Design

In this design, congestion can be detected in three different ways:

1. Re-transmission timer expires.
   action: Reset the current window size to 1.

2. A NACK received with the reason as "congestion".
   If the sequence number of the received NACK is greater than the sequence number of the last received NACK + $k*$(the current window size), where $k < 1$. the consumer should regard this as a sign of congestion. The purpose of doing this is to avoid the sharp drop of the congestion window size caused by the situation where a consecutive sequence of NACKs were received because of a burst of packet drops occurred in the network. In such case, we want only one multiplicative decrease like what TCP did.
   action: multiplicative decrease, fast re-transmission and fast recovery.

3. A hole in the packets sequence detected.
   When the consumer sends interest packets out, it records the order of the sent interests. When the data packets come back, the consumer can check whether the data packets are in the same order as the interests. The assumption is that most time data packets should arrive in order. If the consumer expected the N_{th} data packet to arrive, but instead the M more data after it have arrived, the consumer regards this as a sign of congestion.
   action: multiplicative decrease, fast re-transmission and fast recovery.

Important Data Structures and Variables

sentQueue: this queue is used to hold the interest packets that are in the current window, i.e., they have not been acknowledged yet. Once the consumer received the data packets, it removes the corresponding entries in the queue.

retxQueue: this queue is used to hold the interest packets that should be re-transmitted due to congestion.

retxedList: this list is used to hold the interest packets that have been re-transmitted and not acknowledged yet.

outOfOrderDataList: this list is used to hold the received data packets that are out of order. As long as the consumer received the data packet with the expected sequence number, this list should be set to empty.

cwnd: congestion window size
ssthresh: slow start threshold
nextSeqNum: the sequence number of the next interest packet to be sent out in order. This number should be monotonically increasing.
lastReceivedNack: the most-recently received NACK packet.
RTO: value of the current calculated RTO

Adjustable Parameters

initialCwnd: initial value for cwnd
initialSsthresh: initial value for ssthresh
MDcoef: coefficient for multiplicative decrease. In TCP, its value is 0.5
AIstep: incremental quantity for additive increase. In TCP, its value is 1
maxOutOfOrderData: the maximum number of consecutively arrived out-of-order data packets the consumer

can withstand until triggering a congestion signal
rtoBackoffMultiplier: the multiplier for RTO back off. In TCP, its value is 2

Congestion Control Algorithm

```
Init()
  cwnd = initialCwnd
  ssthresh = initialSsthresh
  nextSeqNum = 1
  lastReceivedNack = null
  sentQueue = []
  retxQueue = []
  retxedList = []
  outOfOrderDataList = []
  maxOutOfOrderData = 3
  ScheduleNextInterest() /* send out the first interest */

ScheduleNextInterest()
  if !retxQueue.empty()
    SendInterest(retxQueue.first()) /* send out interests that need re-transmission first */
    retxedList.add(retxQueue.first())
    retxQueue.removeFirst()
  else
    SendInterest(nextSeqNum)
    nextSeqNum = nextSeqNum + 1

SendInterest(seqNum)
  sentQueue.add(seqNum)
  timer[seqNum] = RTO
  transmitInterest(seqNum)

OnData(data)
  if !retxedList.contains(data.seqNum) /* do not take re-transmitted packets as RTT samples */
    UpdateRTO(EstimateRTT(data.seqNum))
  else
    retxedList.remove(data.seqNum)
  if sentQueue.first() == data.seqNum /* expected data packet */
    outOfOrderData.clear()
    if cwnd < ssthresh
      cwnd = cwnd + AIstep /* additive increase */
    else
      cwnd = cwnd + AIstep/cwnd /* congestion avoidance */
  else /* data arrived out of order */
    outOfOrderData.add(data.seqNum)
    if outOfOrderData.size() > maxOutOfDrderData /* congestion signal */
      ssthresh = MDcoef * cwnd
      cwnd = ssthresh /* fast recovery */
      retxQueue.add(sentQueue.first())
      ScheduleNextInterest() /* fast re-transmission */
    else
      /* don't increase cwnd here */

  sentQueue.remove(data.seqNum) /* remove the acknowledged interest from the queue */
```

```
OnNack(nack)
    if nack.seqNum > lastReceivedNack.seqNum + cwnd
       ssthresh = MDcoef ∗ cwnd
       cwnd = ssthresh /∗ fast recovery ∗/
       lastReceivedNack = nack /∗ update ∗/

    retxQueue.add(nack.seqNum)
    ScheduleNextInterest() /∗ fast re-transmission ∗/

OnTimeout(seqNum)
    ssthresh = MDcoef ∗ cwnd
    cwnd = 1
    RTO = rtoBackoffMultiplier * RTO

    retxQueue.add(seqNum)
    ScheduleNextInterest()
```

RTT Estimation and RTO Calculation

Design Principles

- Compliant with RFC 6298 (https://tools.ietf.org/html/rfc6298).
- Using Karn's algorithm, i.e., don't take RTT sample from re-transmitted packets.
- Adjust parameters to common TCP defaults.

Variables and Parameters

RTO	re-transmission timeout
SRTT	smoothed round-trip time
RTTVAR	round-trip time variation
G	clock granularity (unit: seconds)
alpha	RTT estimation filter gain
beta	variance filter gain
MINRTO	lower-bound of RTO
MAXRTO	upper-bound of RTO
K	coefficient for calculating RTO

Initial Variable and Parameter Settings

```
MINRTO = 1; //initially, the sender should set RTO to 1s, as suggested in RFC 6298
RTO = MINRTO;
alpha = 1/8; // suggested in RFC 6298
beta = 1/4; // suggested in RFC 6298
MAXRTO = 60; // RFC 6289 suggests this value should be at least 60s
G = 0.1; // 0.1 second, RFC 6298 suggested that finer clock granularities (<= 100ms) perform better than coarser granularities.
```

RTT estimation and RTO calculation (R is the current RTT measurement)

```
RttEstimation(R)
      if it is the very first measurement:
            SRTT = R;
            RTTVAR = R/2;
      else: // subsequent RTT measurement
```

RTTVAR = (1 - beta) * RTTVAR + beta * |SRTT - R|;
SRTT = (1 - alpha) * SRTT + alpha * R;

Note that RTT sample should not be taken from the re-transmitted packets according to Karn's algorithm.

After computing SRTT and RTTVAR, sender should update RTO:

RTO = SRTT + max (G, K * RTTVAR)

Note that if the computed RTO is less than 1 second, it should be rounded up to 1 second, as suggested in RFC 6298. RFC 6298 recommends a large minimum value on the RTO to keep TCP conservative and avoid spurious re-transmission.

Algorithm for managing the re-transmission timer

note: the algorithm is recommended in RFC 6298, terminologies are adjusted for NDN. Also section (5.7) is specific to TCP's three-way handshake, so is not included.

1. Every time an interest packet is sent (including a re-transmission), if the timer is not running, start it running so that it will expire after RTO seconds (for the current value of RTO).
2. When the outstanding interest packets have been acknowledged (by their corresponding data packets), turn off the re-transmission timer.
3. When a data packet is received that acknowledges a new interest packet, restart the re-transmission timer so that it will expire after RTO seconds (for the current value of RTO).

When the re-transmission timer expires, do the following:

1. Re-transmit the earliest interest packet that has not been acknowledged yet.
2. The consumer must set RTO = RTO * 2 ("exponentially back off the timer"). The value of MAXRTO may be used to provide an upper bound to this doubling operation.
3. Start the re-transmission timer so that it will expire after RTO seconds (for the value of RTO after the doubling operation).