VStorm: Video Traffic Management By Distributed Data Stream Processing Systems

Le Xu Klara Nahrstedt Indranil Gupta
Department of Computer Science, University of Illinois at Urbana-Champaign
{lexu1, klara, indy}@illinois.edu

Abstract

Recent published work shows that Quality of Experience (QoE) has become one of the major concerns in the area of large-scale multimedia Internet services. Due to the significant increase of video traffic and continuously growing need of video quality experience, we need new management platforms for multimedia traffic. Recent studies propose a variety of solutions that address this challenge. However, due to the complexity of algorithms, these solutions require large amount of computational resources. On the other hand, we have also witnessed the emergence of large-scale distributed stream processing systems. These systems provide real-time results for nearly all types of data streams and computation in a massive scale. In this project, we explore the possibility of deploying distributed data stream processing (DSP) systems in a large-scale multimedia network with dynamically changing Internet resource.

In this paper, we use a popular distributed stream processing system, Apache Storm to implement multiple frameworks proposed by recent published work. Simulation results on our frameworks show that implementing complex stream control strategy in DSPs can be efficient and flexible.

1. Introduction

As more users choose Internet over cable as main resource of video streaming service, Internet video service quality has now become a major concern. Studies show that in year 2014, consumer Internet video traffic has occupied 64% of all consumer Internet traffic, while this number may rise to 80% by year 2019 [5]. The scale of the video traffic makes it challenging to build a fully scalable platform to enable traffic monitoring, analysis and control. Moreover, different from regular data transmitting model, user expectation towards video traffic can no longer be defined by throughput or end-to-end latency, but restrictions such as video quality (frame rate and frame size), re-buffering rate, and quality variations, etc. [13]. We categorize these metrics as Quality of Experience (QoE), a quantifiable measurement designed for Quality of Service for video streaming services. QoE can be affected by a variety of factors, especially unpredictable network throughput and latencies. One of the most common approaches to address this issue efficiently is by using bit-rate adaption [10, 12, 13]; e.g., maximizing the bitrate when network connection has sufficient bandwidth in order to improve QoE; minimizing the bitrate when network become congested to prevent re-buffering events. Many recent studies [13] [10] propose designs and techniques to address these challenges. However, these solutions are mostly implemented inside a lower layer protocol [13], which may cause large computational overhead during data transmitting.

In parallel with the emergence of Internet video streaming network, the big data era has also witnessed the development of distributed stream processing systems (DSPs). These systems are designed to process queries and data streams in real time, which makes it naturally compatible with multimedia streams (video and audio). In the past decade there has been a wave of studies to build this type of systems, targeting shorter query latencies, higher throughput, and scalability. This design leads us to question whether there exists an DSP solution to address the bit-rate adaption challenge. In this project, we choose Apache Storm [3] and GStreamer [8] to implement an application-layer solution based on two existing projects. There are a large number of techniques proposed by multimedia system area to improve video quality in general. In

this work, we are going to majorly focus on solutions proposed by two work, [13] and [10]. [13] proposes a QoE model that defines client-side bit-rate adaption strategy, while [10] proposes an Internet control plane that enables bit-rate adaption and video source adaption simultaneously based on prediction to the server performance.

Note that the goal of this work is not to implement these methods using the identical metric or techniques described in [13] and [10], but to build a prototype framework enabling the possibility of leveraging these computational complex techniques with an scalable, application-level solution. In the following sections we are first going to explored these two works and the systems we are going to deploy in the implementation. Then Section 3 is going to discuss the design of these model in terms of Storm Topologies. Finally we are going to demonstrate simulation results and address some questions and discussions.

2. Related Work

2.1 Client-side Bitrate Adaption

Recent published work [13] has proposed a new notion of Quality of Experience for video streaming systems. Instead of using a single metric that measures the QoE of these systems, this work proposes a flexible mathematical model to combine multiple sub-metrics in the same equation by different contributing parameters. This gives a much more general model for users with different needs to evaluate their QoE in a single metric, while also provides a convenient way to use a optimization algorithm (MPC) to compute a bitrate that can potentially optimize its QoE.

However, this optimization algorithm is rather computationally heavy so that multiple techniques have to be applied to the algorithm for it to become practical. Specifically, in the FastMPC proposed by the paper, it predetermines the range of possible values of each sub-metrics, e.g., buffer level, bitrate, server throughput, etc. The optimization computation is completed before streaming starts, which introduces potential large start-up time. Moreover, the paper also suggests that the video player cannot be bundled with an external solver due to the licencing issue.

One way to solve this problem is to implement the algorithm itself in an underlying transmission protocol such as DASH [12]. However, embedding computationally heavy workload inside DASH protocol may introduce complexity for the user to tune their QoE metric as needed and is not recommended to achieve an End-to-End architecture [11]. In the next section we discuss a design to use a compositional QoE model for bit-rate adaption by a Storm Topology. In this implementation have expanded the source of video stream from one server to multiple servers.

2.2 Internet Video Control Plane

Work [10] discussed another solution to improve streaming video quality. Instead of merely measuring the video quality and network bandwidth from the client side, [10] suggests adapting a middle layer between clients and servers carrying the logic of optimization, i.e., bitrate adaption and selection of CDNs. The paper demonstrates examples from the real-world workload that proves network performances from different CDNs are unpredictable and can be largely varied by space and time. Hence the paper proposes a control plane to 1. select target CDN based based on each client's historical CDN quality data, and 2. select proper bitrate for the stream in order to achieve user-defined global optimization.

The paper proposes a concrete design plan for the control plane. Nevertheless, the paper does not address a detailed implementation of such design, which entails multiple interesting potential opportunities for future research. These problems include the scalability of the algorithm, interactions with CDNs, syncing among different controllers, etc. In this project we aim to explore whether distributed stream processing systems can tackle these challenges.

2.3 Selection of Existing Tools

GStreamer [8] is an open-source multimedia framework that provides programming support for developers to build complex multimedia pipelines (as example shown in Figure 2). In a GStreamer, each element can serve as processing unit and has both source(s) and sink(s) that serves as input and output of video(audio) streams.

We also introduce Apache Storm [3], one of the fastest and most scalable distributed stream processing (DSP) systems. We compare the nature of this programming paradigm (as example shown in Figure 2) with Apache Storm's programming model. The similarities between these systems are obvious: both models process real-time data and both models use acyclic graph of computational component to perform tasks. In the

following sections we discuss the possibility of using Storm (or other DSPs) to perform multimedia streaming tasks.

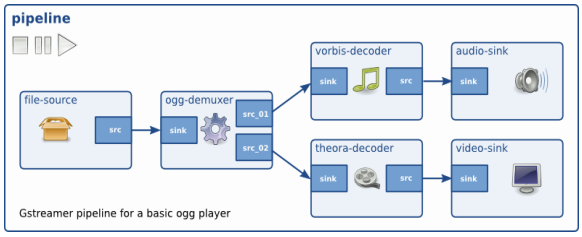

Figure 1. An Example GStreamer Pipeline [4]

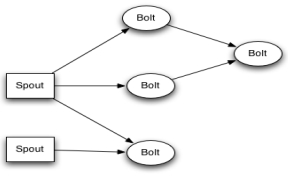

Figure 2. An Example Storm Topology [9]

3. Architecture and Implementation

In this section, we are going to introduce two frameworks built to accommodate computationally-heavy algorithms such as the ones proposed in [13] and [10]. Note that the the goal of this work is not to implement the algorithms proposed in these work, but to demonstrate the possibility of leverage complex video streaming optimization efficiently, on an application-level system. In these frameworks, video/audio clients are implemented by GStreamer [8]. The video quality control and resource management logic are implemented by Storm [3] topology.

3.1 A Client-side Bitrate Adaption Model

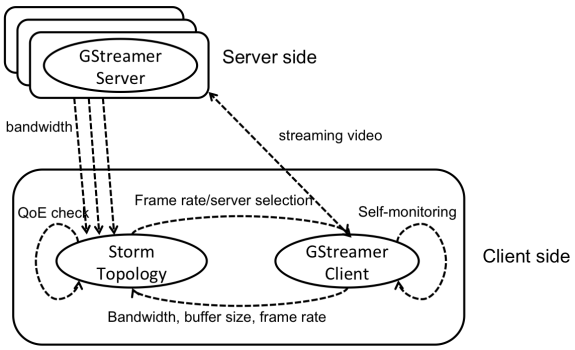

Figure 3. Client-side Bitrate Adaption Architecture

Figure 3 illustrates an architecture enabling client-side bitrate adaption. In this architecture, QoE control and adaption algorithm is run inside a Storm Topology on each client in parallel with the GStreamer video client. This topology relies on statistics periodically measured and pushed by video clients. After receiving statistics input from video client, Storm runs QoE approximation to evaluate the quality of the video stream, selecting appropriate frame rate and source of the streaming. The logical workflow of the storm topology is illustrated in Figure 4.

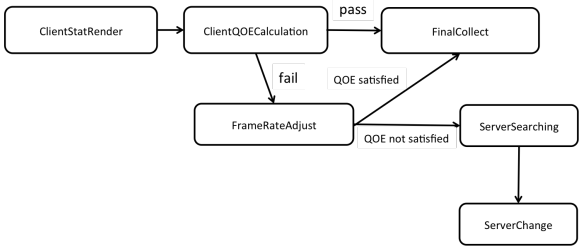

Figure 4. Client-side Bitrate Adaption Topology

The current implementation of a client-side bitrate adaption is composed by 6 components:

- ClientStatRender - This operational component periodically collects and propagates client statistics. These statistics includes client-side bandwidth, buffer size (buffer occupancy) and current rate. Each set of statistics collected from a client are stored in a data structure called *tuple*.
- ClientQoECalculation - This operational component collects video client statistics from ClientStatRender and apply user-defined QoE calculation algorithm. In our implementation, QoE can be calculated by:

  $$QoE = \alpha \cdot bandwidth + \beta \cdot buffer\_size + \gamma \cdot frame\_rate$$

  In this equation, α, β, γ are user defined parameters specifying the importance of each sub-metric. This bolt can also be implemented to read these parameters periodically from a file so that users can change these parameters dynamically without interrupting current workload.
- FinalCollect - This operational component collects the computational results and logs processing latencies. It also overwrites configuration file for GStreamer client if bitrate rate adaption operation needs to be carried out.
- FrameRateAdjust - This operational component handles all tuples that fail QoE checks in ClientQoECalculation. It checks whether the there ex-

ists a frame rate that can optimize current video stream's quality and satisfy its QoE constraints. If this frame rate is different from the current frame rate but within the user defined range then the new frame rate is propagate to FinalCollect. If the frame rate is not within the user defined range the tuple will be further handled by ServerSearching.

- ServerSearching - This operational component serves tuples that cannot achieve their QoE goals streaming from their current sources. This component pulls resource (available bandwidth) data from all sources and searches source servers with adequate bandwidth to improve current streaming quality. All tuples processed by this component are propagated to ServerChange. This process is placed in the downstream of the flow to avoid unnecessary communication with the servers.
- ServerChange - This operational component finalizes server searching process by first overwrites configuration file for GStreamer in order to notify the new target server. It also logs the processing latencies for analyzing purposes.

3.2 A Middle Layer Resource Management Model

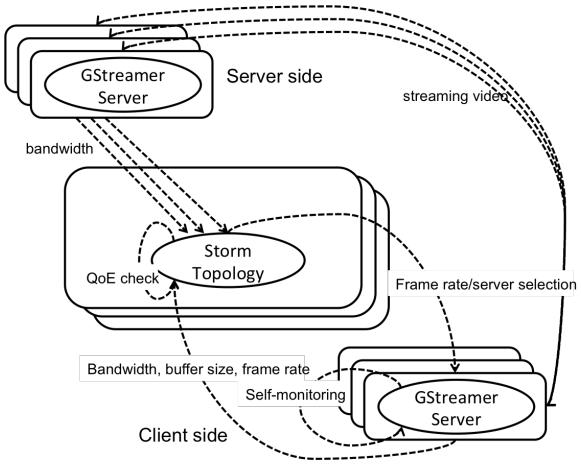

Figure 5. Middle Layer Resource Management Architecture

A middle layer streaming resource management model is shown in Figure 5. A control plane between CDNs and video clients, originally proposed by [10], alleviates heavy computation from both clients and servers. With the knowledge of both servers and clients globally, the control plane runs global optimization algorithm, such as traffic load balance among servers.

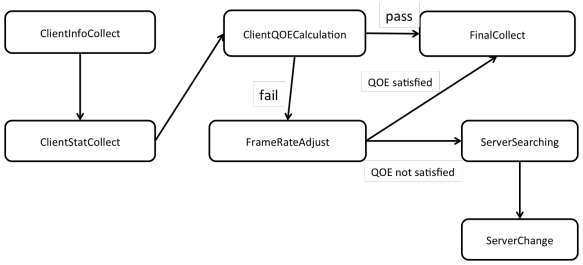

Figure 6. Middle Layer Resource Management Topology

In our implementation, we Storm topology is run in a cluster with connection to both clients and servers. The initial design of Storm topology is shown in Figure 6. Currently, the topology workflow is similar to the topology presented in subsection 3.1. However, in topology deployed in a control plane, we add a computational component "ClientInfoCollectSpout" to periodically collect the information of current client in the system. Once the data is collected, it will be processed by the similar workflow shown in Figure 4.

3.3 Middle Layer Resource Management: An Improved Model

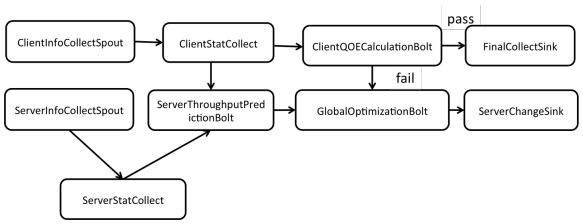

Figure 7. Improved Middle Layer Resource Management Topology

Figure 7 shows a Storm workflow enabling global optimization. In this topology, ClientInfoCollectSpout and ServerInfoCollectSpout learn the number of clients and servers in the system . Then ClientStatCollect and ServerStatCollect collect statistics from both clients and servers. Then all data are delivered to ServerThroughputPredictionBolt. This component predicts performance (bandwidth, throughput) of each server towards every client in the system. As explained in [10], this step is necessary since network condition varied by complex factors. Another component, ClientQoECalculationBolt checks QoE restrictions for every client. Then GlobalOptimizationBolt will subsequently collect clients that fail to satisfy their QoE requirements and assign new target servers based on

user-defined global optimization strategy, e.g., traffic load-balancing. Finally ServerChangeSink and FinalCollectSink finalize the computation and apply computational result to configuration of every GStreamer client.

3.4 Current Development Progress

The integration between GStreamer servers/clients and Storm cluster for some of the models are currently being implemented. All GStreamer servers and clients discussed in this section is largely developed from an implemented architecture from [1]. The model discussed in subsection 3.3 is currently under development and will not be involved in the simulation.

4. Simulation

In order to measure possible performance of Storm topologies discussed in Section 3 we perform groups of simulation for both middle layer resource management model and middle layer resource management model. In order to serve real time video streaming application, it is critical for the control platform to process with minimal latencies. As a result in our simulation we are going to use latencies as our main metric.

4.1 Evaluation Setup

We perform evaluation on a 12-node cluster on Emulab [7]. Each node [6] in the cluster is equipped with 2.4 GHz quad core processor, 12 GB RAM, and 750 GB storage space. The cluster is internally linked by 100 Mb LAN network.

Among 12 nodes, 9 of the nodes are used to set up a Storm cluster (1 for Nimbus host and 8 for Supervisor hosts). We varied the number of servers (in Client-side model) and the number of both clients and servers (in Middle layer model) by increasing the number of asynchronous threads updating the client statistic file and server bandwidth file. These files are synced to Storm cluster (and pushed back to clients) by underlying NFS.

4.2 Execution Latency

Figure 8 and Figure 9 show performance of Storm topology by total execution latencies for Client-side bitrate adaption and middle layer resource management model respectively. In these figure vertical axis marked the number of servers we use in each simulation run, while horizontal axis marked the total execution latency in milliseconds. We performed two sets of experiments with varied fetching rate: 500 ms (show in the plot as blue bars) and 100 ms (show in the plot as red bars). Fetching rate indicates the time period between each data fetch initiated by the Storm topology. And the total execution latency we measure indicates the total amount of time each tuple spends in operational components discussed in Figure 3 and Figure 6.

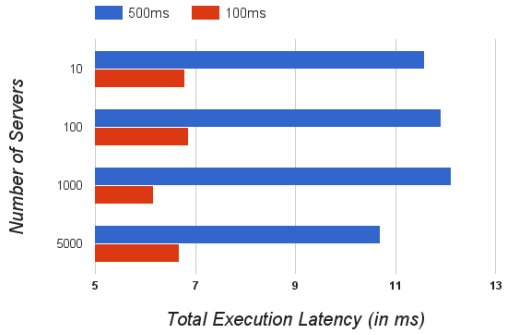

Figure 8. Client-side Bitrate Adaption Model: Total Execution Latencies

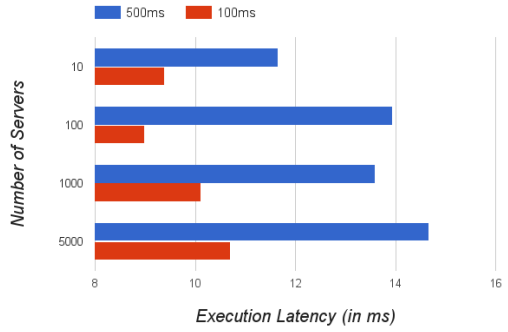

Figure 9. Middle Layer Resource Management Model: Total Execution Latencies

From Figure 8 we observe client-side bitrate adaption model achieves sub-second latencies: less than 12ms for all scenarios for 500ms fetching period and less than 7ms for all scenarios for 100ms fetching period. Interestingly, we can also conclude from this plot that neither increasing number of servers nor increasing the number of data fetchings necessarily affect topology's latency. This is possibly caused by non-congested execution under abundant resources. From Figure 9 we can also make the similar conclusion for middle layer management model.

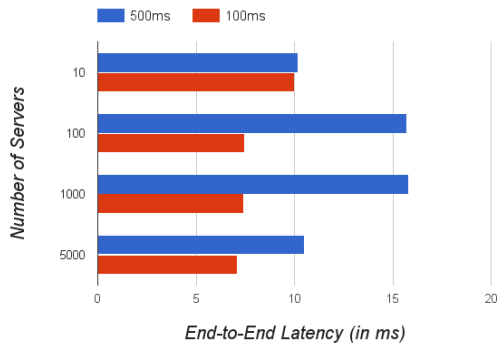

Figure 10. Middle Layer Resource Management Model: Total End-to-End Latencies

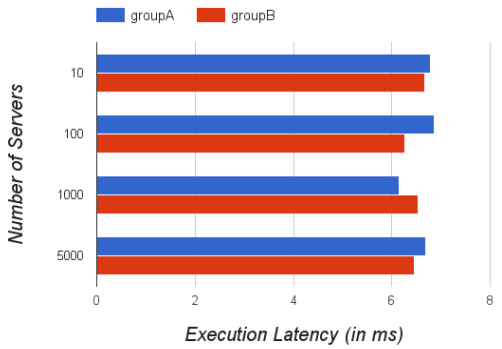

Figure 11. Client-side Bitrate Adaption Model: Varied Update Rate

4.3 End-to-End Latency

Figure 10 illustrates the average End-to-End latency of middle layer management model. End-to-End latency indicates the difference between the time when the tuples being generated and the time when the tuples being processed by the last component in the topology. Comparing to execution latency, end-to-end latency involves both processing time and network transmission time.

Based on Figure 10 we observe that similar to execution latency, end-to-end latency is not affected by increasing amount of servers or fetching rate. Both latency measurements indicate that under changing workload of video streaming network, Storm's performance can be good and stable.

Comparing to Figure 9 the latencies we observe in Figure 10 are generally shorter in the same condition, due to the fact that two measurement methods are used to generate these results. We use Storm internal metric to measure execution latency, this value would also involve initial large outliers generated when the topology starts. For measuring end-to-end latency we manually select data points after the topology has stabilized.

4.4 Varied Data Update Rate

Finally, we study the impact of data update rate towards client-side bitrate adaption model. Figure 11 shows the performance of Storm topology under different data update frequencies. In group A experiment, statistics is updated by clients every 5 seconds and bandwidth information is updated by server every 1 second. In group B experiment we decrease the update period to become 50% of the original. Based on result shown in Figure 11, we can conclude that the data update rate has little effect on topology performance.

5. Discussion

Using Storm To Build Video Streaming Pipeline: One of the initial goals for this work is to explore whether there is a possibility to use Storm (or other distributed data stream processing systems like Spark Streaming [14], Apache Flink [2]). After investigation on GStreamer framework I found it is possible to adapt these framework to video data streaming. However it will take more effort discovering lower level API to combine fundamental video streaming functionality into these applications.

References

[1] CS414: Multimedia System Design. https://courses.engr.illinois.edu/cs414/sp2014/, 2014. ONLINE.

[2] Apache Flink. https://flink.apache.org, 2016. ONLINE.

[3] Apache Storm. http://storm.apache.org, 2016. ONLINE.

[4] Bins and pipelines. https://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/section-intro-basics-bins.html, 2016. ONLINE.

[5] Cisco Visual Networking Index: Forecast and Methodology, 2014-2019 White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html, 2016. ONLINE.

[6] D710. https://wiki.emulab.net/wiki/d710, 2016. ONLINE.

[7] Emulab. http://emulab.net/, 2016. ONLINE.

[8] GStreamer, open source multimedia framework. https://gstreamer.freedesktop.org, 2016. ONLINE.

[9] Storm Tutorial. http://storm.apache.org/releases/current/Tutorial.html, 2016. ONLINE.

[10] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A case for a coordinated internet video control plane. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 359–370. ACM, 2012.

[11] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[12] I. Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, (4):62–67, 2011.

[13] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338. ACM, 2015.

[14] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, pages 10–10. USENIX Association, 2012.