Implementing TCP SACK Conservative Loss Recovery Algorithm within a NDN Consumer

Shuo Yang

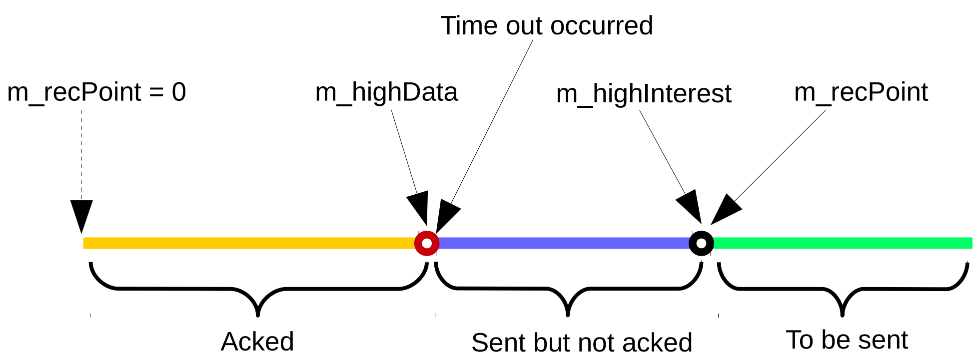
1. Design

- Consumer uses packet timeout as signal of congestion;
- Consumer reacts to one packet loss event per RTT (to handle a burst of packet loss);
- Consumer takes one RTT sample per RTT;
- Consumer uses TCP's AIMD scheme to adjust congestion window size;

2. Algorithm

<u>Parameters:</u>

- *m_highData*: the highest segment number of the Data packet the consumer has received so far;
- *m_highInterest*: the highest segment number of the Interests the consumer has sent so far;
- *m_recPoint*: the value of *m_highInterest* when a packet loss event occurred. It remains fixed until the next packet loss event happens;
- m_cwnd: congestion window size (unit: segment), initial value: 0;
- m_ssthresh: slow start threshold, initial value: 200;



<u>Algorithm description:</u>

- Initially, *m_highData*, *m_highInterest* and *m_recPoint* all set to 0;
- A packet loss event happens when *m_highData* > *m_recPoint*;
- When a timeout occurred, if *m_highData* > *m_recPoint*, this timeout would be considered a packet loss event, consumer should update *m_recPoint* with the value of *m_highInterest*, then adjust congestion window size accordingly (ssthresh = cwnd/2, cwnd = 1); otherwise the timeout wouldn't be considered as a packet loss event and consumer doesn't adjust window size;
- the value of *m_highData* will be updated each time a Data packet was received; the value of *m_highInterest* will updated each time an Interest packet was sent;

In the above figure, initially, m_recPoint = 0. When the time out happened at the segment represented by the red circle, since *m_highData* > *m_recPoint*, it's considered a packet loss,
so *m_recPoint* = *m_highInterest*, and consumer won't react to all the timeouts of the segments in the blue area until the condition *m_highData* > *m_recPoint* is true again. Therefor consumer only reacts to at most one packet loss per RTT.

Pseudo code:

```
Function OnData (data, segmentNo)
    If m_highData < segmentNo then
        m_highData = segmentNo;
    End if

    If m_cwnd < m_ssthreshold then
        m_cwnd = m_cwnd + 1;
    Else
        m_cwnd = m_cwnd + 1 / m_cwnd;
    End if

    SchedulePackets();
```

```
Function OnTimeout ()
    If m_highData > m_recPoint then
        m_recPoint = m_highInterest;
        m_ssthreshold = m_cwnd / 2;
        m_cwnd = m_ssthreshold;
        BackoffRto();
    End if

    SchedulePackets();
```

3. **Implementation**

We updated chunks application of ndn-tools repository with the congestion control algorithm mentioned above. The current version of chunks application uses a fixed window size and a "backoff and retry" strategy to deal with packet loss. Regarding to how chunks application works, please refer to "how-chunks-works.pdf" for details.

Without touching other modules, we mainly modified **pipeline-interest** module with the following changes:

- discard the use of **data-fetcher** module for Interest transmission, **pipeline-interest** directly schedules and sends Interests by itself;
- original **pipeline-interest** module uses NDN's own timeout mechanism (Interest lifetime expiration) to detect timeout, the modified version replies on RTT/RTO estimation as used by TCP.
- An internal class **SegmentInfo** is used to wrap up a sent-but-not-acknowledged segment's related information. It includes: Pending Interest ID (used to remove a timed out Interest from face), state, RTO (used for timeout detection) and time it was sent (used to calculate RTT) for that segment.
- A key data structure is a C++ std::map that maps segment number to its **SegmentInfo** object.

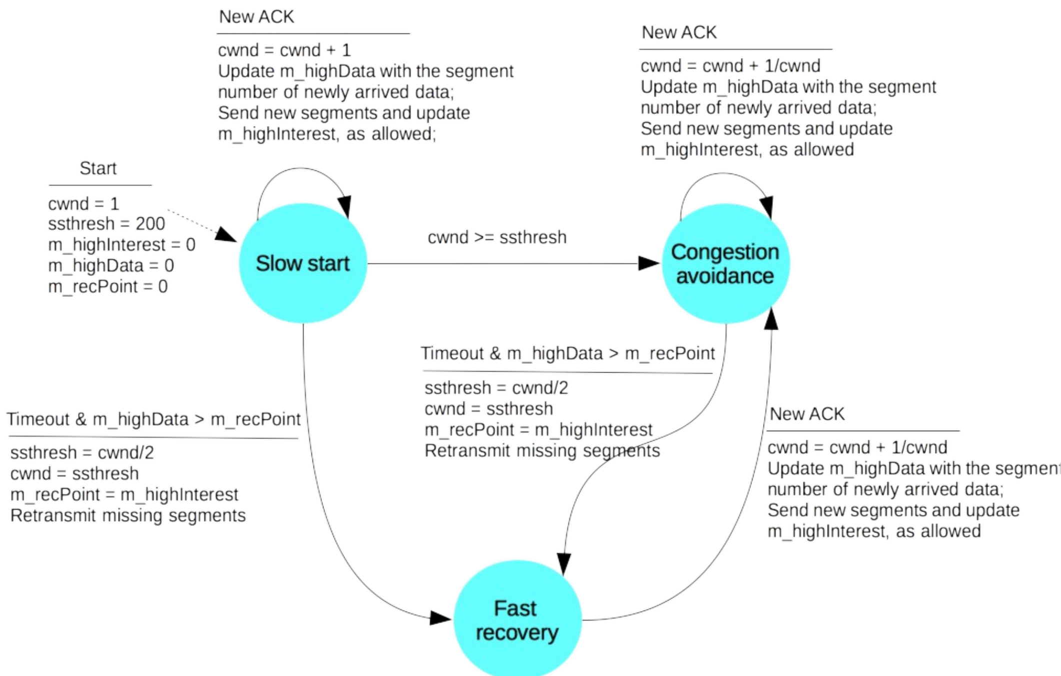
```
std::map<uint64_t, shared_ptr<SegmentInfo>> m_segmentInfoMap;
```

- an event is scheduled every 10ms (configurable) to check timed out segments. It works by scanning the m_segmentInfoMap, for each sent-but-not-acknowledged segment, calculate how long has passed since it was sent out, if greater than the RTO value stored in **SegmentInfo** object associated with that segment, time out that segment.

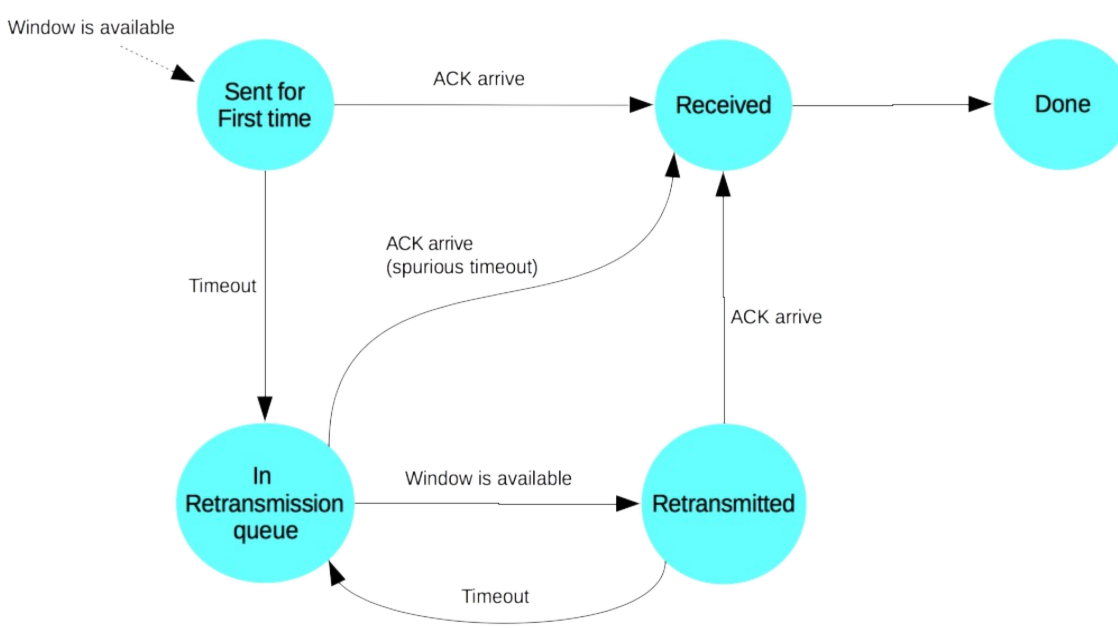
Added modules and features:

- added a **rtt-estimator** module which implements a mean-deviation RTT estimator as elaborated in RFC6298;
- if -v (verbose) option is on, a brief performance summary will be printed out on the stderr after downloading finishes;
- added a new command line option -s (keep stats) to output statistics to files after downloading finishes;

State diagram for congestion control:



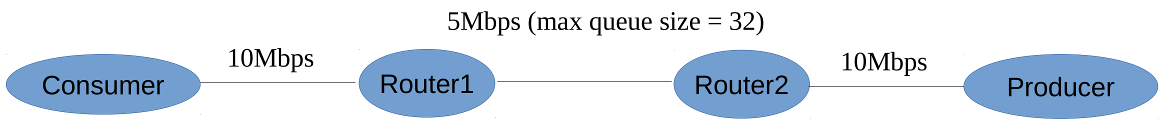
State diagram for segment:



4. **Experimentation**

Experiment environment: Minindn
Size of the file being transferred: 10MB
Topology: linear and dumbbell

linear topology (bottleneck link: Router1 --- Router2):

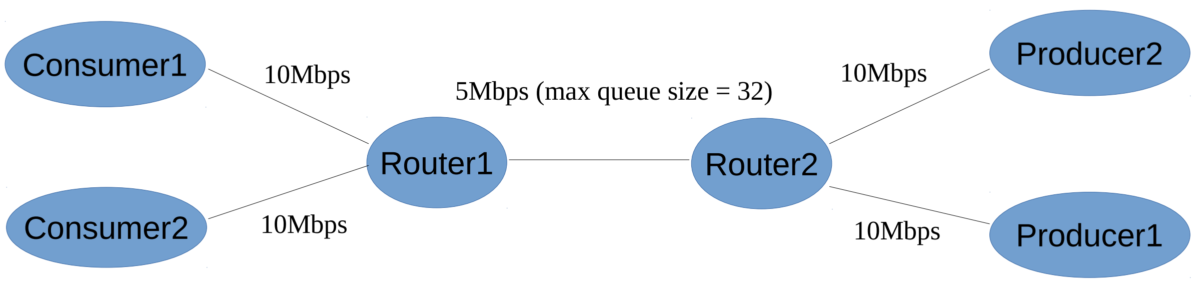


Traffic: consumer downloads file from producer

Minindn configuration for linear topology (linear.conf):

```
[nodes]
consumer1: _
router1: _
router2: _
producer1: _
[links]
consumer1:router1 delay=10ms bw=10
router1:router2 delay=10ms bw=5 max_queue_size=32
router2:producer1 delay=10ms bw=10
```

dumbbell topology (bottleneck link: Router1 --- Router2):



Traffic: cross traffic (consumer1 downloads file from producer1 and consumer2 downloads file from producer2).

Minindn configuration for dumbbell topology (dumbbell.conf):

```
[nodes]
consumer1: _
router1: _
producer1: _
consumer2: _
router2: _
producer2: _
[links]
consumer1:router1 delay=10ms bw=10
router1:router2 delay=10ms bw=5 max_queue_size=32
```

```
router2:producer1 delay=10ms bw=10
consumer2:router1 delay=10ms bw=10
router2:producer2 delay=10ms bw=10
```

Minindn script for running the experiment (ndnchunk_experiment.py): see attached.

Command for running the experiment:

```
mini-ndn$ sudo ./install.sh -i
mini-ndn$ sudo minindn --experiment=ndnchunk ./ndn_utils/topologies/linear.conf
mini-ndn$ sudo minindn --experiment=ndnchunk ./ndn_utils/topologies/dumbbell.conf
```

5. **Results analysis and comparison**

Performance Metrics:

- Download time: total time it takes to download the file
- Effective throughput:
  (number of data received * size of data packet (including header overhead)) / (download time)
- packet loss rate:
  number of packet loss bursts happened / total number of packets received

Plots:

- congestion window size changes over time
- RTT samples taken over time
- RTT measured for each segment and its caculated RTO

Comparison:

- Design #0:
  - Fixed cwnd with optimal value (32 for linear topology, 16 for dumbbell topology)
- Design #1:
  - AIMD scheme
  - Consumer reacts to multiple packet losses per RTT
  - Consumer takes multiple RTT samples per RTT
- Design #2:
  - AIMD scheme
  - Consumer reacts to one packet loss event per RTT
  - Consumer takes multiple RTT samples per RTT
- Design #3: our design

Results:

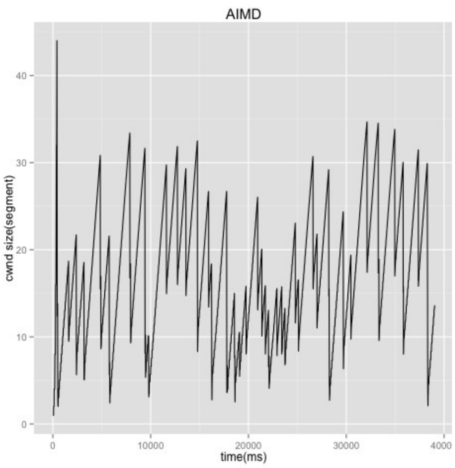
linear topology:

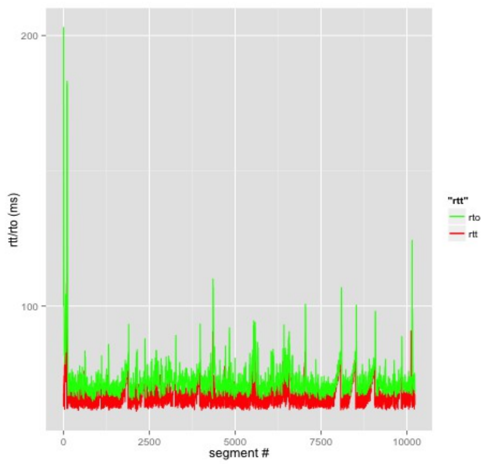
Design #0:

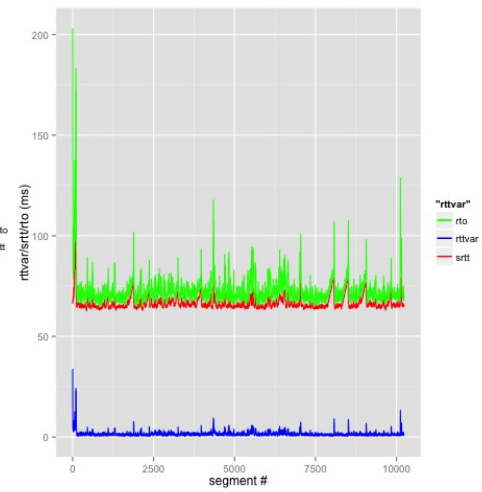
Time (s)	Throughput (kbps)	Timeout percentage
23.8	4986	0%
23.8	4984	0%
23.9	4969	0%

Design #1:

Time (s)	Throughput (kbps)	Timeout percentage
39	3038	1.3%
36.9	3213	1.1%
34	3497	0.85%

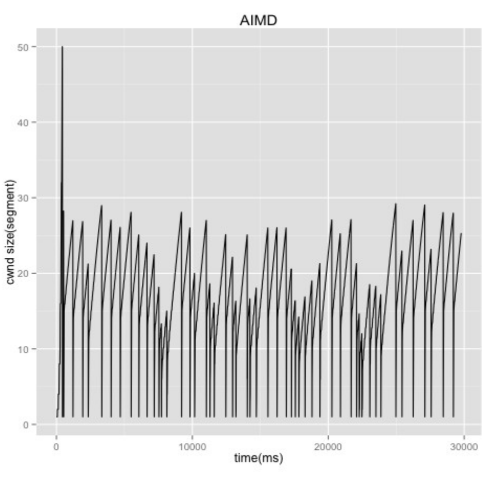


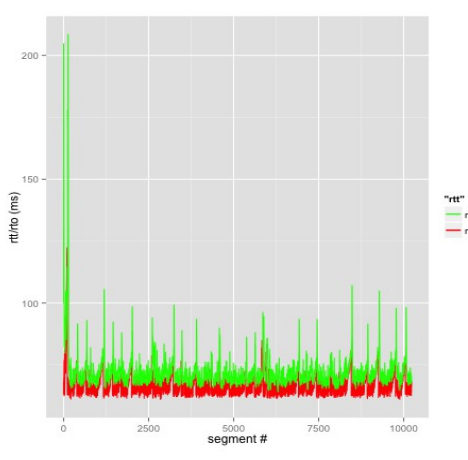


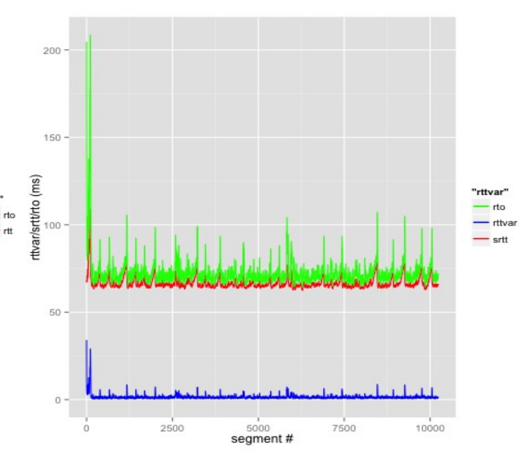


Design #2:

Time (s)	Throughput (kbps)	Timeout percentage
29.8	3983	0.65%
30.4	3900	0.4%
31.1	3808	0.4%

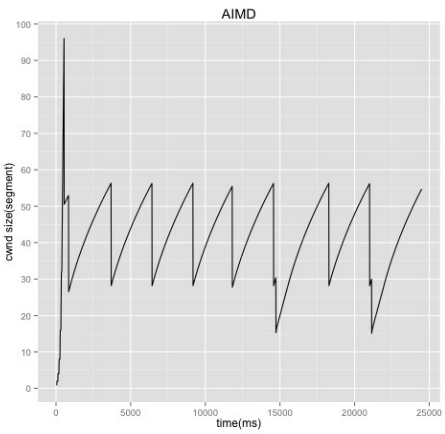


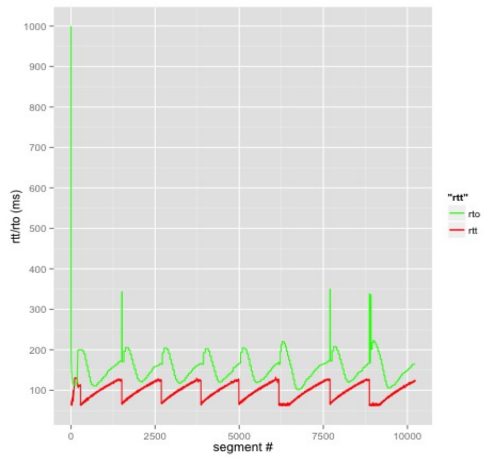


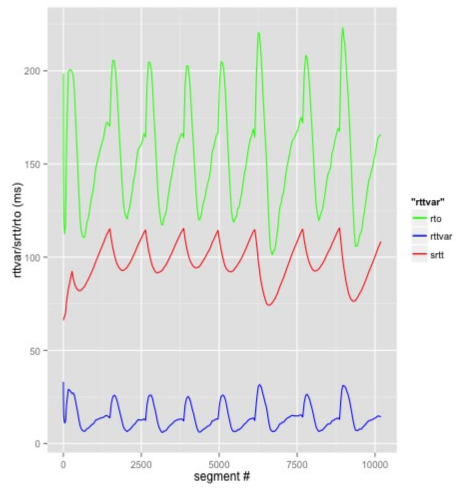


Design #3:

Time (s)	Throughput (kbps)	Timeout percentage
24.5	4844	0.1%
25	4745	0.14%
25	4748	0.13%







dumbbell topology:

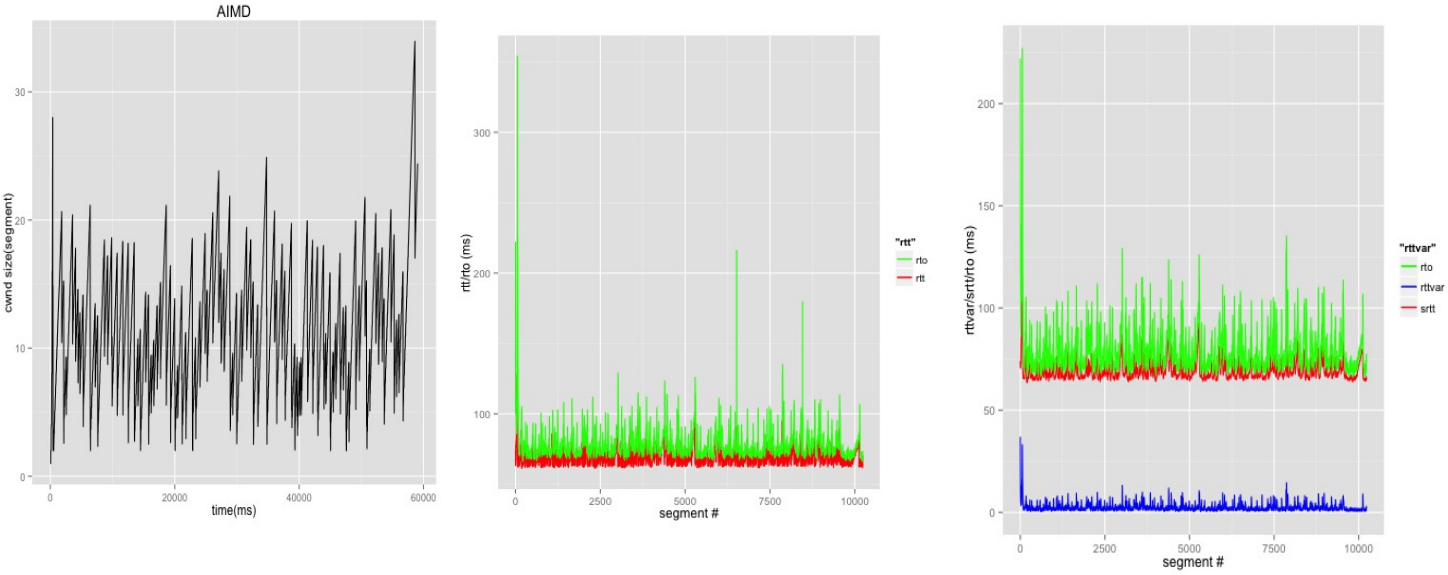
Design #0:

Consumer1			Consumer2		
Time (s)	Throughput (kbps)	Timeout percentage	Time (s)	Throughput (kbps)	Timeout percentage
47.5	2496	0	47.5	2496	0
47.8	2483	0	47.8	2481	0
47.7	2487	0	47.7	2487	0

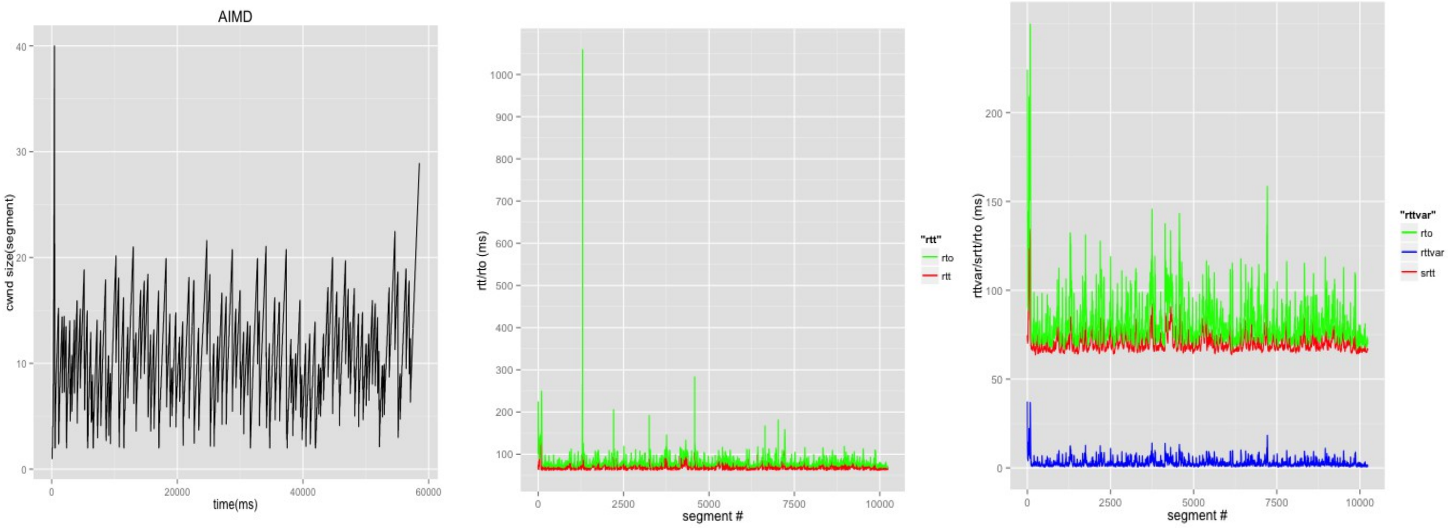
Design #1:

Consumer1			Consumer2		
Time (s)	Throughput (kbps)	Timeout percentage	Time (s)	Throughput (kbps)	Timeout percentage
59	2009	2.3%	56.7	2094	2.7%
57	2082	3.4%	58.5	2027	3.2%
54.3	2183	2.2%	48.4	2452	2.4%

Plots for Consumer1:



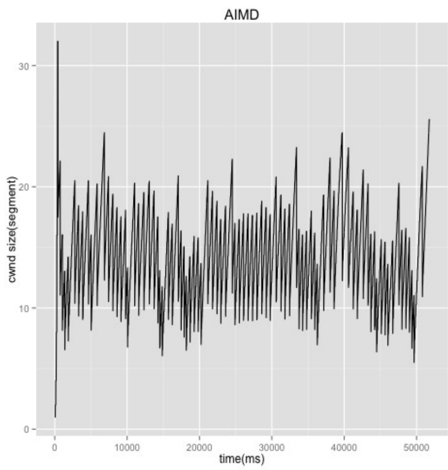
Plots for Consumer2:

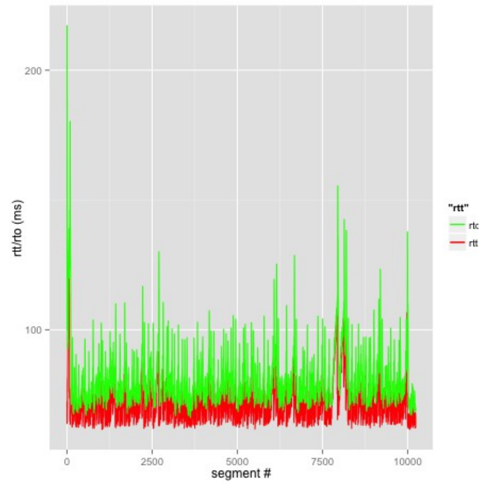


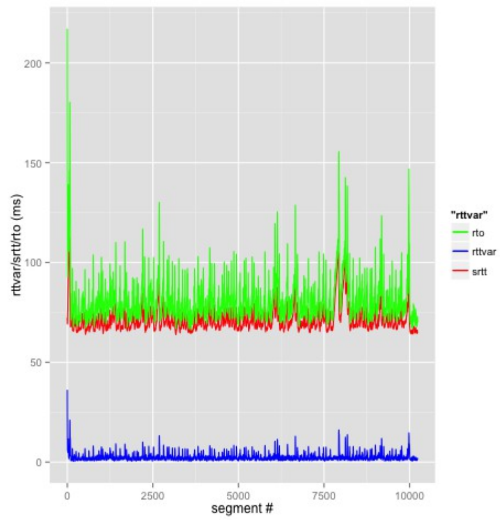
Design #2:

Consumer1			Consumer2		
Time (s)	Throughput (kbps)	Timeout percentage	Time (s)	Throughput (kbps)	Timeout percentage
51.8	2292	0.93%	50.7	2342	0.9%
52.9	2243	1.1%	52.9	2242	1.1%
53.3	2227	1.6%	51.5	2301	1.2%

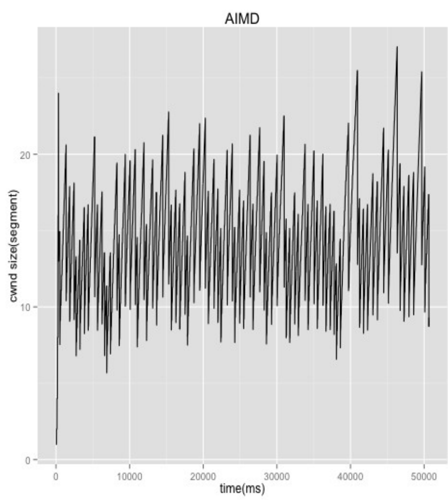
Plots for Consumer1:

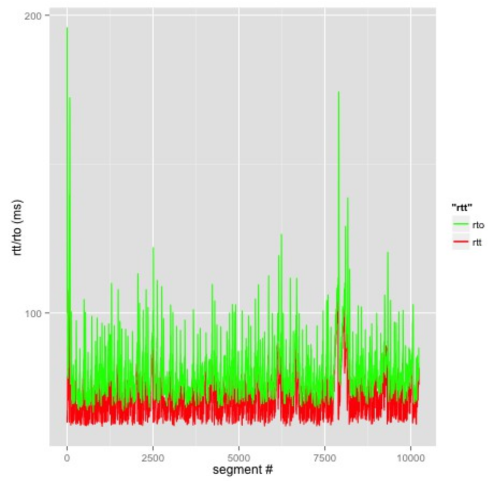


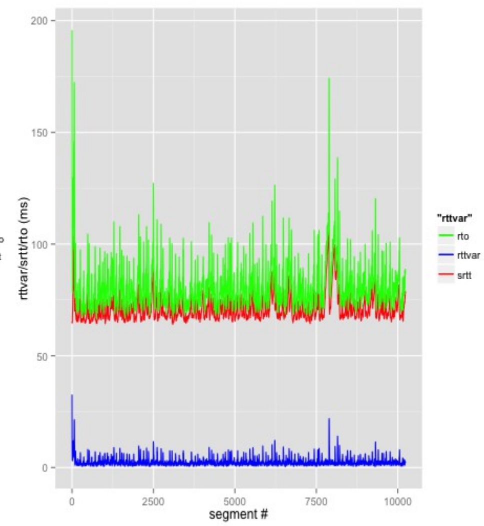




Plots for Consumer2:



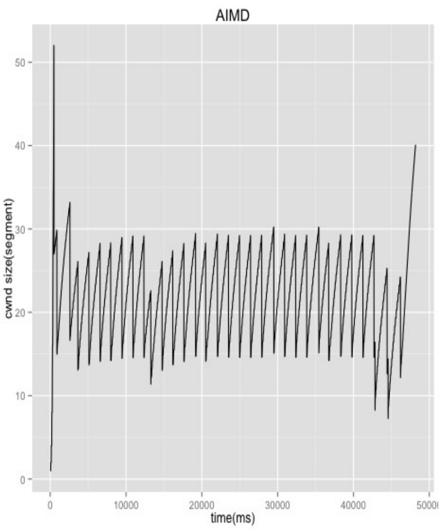


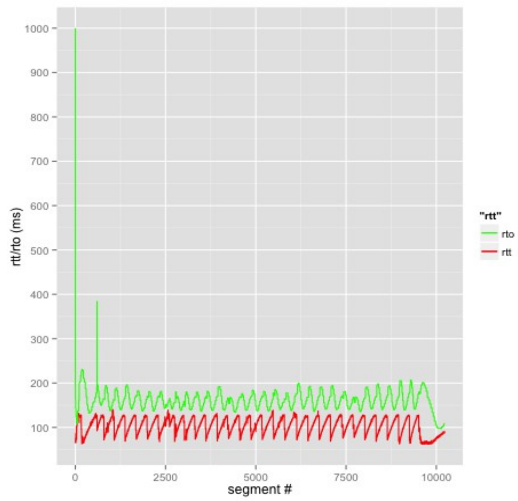


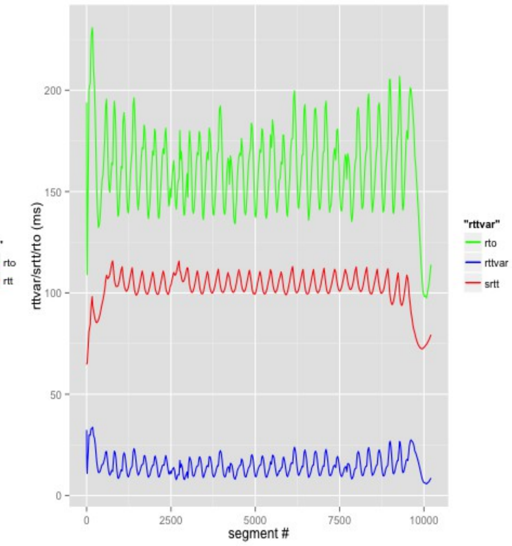
Design #3:

Consumer1			Consumer2		
Time (s)	Throughput (kbps)	Timeout percentage	Time (s)	Throughput (kbps)	Timeout percentage
48.2	2462	0.34%	46.4	2555	0.31%
47.7	2488	0.33%	48	2471	0.34%
48.1	2467	0.34%	45.4	2616	0.29%

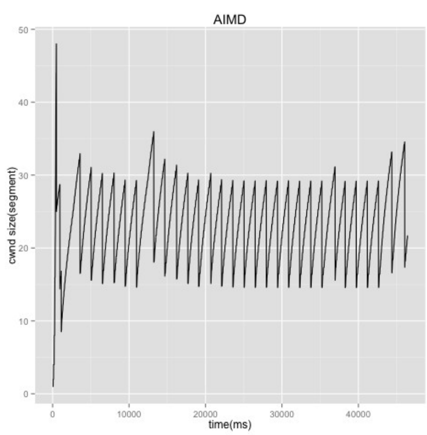
Plots for Consumer1:

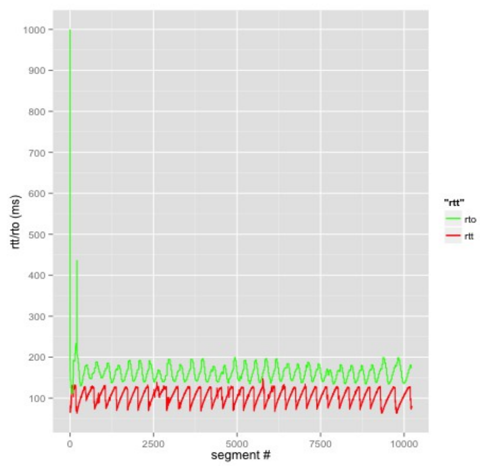


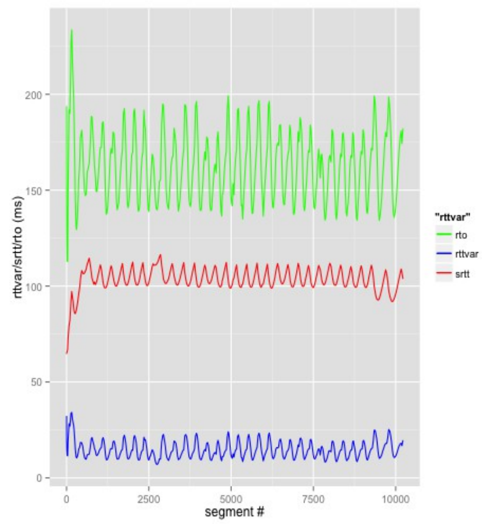




Plots for Consumer2:







Observations & Analysis:

- Design #0 sets a baseline for performance comparison, other designs can perform no better than it.
- Consumer needs to wait for RTO to expire to be aware of congestion.
- In Design #1, due to multiple packet losses within one RTT, the connection cannot reach the equilibrium state, which causes the packet conservation to fail.
- Design #2 has problem of making full use of available window size, most likely due to inadequate estimation of RTT and RTO values.
- Design #3 yields performance very close to that of Design #0, most time it can make full use of congestion window and dynamically react to congestion condition in time. The overhead, comparing to Design #0 would be the process of adjusting window size, especially linear increase of window size. It also shows that taking one sample per RTT yields better RTT & RTO estimation.
- For the dumbbell topology, two consumers can share the bottleneck link bandwidth evenly most of time.