Tengfei Ma 121090406

I. Algorithm Analysis

a) My algorithm is PANDA(Probe and Adapt) according to ***Probe and Adapt: Rate Adaptation for HTTP Video Streaming At Scale***

b) Rate oscillation and instability behaviors are not incidental. To overcome this, envision a solution based on a "probe and adapt" principle. In this approach, the TCP downloading throughput is taken as an input only when it is an accurate indicator of the fair-share bandwidth. In the presence of off-intervals, the algorithm constantly probes the network bandwidth by incrementing its sending rate, and prepares to back off once it experiences congestion.

II. Implementation

a) Theoretical Implementation

i. Estimate the bandwidth share x[n] by

x[n] = kappa * (omega - max(0, x[n-1] - Previous_Throughput+omega))*Previous_actual_inter_request_time+x[n-1]

,where kappa=0.14, omega=0.3,

Previous_actual_inter_request_time=max(Previous_target_inter_request_time, Segment_download_duration)

ii. Smooth out x[n] to produce filtered version y[n] by (EWMA smoother)

y[n] = -alpha*(y[n-1]-x[n])*Previous_actual_inter_Request_time+y[n-1]

,where alpha=0.2

iii. Quantize y[n] to the discrete video bitrate r[n]∈ Available_bitrate by (dead-zone quantizer)

$$r[n] = \begin{cases} r\_up & \text{if } r[n-1] < r\_up \\ r[n-1] & \text{if } r\_up <= r[n-1] <= r\_down \\ r\_down & \text{otherwise} \end{cases}$$

,where r_up = $max_{r \in Available\ bitrate}$ r subject to r <= y[n] - epsilon*y[n],

r_down = $max_{r \in Available\ bitrate}$ r subject to r <= y[n]

iv. Schedule the next download request via

$$target\_inter\_request\_time = \frac{r[n]*VideoSegmentDuration}{y[n]} + beta*(B[n-1]-B\_min)$$

,where B[n-1] is client buffer duration, B_min=26,beta=0.2

b) Code Implementation

i.
```
targetInterRequestTime_previous = 0.0
Video_Time_previous = 0.0
bitrate_previous = 0.0
bandwidthShare_previous = 0.0
smoothedBandwidthShare_previous = 0.0

def student_entrypoint(Measured_Bandwidth, Previous_
    #student can do whatever they want from here goi

    R_i = list(Available_Bitrates.items())
    R_i.sort(key=lambda tup: tup[1] , reverse=True)
    global targetInterRequestTime_previous
    global Video_Time_previous
    global bitrate_previous
    global bandwidthShare_previous
    global smoothedBandwidthShare_previous
```
I set some global variable to provide information to next chunk, and R_i is available bitrate set, which is arranged in descending order.

ii.
```
if (Chunk["current"] == "0"):
    bitrate = int(R_i[-1][0]) # choose lowest bitrate for start
    bitrate_previous = bitrate
    Video_Time_previous = Video_Time
    return bitrate

if (Chunk["current"] == "1"):
    # set initial value
    bandwidthShare_previous = Previous_Throughput
    smoothedBandwidthShare_previous = bandwidthShare_previous
```
Set some initial values for first two chunks.

iii.
```
segmentDownloadDuration = Video_Time - Video_Time_previous
Video_Time_previous = Video_Time # prepare for next chunk
actual_interRequestTime_previous = max(targetInterRequestTime_previous, segmentDownloadDuration)
```
Calculate actual inter-request time

iv.
```
# step 1: Estimate the bandwidth share
kappa = 0.14 # according to article, all parameter is picked as default value
omega = 0.3
bandwidthShare = kappa*(omega - max(0.0,bandwidthShare_previous - Previous_Throughput + omega)) * actual_interRequestTime_previous + bandwidthShare_previous
if bandwidthShare < 0: # bandwidth share can not be negative
    bandwidthShare = 0
bandwidthShare_previous = bandwidthShare # prepare for next chunk
```
calculate bandwidth share for step 1.

v.
```
# step 2: Smooth out estimated bandwidth share by EWMA smoother
alpha = 0.2
smoothedBandwidthShare = (-alpha)*(smoothedBandwidthShare_previous - bandwidthShare) * actual_interRequestTime_previous + smoothedBandwidthShare_previous
smoothedBandwidthShare_previous = smoothedBandwidthShare # prepare for next chunk
```
Calculate smoothed bandwidth share for step 2.

vi.
```
# step 3: Quantize smoothed bandwidth share to the discrete video birate by dead-zone quantizer
epsilon = 0.15
delta_up = epsilon * smoothedBandwidthShare
r_up = 0.0
for i in range(len(R_i)): # find max available bitrate subject to the bitrate <= (smoothedBandwidthShare - delta_up)
    if int(R_i[i][0]) <= (smoothedBandwidthShare - delta_up):
        r_up = int(R_i[i][0])
        break

delta_down = 0.0
r_down = 0.0
for i in range(len(R_i)): # find max available bitrate subject to the bitrate <= (smoothedBandwidthShare - delta_down)
    if int(R_i[i][0]) <= (smoothedBandwidthShare - delta_down):
        r_down = int(R_i[i][0])
        break

if r_down == 0.0: # if we can not find the value, we pick min available bitrate
    r_down = int(R_i[len(R_i)-1][0])
if r_up == 0.0:
    r_up = int(R_i[len(R_i)-1][0])
# calculate bitrate by "dead zone" [r_up, r_down]
if bitrate_previous < r_up:
    bitrate = r_up
elif bitrate_previous <= r_down:
    bitrate  = bitrate_previous
else:
    bitrate = r_down

bitrate_previous = bitrate # prepare for next chunk
```
Calculate bitrate for this chunk request. The bitrate is what we want.

vii.
```
# step 4 : Schedule the next download request
B_min = 26
beta = 0.2
chunkTime = Chunk["time"]
B_n_prev = Buffer_Occupancy["time"]
targetInterRequestTime_previous = bitrate * chunkTime / smoothedBandwidthShare + beta * (B_n_prev - B_min)
```
Calculate target inter-request time for next chunk.

III. Evaluation

a)
```
testHD:
Results:
Average bitrate:983333.3333333334
buffer time:0.202
switches:1

Score:895341.5864155713
```
```
badtest:
Results:
Average bitrate:500000.0
buffer time:1.012
switches:0

Score:474707.7181838023
```
```
testALTsoft:
Results:
Average bitrate:983333.3333333334
buffer time:0.202
switches:1

Score:895341.5864155713
```
```
testALThard:
Results:
Average bitrate:500000.0
buffer time:1.001
switches:0

Score:474975.6363100182
```
The test result is shown.

b) The PANDA player has the best stability-responsiveness trade-off, and also has the best bandwidth utilization. In competing with long-running TCP or the most aggressive HAS client, PANDA shows fair competency. The absolute efficiency may not be excellent enough due to some trade-off. The performance of this algorithm has its own strength on estimating the next round bandwidth, which can help to choose next bitrate for client and control both download time and video quality.