Video Streaming Using AIMD Algorithm Over UDP

Temitope Akintunde

Rebecca Battat

I4722 High Performing Networks

City College of New York

Professor Kaliappa Ravindran

Project Overview

In this project, we send a video from client to server using a UDP connection. We use UDP instead of TCP to be able to control the send rate and detect packet loss on our own. To send the best quality video given congestion on the network we use the AIMD algorithm. AIMD stands for Additive Increase Multiplicative Decrease. If there is congestion on the network we decrease the flow by multiplying by a constant - in this case, .80. If there is no congestion on the network, we increase the flow by adding a constant - in this case, 50. That way when congestion exists, we quickly decrease until we get to a stable rate and then we can add until we find the perfect rate given the network traffic.

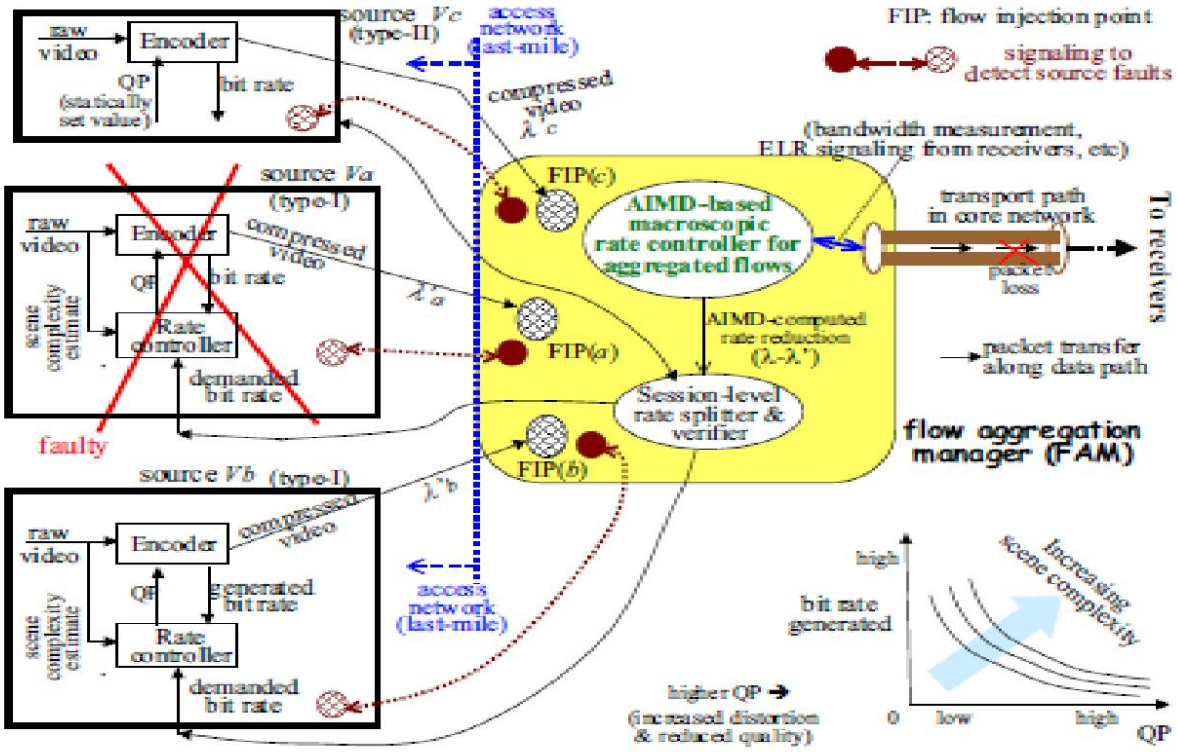
This flow rate does not remain constant due to the bursts of the video and due to changes in network congestion. When the video contains more action, there is more information to be sent. Therefore, more information is sent across the network. Both of these factors contribute to the unpredictability of the network, making the AIMD algorithm shine.

This project contains both a client and a server. The client chooses a video from the server and a desired bit rate. The server encodes the video and sends it using ffmpeg.

Implementation

For this project we used C# on Visual basic to create the GUIs for both the client and the server. We also used ffmpeg to encode the video. To send the video we used a UDP connection. UDP connection cannot send large files so we changed the file to a byte array. Each time we sent a variable rate, called currentRate, which was determined by the aimd algorithm. This means we sent currentRate number of bytes during each iteration. The client sent an acknowledgement for each packet it received. If the server did not get an acknowledgement, it assumed the packet was lost. It then proceeded to resend the set of bytes. As the client received packets it wrote the data to a file. After the entire file was sent, it played the video on the client side.

AIMD

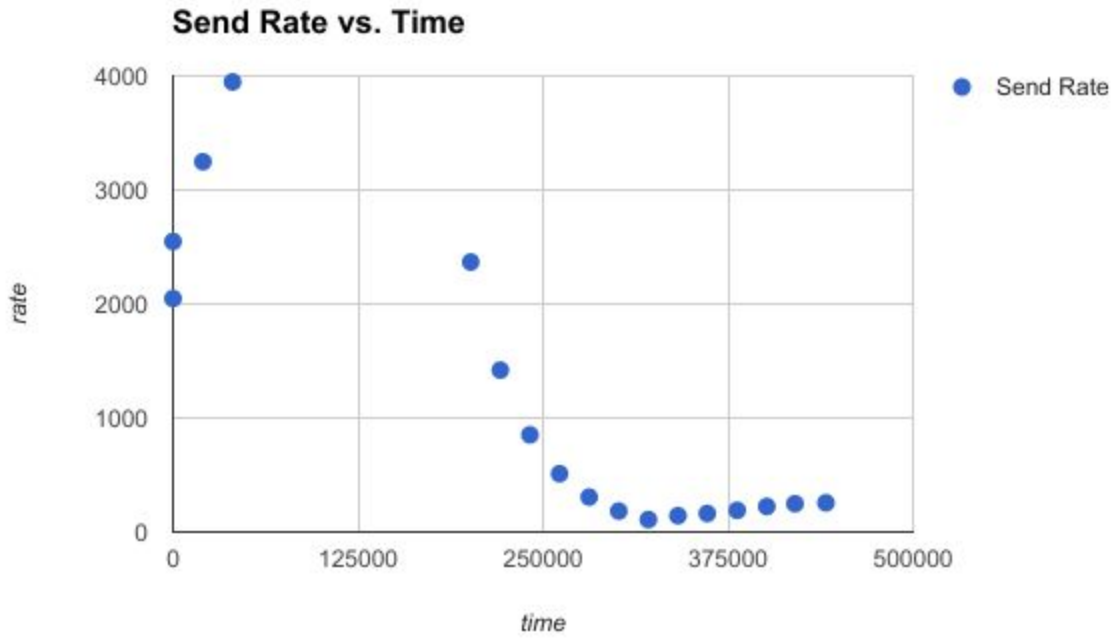


The figure above shows the AIMD algorithm in action. AIMD is a control plane algorithm. The AIMD algorithm helps to control congestion. Congestion may be caused by high levels of traffic and by the unpredictability and burstiness of flows. It is difficult to estimate the exact flow each person should get because it hard to measure the bandwidth capabilities of the path. Also videos are not steady since during moments of increased action in the video, the viewer requires more bandwidth. But if the scene is relatively motionless, less bandwidth is needed. This is called burstiness. The AIMD algorithm seeks to control this in a heuristic way. If our loss rate is low,

we keep adding to the rate. However, once the loss rate reaches a certain threshold, the AIMD algorithm tells each flow to decrease by a certain percentage. We decrease multiplicatively so we back away from the congestion quickly. But we creep up additively, to test the limits of how much we can send.

GRAPHS

The following graphs depict the AIMD algorithm at work.



FFMpeg

FFmpeg is a multi platform solution software that records, converts and edits both audio and video depending on the user's request. At its core, it is a command line tool that converts one video file format to another type (could be .wmv, .3pg , .avi , .mp4, etc).

In this project we downloaded FFmpeg and uncompressed the files to a folder. After ensuring the .exe file was placed in the bin folder of our program, the ffmpeg video format encoded and converted the video files and added to our line of code on the server side and the client side.

Components of FFMpeg are ffserver, ffprobe, ffplay, ffmpeg, and ffmpeg.exe.

Source:https://ffmpeg.org/

Compilation

The link below provides a detailed description for the installation of Ffmpeg a Windows OS System. We also tried the installation of FFmpeg in Linux on a virtual machine while trying to simulate a server and client.

https://trac.ffmpeg.org/wiki/CompilationGuide/MinGW

Input/Output file format

The FFmpeg can be used to convert video formats. There no restrictions on the type of video file formats that can be used for H.264 encoding as input and output files.

H.264 Encoding using FFMpeg

The below link provides a detailed description of producing high quality H.264 encoded video using the software.

https://trac.ffmpeg.org/wiki/Encode/H.264

Command Line Parameters

Command line for encoding standard webvideo

ffmpeg [input options] -i [input filename] -codec:v [video options] -codec:a [audio options] [output file options] [output filename]

example

ffmpeg -i m1.VOB -codec:v libx264 -profile:v high -preset slow -b:v 400k -maxrate 500k -bufsize 1000k -vf scale=720:576 -threads 0 -r 25 out3.mp4 .

You can also do a two pass encoding and it gives the advantage of quality increase and more accurate file size for given bitrate.. The command line arguments are position sensitive.

meaning of the parameters are:-

-i [input file] this specifies the name of input file – here m1.VOB

-codec:v libx264 - to encode video to H.264 using libx264 library

-profile:v high sets H.264 profile to “High” . Other valid options are baseline, main

-preset slow sets encoding preset for x264 – slower presets give more quality at same bitrate, but need more time to encode. “slow” is a good balance between encoding time and quality. Other valid options are:ultrafast, superfast, veryfast, faster, fast, medium, slow, slower,*etc.*

-b:v sets video bitrate in bits/s

-maxrate and -bufsize forces libx264 to build video in a way, that it could be streamed over 500kbit/s line considering device buffer of 1000kbits. Very useful for web - setting this to bitrate and 2x bitrate gives good result

-vf scale “scale” filter, which resizes video to desired resolution. “720:480” would resize video to 720x480, “-1” means “resize so the aspect ratio is same.” Usually you set only height of the video, so for 380p you set “scale=-1:380”, for 720p “scale=-1:720” etc.

-threads 0 tells libx264 to choose optimal number of threads to encode, which will make sure all your processor cores in the computer are used.

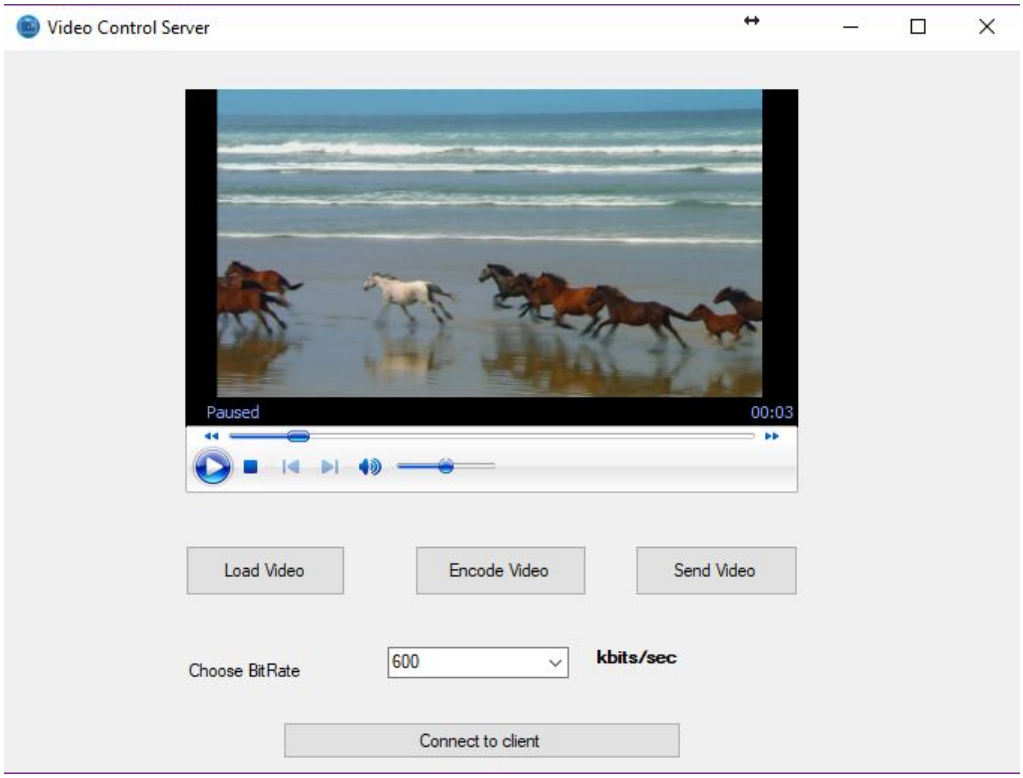
-codec:a libfdk_aac tells FFmpeg to encode audio to AAC using libfdk-aac library

-b:a sets audio bitrate in bits/s

Sample Output :- sample ffmpeg output is given in the inserted file below

Output Screens

Server output



Client Output