15-441/641: Networking and the Internet
Project 3: Video Streaming Service

Lead TAs:
Wuwei Lin <wuweil@andrew.cmu.edu>
Zhenzhen Liu <zhenzhel@andrew.cmu.edu>

1 Introduction

In this project you will explore how video content distribution networks (CDNs) work. In particular, you will implement a video distribution system that uses adaptive bitrate selection (described in the *Video Distribution* lecture) and a load balancer for a server cluster in a datacenter. *The load balancer part of the project is optional*, and will allow you to increase your score on this project by as much as 10%.

The grade distribution and deadlines for the two checkpoints are shown below:

CP	Goal	P3 Grade	Deadline
1	Proxy and Bit rate adaptation.	100%	Dec 9, 2020
2	Load Balancing.	+10% (EC)	Dec 11, 2020

The process of team formation is the same as project 2. If you have any questions about it, please let us know ASAP. For this project, we will be used Gradescope for most of the grading. For some parts of the project, we will be using manual grading, similar to the last checkpoint of Project 2. We will have some hidden tests (like what we do in project 1 and 2), so please make sure to make your code robust.

2 Your Video Distribution System

Figure 1 depicts (at a high level) what the system might look like in the real world. We have a set of clients on the right downloading video from a CDN, which has several CDN instances (datacenters). In this project we will focus on one data center, i.e., you do not have to implement DNS or a similar technique to select what CDN datacenter to use. Clients first download a manifest file will then download individual video chunks from the CDN. Inside the datacenter, the CDN uses a load balancer to distributed the load across

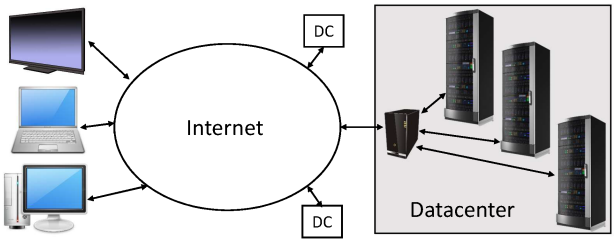

Figure 1: In the real world...

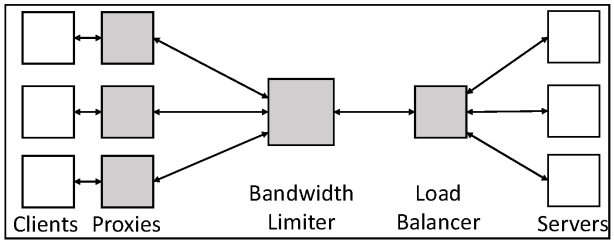

Figure 2: Your system.

Figure 3: System overview.

a set of servers. While playing the video, the video player optimizes the quality of the video by picking the highest bit rate the network can support, based on the throughput observed for earlier chunks.

Implementing an entire video distribution system is clearly a tall order, so let's simplify things. For this project, you will do your development and testing inside a docker container we provide (§5). The starter code also includes a network simulator that allows you to run several clients and a server and you can control the client-server bandwidth. Figure 2 shows simplified video distribution system you will use build in this assignment. The shaded components in Figure 2 will be written by you.

In this project you will implement two critical components of a video distribution system. In the first checkpoint, you will implement a proxy that implements video bit rate adaptation. In the second checkpoint, you will implement a layer 4 load balancer that directs requests for video chunks from clients to a specific server with the goal of balancing the load evenly across the available servers. Note that each checkpoint only uses a subset of the components in Figure 2.

The next two sections describe the two components of the project in more detail. §5 describes the development environment for project 3, while §**??** lists what you need to complete and hand in. Finally, §**??** summarizes grading information for checkpoint 1 and the optional checkpoint 2.

3 Checkpoint 1: Video Bit Rate Adaptation

Checkpoint 1 includes three tasks:

1. Implement a proxy
2. Implement bitrate adaptation
3. Analyze the behavior of your proxy in a writeup

The details of each task are discussed below in this section.

The system you will use for this part of the project consists of the following components in Figure 2:

Browser. You will use an off-the-shelf web browser to play videos served by your CDN (via your proxy). You will do experiments with multiple browsers accessing the Internet.

Proxy. Rather than modify the video player itself, you will implement adaptive bit rate selection in an HTTP proxy. The player requests chunks with standard HTTP GET requests; your proxy will intercept these and modify them to retrieve whichever bitrate your algorithm deems appropriate. To support multiple clients, you will launch multiple instances of your proxy. More detail in §3.1.

Bandwidth Limiter. To model the variable amount of bandwidth this available on the Internet between a set of clients and a Web Server, you will use a bandwidth limiter. The bandwidth limiter allows you to change the available bandwidth so you can evaluate how well your bit rate adaptation algorithm performs in various scenarios.

Web Server. Video content will be served from an off-the-shelf web server (Apache) that comes with the starter code. It uses `127.0.0.1` as the IP address. To simulate a CDN with several content servers, we will use different ports numbers. More detail in §5.

This project has two checkpoints. For the first checkpoint you have to implement a proxy that implements video bit rate adaptation, and in the second checkpoint you will add DNS redirect to pick the "best" server for each client, considering both server load and network distance. This section describes the features of these three components in detail.

3.1 Implementating a Proxy

You are implementing a simple HTTP proxy. It accepts connections from one web browser, modifies video chunk requests as described below, resolves the web server's DNS name (part of the HTTP request), opens a connection with the resulting server IP address, and forwards the modified request to the server. Any data (the video chunks) returned by the server should be forwarded, unmodified, to the browser.

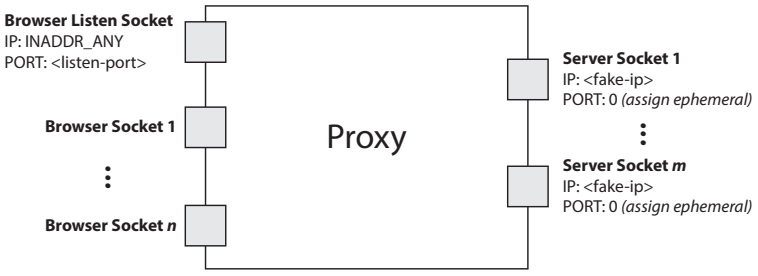

Figure 4: Your proxy should listen for browser connections on `INADDR_ANY` on the port specified on the command line. It should then connect to web servers on sockets that have been bound to the proxy's fake IP address (also specified on the command line).

Your proxy should listen for connections from a browser on any IP address on the port specified as a command line argument (see below). When it connects to a server, it should first bind the socket to the fake IP address specified on the command line (note that this is somewhat atypical: you do not ordinarily `bind()` a client socket before connecting). Figure 4 depicts this.

Your proxy should accept multiple concurrent connections using `select()`, as in project 1. The reason is that the browser may open multiple TCP connections to download objects (chunks in our case) in parallel. Similarly, your proxy should open multiple connections to the server (Figure 4). **It is encourage to re-use `select()` and HTTP parsing code from Project 1.**

3.2 Video Bitrate Adaptation

Many video players monitor how quickly they receive data from the server and use this throughput value to request better or lower quality encodings of the video, aiming to stream the highest quality encoding that the connection can handle. Rather than modifying an existing video client to perform bitrate adaptation, you will implement this functionality in the HTTP proxy you implemented.

Video bit rate adaptation is described in pages 7-8 of the handout of the Video Distribution Lecture. It involves a number of steps: (1) throughput estimation, (2) choosing a bit rate,

3.2.1 Throughput Estimation

The goal of bit rate adaptation is to pick the best bit rate for each video chunk, as network conditions change. In the real world, this means that the video player picks a new rate for each chunk; this function is delegated to the proxy in our case. The proxy determines the bit rate for each chunk based on the past history of the throughputs observed for downloads of earlier chunks from the same server, i.e., the throughput estimation is for the path between your proxy and a specific server IP address. To measure the throughput for each chunk you will use a "black box" approach: you simply measure the throughput for each chunk as described below, while ignoring external information (e.g., how many videos are being streamed, how many chunks are being fetched in parallel, etc.).

Your proxy should estimate the throughput for each video chunk as follows. Note the start time, t_s, of each chunk request (i.e., include `time.h` and save a timestamp using `timeofday()` when your proxy receives a chunk request from the player). Save another timestamp, t_f, when you have finished receiving the chunk from the server. Now, given the size of the chunk, B, you can compute the throughput for the chunk, T, your proxy saw for this chunk:

$$T = \frac{B}{t_f - t_s}$$

To smooth your throughput estimation, your proxy should use an exponentially-weighted moving average (EWMA). Every time you make a new measurement (T_{new}), update your current throughput estimate as follows:

$$T_{current} = \alpha T_{new} + (1 - \alpha) T_{current} \quad (1)$$

The constant $0 \leq \alpha \leq 1$ controls the tradeoff between a smooth throughput estimate (α closer to 0) and one that reacts quickly to changes (α closer to 1). You will control α via a command line argument.

Since the bit rate depends to the path to the server, you need to maintain a separate throughput estimate for each server. When a new server is used, e.g., for a new stream, set $T_{current}$ to the lowest available bitrate for that video and server. Note that however that for CP 1, you can assume that (1) there is only one client per proxy, (2) each client client will only download one video at the time, and (3) all the chunks for a video will be fetched from the same server . That simplifies the data structures you need to maintain in the proxy. However, for CP 2, you may talk to multiple servers, so separate estimates are needed.

3.2.2 Choosing a Bitrate

Once your proxy has calculated the connection's current throughput, it should select the highest offered bitrate the connection can support. For this project, we say a connection can support a bitrate if the average throughput is at least 1.5 times the bitrate. For example, before your proxy should request chunks encoded at 1000 Kbps, its current throughput estimate should be at least 1.5 Mbps.

Your proxy should learn which bitrates are available for a given video by parsing the manifest file (the ".f4m" initially requested at the beginning of the stream). The manifest is encoded in XML; each encoding of the video is described by a `<media>` element, whose `bitrate` attribute you should find.

Your proxy replaces each chunk request with a request for the same chunk at the selected bitrate (in Kbps) by modifying the HTTP request's Request-URI. Video chunk URIs are structured as follows:

`/path/to/video/<bitrate>Seg<num>-Frag<num>`

For example, suppose the player requests a chunk that corresponds to fragment 3 of sequence 2 of the video Big Buck Bunny at 500 Kbps:

`/path/to/video/500Seg2-Frag3`

To switch to a higher bitrate, e.g., 1000 Kbps, the proxy should modify the URI like this:

`/path/to/video/1000Seg2-Frag3`

IMPORTANT: When the video player requests `big_buck_bunny.f4m`, your proxy should instead return `big_buck_bunny_nolist.f4m`. This file does not list the available bitrates, preventing the video player from attempting its own bitrate adaptation. You proxy should, however, fetch `big_buck_bunny.f4m` for itself (i.e., don't return it to the client) so you can parse the list of available encodings as described above.

3.2.3 Implementation Details

Logging: We require that your proxy create a log of its activity in a very particular format. After each request, it should append the following line to the log:

`<time> <duration> <tput> <avg-tput> <bitrate> <server-ip> <chunkname>`

`time` The current time in seconds since the epoch.

`duration` A floating point number representing the number of seconds it took to download this chunk from the server to the proxy.

`tput` The throughput you measured for the current chunk in Kbps.

`avg-tput` Your current EWMA throughput estimate in Kbps.

`bitrate` The bitrate your proxy requested for this chunk in Kbps.

`server-ip` The IP address of the server to which the proxy forwarded this request.

`chunkname` The name of the file your proxy requested from the server (that is, the modified file name in the modified HTTP GET message).

Running the proxy: By running `make` in the root of your submission directory, we should be able to create an executable called `proxy`, which should be invoked as follows, *even if not all arguments are functional at the first checkpoint*:

`./proxy <log> <alpha> <listen-port> <target-port>`

`log` The file path to which you should log the messages described in §3.2.3.

`alpha` A float in the range [0, 1]. Uses this as the coefficient in your EWMA throughput estimate (Equation 1).

`listen-port` The TCP port your proxy should listen on for accepting connections from your browser.

`target-port` The port for the proxy to fetch video chunks from the server.

To play a video through your proxy, point a browser on your VM to the URL `http://localhost:<listen-port>/index.html`. (You can also configure VirtualBox's port forwarding to send traffic from `<listen-port>` on the host machine to `<listen-port>` on your VM; this way you can play the video from your own web browser.)

3.3 Proxy Behavior Analysis

Once your proxy is working, launch two instances of it on the "sharelink" topology we provide. Running the "sharedlink" topology using `./netsim.py sharelink start` will also create two servers (listening on the port number in `sharelink.png`); you should direct one proxy to each port. Now:

1. Start playing the video through each proxy.
2. Run the `onelink.events` file and direct `netsim.py` to generate a log file: `./netsim.py sharelink run -l <netsim-log> -e onelink.events`
3. After 1 minute, stop video playback and kill the proxies.
4. Gather the netsim log file and the log files from your proxy and use them to generate plots for link utilization, fairness, and smoothness. Use our `grapher.py` script to do this: `./grapher.py <netsim-log> <proxy-1-log> <proxy-2-log>`

Repeat these steps for $\alpha = 0.1$, $\alpha = 0.5$, $\alpha = 0.9$ (see §3.2.1). Compile your 9 plots, labelled clearly, into a single PDF named `writeup.pdf`. You should divide your writeup into three sections: (1) Link Utilization, (2) Fairness, (3) Smoothness. Under each section, present the three plots for the the three different alpha features and provide a 1-paragraph explanation of how and why these features are impacted by your choice of α.

3.4 Grading

You will turn in the following files to Gradescope:

- `Makefile` — Running `make` should produce an executable named `proxy`, as described in §3.2.3.
- `src` — A directory named `src` containing your source code. You may organize your code within this directory as you see fit.
- `writeup.pdf` — Your analysis of your proxy as described in §3.3.

The breakdown of grading for Checkpoint 1 is below.

Task	Weight	Subcriteria
Format	5%	*Assigned by human grader:* • Correct turnin – Makefile, compilation, properly tagged repo, TA's don't have to edit or search for files (5%)
Basic Proxy	30%	*Assigned based on tests:* • Receives requests from browser and forwards them to server. • Forwards the chunks it receives to the browser.
Bit Rate Adaptation	40%	*Assigned on tests:* • Implements throughput estimation • Bitrate selection
Writeup	25%	*Assigned by human grader:* • Correctness of graphs. • Discussion of the graphs.

4 Checkpoint 2 (Optional): Layer 4 Load Balancing

We now describe the load balancing part of the project, which is optional (see §**??**).

To spread the load of serving videos among a group of servers, datacenters use load balancers that receive client traffic with incoming requests (e.g., HTTP requests over TCP) and forward the traffic to individual servers. Load balancing can be done in many different ways. For example, the load balancer can terminate TCP sessions and forward requests to servers over a new TCP session based on the nature of the requests (e.g., the URL in an HTTP GET). An alternative is to use a flow-level load balancing (TCP in practice) that only uses information in the packet headers at layer 4 and below. This is a much more light-weight solution, so it is widely used.

In this components of the project, you will build a flow level load-balancer for video. The system you will use consists of the following components in Figure 2:

Browser. The browser is a traffic generator that models a video player by requesting video chunks with a size and frequency that can be controlled via a script.

Load balancer. You will develop a load balancer, using the proxy starter code used in the first part of the project as a starting point.

Web Server. We will use the same Apache server as in Checkpoint 1, and simulate a CDN with multiple servers by using different port numbers. More detail in §5.

4.1 Implementing the Load Balancer

Layer 4 load balancers are normally implemented entirely at the packet level. The load balancer extracts fields from the packet headers (e.g., client IP address and port number) and uses his information to identify individual flows. It then assigns each flow to a server and forwards all packets of the same flow to the same server by looking only at the IP and TCP header. This is somewhat tricky to implement in a short amount of time, so we will implement the load balancer as a TCP proxy, using the proxy code you already used in the first part of the project as a starting point.

The TCP proxy behaves the same way as the proxy you already wrote. It accepts and terminates client TCP connections. For each new client TCP connection, it opens a new TCP session to a server and then forwards all traffic between the two TCP sessions in both directions. Note that your proxy does not have any access to the IP and TCP headers, which is normally the information that is used by a flow-level load

balancer to identify all packets belonging to the same flow. However all packets with the same client IP address and port number will be received by the proxy through the same socket, so the socket corresponds to a specific flow.

There are a few differences with proxy in the first part of the project. First, the original starter code is a HTTP proxy that only forwards requests when it finds an HTTP message in the byte stream. This is not necessary in our our layer 4 load balancer, so you should remove that part from the starter code. Second, instead of doing doing bit rate adaptation, it does load balancing.

4.2 Load Balancing Algorithm

The unit of load used by your load balancer is an entire video requested by a client since all the chunks of a video are sent over the same TCP connection. Since videos differ in number of chunks and the chunk size, the load a video places on a server differs across videos. The load balancer can use various strategies to spread the load. In this project we want you to implement two algorithms: **round-robin** load balancing and **load-aware** load balancing.

The first algorithm does round-robin assignment of flows to servers. It simply iterates through the list of servers as it received new client TCP session requests. Round-robin load balancers are not sensitive to load so for some video load they may results in an uneven load distribution across servers.

The second algorithm is a load-aware load balancer that explicitly considers the current load on the servers when assigning new incoming TCP sessions to servers. In our case, server load is number of bytes served by the server to clients in a certain time interval. Since you are implementing a layer 4 load balancer, the load balancer does not have any information on the nature of the video, such as its length or chunk size. As a result, the load balancer must monitor the load on each server and use this information to guide the assignment of requests to servers. The load balancer keeps track of the number of bytes served by each server. At time T, it estimates the estimates the future server load as bytes served in time window $[T-x, T]$. Note that this assumes that the load a video places on the server is fairly stable. Clearly because of bit rate adaptation, this is really an approximation. However, when a server handles a sufficiently large number of vidio flows, these effects should generally average out.

4.3 Implementation Details

You can reuse your code in checkpoint 2. Your load balancer should use the arguments below:

```
./loadbalancer <listen port> <server number> <server ports> <algorithm version>
```

`listen port` The port that your load balancer will listen to. All the incoming connection would visit this port.

`server number` The total number of servers.

`server ports` The port file specifying port of each server. See §5.2 for detail.

`algorithm version` Which load balancing algorithm to use. 0 for round robin and 1 for load aware load balancing. See the description of these two algorithms in the section below.

As you may noticed, the time window plays a crucial role in your load balancer. It will determine the performance of your load balancer, and it also depends on the fetching pattern inside each connection. We recommend you write your own load event and play with different time window. Generally, your time window should be smaller than average connection lifetime and bigger than fetching interval. In our tests, we use intervals between video requess in the 0.05s to 0.1s range, so please make your time window smaller, e.g., 0.2 - 0.4s.

4.4 Testing

We will test your load balancer with a variety of video connections – *some of which request larger or smaller chunks*. We will use Jain's Fairness Index across different servers to determine whether the load balancer is

evenly distributing the load. For n servers, each serving bandwidth $x_1, x_2 ... x_n$ on average, JFI is calculated as follows:

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2} = \frac{\overline{\mathbf{x}}^2}{\overline{\mathbf{x^2}}} = \frac{1}{1 + \widehat{c_\mathrm{v}}^2}$$

You can use the tools in `/autograder/netsim` to test your load balancer. It works as follows:

1. Run `./netsim.py servers start -s [server port file]`. This starts the network simulation.
2. Run `./monitor.py -s [server port file]`. This starts monitoring the load on each server.
3. Start your load balancer.
4. Run `./loadgen.py -e [load event file]` to generate load.
5. When the loads are done, use `CTRL+C` to stop `monitor.py`. It will print out the load on each server, which looks like {`8080:'0.124575 MB', 8081:'3.453783 MB'`}.
6. Calculate the JFI using the equation above.

4.5 Comparing your Two Algorithms

Once you have completed your two algorithms, we want you to compare their performance. We will provide you two workloads: one workload where all video connections request chunks that are the same size, and one workload where all video connections request chunks that are different sizes. Set up your testbed with 5 different video servers and use your load balancer to divide the client connections between the servers.

You will then need to measure the load on each server for 2, 10, 20, and 100 concurrent connections. Using the load for each server, calculate the JFI for your load balancer. Repeat this experiment 5 times and then take the median JFI you computed.

Generate two graphs: one for your load balancer running in 'round robin' mode and one using 'adaptive' load balancing mode. On the x-axis, you should plot the number of concurrent connections, and on the y-axis you should plot the median JFI from your experiments.

Place these graphs in a writeup called `evaluation.pdf` and answer the following questions:

1. Is one algorithm generally more 'fair' than the other? Which? Why?
2. What is the difference between JFI for Round Robin versus Adaptive Load Balancing when there are only 5 flows? What is the difference between JFI for RR vs ALB when there are 100 flows? Does JFI improve, get worse, or stay the same as the number of flows increases and why?

4.6 Grading

Turn in the following files to Gradescope:

- `Makefile` — Running `make` should produce an executable named `loadbalancer`, as described in §4.3.
- `src` — A directory named `src` containing your source code. You may organize your code within this directory as you see fit.
- `Write up` — A write up named `evaluation.pdf` showing the results of the experiments you ran as described above

Grading will be calculated as follows:

Task	Weight	Subcriteria
Format	20%	*Assigned by human grader:* • Correct turnin – Makefile, compilation, properly tagged repo, TA's don't have to edit or search for files (5%) • Code style (10%) • Code commenting (5%)
Load Balancer	40%	*Assigned based on tests:* • Basic Correctness: Load balancer steers flows to servers. • Round robin load balancing • Adaptive load balancing
Writeup	40%	*Assigned by human grader:* • Correctness of graphs • Discussion of graphs and answers to questions

5 Development Environment

For this project, we are providing a docker image pre-configured with the software you will need and you need to create a docker container of this image. We strongly recommend that you do all development and testing in this container. Your code must compile and run correctly on this container as we will be using the image with same environment for grading. This section describes the docker container and the starter code it contains.

5.1 Docker Container

Containers are a standardized unit of software that allows developers to isolate their app from its environment. And Docker provides a way to manage, build and share containers. For more info on this, you could refer to `https://www.docker.com/` to learn more about this.

The docker image we provide was created using Docker, so you need to install Docker in your machine to use it. The detailed process to set up the docker container and do port mapping are provided in the README of the GitHub repository. Please refer to it to set up your container. Docker is a free download software for Windows, OSX, and Linux on `https://www.docker.com/get-started`.

5.2 Starter Files

The starter code includes two folders: `docker_setup` and `starter_proxy`. `starter_proxy` provides some starter code to build a proxy, and a script to generate the graphs for Checkpoint 1; the files are listed below:

`grapher.py` : Generate plots for CP1 writeup. Usage: `python grapher.py <netsim log> <proxy1 log> <proxy2 log>`

`inc/` : Contains the header files for the .c files in `src`

`src/proxy.c` : Contains the major network and multiplexing related code

`src/httpparser.c` : Contains some http parsing and header value extraction code

`src/customsocket.c` : Contains some helper functions for creating sockets

`docker_setup` contains the files to set up the docker container; see `docker_setup/README.md` for details. After you build the container, you will find the following files in `/autograder` inside the container.

`netsim`

`/autograder/netsim/netsim.py` This script controls the simulated network; see §5.3.

`/autograder/netsim/util.py` This script contains code used by `netsim.py` to check process status and output; you do not need to interact with it directly.

`/autograder/netsim/topology` This directory contains three topology you could used in netsim: onelink, twolink and sharelink. In each directory, you could find a .png file describes the topology and a event file we used to change the bandwidth of link inside; see §5.3.

`/autograder/netsim/loads` This directory contains the load event file samples used by `loadgen.py` to generate the load in checkpoint 2. We have provided 3 events files: `simple.load`, `hybrid.load`, and `test.load`. `simple.load` defines concurrent connections with the same load. `hybrid.load` and `test.load` define connections that start at different times with different loads. You can also define your own load event files and play around with it. See §5.5

`/autograder/netsim/servers` This directory contains the server port file you will use for `monitor.py`, `netsim.py` and your load balancer. Each line in it represent a port where a server listen on it. We have provided 3 files, `2servers`, `5servers`, and `10servers`. See §5.6, §5.3 and §4.3.

`/autograder/netsim/apache_setup.py` This file contains code used by `netsim.py` to start and stop Apache instances on the port number described in `/autograder/netsim/*.png` or `/autograder/netsim/servers/*server`; you do not need to interact with it directly.

`/autograder/netsim/loadgen.py` This file contains code that could generate a load event which is incoming TCP connections fetching video chunks in different bit rate; See §5.5

`/autograder/netsim/monitor.py` This file contains code that could monitor the load on each server specified in server port file; See §5.6

`/autograder/netsim/*.events` These events files are used by `netsim.py` to dynamiclly change the bandwidth of the link in your topology; see §5.3.

5.3 Network Simulation

5.3.1 Bandwidth Limited Server

To test your system, you will run everything (proxies, servers, load balancer) on a simulated network in the container. You control the simulated network with the `netsim.py` script. You need to provide the script with a mode refering to a specific topology, which contains server instances and bandwidth limited link. We provide three topology for you to use and also a png image to describe it in a visible way. You should open the image and understand each topology. To start the network from the `netsim` directory:

```
./netsim.py <topology> start
```

You should only use `onelink, twolink, sharelink` in topology argument to start the corresponding topology. Starting the network creates several apache server instances and build links in front of it to limit the bandwidth and the way to access the server through links is to access the port it opens. Your proxy should connect to the port specified in `/autograder/netsim/*.png`. For example, you could use

```
./netsim.py onelink start
```

to start a onelink topology which has only one server and listens on port number 15641.

Note that each link's bandwidth limitation in new started topology is set to `1000kbit/s`. You could change it by using events described below or modify it in `./netsim.py`

To stop it once started (shut servers and links), run:

```
./netsim.py <topology> stop
```

To facilitate testing your adaptive bitrate selection, the simulator can vary the bandwidth of any link designated as a bottleneck in your topology's `<topology>.png` image. (Bottleneck links must be declared in our topology.) To do so, add link changes to the `.events` file you pass to `netsim.py`. Events can run automatically according to timings specified in the file or they can wait to run until triggered by the user (see `/autograder/netsim/onelink.events` for an example). When your `.events` file is ready, tell `netsim.py` to run it:

```
./netsim.py <topology> run -e <events>
```

Note that you must start the network before running any events. You can issue the `run` commands as many times as you want without restarting the network. You may modify the `.events` file between runs without restarting the network. Also note that the links stay as the last event configured them even when `netsim.py` finishes running.

5.3.2 Bandwidth Unlimited Server

In checkpoint 2 and 3, we are not limiting the bandwidth of each server any more, so you should use a new topology called `servers` and it will need you to specify the ports used for these server by using `-s` arguments. Such port file are provided in `/autograder/netsim/servers` directory. You could use command below to start 2 servers without bandwidth limitation.

```
./netsim.py servers start -s /autograder/netsim/servers/2servers
```

Note that because we're not limiting the bandwidth in this mode, you can't run event when using servers topology.

5.4 Apache

You will use the Apache web server to server the video files. `netsim.py` automatically starts instances of Apache for you based on the topology. Each instance listens on a port according to the topology it uses and is configured to serve files from `/var/www`; we have put sample video chunks here for you. Note that in checkpoint 1, you should connect to apache server through the port number provided in `.png` file rather than connect to server using 8080 or 8081 directly. But in checkpoint 2, you could access the port in the port file located in `/autograder/netsim/servers` directory directly.

5.5 Load Generator

In checkpoint 2, we provided a load generator for you to simulate the load in the real world. The load behaves like many incoming TCP connection to a specified port. The TCP connections could arrive at same time or arrive in some pre-defined order. In each connection, HTTP requests will be sent to fetch video chunks, but the size of requested chunk may alter. Between each HTTP request, there could be an idle interval and each TCP connection will end after it fetches a specific number of chunks. It seems a little bit of complex, but all these behavior are defined in six arguments and these six arguments becomes a load event. Each line of a load events file is a event which contains these six arguments. We will explain it in next section.

5.5.1 Load Event

As you may noticed, in the load events file located at `/autograder/netsim/loads` are made of lines with six number. These numbers are formatted as below:

```
<start_time> <port> <bitrate> <interval> <chunk_num> <threads>
```

`start_time` The start time of this event (in seconds) after this script starts to be executed. Float number is supported.

`port` The port number this event will access. You should use the listen port of your load balancer in this part.

`bitrate` The bitrate here refers to the bitrate of the chunk in checkpoint 1. So it will determine the size of chunk it requests. So, there is only 3 optional here: 100, 500 and 1000. Please don't give number outside of these three options.

`interval` The interval (in seconds) between the last fetched chunk and new request. If set to 0, the TCP connection will send request right after last fetched chunk. Float number is supported.

`chunk_num` The number of chunk need to be fetched in this TCP connection, after the TCP connection fetched this number of chunks, it will terminate.

`threads` The number of concurrent TCP connection in this event.

You could find example in `/autograder/netsim/loads` and could use these example to test your load balancer. But we recommend you to write your own load events file and play with it to see how your load balancer work.

5.5.2 Usage

Each time you run your load generator, you need to specify a load events file using `-e` argument, so you use command like below to run a load:

```
./loadgen.py -e <load events file>
```

5.6 Monitor

To measure the load generated by load generator, we also provide a load monitor to calculate the load for each server. It takes the same server port file you used in netsim servers topology located at `/autograder/netsim/servers` directory and will monitor the load on these ports. You could use CTRL + C to stop the load monitor and it will print out the load on each server.

You need to specify the server port file using `-s` argument, use command like below to start your monitor:

```
./monitor.py -s <server port file>
```

6 Hand In

You will submit your code using Gradescope (`https://www.gradescope.com/`). The process of team formation and submission is the same as project 2. If you have any questions about it, please let us know ASAP. For this project, we will use the autograder on Gradescope to grade your submission.