Implementing TCP SACK Conservative Loss Recovery Algorithm within a NDN Consumer

Shuo Yang

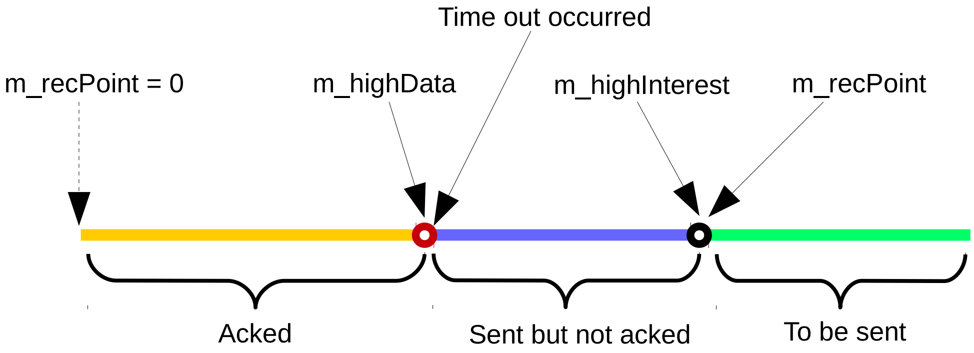
1. **Design**

- 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.

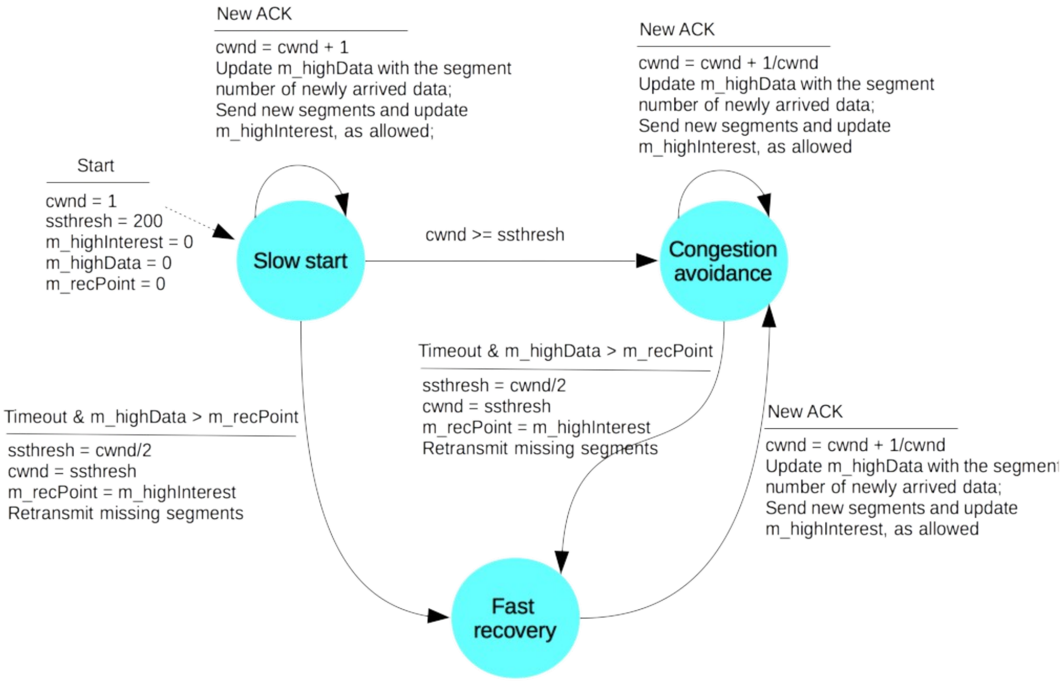
```
std::map<uint64_t, shared_ptr<SegmentInfo>> m_segmentInfoMap;
```

- 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.

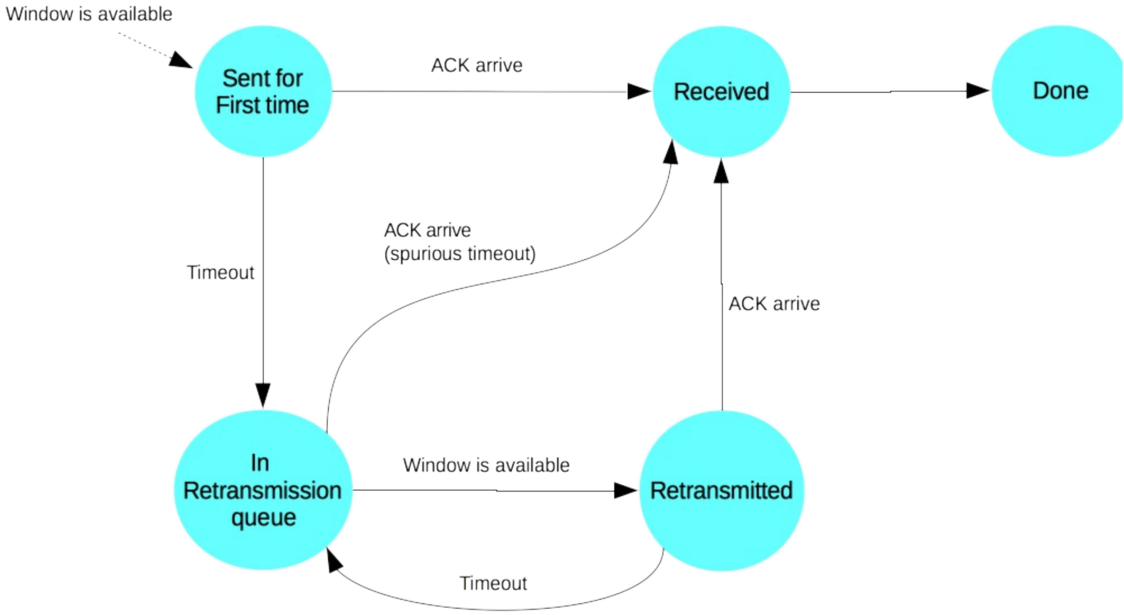
Added modules and features:

- 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:


New ACK
cwnd = cwnd + 1
Update m_highData with the segment
number of newly arrived data;
Send new segments and update
m_highInterest, as allowed;
New ACK
cwnd = cwnd + 1/cwnd
Update m_highData with the segment
number of newly arrived data;
Send new segments and update
m_highInterest, as allowed
Start
cwnd = 1
ssthresh = 200
m_highInterest = 0
m_highData = 0
m_recPoint = 0
Slow start
cwnd >= ssthresh
Congestion avoidance
Timeout & m_highData > m_recPoint
ssthresh = cwnd/2
cwnd = ssthresh
m_recPoint = m_highInterest
Retransmit missing segments
New ACK
cwnd = cwnd + 1/cwnd
Update m_highData with the segment
number of newly arrived data;
Send new segments and update
m_highInterest, as allowed
Timeout & m_highData > m_recPoint
ssthresh = cwnd/2
cwnd = ssthresh
m_recPoint = m_highInterest
Retransmit missing segments
Fast recovery

State diagram for segment:


Window is available
Sent for First time
ACK arrive
Received
Done
ACK arrive
(spurious timeout)
Timeout
ACK arrive
In Retransmission queue
Window is available
Retransmitted
Timeout