VidMid: Middleware for Enhanced Video Streaming and Download Performance

Mangla, Abhishek
amangla@uci.edu

Munshi, Faramarz
fmunshi@uci.edu

Guo, Yuzhou
yuzhoug1@uci.edu

June 2019

1 Introduction

In this project, we intend to solve a key issue that plagues Millenials and Generation Z in particular: video playback in the presence of overwhelming network congestion. A prime example of this issue in daily life is the popular TV series, Game of Thrones, which has recently begun to air its final season with a proportion of users experiencing minor to major delays in streaming that video for one reason or another. A large portion of the current research is being done in improving video transmission on cellular networks via device-to-device (D2D) by using devices to cache video files (Do et al., 2012). This can increase the capacity to stream and reduce the end-to-end delay especially in unreliable networks. The general framework of enabling cellular D2D communication is the interplaying frequency domains and establishment of appropriate radii for communication bubbles to optimize the chances that the cell a user is in contains users who may have the content a user wishes to access(Saito et al., 1999). We leverage this concept to enhance the experience for a user on a reliable network.

Our project's core problem is video streaming in time-varying network environments. In media streaming systems, a client consumes the content of a media file while the file is being downloaded, commonly referred to as "play-while-downloading" mode. At any point of time (or, in industry, frequently referred to as On-Demand), a client machine can request an audio or video file from a server. In most of the existing stored audio/video applications, after a delay of a few seconds the client begins to playback the audio file while it continues to receive the file from the server. The feature of playing back audio or video while the file is being received is called streaming (Huang et al., 2009). Streaming media enables real-time and continuous delivery of video and audio data in a fashion of "flow", i.e., once the sender begins to transmit, the receiver can start playback almost simultaneously as it is receiving media data from the sender, as opposed to waiting for the entire media file to be ready in the local storage. This imposes stringent demand in the timing of packet delivery (Liu et al., 2008).

With the explosive growth of video applications over the Internet, many approaches have been proposed to stream video effectively over packet switched, best-effort networks (Biersack et al., 2004). A number of these use techniques from source and channel coding, implementing transport protocols, or modifying system architectures in order to deal with delay, loss, and the time-varying nature of the Internet. A number of these schemes assume a single fixed route between the receiver and the sender throughout the session. If the network is congested along that route, video streaming suffers from high loss rate and jitter. Even if there is no congestion, as the round-trip time between the sender and the receiver increases, the TCP throughput may reduce to unacceptably low levels for streaming applications.(Bhattacharyya et al., 2019) Based on the aforementioned issues, it is

conceivable to make content available at multiple sources so that the receiver can choose the "best" sender based on bandwidth, loss, and delay (Huang et al., 2009).

In addition to the previous work mentioned above, there are also many other algorithms that help with understanding the flow of information from the host to the receiver using a buffer-based approach to understand which channel is ideal. The study from Stanford using the popular video streaming company's (Netflix) data, suggests that "existing ABR algorithms face a significant challenge in estimating future capacity: as capacity can vary widely over time, a phenomenon commonly observed in commercial services."(Huang et al., 2017) They suggest an alternative approach, that capacity estimation is not required, and instead to begin the work with a buffer and then understand when capacity estimation would become a necessary feature. They produce a variety of experiments to test the viability of a similar approach using millions of real users from Netflix. They "start with a simple design which directly chooses the video rate based on the current buffer occupancy" (Huang 2017).[6] Their investigation shows that in a steady state, capacity estimation is a wholly unnecessary activity, but becomes important only really during the startup phase when the "buffer itself is growing from empty."(Huang et al., 2009)

The chief goals of our experiment, although linked, are three-fold: to decrease network congestion near server hubs, as a result, bringing more relevant traffic closer to the edge, and subsequently, to the clients with lower latency and more reliably.

2 Related Work

In this section, we explore different techniques to realize the concepts of multi-helpers to improve streaming experience. Nguyen et. al. proposed a distributed streaming protocol to get video segments from multi mirror sites based on TCP-friendly protocols (Pq Nguyen, 2002). They proposed a receiver-driven protocol on which the receiver of the video coordinates transmission from multiple sender based on the information it receives:

> "Each sender estimates and sends its round trip time to the receiver. The receiver uses the estimated round trip times and its estimates of sender's loss rates to calculate the optimal sending rate for each sender. When the receiver decides to change any of the sender's sending rates, it sends an identical control packet to each sender. The control packet contains the synchronization sequence number and the optimal sending rates as calculated by the receiver for all senders. Using the specified sending rates and synchronization sequence number, each sender runs a distributed packet partition algorithm to determine the next packet to be sent." (Pq Nguyen, 2002)

There are two fundamental components to this protocol, one of which is deciding each sender's sending rate upon starting delivery of the segment to the receiver. This is important because they want to ensure that network jitter and segment loss rates are reduced. The second component is the partitioning algorithm which decides which sender to send a particular video segment at any specific point of time. This algorithm is distributed, which means that it's implemented at the edge, by each sender.

Huang et. al. proposed a scaled down version of the mega-giant media company, Youtube(Do., 2007), coined YuTube (Huang and Yu, 2008), a scalable distributed video-streaming system for fast and fault-tolerant streaming. In YuTube, video files are partitioned into various segments, which are distributed across their streaming cluster. YuTube consists of both a client player and a streaming server, where a master is elected to manage the cluster and maintain the features of the system. The master should be the least-loaded node in the cluster in order to maintain

load-balancing (Huang and Yu, 2008). The notion of a "Mesh" is utilized for YuTube to cache the mapping of each node to their respective states, including load, free disk space and a list of video segments hosted, and one can consider Mesh as a log of current global state of the cluster. Every node in the cluster keeps a copy of the mesh so that the overhead for transmitting these data can be saved. Additionally, features such as video segment replication, data consistency and manageability are introduced in the paper, making it relatively more scalable, as well as manageable (Huang and Yu, 2008). (Huang et al., 2017)

3 System Architecture

We implemented the basic framework of the system to test the basic functionality and understand whether the premises of our system architecture were sound. We introduce the concept of a cluster, consisting of a client and several neighbours acting as the helpers. Among these nodes the least-loaded neighbours should be selected as the master of this cluster to manage the network and meta-data transmission. Our middleware clients run a Python script which will take one of potentially three already defined video files that are being served by our video server on an Amazon EC2 instance. The video server is hosted using Flask's REST protocol. The client's video requests are forwarded to our middleware which runs on a separate thread to make a connection with a master node. It is the client side middleware's responsibility to retrieve video segments in parallel and append all of them together and signal the end of the video request to the python video request. The client's middleware also interacts with the master node to send heartbeats and other node information. Furthermore, we implement Mesh as a Python dictionary for every node in the cluster so that it would serve as the basis of implementing other distributed system property. Mesh can be updated due to specific reason on certain node in the cluster, so every time it's updated the slave informs the change to the master and the master broadcast this update to all others. This is where we have reached so far, and based on this, we can extend our system by adding more features on it:

- **Load Balancing:** By adding a load field, which is defined as the number of streaming requests, to Mesh structure, every node in the cluster will have a clear view of the current load balancing. According to the view provided by Mesh, election can be held at appropriate time, and proper sender can be chose to help the requesting client.
- **Congestion Control:** By specifying a bounded sending rate to every node in the cluster, the master have the control over the data flow happening in the cluster. This should largely mitigate network congestion and thus reduce the loss rate of video segments.
- **Manageability:** To add access control for security reason, master is supposed to decide whether to give permission when a new device arrives. Mesh is updated once the newcomer complete the configuration.

3.1 Group View Synchronization

We used YuTube's concept of network meshes (Huang and Yu, 2008) to maintain a group view of connected clients for video downloads. When a client connects, it sends its system information to the server which then updates the master mesh and proceeds to broadcast this new mesh to all connected clients. For fault tolerance, if a client does not send data after a while or quits its connection with the server, its information is removed from the list of clients maintained by the server and its mesh information is removed from the group mesh with a broadcast as well. This

ensures consistent cuts in terms of the group view of the network mesh because video downloads will work properly as long as they happen in between changes in the clients connected to the network. This is related to consistent cuts in messaging as shown in Figure 1 because as long as a video request from a participating client occurs (the first straight square line) before a change in client conditions is broadcast, the download will be successful.

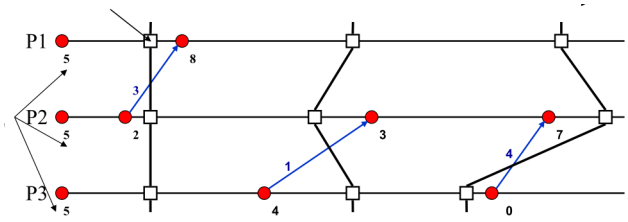


Figure 1: Cuts and Fault Tolerance

4 Experimental Plan

The experiments we intend to conduct on our system are the following:

- **The Naive Experiments or the Fair Partitions:** A series of two experiments, using both one and two neighbours, using the partitioning algorithm that simply partitions the file equally to each given client, sending the partitioned chunks to each helper as well as the main requesting client device.
- **The Dynamic Experiments**, or the Unfair Partitions: A series of two experiments, using both one and two neighbours, using an unfair partitioning algorithm that dynamically partitions the files based upon the values of the two aforementioned constraining resources (RTT and available memory).
- **K-Hops Experiment, or an alteration of the definition of neighbours:** Currently, for all of the above experiments, any client that connected to the middleware would be classified as a neighbour regardless of physical or network distance between client and server and between client and client. The definition of neighbour should and would be more limiting in a proper application setting, defined by the number of hops away each client is from one another as well as from the server. Instead of assuming any client that connects is a neighbour, apply a more stringent definition on the above experiments for both the single and dual neighbour case, like only clients that are one network hop away would be classified as true neighbours and the rest would be ignored. This proves to be the hardest of the experiments as our access was limited to 3 clients and a server, meaning this redefinition would and could prove troublesome.

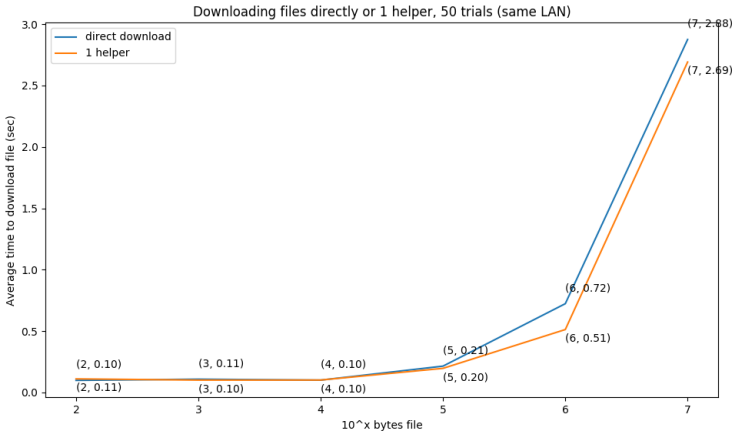
The experimental plan was more comprehensive than the results we were able to produce as a result of complications with regards to firewalls, ISP restrictions, bandwidth restrictions, and limited resource availability. Hence, we limited the results to, at max, a dual client setting to test base cases and the premises of our claims.

5 Results

All experiments conducted in our project had two computers on the same LAN- a home wifi. The two clients ran on different machines: a raspberry pi 3 and a 2017 Macbook Air, each with different storage capacities.

5.1 Fair Partitions Experiment

In experiment 1, we wanted to justify the entire basis of this paper; whether or not the middleware with even fair partitioning of the file would speed up the streaming or download speed. This is the aforementioned naive experiment. When file sizes are small or less than 100KB, our strategy of using 2 computers to download a file has the same performance as the direct download of the same file. But as file sizes get larger and on the scope of several MB, a drop in time to download using our fair-partitioning strategy can be seen. Our method cut down about 0.2 seconds for the 1 MB and 10MB file size cases. We have reasonable confidence in our results because we did 50 trials for each file size. It might have been a bit more informative to have tested more file sizes in between to see the shape of the curve.



Experiment 1: fair partitioning of files

5.2 Dynamic Experiment using Disk Space

In experiment 2, we partition the requested file based on available disk space. Our algorithm was considerably simpler than the one employed in the (Pq Nguyen, 2002), but the results show that even a simple disk-sized based partitioning algorithm helps the overall file download time. We see, instead of the ordinary 9.7% increase in speed for the largest file size in the naive approach, an improvement of 19%, clearly edging out the naive fair partitioning algorithm.

This is probably because in fair partitioning, the ratio is 1:1 whereas with disk space, the Macbook has a much larger chunk to download than the raspberry pi. Specifically, we calculated the portion a client must download as follows. Client 0's percentage of file =

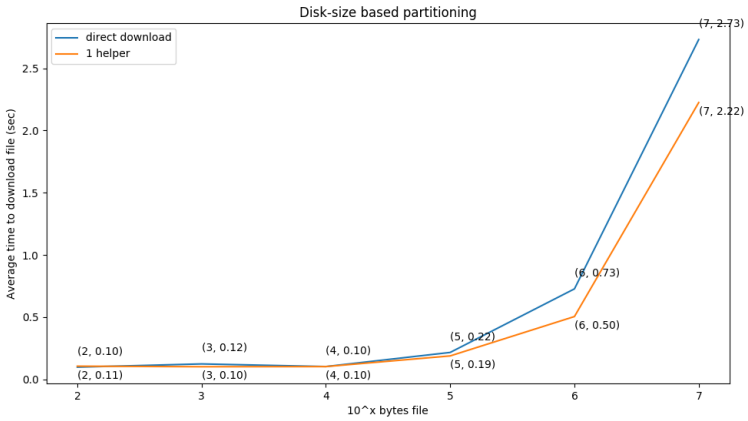
$$disk_0/(\Sigma_i disk_i)$$

The specific ratios for the macbook and raspberry pi was far from even, which indicates disk-space partitioning is a valuable metric for considering how to allocate chunks to participating

clients.

$$MAC = 0.79131832221$$

$$Pi = 0.20868167779$$



Experiment 2: Partitioning based on disk space

5.3 Dynamic Experiment using RTT

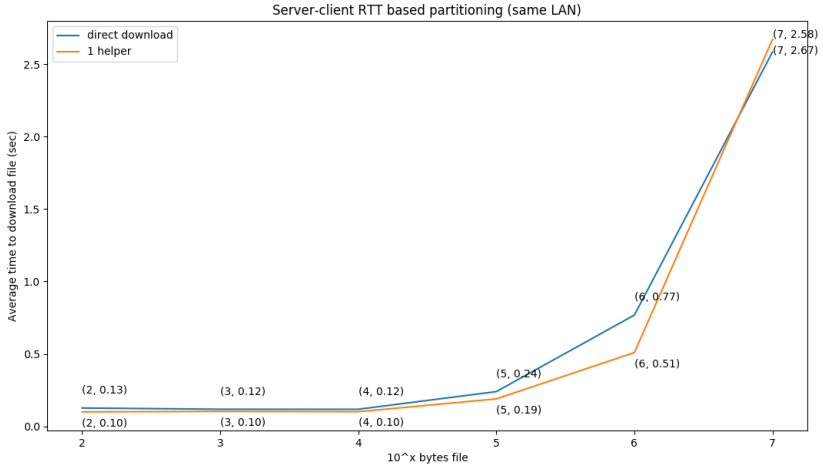
In experiment 3, we partitioned based on the RTT or round-trip time to the server. The time to download was not faster than direct download on the most part and actually may have inhibited or worsened the overall system with this partitioning algorithm. At the largest file size, we see a 3.4% slow down, while for smaller sized files, we see at max a 35% speedup. On average, RTT partitioning seemed to matter more for smaller files rather than larger files, or in our case, video files. Furthermore, this may have occurred because RTT ratios (calculated the same way as disk space ratio above in experiment 2) are essentially equal rendering this to be the same as experiment 1. We performed 10 trials of this experiment which speaks to why its results are a little bit different from the fair partitioning strategy. Figure 2 shows the comparison of all 3 of our partitioning strategies: equal and RTT based partitioning were essentially the same while disk based partitioning has better download times for larger files.

5.4 K-hops Experiment

We were not able to test the K-hops experiment or change the definition of neighbours as we could not get the WAN version functional as a result of firewall issues, ISP restrictions, and network limits.

6 Conclusion

In this paper, we proposed a middleware for fast video streaming and download. We built the system on the basis of network meshes for all nodes in the cluster to synchronize a global state and utilized different partitioning algorithm to realize streaming with the help of multi-senders. Our evaluation result shows that disk-based partitioning is the best strategy, but in general, parallel



Experiment 3: Partitioning based on ping rates

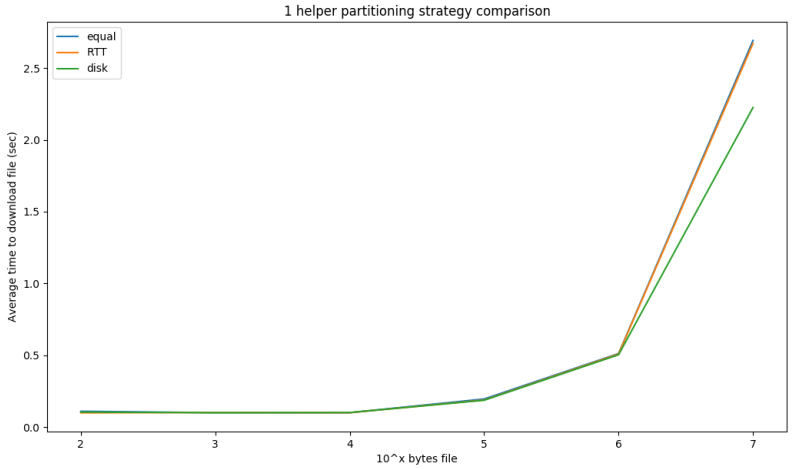


Figure 2: Comparing the partitioning strategies

video downloads from clients in the same LAN can speed up video file's (or any type of file's for that matter) download time.

For future efforts, we would like to focus on fault tolerance in the event of failure of video downloads from the EC2 server and implementing an election algorithm in the case that the managing middleware server crashes or has some error. Another possible extension of this work is getting clients to interact with clients on other LANs, but dealing with open ports, port forwarding, private/public IPs can be challenging for this endeavor. A third future project could be testing video streaming instead of static file downloads and showing a reduction in buffering times using our algorithm and strategy. Finally, the way partitioning and chunks are assigned could be optimized as a combination of many more factors aside from RTT to the video server and disk space. Using a combination of these factors could lead to the best parallel download strategy.

References

Bhattacharyya, R., Bura, A., Rengarajan, D., Rumuly, M., Shakkottai, S., Kalathil, D., and Mok, R. K. P. (2019). Qflow: A reinforcement learning approach to high qoe video streaming over wireless networks.

Biersack, E. W., Rodriguez, P., and Felber, P. (2004). *Performance Analysis of Peer-to-Peer Networks for File Distribution*. France and Microsoft Research, Institut EURECOM.

Do., C. (2007). In *Seattle Conference on Scalability: YouTube Scalability*. Google Tech Talks.

Do, N., hsin Hsu, C., and Venkatasubramanian, N. (2012). *Video Streaming over Hybrid Cellular and Ad Hoc Networks*. IEEE Transactions on Mobile Computing.

Huang, D. Y. and Yu, E. (2008). Yutube: A scalable distributed video-streaming system.

Huang, T.-Y. et al. (2017). A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *Stanford University*, (stanford.edu).

Huang, Y., Gao, Y., Nahrstedt, K., and He, W. (2009). Optimizing file retrieval in delay-tolerant content distribution community. Proc. IEEE 29th Int'l Conf. Distributed Computing Systems (ICDCS 09).

Liu, Y., Guo, Y., and Liang, C. (2008). A survey on peer-to-peer video streaming systems. *Peer-to-Peer Networking and Applications*, 1(18-28 10.1007/s-0006-y):12083–007.

Pq Nguyen, Thinh & Zakhor, A. (2002). Distributed video streaming over internet. page 10.1117/12.449979.

Saito, Y., Bershad, B. N., and Levy, H. M. (1999). Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. *ACM SIGOPS Operating Systems Review*, ACM Vol. No. 5:33.