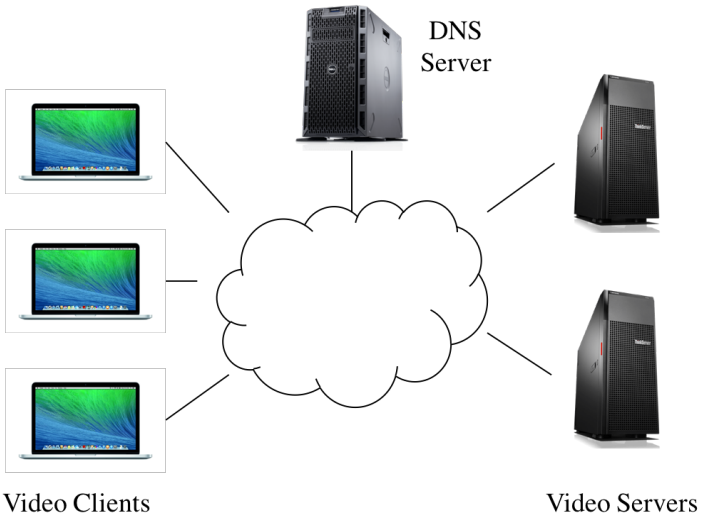
Assignment 2: Video Streaming via CDN

Due: Feb 27th, 2022 at 11:59 PM

Table of contents

Overview

Video traffic dominates the Internet. In this project, you will explore how video content distribution networks (CDNs) work. In particular, you will implement adaptive bitrate selection and an HTTP proxy server to stream video at high bit rates from the closest server to a given client.



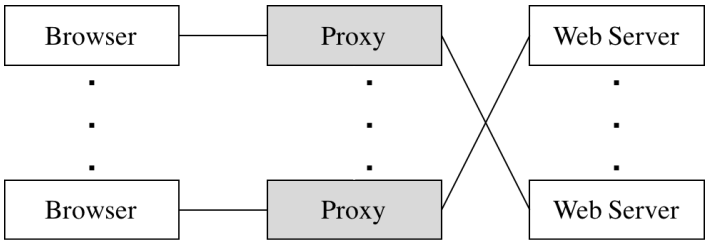
Video CDNs in the Real World

The figure above depicts a high level view of what this system looks like in the real world. Clients trying to stream a video first issue a DNS query to resolve the service's domain name to an IP address for one of the CDN's content servers. The CDN's authoritative DNS server selects the “best” content server for each particular client. (To simplify our assignment, we do not need to implement the DNS server except for the bonus part.)

Once the client has the IP address for one of the content servers, it begins requesting chunks of the video the user requested. The video is encoded at multiple bitrates; as the client player receives video data, it calculates the throughput of the transfer and requests the highest bitrate the connection can support.

Video CDN in this Assignment

Implementing an entire CDN is difficult; instead, you'll focus on a simplified version. First, your entire system will run on one host and rely on mininet to run several processes with arbitrary IP addresses on one machine. Mininet will also allow you to assign arbitrary link characteristics (bandwidth and latency) to each pair of “end hosts” (processes).



You'll write the gray-shaded components (the proxy) in the figure above.

Browser. You'll use an off-the-shelf web browser (Firefox) to play videos served by your CDN (via your proxy).

Proxy. Rather than modify the video player itself, you will implement adaptive bitrate 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.

Web Server. Video content will be served from an off-the-shelf web server (Apache). As with the proxy, you will run multiple instances of Apache on different IP addresses to simulate a CDN with several content servers.

To summarize, in this assignment you will implement the "Bitrate Adaptation in HTTP Proxy" which will determine bitrate for the played video.

Learning Outcomes

After completing this programming assignment, students should be able to:

- Explain how HTTP proxies and video CDNs work
- Create topologies and change network characteristics in Mininet to test networked systems

Clarifications

- For the proxy, you will need to parse some HTTP traffic. To make your life easier for this project, you don't need to be concerned about parsing all the information in these HTTP messages. There are only two things that you need to care about searching for: `\r\n\r\n` and `Content-Length`. The former is used to denote the end of an HTTP header, and the latter is used to signify the size of the HTTP body in bytes.
- The proxy should be able to support multiple browsers playing videos simultaneously. This means you should test with multiple browsers all connecting to the same proxy. In addition you should also test with multiple proxies each serve some number of browser(s), in order to make sure that each proxy instance does not interfere with others.
- While testing the proxy, you may notice that one browser may sometimes open multiple connections to your proxy server. Your proxy should still continue to function as expected in this case. In order to account for these multiple connections, you may use the browser IP address to uniquely identify each

connection (this implies that while testing your proxy server, each browser will have a unique IP address. For example, only one browser will have an IP address of 10.0.0.2).
- Throughput should be measured on each fragment. For example, throughput should be calculated separately for both Seg1-Frag1 and Seg1-Frag2.
- You are recommended to test your code with Mininet. But testing with Mininet is not necessary in the final grading process.

Environment Setup

We will provide a VM that has all the components you need to get started on the assignment. While we tried to make the base VM work for all the projects, unfortunately this didn't come to fruition. Starting fresh also ensures a working environment free from accidental changes that may have been made in the first project.

We encourage you to use VMware instead of Virtual Box for this and following projects, which is more compatible with different OS and is free with personal license. For Windows and Linux users, we recommend VMware Workstation Player. For Mac users, we recommend VMware Fusion Player.

You may install tools that suit your workflow. **But DO NOT update the software in the VM.** You can find a guide on how to troubleshoot the VM here.

This VM includes mininet, Apache, and all the files we will be streaming in this project. Both the username and password for this VM are `csci4430`. To start the Apache server, simply run the python script we provide by doing the following:

```
python start_server.py <host_number>
```

Here `<host_number>` is a required command line argument that specifies what host you are running on Mininet. This is important as if you're running on h1 in Mininet (which is given the IP address 10.0.0.1), passing in `1` into the `<host_number>` argument will help ensure that the Apache server instance will be bound to the 10.0.0.1 IP address. The `<host_number>` argument must be between 1 and 8.

The Apache servers would not automatically stop after mininet is closed. You MUST manually stop the server. To stop the Apache server, run:

```
sudo killall httpd
```

Like any HTTP web server (not HTTPS) these instances of Apache will be reachable on TCP port `80`. For simplicity, all of our web traffic for this assignment will be unencrypted and be done over HTTP.

For this project, we will be using an off the shelf browser (in this case, Firefox). To launch Firefox for this project, run the following command:

```
python launch_firefox.py <profile_num>
```

Here `<profile_num>` is a required command line argument that specifies the instance of Firefox you are launching. We support launching profiles 1-8, however, should you feel the need to test more thoroughly, you can launch it with a different number and simply create a new profile as needed. To ensure a separate connection for each instance of Firefox, we recommend that you launch Firefox with a different profile number (otherwise you might notice that different Firefox instances will sometimes share a connection with your proxy server).

Also make sure you don't modify the firefox profiles we set up as well as the configuration files inside the current firefox build directory.

You are highly recommended to test your code in Mininet. To be clear, you launch the web server, the firefox browser, and the proxy server all inside Mininet.

You may create your own Mininet topology script for testing the package as a whole. A simple Starfish topology (all hosts connected to one switch in the middle) should suffice for testing.

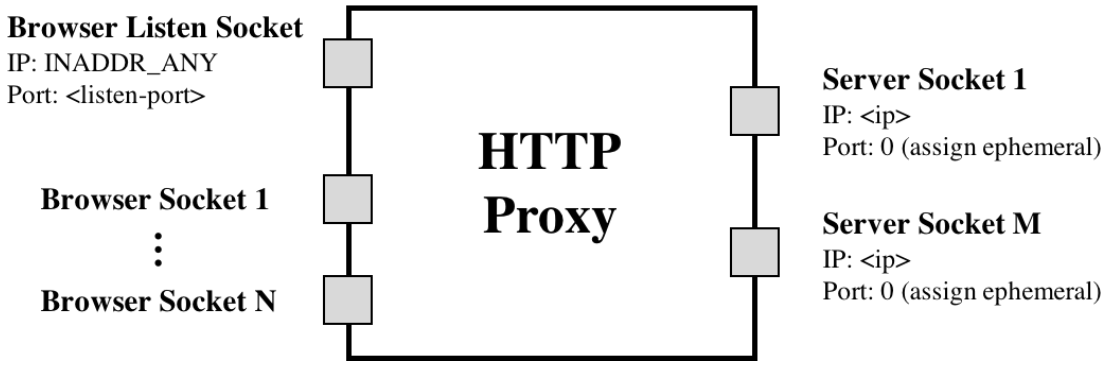
> **NOTE:** For this project, we are disabling caching in the browser. If you do choose to create a new profile, please double check if caching is disabled by going to the `about:config` page and setting both `browser.cache.disk.enable` and `browser.cache.memory.enable` to `false`.

Bitrate Adaptation in HTTP Proxy

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. Instead of modifying an existing video client to perform bitrate adaptation, you will implement this functionality in an HTTP proxy through which your browser will direct requests.

You are to implement a simple HTTP proxy, `miProxy`. It accepts connections from web browsers, modifies video chunk requests as described below, opens a connection with the resulting 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.

`miProxy` should listen for browser connections on `INADDR_ANY` on the port specified on the command line. It should then connect to a web server specified on the command line.



`(assign ephemeral)` is referring to the fact that the kernel will pick the proxy's TCP port when it connects to the web server's port `80`. Nothing more than the proxy calling `connect()` is happening here.

`miProxy` should accept multiple concurrent connections from clients (Firefox web browser) using `select()` and be able to handle the required HTTP 1.1 requests for this assignment (e.g., HTTP `GET`).

The picture above shows `miProxy` connected to multiple web servers. However, in our assignment, each proxy will be connected only one server which is specified in the running command.

We will cover the basic usage of `select()` in the tutorial.

> *Note: A good resource for socket programming is [Beej's Guide to Network Programming Using Internet Sockets](#).*

Throughput Calculation

Your proxy measure the the throughput between the server and ifself to determine the bitrate. Your proxy should estimate each stream's throughput once per chunk. Note the start time of each chunk when your proxy started receiving the chunk from the server and save another timestamp when you have finished receiving the chunk from the server. Given the size of the chunk, you can now compute the throughput by dividing chunk size by time window.

Each video is a sequence of chunks. To smooth your throughput estimation, you should use an exponentially-weighted moving average (EWMA). Every time you make a new measurement (as outlined above), update your current throughput estimate as follows:

```
T_cur = alpha * T_new + (1 - alpha) * T_cur
```

The constant `0 ≤ alpha ≤ 1` controls the tradeoff between a smooth throughput estimate (`alpha` closer to 0) and one that reacts quickly to changes (`alpha` closer to 1). You will control `alpha` via a command line argument. When a new stream starts, set `T_cur` to the lowest available bitrate for that video.

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 fragment 3 of chunk 2 of the video `big_buck_bunny.f4m` 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`, you 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. Your proxy should keep this list of available bitrates in a global container (not on a connection by connection basis).

Running `miProxy`

You will run the miProxy with the following command.

`./miProxy <listen-port> <www-ip> <alpha> <log>`

- `listen-port` The TCP port your proxy should listen on for accepting connections from your browser.
- `www-ip` Argument specifying the IP address of the web server from which the proxy should request video chunks. Again, this web server is reachable at port TCP port `80`.
- `alpha` A float in the range [0, 1]. Uses this as the coefficient in your EWMA throughput estimate.
- `log` The file path to which you should log the messages as described below.

> *Note: for simplicity, arguments will appear exactly as shown above (for both modes) during testing and grading. Error handling with the arguments is not explicitly tested but is highly recommended. At least printing the correct usage if something went wrong is worthwhile.*

miProxy Logging

`miProxy` must create a log of its activity in a very particular format. If the log file already exists, `miProxy` overwrites the log. *After each chunk-file response from the web server*, it should append the following line to the log:

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

- `broswer-ip` IP address of the browser issuing the request to the proxy.
- `chunkname` The name of the file your proxy requested from the web server (that is, the modified file name in the modified HTTP GET message).
- `server-ip` The IP address of the server to which the proxy forwarded this request.
- `duration` A floating point number representing the number of seconds it took to download this chunk from the web 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.

Testing

To play a video through your proxy, you launch an instance of the Apache server, launch Firefox (see above), and point the browser on your VM to the URL `http://<proxy_ip_addr>:<listen-port>/index.html`.

Bonus Task: DNS Load Balancing

NOTE: The bonus task will be updated soon. This part is optional but you will receive some bonus marks after finishing this part.

Submission Instructions

Your assigned repository must contain:

- The source code for `miProxy`: all source files and a Makefile for `miProxy` should be in the path `miProxy`.

Example final structure of repository:

```
$ tree ./p2-<your-SID>/
./p2-<your-SID>/
├── miProxy
|   ├── Makefile <- supports "make clean" and "make"
|   ├── ** source c or cpp files **
|   └── miProxy  <- Binary executable present after running "make"
└── starter_files
    ├── launch_firefox.py
    └── start_server.py
```

Autograder

The autograder will be released roughly halfway through the assignment. You are encouraged to design tests by yourselves to fully test your proxy server. You should *NEVER* rely on the autograder to debug your code. Clarifications on the autograder will be added in this section:

Our autograder runs the following versions of gcc/g++, please make sure your code is compatible.

```
$ gcc --version
gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ g++ --version
g++ (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Acknowledgements

This programming assignment is based on Peter Steenkiste's Project 3 from CMU CS 15-441: Computer Networks.