Reproducing AWStream's Pedestrian Detection

Eric Prokop
Stanford University
ecprokop@stanford.edu

Peiqian Li
Stanford University
peiqian@stanford.edu

ABSTRACT

AWStream (Adaptive Wide-Area Streaming Analytics) [7] is a framework for streaming data analytics applications to smartly trade off between latency and application performance in the face of limited network bandwidth. It aims to simplify application development by providing an interface through which developers can specify configuration "knobs" that control how data is degraded when bandwidth is constrained. AWStream learns the best performing configuration for a given available bandwidth, and uses that to adjust the data send rate when it detects congestion in the network.

We present our efforts to reproduce Figure 12(a) from the original AWStream paper, which compares latency and accuracy for a pedestrian detection application both running with and without AWStream. We also show our reproduction of two intermediate results, including an offline-trained profile and a time series plot for throughput, latency and accuracy.

1 INTRODUCTION

Streaming data analytics applications have become increasingly pervasive in recent years: real-time log processing, pedestrian detection and vehicle tracking for surveillance camera footage, and Internet of Things (IoT) data analytics are all examples of applications that often require data to be streamed from a source to a remote analytics server. When data sources are numerous and widely distributed, the most cost-effective solution is to stream data over a wide-area network (WAN) such as the public Internet. In many cases (IoT devices, surveillance cameras) a wireless link is used for the last hop.

1.1 Problem Background

The main challenge that this architecture presents for applications is that WAN bandwidth is variable, scarce, and expensive. Bandwidth variability and scarcity stems from the fact that a data stream shares the network with many other flows, all of which are non-deterministic in terms of arrival, length, and bandwidth consumption. Wireless links over the last hop also contribute to scarcity. Furthermore, data center operators often charge more for wide-area bandwidth between two sites. Applications relying on low-latency between data sources and servers must therefore properly detect and deal with congestion.

Adaptive streaming applications aim to solve this problem by dynamically trading off accuracy for less bandwidth consumption during periods of congestion. The client follows some policy that determines how to degrade the data in order to reduce the send rate. In practice, many streaming applications utilize manual policies; for example, a surveillance system for pedestrian detection may have a policy stipulating that frame rate be reduced by 1/3 when latency exceeds a certain threshold. Zhang et al. discuss three issues with the policy-based approach:

- Policies are application-specific and do not generalize. For example, reducing the frame rate may be one policy for video-based streaming applications, but does not apply to non-video applications such as streaming log analysis.
- Policies are often written by developers based on heuristics, rather than driven by evaluation metrics, and therefore are often suboptimal.
- Specifying policies manually is tedious, error-prone, and therefore not scalable.

1.2 AWStream's Contribution

AWStream is a framework with three modes of execution (profiling, client, and server) that work in tandem to alleviate each of these challenges for applications:

- AWStream features a generic `maybe` API to interface with any kind of streaming application. Developers specify a set of degradation functions with parameters ("knobs") that AWStream can tune when generating adaptive policies.
- AWStream generates adaptive strategies via a data-driven approach by employing a combination of offline and online profiling to learn a Pareto-optimal policy for when and how to degrade data.
- AWStream client detects congestion and adapts the send rate accordingly by following the learned policy.

The authors evaluated AWStream by comparing latency and application accuracy results for an application with no adaptation (using TCP and UDP as transport protocol), as well as various adaptive approaches (JetStream, JetStream++, HTTP Live Streaming, and AWStream). The authors implemented a pedestrian detection application which uses a GPU-accelerated histogram of oriented gradients algorithm with an SVM classifier to output bounding boxes with normalized

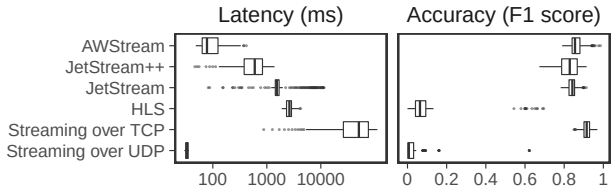

Figure 1: Figure 12(a) from the original AWStream paper. The box plot shows latency and accuracy for the pedestrian detection application during a period of 240 seconds when bandwidth is constrained. When run with AWStream, the application has much lower latency and only a small reduction in accuracy when compared with streaming over TCP (no client-side adaptation).

coordinates on the image. The outputs are compared against the reference result from the raw data, and declared a success if the intersection over union (IOU) of the bounding boxes is greater than 50%.

Figure 1 shows the results for a pedestrian detection application when bandwidth is constrained at the client using the Linux `tc` utility. It can be seen that AWStream achieves sub-second latency with only a small reduction in accuracy when compared with streaming the raw data over TCP.

Using the AWStream code on Github[1], we attempt to reproduce Figure 1. We run the same pedestrian detection application as the original authors and use the same dataset (MOT16 [3]) for offline profiling and evaluation. We run our client and server in separate regions in Google Cloud Platform (the original authors used Amazon Web Services) with Linux `tc` to constrain bandwidth at the clients.

In the next section we discuss related work. Then, we describe our reproduction efforts in detail, including setup, methodology, results, and future work.

2 RELATED WORK

Streaming analytics systems. JetStream [5] is a wide-area streaming system that Zhang et al. compare with AWStream. JetStream has a policy framework that allows developers to formulate policies for sending data at a rate that maximizes value (e.g. accuracy). Compared to JetStream, AWStream has the advantages we mentioned in Section 1.2, as well as the support for different resource allocation policies for multiple applications. In addition, [7] shows that AWStream achieves a lower latency when evaluated against JetStream for two video applications, augmented reality and pedestrian detection.

Multimedia streaming systems. For two video-based applications, Zhang et al. also compared AWStream against a few multimedia streaming protocols including RTP [6] and HLS [4] (an HTTP-based protocol focused on optimizing adaptation strategies for quality of experience). As the AWStream authors argue, RTP only aims to achieve low latency often at the expense of very poor accuracy; HLS adaptation reacts slowly to changing network conditions, and its strategy is often suboptimal for applications such as pedestrian detection relying on image detail.

Salsify [1] is an architecture for adaptive video streaming that tightly couples the network transport protocol with the video codec. The system aims to encode each frame near and below a target size which is dynamically estimated by observing network conditions. While Salsify aims to optimize for low latency and high SSIM (structural similarity), AWStream (through its offline/online profiling process) attempts to maximize an application-specific metric, such as accuracy in a pedestrian detection application.

3 EXPERIMENT

3.1 Setup and Methodology

Our environment consisted of two nodes provisioned in Google Cloud Platform: a server in Belgium and a client in Oregon, USA. We forked[2] the AWStream GitHub repository, which hadn't been significantly updated in over a year. We made various patches to the Rust code and helper scripts to accommodate API changes in the dependencies. We also modified hard-coded configurations and logic specific to the computing environment, application, and source data.

After resolving all build and runtime errors, we reverse-engineered the steps needed to generate a profile from the pedestrian detection dataset. We subsequently ran an AWStream client and server to measure runtime latency, throughput, and accuracy. The project's documentation provided a rough outline on how to achieve this, but overall we found it to be outdated and at times misleading. We relied mostly on close reading of the code and email correspondence with the original paper's authors for guidance.

3.2 Reproduction Steps

Offline profiling. The goal of profiling is to find the optimal configuration for each available bandwidth for some spectrum of bandwidths. To find the Pareto-optimal profile, we ran AWStream in profiling mode (a series of binaries) on the MOT-16-04 dataset to record bandwidth and application accuracy for every combination of knobs. In this case, knobs include video frame rate, frame width, and quantization (an H.264 parameter[3]). Using an Ubuntu VM with one Nvidia K80 GPU on a 35 second (1050 frame) video, this step took about four hours. From there, we used the provided code

[1] github.com/awstream/awstream

[2] Fork available at: bitbucket.org/peiqian/cs244-awstream

[3] https://www.vcodex.com/h264avc-4x4-transform-and-quantization/

to calculate summary statistics over time intervals and determine the profile. This step produced Figure 4, which is discussed in Section 4.

Streaming across the WAN. Rather than encoding video on the client, streaming it across the network, and running the pedestrian detection histogram of oriented gradients algorithm on the server, the AWStream client and server code runs a simulation. The client accepts a `source.csv` file that contains, for each of the Pareto-optimal configurations in the profile, the size of the data to be sent at each frame. At runtime, the client adaptation code measures the available bandwidth (as described in the original paper), selects the configuration to use, and sends an appropriately-sized "dummy frame" (a stream of zeroes of the desired size in terms of the number of bytes) across the network, along with the configuration itself. Upon receiving a "dummy frame", the server uses the same configuration to look up the application accuracy for that frame based on precomputed statistics.

We wrote a script to generate `source.csv` (as it is missing in the original repository), and ran both the client and server for ten minutes using the Linux tool `tc` to restrict bandwidth at the client in the same manner as the original paper: no shaping before t=200s; at t=200s limit bandwidth to 7.5 Mbps for 3 minutes; at t=380s limit bandwidth to 5 Mbps; at t=440s remove all shaping.

We also ran the client with no adaptations to reproduce the "streaming with TCP" box plot in the original Figure 12(a). Since sending raw data requires around 238 Mbps bandwidth, we send frames with H.264 quantization level equal to 10, which only requires about 72 Mbps throughput (our estimated network bandwidth on Google Cloud between our server and client nodes was about 150 Mbps). In the original paper, the authors set quantization level equal to 20, which required 17 Mbps bandwidth (they had imposed a background bandwidth upper limit of 25 Mbps).

4 RESULTS

Figure 4 is a plot of the profile we generated for MOT-16-04 using the steps outlined in Section 3. Each tick on the plot represents one of the 216 possible configurations (unique combination of knobs). Highlighted are the Pareto boundary as well as the configurations in which only a single knob is modified. It's nearly identical to Figure 3, the profile plot from the original paper. This is to be expected, but provides a good intermediate sanity check, which was useful given the difficulties we ran into running the code.

Figure 2 shows our reproduction of Figure 12(a) from the original paper (Figure 1). Latency results are quite similar: AWStream is able to adapt to the imposed bottleneck by detecting congestion and lowering the data rate. As a result, latency is relatively low since it only sends a quantity of data that can fit in the pipe. Unsurprisingly, latency for plain TCP balloons to well over 10 seconds in both cases because it attempts to send raw, uncompressed frames through a bandwidth-restricted network. Our latency is slightly higher (never dips below 100ms) which can probably be explained by the fact that our client-server setup sends from Belgium to Oregon, while the original setup sends from the east coast of the United States to the west coast.

Accuracy results hold true as well. In both cases, the F1 score when running with AWStream hovers around 80%. Although we'd expect TCP accuracy to be higher in our reproduction because we used a quantization level of 10 rather than 20, the results appear to be the same. We cannot explain why this is the case, but we noticed that the AWStream reference data available on GitHub suggests that TCP accuracy should be closer to 0.89.[4] while in Figure 1 the accuracy appears to be well above 0.9.

We also produced a time series plot showing throughput, latency, and accuracy (Figure 6). We can clearly see the period when traffic shaping occurs from 200s to 440s. TCP throughput drops while the latency jumps, which is expected. However, we do not observe AWStream taking full advantage of the available capacity. This is surprising, because AWStream is supposed to probe for additional network capacity and increase the data quality when bandwidth becomes available. We see this starting to occur in the blue spike at the beginning of the throughput plot, but it drops off shortly after. One possible explanation is that AWStream detected congestion, scaled back the data rate, and never recovered for some reason. We observed this behavior for several runs, which suggests there could be a bug in the adaptation code, but the results are inconclusive. For comparison, we also provide Figure 5, which is a time series plot from the original paper for the augmented reality application. A time series plot for pedestrian detection wasn't provided so we can't make a direct comparison, but we do see similar patterns for latency and accuracy between the two applications.

Overall, we found that the results from the original paper are reproducible. The profile plots are nearly identical, which is to be expected because there is little to no non-determinism involved in profile generation. More importantly, the AWStream code performs close enough to the paper's description to reasonably reproduce the results of the original paper for the latency and accuracy and box plots, with some caveats as described above. What is less clear to us is how AWStream, when used for video streaming-based applications, compares to existing video streaming systems which are unaware of

[4] https://github.com/awstream/data/blob/master/reference-data/mot.tcp.csv

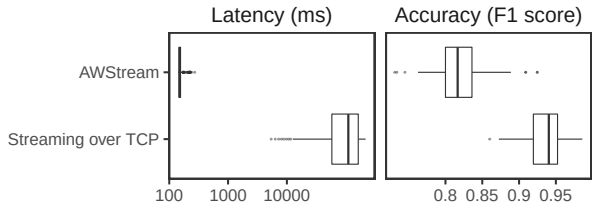

Figure 2: Our reproduction of Figure 12(a).

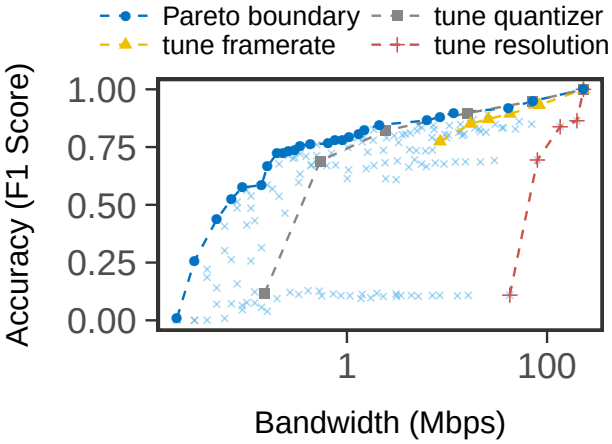

Figure 3: The original profile plot.

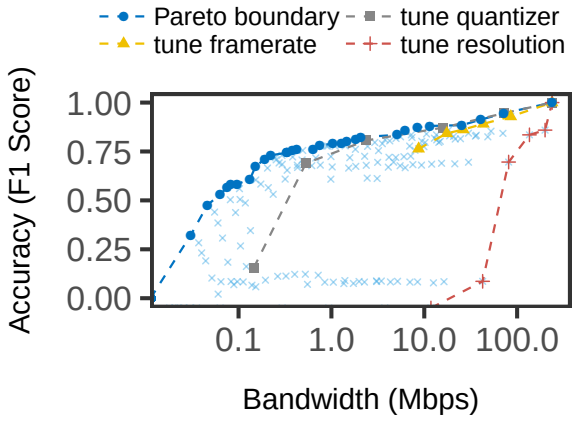

Figure 4: Our reproduction of the profile plot.

the particular application running on top. We discuss this more in the next section.

5 DISCUSSION AND FUTURE WORK

In the original paper, the authors compare AWStream to five other approaches to sending data across a network: HTTP

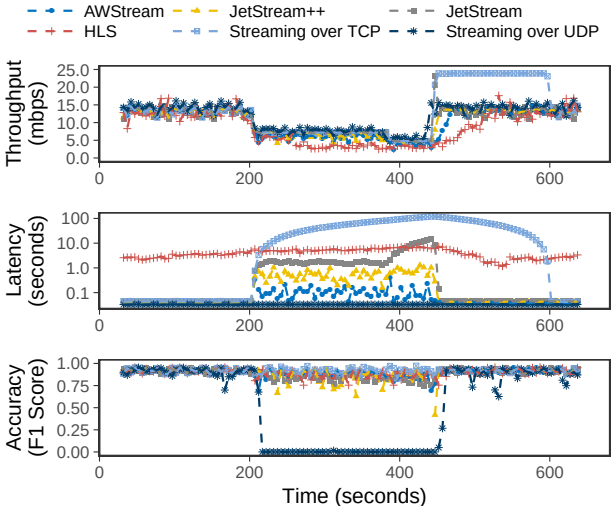

Figure 5: The original paper's time series plot for an augmented reality application.

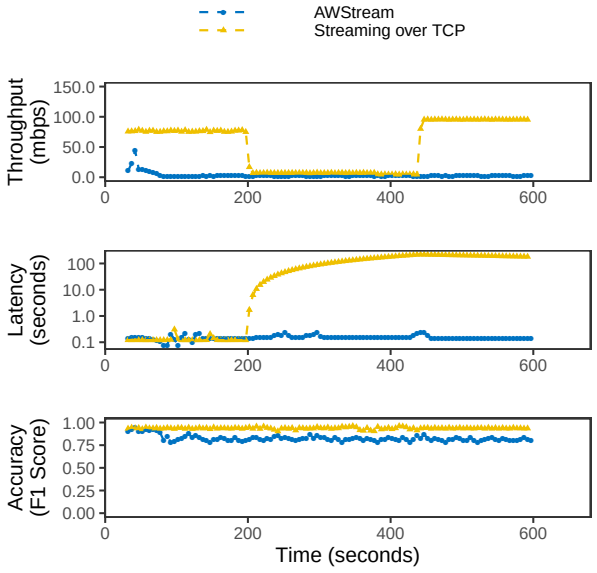

Figure 6: Our time series plot for a pedestrian detection application.

Live Streaming, plain TCP, plain UDP, JetStream, and JetStream++. Of these, we believe only JetStream can be considered a fair comparison: HTTP Live Streaming with 1 second chunks doesn't have low enough latency; plain TCP with no adaptation sending data at a rate higher than the network can handle will naturally cause network congestion; UDP with no congestion control isn't a realistic approach if one is concerned about packet loss, and is frowned upon by

IETF[2]; JetStream++ is JetStream using the profile generated by AWStream, and isn't an existing system.

Although AWStream is meant to be a general framework for any real-time streaming analytics applications (not just video streaming applications), it's unclear if it actually performs better than other video streaming systems which are application-unaware (but *are* optimized for video), such as WebRTC, Skype, Zoom, or, more recently, Salsify[1]. It would be interesting to compare performance for various video-based applications using AWStream vs. one of these existing frameworks. We intended to compare AWStream against WebRTC for the pedestrian detection application, but were unable to finish the implementation before our deadline.

6 CONCLUSION

The original paper's authors presented AWStream as a software framework with an innovative, well-defined API for developers of real-time streaming analytics applications to smartly trade off between latency and throughput. Its goals were two-fold: provide a boost to application performance by learning an optimal profile, and make developers' lives easier.

We were able to successfully reproduce partial results from the paper around application performance: AWStream learns a profile based on training data and uses it perform real-time adaptation. For applications that care about latency, AWStream presents a clear improvement over plain CUBIC TCP when streaming across bandwidth-constrained networks. Unfortunately, we were not able to determine if AWStream performs better than any other video streaming systems for video-based applications, since our deadline arrived before we were able to finish implementing a comparison with WebRTC.

Although AWStream is presented as a library to be used by streaming applications, we found that the current Rust implementation doesn't quite realize this vision. The code base is coupled to the application simulation code used to produce the results for the original paper, and required patches to build and run. Areas for future work include more easily enabling integration with external applications and comparing AWStream with other existing video streaming systems to more concretely measure its benefit to video streaming applications specifically.

7 ACKNOWLEDGEMENTS

We'd like to thank Keith Winstein, Sachin Katti, and Bruce Spang for guidance throughout this project. We're also very grateful to Ben Zhang, who patiently corresponded with us over the course of several weeks to help get AWStream up and running.

REFERENCES

[1] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. 2018. Salsify: low-latency network video through tighter integration between a video codec and a transport protocol. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 267–282.

[2] G. Shepherd L. Eggert, G. Fairhurst. 2017. *UDP Usage Guidelines*. RFC 8085. Internet Engineering Task Force. https://tools.ietf.org/html/rfc8085

[3] Anton Milan, Laura Leal-Taixé, Ian Reid, Stefan Roth, and Konrad Schindler. 2016. MOT16: A Benchmark for Multi-Object Tracking. *arXiv preprint arXiv:1603.00831* (2016). https://motchallenge.net/

[4] Roger Pantos and William May. 2016. HTTP Live Streaming. (2016). https://tools.ietf.org/html/draft-pantos-http-live-streaming-19

[5] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. 2014. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 275–288. http://dl.acm.org/citation.cfm?id=2616448.2616474

[6] H Schulzrinne, S Casner, R Frederick, and V Jaconson. 2006. RTP: A Transport Protocol for Real-Time. (2006).

[7] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. 2018. AWStream: Adaptive Wide-area Streaming Analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 236–252. https://doi.org/10.1145/3230543.3230554