Project 3: Internet Video Delivery

Project Goals and Background:

This project pertains to Adaptive Bit Rate (ABR) algorithms for Internet video delivery. The project will involve reading papers published in top scientific conferences on computer networking, and implementing and evaluating ABR algorithms proposed in them on a custom ABR simulator that we have built. You will also critique these algorithms, identify when they work well and when they do not, explore design variants, and turn in your results and findings in a research report that you will submit.

When delivering Internet video, content is split into chunks (each chunk corresponding to a few seconds of video play time). Each chunk is encoded at different video qualities. An ABR algorithm (typically implemented in the client) must decide what quality to fetch each chunk at while simultaneously optimizing several objectives (i) avoiding rebuffering events at the client; (ii) maximizing the average quality fetched across all chunks; and (iii) minimizing variation in quality across chunks. It is common to capture performance using a composite metric that combines all of these metrics. The Composite Metric comprises a reward for fetching higher quality chunks, and penalties for rebuffering and quality variation, with the weights for rebuffering and quality change metrics set by each video publisher based on its preferences (see "Overview on Video Streaming and User Experience" or the ***MPCSigcomm15*** paper below for more details). In making its decisions, some ABR algorithms require a prediction of future bandwidth, with methods for predicting bandwidth itself ranging from simple ones (e.g., take the mean of the throughput of the last few chunks) to more sophisticated techniques actively being researched.

There are two mandatory readings associated with the project:

[MPCSigcomm15] "A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP", Yin et al., ACM Sigcomm 2015
[BufferSigcomm14] "A buffer-based approach to rate adaptation: evidence from a large video streaming service", Huang et al., ACM Sigcomm 2014.

Specific requirements for this project:

In this project, we require you to:

1. Implement, evaluate, and compare the (i) RobustMPC algorithm in the MPCSigcomm15 paper, as well as a variant of the algorithm that you devise; and (ii) the BBA-2 algorithm in the BufferSigcomm14 paper, as well as a variant of the algorithm of your choosing/that you devise. RobustMPC and BBA-2 have been chosen as the base algorithms not necessarily because they are the best algorithms, but because they have been published in reputable scientific forums, are well cited, represent interesting design decisions, and we believe the algorithms here are feasible

to implement within the constraints of a course project, while not being too trivial to implement

2. For the variant of RobustMPC, you may implement FastMPC (where throughput estimates are not discounted). For the variant of BBA-2, you may implement BBA-0, or BBA-1. However, we encourage you to design and implement more innovative variants (for at least one base algorithm), including possibly variants that out-perform the RobustMPC and BBA-2 algorithms. Note that our emphasis here is on the variant being interesting/ thought-provoking – if it ultimately did not result in performance that met your expectations, but you can document it well, and explain why performance fell short, that would still be considered a worthwhile effort. You are welcome to implement more than one variant of each algorithm, though quality (more interesting variants with interesting insights) is better than quantity (a large list of minor variants, but no particularly interesting nuggets).
3. Turn in a report which provides an analysis of these algorithms and the variants. The report should document the performance of the two algorithms (and their variants) and a comparison on a set of benchmark configurations/bandwidth traces that we provide.

Hints on implementing the MPCSigcomm15 algorithm:

The algorithm published in the MPCSigcomm15 paper may feel intimidating at first, with a fair bit of math. However, the algorithm is not difficult to implement, and some useful hints are provided below.

Consider that chunks ***1,2,..i*** have been downloaded. The algorithm must determine what quality to download for chunk ***i+1***. We suggest the following for the implementation:

1. Consider a look-ahead window of ***W*** chunks.
2. Evaluate all possible choices of chunk qualities for the next W chunks, and compute a score for each possible choice. Concretely, assume W=5, and there are 3 different chunk qualities. Then, there are 3^5=243 different possible sequences to evaluate. Find the sequence with the best score, and accordingly choose the quality for chunk ***i+1***
3. Computing the score for each sequence itself involves simulating the effects of each sequence (e.g., determining if that sequence would trigger a rebuffering event).

Step 2 is not clearly described in the paper. While we recommend a brute force enumeration here for ease of implementation, the paper talks about solving an optimization problem, but does not detail it sufficiently. For those interested, a more efficient implementation is feasible using a dynamic program, but that is not required for the course project. That said, an ambitious student may choose to implement a dynamic program as the RobustMPC variant (with the benefit being computation efficiency).

Overview on Video Streaming and User Experience

Video Streaming

When videos are streamed from the internet, they are typically segmented into chunks. Each chunk of video contains the information for a few seconds of video, which is typically anywhere from one to four seconds.

For example, if our chunk size were two seconds, then the first chunk streamed contains the video file from time t=0 to time t=2 second. The next chunk contains t=2 to t=4 seconds, and so on. Client applications request chunks to download one at a time and save them in the buffer before streaming them to the user.

While the duration of each chunk is the same, chunks can be different sizes for two reasons: Variable Bitrate and Quality.

Variable Bitrate (VBR) is a method of video encoding that allows videos to be streamed more efficiently. In low-motion scenes (such as still images or an opening credit with a black background), VBR can encode chunks with fewer bits. What this means for your streaming application is that chunks can vary in size significantly depending on the content being streamed.

Quality of video also affects the size of chunks. When streaming very high-quality video, chunks will take a lot of data to be encoded. The opposite is true for low quality chunks. Typically, a server will save several copies of a video encoded with different bitrates. For each chunk fetched, a client can choose which of these videos to stream.

For the purposes of this project, each chunk has a “chunk size ratio” which controls how much data it takes to encode a chunk using VBR. Additionally, chunks of video can be fetched in one of several quality levels. Your algorithm is responsible for choosing which quality to fetch.

Calculating User Quality of Experience

Two factors very important to users are the quality of the video and the rebuffer rate. Ideally, your algorithm will pick the highest quality chunks it can while also avoiding rebuffering as much as possible.

Additionally, users are worried about the overall variation in quality. Imagine watching a video that changes quality every three seconds-- annoying! Therefore, there are three variables to optimize when building your algorithm:

1. Maximize overall chunk quality. Give the user the highest quality video possible.
2. Minimize rebuffer time. Don’t have the user waiting while the next chunk is fetched (alternatively, “keep the client buffer > 0 seconds at all times”)
3. Minimize quality variation. Give the user a consistent viewing experience.

The relative importance of these three metrics is controlled by three coefficients: The quality coefficient, variation coefficient, and rebuffer coefficient. If a user is very interested in one aspect, the corresponding coefficient will be higher: e.g. a user interested in quality will have a high-quality coefficient.

Overall user Quality of Experience (QoE) for viewing one video is calculated as follows:

$$QoE = [(Qual\ Coef)\ (Total\ Chunk\ Qual)\ - (Var\ Coef)\ (Total\ Quality\ Var) - (Rebuff\ Coef)\ (Rebuff\ Time)]\ /\ (Video\ Length)$$

Additionally, many papers consider join latency, or the time between a user starting the video and the first chunk arriving. In our simulator, we consider join latency as rebuffering time.

ABR Simulator Test Cases

The ABR simulator test cases are found in the tests/ directory of the starter code. Each defines parameters that control the video, user quality calculation, and client download throughput for the current viewing session. A description of each parameter is below. **More details can be found by reading the comments in each test file.**

Video Parameters

This defines the parameters for the specific video file being streamed. There are three parameters in this section.

- chunk_length: The number of seconds of video per chunk.
- base_chunk_size: The size of an average chunk (MB) at the lowest quality. Chunk size may still vary due to VBR encoding.
- client_buffer_size: The max seconds of video stored by receiver/client

Quality Parameters

This defines the parameters for calculating the user quality of experience. There are four parameters in this section.

- quality_levels: The number of quality levels available for streaming. In our simulator, streaming at a higher quality takes exponentially more bits; quality level 2 is twice as expensive as 1 and quality level 3 is four times as expensive as quality level 1.
- quality_coefficient: Used for calculating User Quality of Experience.
- variation_coefficient: Used for calculating User Quality of Experience.
- rebuffering_coefficient: Used for calculating User Quality of Experience.

Throughput Parameters

This defines the client throughput and chunk download rate for the given viewing session. Because client bandwidth changes, it is formatted as a list of bandwidths that the client application sees over time. **It is recommended that you write a separate script to parse this list and build a graph of the client throughput over time. This will greatly improve your understanding of what is going on for each algorithm running each test case.**

Chunk Parameters

This defines the VBR behavior of the video being streamed by the client. It is formatted as a list of chunk size ratios, one for each chunk. Again, **it would be greatly beneficial to your understanding to write a script that parses and graphs this list.**

The size for streaming a particular chunk at a particular quality level is given by:

$$Total\ size = base\ chunk\ size\ \times chunk\ size\ ratio\ \times\ 2^{quality\ level - 1}$$

There are three things that define the size in bytes:

1. Base chunk size, which is statically defined in the video parameters section
2. Chunk size ratio, which comes from the chunk parameters lists. This represents how difficult it is for VBR to encode the chunk (e.g. a chunk with a ratio 2 takes about twice the bits to encode as a chunk with ratio 1, all others held constant).
3. Quality level, which is chosen by your algorithm.

Note that chunks vary in number of bytes, but each chunk has the same number of seconds of video. As an example, let's say you'd like to stream the first chunk choosing from quality levels 1, 2, 3. The base chunk size is 1Mb and let's say the first value in the chunk size ratio list is 1.

$$Size\ corresponding\ to\ quality\ level\ 1 : 1\ \times 1\ \times 2^{1-1} = 1\ MB$$
$$Size\ corresponding\ to\ quality\ level\ 1 : 1\ \times 1\ \times 2^{2-1} = 2\ MB$$
$$Size\ corresponding\ to\ quality\ level\ 1 : 1\ \times 1\ \times 2^{3-1} = 4\ MB$$

For any given chunk, streaming at a higher quality takes more bits. However, things are more nuanced when comparing across chunks, since the chunk size ratio varies as well. Let's say the quality level chosen for chunk 1 is 1, the quality level chosen for chunk 2 is 2, and the quality level chosen for chunk 3 is 3. However, the chunk_size_ratios for the first 3 chunks are 1, 2, .5. Then,

$$Size\ of\ first\ chunk : 1\ \times 1\ \times 2^{1-1} = 1\ MB$$
$$Size\ of\ second\ chunk : 1\ \times 2\ \times 2^{2-1} = 4\ MB$$
$$Size\ of\ third\ chunk : 1\ \times 0.5\ \times 2^{3-1} = 2\ MB$$

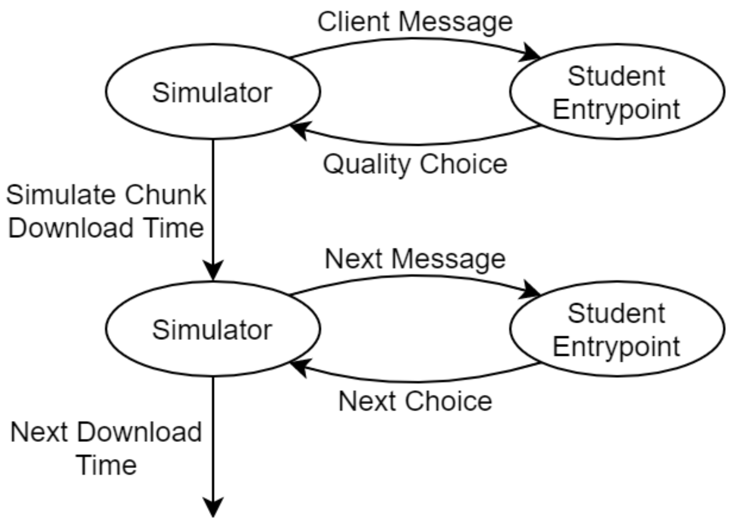
Because variable-bitrate encoding allowed the third chunk to be encoded more cheaply, we could stream at a higher quality and use fewer bits than the second chunk. Algorithms (MPC, BBR 2) can take advantage of this by increasing the quality of "cheap" chunks such as the third while decreasing the quality of "expensive" chunks such as the second.

Building and Running Your Solution

The Simulator and Your Code

For this project, we have created an ABR simulator that replicates a client streaming session. Your code will reside in the student/ folder of the simulator in the student_entrypoint() function. For each chunk, the simulator will send a "client message" to your student_entrypoint(). Your function will then select a quality level using one of the algorithms above, and return it to the simulator. The simulator simulates the chunk download and repeats the process.

The overall control flow is shown below.



The client message sent will contain everything you need to implement this lab's algorithms. It contains both dynamic variables that reflect the current stream state (seconds elapsed, previous chunk throughput, bitrates for upcoming chunks…) and static variables that are needed to calculate the algorithm's choices (Quality of Experience coefficients, number of seconds per chunk). These variables are well documented in the student code ClientMessage class.

ALL necessary information is provided in the ClientMessage. Your code **SHOULD NOT** attempt to read the test case files for the purpose of making quality choices.

Simulator Code Structure

The code structure is as follows,

- simulator.py: This is the top-level simulator file. It parses the test case and simulates the viewing session using the choices from your ABR algorithm. **DO NOT MODIFY.**
- tester.py: This script will run simulator.py for all test cases in the tests/ directory. **DO NOT MODIFY.**
- Classes/: This contains supporting code for the simulator. **DO NOT MODIFY.**
- student/student1.py: This is where you are to implement your first ABR algorithm. It contains a predefined class "ClientMessage" containing all the various metrics that might be used by your algorithm. Fill out the student_entrypoint() section of this file.
- student/student2.py: This is where you are to implement your second ABR algorithm.
- student/studentX.py: If you would like to implement more algorithms, you may copy over student1.py or student2.py to make a student3, student4, ... and run them the same way.

Helper Functions and Global Variables

Because the student code is called from one function (student_entrypoint()), you are encouraged to implement any necessary classes, helper functions, and global variables in the studentX.py classes.

Some algorithms, such as RobustMPC, require knowledge of previous chunks in order to make predictions on future chunks. To capture this behavior, **you should save any necessary information in global or module-level variables between student_entrypoint() calls**. If you do not know how to do this, search online for a tutorial on the Python "global" keyword.

Submitting your Code

Submit each algorithmic variant as "*student1.py*", "*student2.py*" and so on. For example, iimplement BBA-2 in "student1.py", a variant of BBA-2 in "student2,py", the RobustMPC algorithm in "student3,py", and a variant in "student4.py". You may submit additional variants if you prefer. Please add a comment to each file indicating which algorithm and variant it corresponds to

Running your Code

The simulator runs your algorithm and outputs statistics on a per-chunk basis as well as for the complete video as a whole. You can run the simulator with the following command:

python simulator.py <path to the test file (.ini)> <Student algorithm to run> ***-v***

The path to the test file should be the path to one of the .ini files in the tests/ directory. The student algorithm to run should be an integer 1 or 2 (or higher if you made more algorithms) to run student1.py or student2.py. '**-v**' is an optional flag that enables verbose output for the simulation. This prints the download times and quality selections for each chunk.

For example, running:
python simulator.py test/hi_avg_hi_var.ini 2 -v

will start the simulator running the test hi_avg_hi_var.ini using the algorithm in student2.py and enable verbose logging.

The tester will run your algorithm and output statistics for all test cases. It is called with

python tester.py <Student algorithm to run>

References and Useful Information

The configuration file is to be extracted and read using the configparser library. See the link below for more information. Using pip, can be installed with the command "pip3 install –user

configparser”. This command is provided in the starter code makefile and can be run with “make configparser”.
https://docs.python.org/3/library/configparser.html

Grading:

The project has open-ended components. Getting a decent grade will require implementing (i) both the RobustMPC and BBA-2 algorithms, and a variant of each; and (ii) reporting results clearly and in a well thought out manner, presenting good quality graphs, and clearly interpreting results, However, the very best grades will be obtained by students that explore particularly new and interesting variants of these algorithms, and show creativity, effort and initiative in the design and implementation of the variants, and in the open-ended components. We may award a bonus to students that go particularly beyond the norm in terms of the open-ended components, and exhibit a high degree of passion and effort in the project. Note that the bar for a bonus will be high and subjective.