version specification are handled transparently.

For example, to communicate using TCP Reno, The sender and reciever create tcp.Reno instances and establish a TCP connection. Relevant statements would follow the following pattern.

```
from tcp.reno import Reno
Reno().send('inputfile').run()
```

An actual low level implementation would be listening on all ports for data, handling it as appropriate, but at the application level, each port has to be bound to separately.

D. Implementation

The implementation depends on a central event loop to fire various events to which each protocol implementation will register a set of handlers. The event loop is implemented using a library named `circuits`. A UDP server is created and bound to the port specified by the user. The base class `TCP_over_UDP` implements data serialization and deserialization routines that pack a byte stream along with TCP metadata into a UDP packet, and unpack the same. It manages

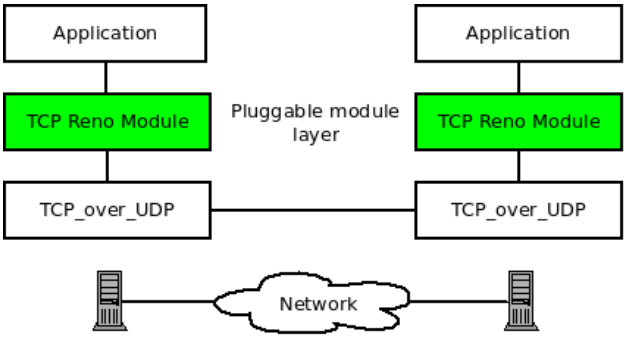

Figure 1. Layered architecture - The application at the top layer uses a common API provided by one of many pluggable modules in the middle layer, each of which provides one variant of TCP algorithm. The modules communicate with a lower level meta-module that translates the data to UDP packets and vice versa.

the central event loop, and fires an acknowledgement event whenever an ack packet is received. When data is submitted to be sent, it reads a chunk of this data into a send buffer, which triggers a data event. Based on the current values of congestion window and flight size, a decision is made if more data can be injected into the network. The buffers are emptied by packing TCP like UDP packets and writing to the link at maximum speed. The transmission is paused when disallowed by the congestion window. When acknowledgements arrive, protocol implementations update their congestion window as appropriate, and a congestion window update event is triggered for each such update. If the send buffer is not empty, more data is written to the line as allowed by the updated value of congestion window.

IV. TCP Reno

The proposal outlined in RFC2581 is used as the basis for TCP Reno implementation. RFC2581 defines TCP Reno's four intertwined congestion control algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery. In addition, the document specifies how TCP should begin transmission after a relatively long idle period, as well as discussing various acknowledgment generation methods. [RFC2581]

Our implementation of TCP Reno reacts to three events on the sender side.

- An ACK for a unacknowledged packet.
- Triple duplicate ACK.
- Timeout for a packet.

The behaviour of Reno depends on the current "state" the sender is in. At any instant, the sender may be in any one of the following "states".

- Slow Start
- Congestion Avoidance
- Fast Retransmit and Fast Recovery

A timer is set for each packet that gets sent, and is used to detect the timeout event. The TCP headers (that were mapped to a portion of UDP payload) can be used to detect the ACKs reieved. An event loop fires these events as they happen, by calling respective response functions of Reno. These response functions are designated as `on_<event>`, for each event.

Each such response function checks for the current state, and delegates execution to an appropriate state specific event handler. The state specific handler processes the event and updates the current state if necessary.

The following subsections describe the Reno algorithm we used as the basis for implementation. These sections, being standard protocol descriptions are quoted or paraphrased from RFC2581 heavily, so as to not deviate from the accepted standard.

A. Slow Start

In Slow-start phase, TCP increases the congestion window each time an acknowledgement for new data is received, by at most Sender Maximum Segment Size (SMSS) bytes. When the congestion window reaches/exceeds a state variable called `ssthresh` - Slow Start Threshold - Reno enters Congestion Avoidance mode. This strategy effectively doubles the TCP congestion window for every round trip time (RTT). (Assuming no losses)

B. Congestion Avoidance

In Congestion Avoidance Phase, TCP congestion window is increased by 1 full sized segment for each RTT until a loss event occurs.

The following formula was suggested for updating the congestion window (cwnd) during this phase.

$$cwnd = cwnd + \frac{\text{SMSS} \times \text{SMSS}}{cwnd}$$

This adjustment is executed on every incoming non-duplicate ACK. [RFC2581]

When a loss occours, as detected by the timer, the value of *ssthresh* must be updated as:

$$ssthresh = \max\left(\frac{flightsize}{2}, 2 \times \text{SMSS}\right)$$

Flightsize represents the amount of data sent, that is yet to be acknowledged by the reciever.

In the event of a triple duplicate acknowledment being recieved, Reno enters Fast Retransmit Phase

C. Fast Retransmit and Fast Recovery

TCP maintains a timer after sending out a packet, if no acknowledgement is received after the timer is expired, the packet is considered as lost. However, this might take too long for TCP to realize a packet is lost and take action. A fast retransmit algorithm is used to make use of duplicate ACKs to detect packet loss. (Using Cumulative ACKs) In fast retransmit, when an acknowledgement packet with the same sequence number is received a specified number of times (Usually 3), TCP sender is fairly confident that the TCP packet is lost and will perform a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire.

When the third duplicate ACK is recieved,

- $ssthresh$ has to be adjusted to be no more than the value obtained using the formula in previous section.

- The lost segment is retransmitted.
- Congestion window is updated as follows.

$$cwnd = ssthresh + 3 \times \text{SMSS}$$

- For each additional duplicate ACK, $cwnd$ is incremented by one SMSS.
- Send more segments into the network if allowed by current values of $cwnd$ and $rwnd$.
- When an ACK acknowledges all data sent before the third duplicate ACK is recieved, $cwnd$ is set to the $ssthresh$ caluculated above.

Reno remains in this state till a non-duplicate ACK is received or a timeout occurs.

D. Timeout

Irrespective of the state Reno is in, a timeout event is considered an indicator of severe network congestion, and hence dratic measures are taken.

The $cwnd$ is set to $2 \times$ SMSS and Reno enters slowstart phase, beginning at the first unacknowledged packet.

E. Computing the Retransmission Timer

The process of computing retransmission timer, which is used to trigger the timeout event is implemented as described in RFC 2988.

V. TCP CUBIC

The proposal outlined in [6] is the basis for implementation of TCP CUBIC. It is an improvement over BIC, and the congestion window update follows a cubic curve.

The "states" and "events" of CUBIC are defined in exactly the same manner as for TCP Reno, as described in the previous section. However, the actual implementation differs from Reno in the congestion window update algorithm as a response to ACK for unacknowledged data and $ssthresh$ update on loss as follows.

The congestion window update during the Congestion Avoidance state is governed by the relation:

$$\begin{aligned} cwnd &= C(t-K)^3 + W_{max} \\ K &= (W_{max} * \beta / C)^{1/3} \end{aligned}$$

where
C is the scalability factor,
t is the elapsed time from the last window reduction,
W_{max} is the window size just before reduction, and
β is the multiplicative decrease factor.

When a loss occours, as detected by the timer or on reciept of triple duplicate ACK, $ssthresh$ is updated as:

$$ssthresh = \max(flightsize \times \beta, 2 \times \text{SMSS})$$

This is followed by state transition to Fast Retransmit and Fast Recovery, and the algorithm proceeds similar to Reno.

VI. TCP Scalable

The proposal outlined in [3] is the basis for implementation of TCP Scalable. TCP Scalable follows MIMD (Multiplicative Increase Multiplicative Decrease) pattern for congestion window update.

The "states" and "events" of TCP Scalable are defined in exactly the same manner as for TCP Reno, as described earlier. However, the actual implementation differs from Reno in the congestion window update algorithm as a response to ACK for unacknowledged data and $ssthresh$ update on loss as follows.

On a successful ACK (for unacknowledged data), the window is updated as:

$$cwnd = cwnd + a$$

When a loss occours, as detected by timer, or on reciept of triple duplicate acknowledgement, $ssthresh$ is updated as:

$$ssthresh = \max(flightsize \times (1-b), 2 \times \text{SMSS})$$

where β is the multiplicative decrease factor.

This is followed by state transition to Fast Retransmit and Fast Recovery, and the algorithm proceeds similar to Reno.

VII. TCP FAST

The proposal outlined in [5] was used as the basis for our implementation of TCP FAST. It is a high speed delay based protocol that attempts to maintain a low delay by indirectly measuring the buffer lengths along the path and adjusting the window size to stabilize them.

The proposal claims that packet loss as an indicator is inherently not ideal and instead takes a delay based approach. Current RTT time in comparision with the smallest RTT observed so far was considered an indicator of queueing delay. Instead of a binary switch, FAST proposes a continuous correction for window size that can vary over an intrerval based on the deviation of observed delays from a target value. This allows for a finer tuning, and stabilizes around the target value. Also, this approach has the advantage that if the equillibrium is detected to be far, the correction would be very high, and will gradually reduce to zero once the target is reached.

FAST periodically updates the window according to the following equation.

$$w \leftarrow \min\left\{2w, (1-\gamma)\, w + \gamma\left(\frac{baseRTT}{RTT} w + \alpha\right)\right\}$$

where $\gamma \in (0, 1]$, baseRTT is the minimum RTT observed so far, and α is a positive protocol parameter that determines the total number of packets queued in routers in equilibrium along the flow's path. The window update period is 20 ms in our prototype.

A. Implementation

To measure the RTT, each packet is marked with the time at which it is sent. The acknowledgemnt handler fires an acknowledgement event whenever an acknowledgement is received, and the event handler can access the time of its

arrival and time at which its corresponding packet was sent. On demand, a property allows computing the corresponding RTT from these values.

A FIFO Queue of constant length was constructed using `collections.deque`, and as acknowledgements arrive, the corresponding RTTs are pushed into it. The queue is constructed in such a way that when the queue is full, pushing another value deletes the first element, to maintain the queue length. This is used to maintain a list of last n RTTs. Whenever the event loop timer fires an update event, this list is consulted to determine the minimum RTT, as baseRTT, and the average of last three RTTs is used as the value of RTT. This averaging dampens the effect of large spikes, if and when they occour.

The event loop iself registers two timers. One of them is used to update a timer counter, which is used as a proxy for determining the time. This is used for effeciency reasons, since querying the system time frequently creates low level interrupts that severely affect the performance. A second timer fires an update event on the top level event loop, to which the FAST implementation responds by updating the RTT.

Since the FAST proposal makes no mention of timeouts, and the delay based calculations need some time at the beginning of the connection to be of any use, we have taken the liberty to implement Slow Start and Fast Retransmit & Fast recovery phases similar to Reno. Ideally, timeouts should not occour in FAST, and the initial few roundtrips of the conversation may have a highly varying congestion window without our additions, but they will not have any adverse affects on the usability of FAST. However, in the rare event that timeouts do occour, we need a mechanism in place to detect and respond to these events by resending the data that is yet unacknowledged. Since a well-established mechanism is described by TCP Reno, and has already been implemented, we found it fit to extend the same to FAST.

VIII. Testing and Analysis

After the complete implementation of all the TCP variants, a pilot test is used for the purpose of debugging. The pilot test is performed between a HP laptop and a DELL PC that are connected by a bottleneck link which offers a speed of about 850 Mbps. In this test, a video file of size 600 MB is used for testing the protocols. After a rigorous testing and debugging by the use of Pdb (Python debugger), the project is made ready for the actual measurement of performance metrics.

A. Performance Metrics

The test is performed on two GCL machines. The test is deliberatelly carried out at a time when the users in GCL are considerably high. TCP Reno is first choosen and a file of size 500 Megabytes is used an input. The trace of the congestion window for every time-tick of 200ms is traced out. The amount of data (in bits) received at the receiver for every second (the goodput) is also traced out for analysis. This has been used as a bench mark for other protocols. The plot of congestion window that has been obtained is in consistence with the thoeritical analysis. TCP Reno has handled several duplicate acknowledgements in a effecient manner in accomplishing the file transfer.

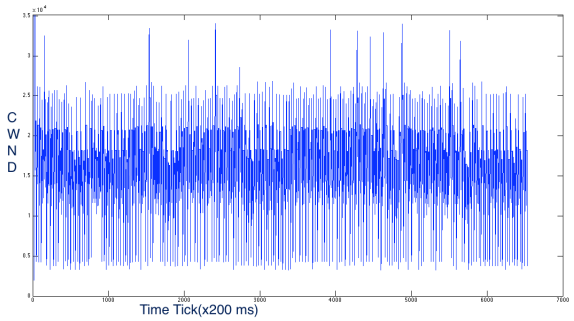

Figure 2. TCP Reno - Variation of congestion window

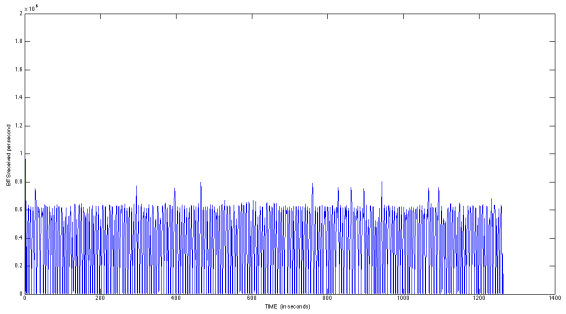

Figure 3. TCP Reno - Variation of Goodput

1) TCP Scalable: TCP Scalable, a MIMD protocol has been tested with a file size of 272 megabytes. The parameters of Scalable viz., a and b are 2 and 0.2 respectively. The $cwnd$ values at the sender and the data received per second at receiver are captured for analysis. From the plot of congestion window, TCP Scalable clearly follows multiplicative increase. Scalable also handled timouts and duplicate acknowledgements relatively well to deliver high goodput.

TCP Scalable is tested against Reno for inter protocol fairness. Two different interfaces are used on each machine each for scalable and reno. First, Reno is initiated for file transfer followed by Scalable with a short delay in between them. Scalable increased its window much aggressively and also delivered the data much faster. The Reno suppressed a little due to Scalable.

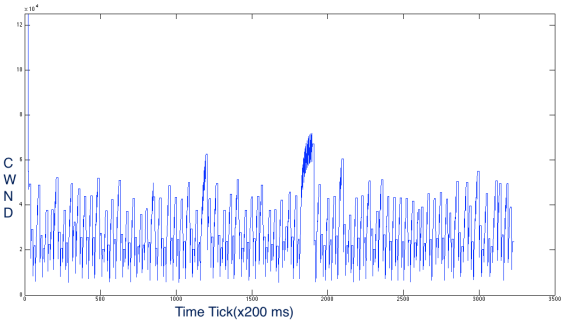

Figure 4. TCP Scalable - Variation of congestion window

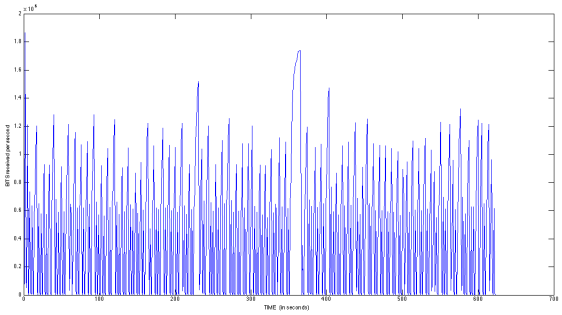

Figure 5. TCP Scalable - Variation of Goodput

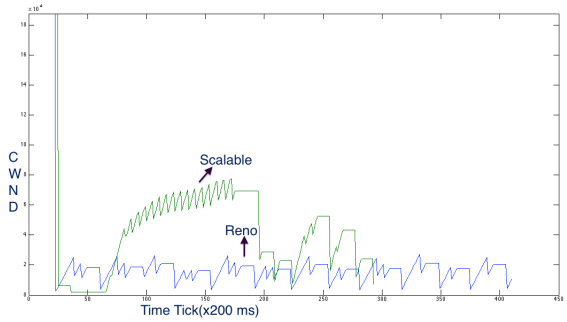

Figure 6. Inter-Protocol Fairness - Reno vs Scalable

2) TCP CUBIC: TCP CUBIC, a cubic window update protocol is tested using a file of size 244 megabytes. The parameters of CUBIC viz., C, β and S_max are 0.4, 0.8 and 160 respectively. The window is updated for a periodic time interval of 200ms. The variation of congestion window observed does not match the theoritical pediction. This is due to the granularity problem that the implementation suffers from. The implementation works such that after every time tick, all events are fired and then the state changes happen. In CUBIC, after $cwnd$ is updated for every time tick. But soon after the increase in the same tick, if a duplicate acknowledgement is encountered, the window is decremented limiting the window increase to linear instead of cubic. Despite this, the goodput observed is much better.

TCP CUBIC is also tested against Reno for inter protocol fairness. Two different interfaces are used on each machine

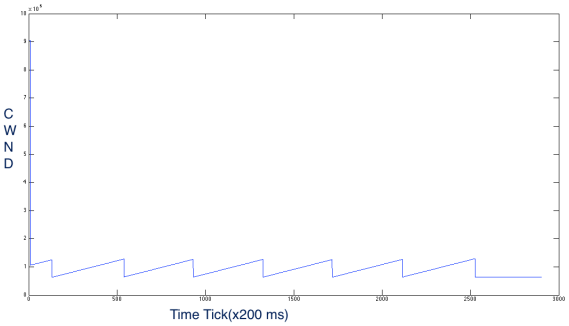

Figure 7. TCP CUBIC - Variation of congestion window

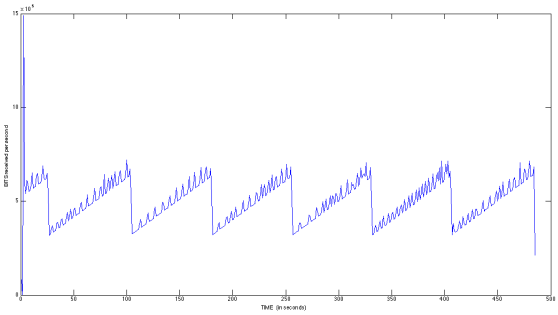

Figure 8. TCP CUBIC - Variation of Goodput

each for scalable and reno. First, Reno is initiated for file transfer followed by CUBIC with a short delay in between them. CUBIC increased its window much aggressively although in a non-cubic fashion due to the above mentioned reason.

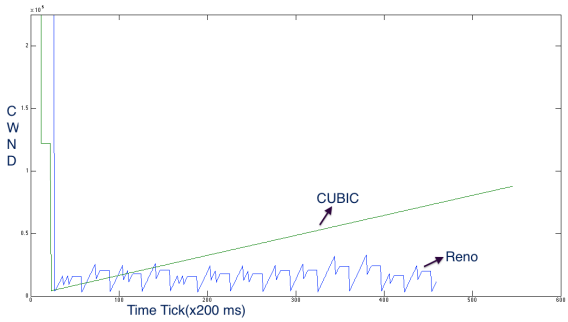

Figure 9. Inter-Protocol Fairness - Reno vs CUBIC

3) TCP FAST: TCP FAST,a delay based protocol is tested using a file of size 60 megabytes. The parameters of FAST viz., γ and α are 0.6 and 500 respectively. The window is updated periodically for every 200ms. Due to the implementation being in a higher level language, we were limited to a high interval between updates to cwnd, and the duplicate acknowledgements and timeouts caused before fast could adapt caused severe timeouts to occour, allowing no recovery. A flat profile is observed due to this.

TCP FAST is also tested against Reno for inter protocol fairness. Two different interfaces are used on each machine

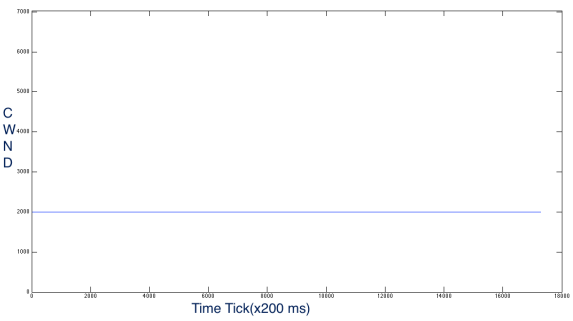

Figure 10. TCP FAST - Variation of congestion window

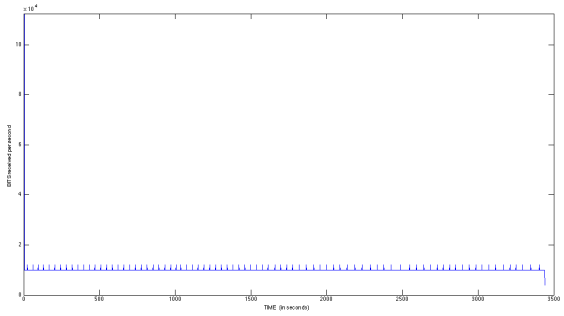

Figure 11. TCP FAST - Variation of Goodput

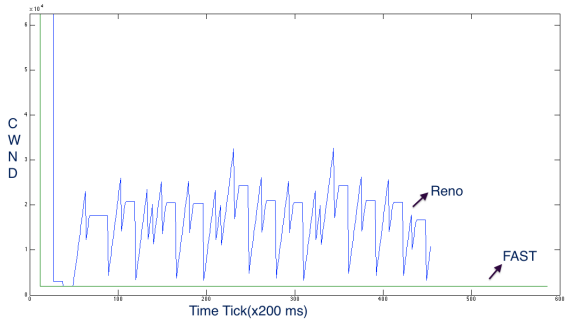

Figure 12. Inter-Protocol Fairness - Reno vs FAST

each for scalable and reno. First, Reno is initiated for file transfer followed by CUBIC with a short delay in between them. In this case, Reno performed much better than FAST as FAST could not hanle the timeouts properly due to the granularity problem stated above.

IX. Conclusion

The choice of an evented asynchronous model gave us flexibility in dealing with the events, and Reno and Scalable perform reasonably well. However, protocols that suggest periodic updates to cwnd are not highly performant using our architecture and choice of toolchain and deviate from the expected behaviour as was seen in the case of Cubic and FAST.

We have successfully tested the interprotocol fairness and the results were as expected. In cases where there was significant deviation, it has been traced back to the limitations imposed by the high level language, its execution speed, and the event loop based architecture we used.

X. Challenges faced

The design of the base architecture demanded a considerable amount of thought and effort. Since our implementation is single threaded, we had to take extreme care to never block on any action since that would stall the entire sender process and this necessitated some clever engineering design.

The asynchronous design made it extremely hard to debug, since it is not very clear the sequence of events that triggered a particular action. However, we managed to iron out most bugs with little effort using circuits Debugger module and using pdb, python debugger to drop to interactive prompt while the program is running and modifying the code and inspecting the live state.

One notable bug that deserves a mention is an accidental duplication of one byte in the receiver while writing to file which caused the complete transfer checksum match to fail. It demanded about 40 hours of time and was traced to an expectation of received packets not partially overlapping with a previously sent packet, which was being violated. A check in place to detect partial overlaps and discard the overlapping portion fixed it.

It took us considerable time and effort to understand the behaviour of Cubic and Fast, and their deviations from the expected norm. It has been conclusively traced back to a design limitation of our architecture, and it was too late to redesign the architecture by then.

XI. References

1) RFC2988: Computing TCP Retransmission timer - November 2000
2) RFC2581: TCP Congestion Control - M. Allman, V. Paxson, W. Stevens - April 1999
3) Kelly: Scalable TCP: improving performance in high-speed wide area networks - CCR 2003
4) Floyd: HighSpeed TCP for Large Congestion Windows, 2003
5) Low et al: FAST TCP: motivation, architecture, algorithms, performance, INFOCOM 2004
6) Ha et al: CUBIC: a new TCP-friendly high-speed TCP variant.ACM SIGOPS Operating Systems Review (2008)