A Proposal for Adaptive Wide-area Streaming Analytics

by

Ben Zhang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Wawrzynek, Chair
Professor Edward A. Lee
Professor Sylvia Ratnasamy
Professor John Chuang

XX XXXX

Abstract

A Proposal for Adaptive Wide-area Streaming Analytics

by

Ben Zhang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

The growing number of Internet-connected devices (sensors and actuators) over the wide-area are challenging how we construct analytical applications. Traditional approaches separate data collection, transportation and distillation as individual one-shot tasks, often performed offline. They are a poor match to the need of acting on data in real time. Recently, the recognition of timely decision-making has spawned many stream processing systems for "big data." However, these systems are primarily tailored towards the infrastructure within a single cluster. In a cluster, while resource allocation is challenging, there are usually enough resources and the allocation is a management problem for maximal utilization. In contrast, the wide-area faces resource scarcity and variability, making it not possible to guarantee enough allocation for applications. Instead, applications have to adapt their behaviors to match the available resources.

While developers can design each individual application to match a specific resource configuration, the ad-hoc solution does not generalize across applications and different data distributions. Besides, optimizations created at design time often don't take the dynamics of the environment into consideration and will behave sub-optimally at runtime.

Recognizing the need for an adaptive behavior across diverse wide-area streaming applications, this manuscript proposes a system-level approach that separates the application logic from adaptation mechanisms. To achieve this goal, I propose a three-stage design framework: (i) a high-level programming abstraction that allows developers to express adaptation options; (ii) an offline profiling tool that learns the resource demand and the impact on application utilities for a specific adaptation strategy—generating an application profile; (iii) a runtime system responsive to environment changes, maximizing application utility according to the learned profile.

The resource and adaptation above are intentionally not specific as it provides a general framework applicable to network resources, computing resources, as well as storage resources. In this proposal, I focus on the adaptation with regards to network resources in the wide-area. Adapting to the heterogeneous computing infrastructures is planned as future work. It will be another part of the final thesis. There is an ongoing research effort (led by my colleagues and I had participated) on storage resources; the final thesis will also briefly discuss the design and some preliminary results.

The bulk body of this proposal focuses on network resources. Specifically, I present AdaptiveStream, a stream processing system for the wide area where the network capacity is scarce and variable. The key observation is the explicit trade-off between application accuracy and bandwidth demand. The system design follows the three-stage framework above: degradation APIs, offline profiling and a runtime system.

Using AdaptiveStream, I have built three real-world applications: a pedestrian detection surveillance application, augmented reality for mobile devices and a distributed top-k. At places where traditional non-adaptive approaches would lead to either significant application accuracy drop or long tail latency, AdaptiveStream gracefully adapts to the network changes, maintaining the balance between application utility and system performance.

Chapter 1

Introduction

Wide-area streaming analytics are becoming pervasive, especially with the emerging Internet-of-Thing (IoT) applications. Large cities such as New York, Beijing and Seattle are deploying millions of cameras for traffic control [39, 49]. Retails stores and critical areas such as railway stations are also being monitored for abnormal activities. Buildings are increasingly instrumented with a wide variety of sensors to improve building energy use, occupant comfort, reliability and maintenance [30]. Geo-distributed infrastructure, such as Content Delivery Network (CDN), needs to analyze user requests (machine logs) and optimize data placement to improve delivery efficiency. In these problems, the data collected at the edge needs to be transported acoross the wide-area and analyzed in real-time.

Existing stream processing for "big data," such as Storm [52] or Spark Streaming [56], only work in the context of a single cluster, where the bandwidth is sufficient (or at least easy to provision). While they are the perfect back end for analyzing streams once data arrive, in the wide-area, the network, with scarce and variable bandwidth, easily becomes the bottleneck [43]. What's worse, the growth rate of wide-area network capacity is not keeping up with the increasing rate of traffic [26].

When facing situations where the bandwidth is not sufficient, applications deployed today either choose a conservative setting (e.g. only delivering 360p videos) or leave their fate to the underlying transport layer: (1) in the case of TCP, the sender will be blocked and data are backlogged, leading to severe delay; (2) in the case of UDP, uncontrolled packet loss occurs, leading to application performance drop. Instead of "suffering" from a degraded network, applications can act proactively by adjusting their behavior: reducing the data rate to ensure that important data are delivered in time.

The idea of adapting data rate for data freshness is not new. Multimedia applications and their adaptations [36, 48] have been extensively studied in the past because their high demands on the network are rarely met. However, since their goal is to improve **user experience**, their adaptation strategies are not necessarily optimal for streaming analytics. For example, humans can tolerate loss on video frame details but expect a smooth frame transition—at least 25 frames per second (FPS). But many vision analytics are based on algorithms that extract edge information from a frame—they will perform poorly if frame details are lost after an aggressive quantization.

The observation that different applications demand different strategies in order to achieve the optimal adaptation calls for a system-level approach. In this proposal, I present the design and implementation of AdaptiveStream, an adaptive stream processing system for the wide-area. Under normal operations, AdaptiveStream applications work with maximal data fidelity; when the network's capacity changes, applications will react by controlling the level of **degradation**: trading data fidelity for freshness.

Although the basic idea behind degradation is intuitive, realizing them in practice is non-trivial. Firstly, as has been discussed earlier, different degradations have different impacts across applications and data distributions. Secondly, each degradation is often more than a binary decision; they can be parameterized with a wide range of tuning space. It's not always possible to derive a close-form analytical relationship between the degradation parameter and its impact on the bandwidth and accuracy. What's more, real-world applications often have multiple dimensions to tune—prohibiting a manual exploration of all the design space. Take video analytics as an example, reducing image resolution, frame rate or changing the video encoding quality are all possible degradation operations. Each operation will affect data rate and application accuracy but its impact to the video quality is not immediately obvious for developers. The situation is worse when multiple degradations are employed. These challenges are elaborated in Section 2.1.

To address the challenges, AdaptiveStream employs a data-driven empirical-analysis approach that's analogous to machine learning. First, AdaptiveStream separates application logic from adaptation strategies by providing a clean abstraction (`maybe` APIs) for developers: this allows expressibility without burdening the developers for an exact strategy. AdaptiveStream then profiles the application using representative datasets and application-specific utility functions to "learn" the optimal strategies under different network conditions. The profile is then used to guide the runtime adaptation, for which AdaptiveStream provides all the necessary "plumbing" modules to without application developers' effort. In Section 2.2, I will present a detailed description of AdaptiveStream's architecture.

To study and validate the effect of degradation, I've built a prototype framework and three real-world applications using AdaptiveStream: a street surveillance application performing pedestrian detection, an augmented reality application recognizing everyday objects and a distributed top-k application analyzing web server access logs. The first two video streaming applications have three degradation operations: reducing resolution, frame rate and video encoding quality. For the distributed top-k application, two degradation operations are envolved: N in a local top-N operation and T as a local threshold. Section 3.2 discusses the considerations and details about the prototype.

The evaluation shows that the offline profiling is able to explore the design space for all three applications and generate the Pareto-optimal adaptation strategies that was not achievable with manual developer configurations or with only a single degradation operation. These profiles offer a quantitative understanding of each application's behavior. During runtime, AdaptiveStream applications are compared against traditional transport protocols including TCP and UDP with a controlled experiment. When the network capacity drops, TCP creates a severe delay; and the delay increases linearly as time goes; for UDP, a severe packet loss makes the application unusable. Applications built with AdaptiveStream handles the network variation gracefully: video streaming analytics is able to keep the latency bounded with 2 seconds and the accuracy higher than

80%. For the top-k application, although the feedback is less frequent and adaptation is not always immediate, the worst-case latency is 12 seconds at most and the accuracy is mostly above 75%.

To this end, this thesis makes the following contributions:

- An in-depth study of wide-area streaming applications in the case of network resource variation.
- The proposal of novel APIs to allow for a design space exploration between bandwidth and accuracy with minimal developer effort.
- A prototype implementation that demonstrates application profiling and runtime adaptation.
- Thorough evaluations based on three real-world applications under different scenarios.

1.1 Motivating Applications

There is a variety of wide-area streaing applications. We discuss three applications with their implications.

Video Surveillance: We envisage a city-wide monitoring system that aggregates camera feeds (both stationary ground cameras and moving aerial vehicles) and analyzes video streams in real-time for surveillance, anomaly detection or business intelligence [38]. While traditionally human labors are involved in analyzing abnormal activities, recent advances in computer vision and deep learning has dramatically increased the accuracy for automatic analysis of visual scenes, such as pedestrian detection [21], vehicle tracking [18], or facial recognition to locate people of interest [34, 41].

Electrical Grid Monitoring: While traditional environmental sensors are slow [9], we are seeing an increasing trend with high-frequency, high-precision sensors being deployed. For example, the microsynchophasors monitoring system for the electrical grid consists of a network of 1000 devices; each produces 12 streams of 120 Hz high-precision values accurate to 100 ns. This amounts to 1.4 million points per second that requires specialized timeseries database [8].

Log Analysis: Large organizations today are managing 10–100s of datacenters (DCs) and edge clusters worldwide [13]. While most log analysis today runs in a batch mode and at most on a daily basis, there is a trend in analyzing logs in real time for quicker optimization [6]. For example, by analyzing the access logs in real time, a content distribution network (CDN) can improve the overall delivery efficiency with optimized data placement.

These applications share a similar structure: a large volume of data generated at the edge need to be transported across the wide area network for real time analysis. Because of the limited network resources in the wide area, they will face practical issues when deployed at a scale.

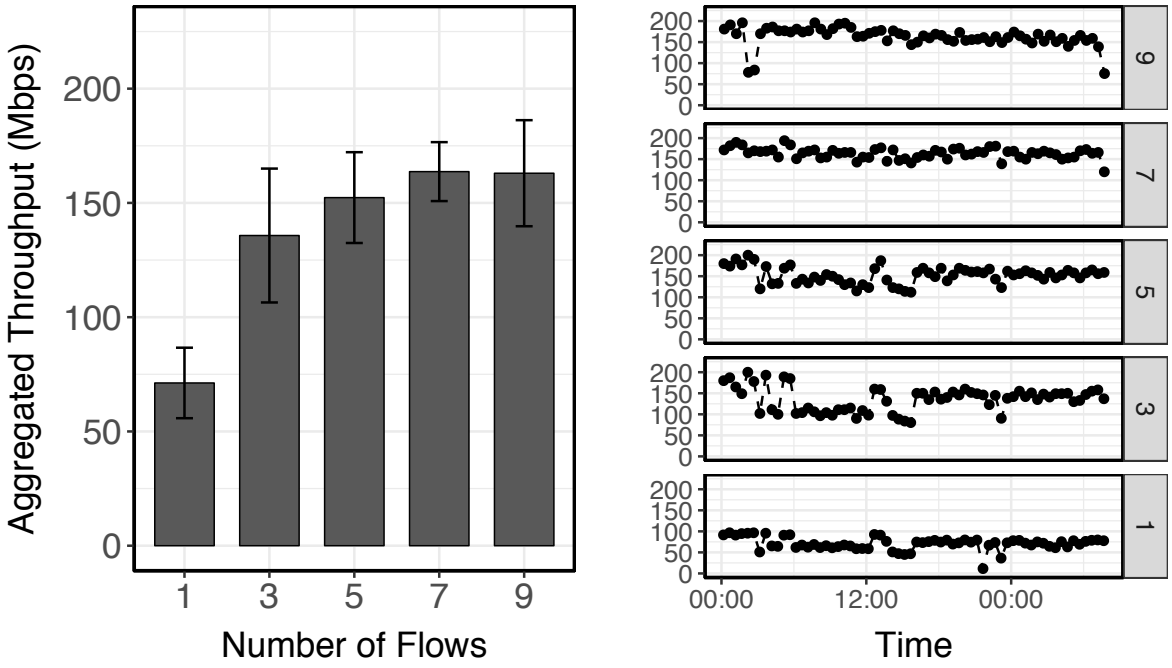

Figure 1.1: Bandwidth measurement between Amazon EC2 sites (from Ireland to California). Note the time-series plot has a resolution of 30 minutes; each point is more than a short period of transient degradation.

1.2 Wide-Area Bandwidth Characteristics

To understand the bandwidth characteristics in the wide-area, I conducted a simple measurement using Amazon EC2. iPerf [27] was used to measure the pair-wise bandwidth between four geo-distributed sites throughout the day. The measurement shows large variance in the measured bandwidth and one such pair of sites is shown in Fig. 1.1. Regardless of the number of flows[1], there exist occasions when the available bandwidth is almost halved. Generally speaking, the back-haul links between EC2 sites are better (if not at least representative) in comparison to the overall wide-area link quality. This varying nature poses real challenges to the realization and successful deployment of wide-area streaming applications.

1.3 Making the Case for a System Approach

When facing insufficient network bandwidth, applications that do not adapt will suffer severe performance penalty: such as a backlog of data for TCP or uncontrolled packet loss for UDP. Instead of giving up control to the underlying transport layer, applications can react and adapt their behaivors to the resource changes.

[1] EC2 has a per-flow and per-VM rate limiting [58].

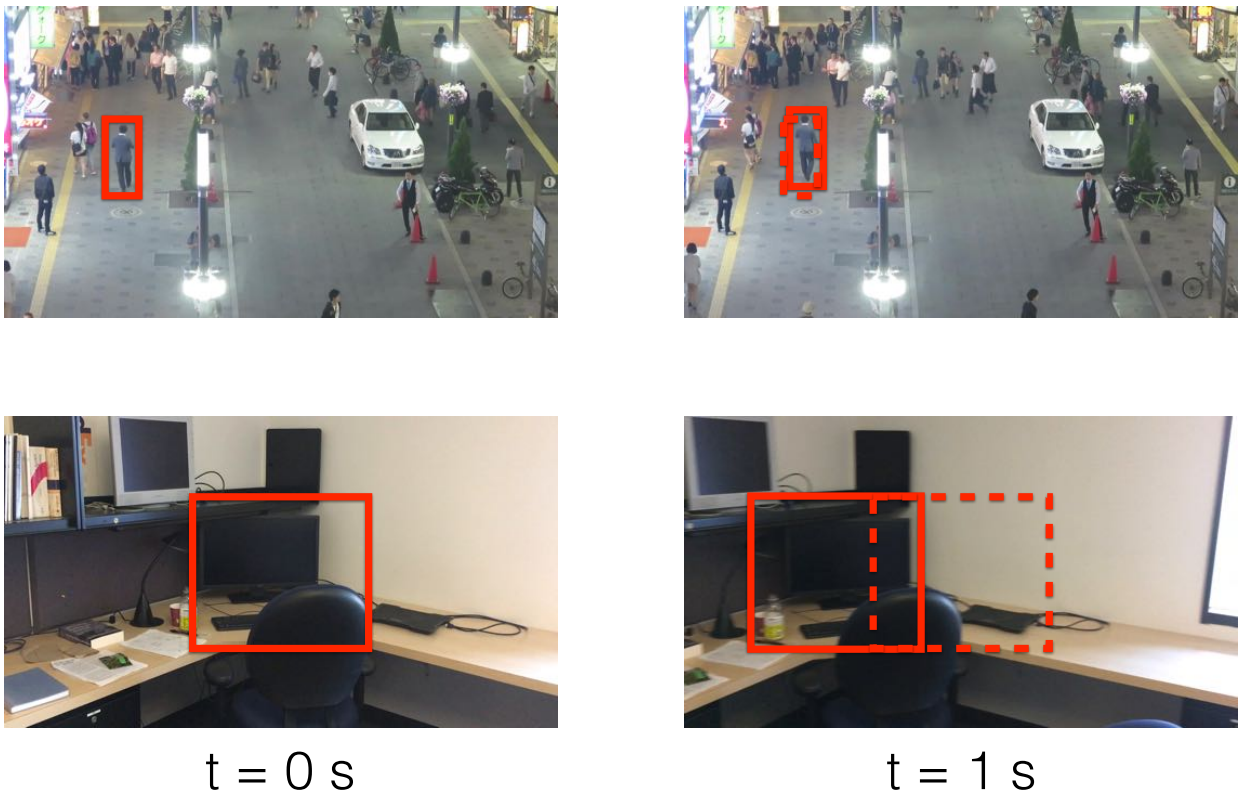

Figure 1.2: The frame difference between two images with one second difference. These are two different deployment scenarios: a stationary camera in a far field (upper) and a mobile camera for nearby objects (lower). The solid rectangle in each scene is the detection target; the dotted rectangle on the right side mirrors the detection on the left side.

While adaptive streaming exists in certain application domains, there has not been a general solution. Consider video streaming applications that have been extensively studied in the literature. There are a plethora of encoding techniques [45, 24] with adaptive strategies [55, 36, 40], however, their primary goal is to optimize end-user quality of experience (QoE). When end users consume a video clip, a smooth video is often more enjoyable than videos with intermittent pauses, even though each pause has crisp images.

Optimizing for QoE doesn't work for our target applications. Each video analytics has its own goal, entailing different adaptive strategies for different applications. For example, some computer vision detection algorithms rely on the edge information [14, 33, 53] while object tracking applications works best when the inter-frame difference is small [5].

Furthermore, even similar applications, when used in different context, require different strategies. Fig. 1.2 offers an example. The pedestrian detection is deployed on a ground stationary camera in a far-field view. When taking pedestrian walking speed into consideration, there is so little difference between frames that it's not necessary to guarantee a high frame rate. But because the camera is far from the targets, it's crucial to have a high-resolution and sharp image. On the

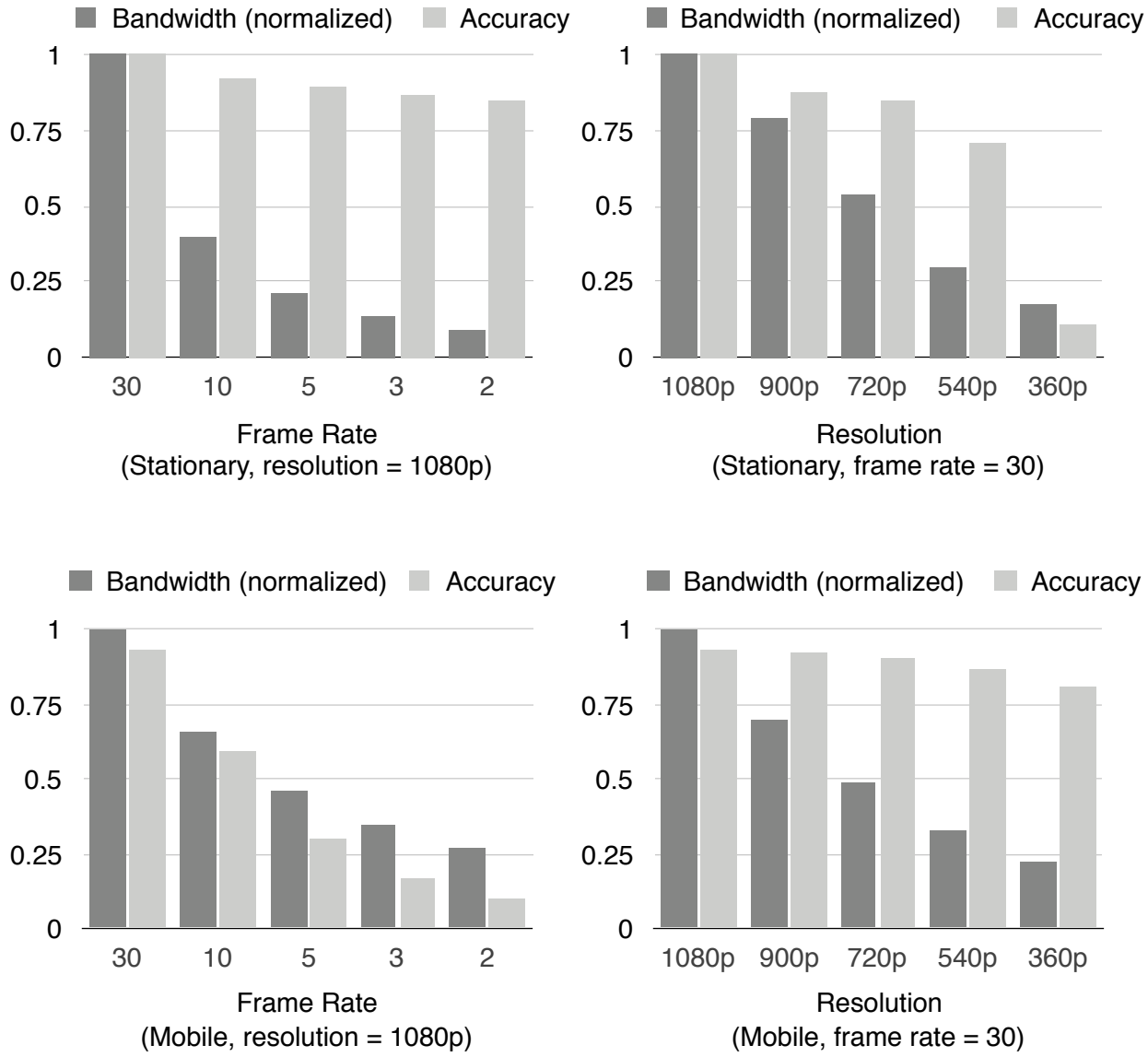

Figure 1.3: Horizontally, how reducing the frame rate or resolution affects the bandwidth requirement and appliation accuracy. Vertically, how the same degradation have different impact for different data characteristics; the upper figures are for a stationary camera deployment while the lower figures are for mobile applications.

other hand, object recognition on a mobile phone captures nearby objects. Due to the movement of the camera, reducing frame rate will introduce significant errors. Fig. 1.3 shows the different rate in accuracy drop when image resolution or frame rate is reduced for the two use cases.

This motivates us to take a system-level approach that synthesizes different adaptation strategies for different queries and contexts.

1.4 Related Work

Stream processing systems: Streaming databases, such as Borealis [1], TelegraphCQ [16], are the early academic explorations. They pioneered the usage of dataflow models with specialized operators for stream processing. Recent research projects and open-source systems, such as MillWheel [4], Storm [52], Heron [32], Spark Streaming [56], Apache Flink [15], primary focus on fault-tolerant streaming in the context of a single cluster. While this thesis has a large debt to the prior streaming work, AdaptiveStream is designed for the wide-area and explicitly trades data fidelity for data freshness; many other stream processing systems choose to throttle the source when backpressure happens [32].

WAN-aware: There is a growing interest in building systems that optimizes data transfers for the wide area, such as GDA [42], Clarinet [54] and OWAN [28]. Most of these works focus on one-time queries or operations (such as data transfer). JetStream [43] is the first that studies streaming analytics in wide area and proposes to use structured storage (data cubes) and explicit degradation policies. While JetStream has demonstrated application responsiveness with hand-written degradation policy, these policies are often developer heuristics that are not backed up by measurements. This thesis extends the idea of degradation with an automatic policy synthesis.

Approximate analytics: The idea of degrading computation fidelity for responsiveness has also been explored in other contexts, primarily SQL queries. Online aggregation [25], BlinkDB [3] and GRASS [7] speed up queries with partial data based on a statistical model of SQL operators. AdaptiveStream is different from these approximatte analytics as it supports arbitrary data processing pipelines where no close-form solution exists to evalute the impact of a particular degradation.

Adaptive video streaming: This is both an active research topic [50, 55] with many trending industrial efforts. Because they target at video delivery for web applications, many have chosen to tune HTTP protocols for video adaptation, such as HLS [40] by Apple and DASH [36] as the new standard. The main technique is to adjust the video resolution and encoding bitrate but they often guarantee a smooth video with high frame rate (at least 25 FPS). This thesis generalizes the adaptation to a wider range of streaming analytics and allows more custom control over what parameters can be adjusted. AdaptiveStream applications utilize existing techniques from adaptive video streaming (such as H.264 [45] and VP9 [24]) instead of reinventing the wheel.

Chapter 2

AdaptiveStream Design

In this chapter, I present the design of AdaptiveStream. The primary goal of AdaptiveStream is to empower applications with the ability to adapt its communication with a maximal utility.

2.1 Challenges

There are four challenges in realizing AdaptiveStream.

C1: Diverse application and data: As discussed in Section 1.3, the best adaptation scheme is often application- and context-specific optimization. It becomes important to separate individual application logic from specific degradation strategy as well as the concrete mechanisms.

C2: No analytical solutions: Unlike SQL queries whose demand and accuracy can typically be estimated using analytical models [19], many of our streaming applications are dealing with unstructure data using either blackbox operations (such as H.264 encoding) or non-linear operators (such as thresholding). The effect of these degradations is not immediately available.

C3: Multi-dimensional exploration: Real-world applications typically have more than one tunable parameters; leading to a combinatorial space for exploration. In addition, these parameters are not necessarily orthognal. The optimal degradation strategies may only be achievable when more than one degradation is in effect.

C4: Runtime adaptation at application layer: Although recent work on resource reservation makes it possible to guarantee quality of service with new IP or MAC layer protocols in LAN (e.g. TSN [29]), we target at WAN analytics where most of the infrastructure is owned by others and shared among many users. An application-layer solution is in favor to those that require special hardware or software upgrade.

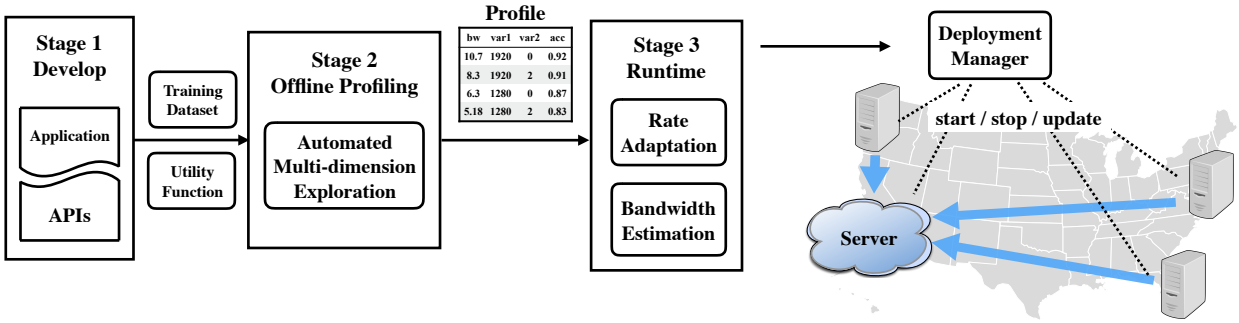

Figure 2.1: The three-stage architecture of AdaptiveStream.

2.2 System Architecture

To address the aforementioned challenges, AdaptiveStream's solution is split into three parts (Fig. 2.1 illustrates them):

Programming abstraction (Section 2.3): Applications are modelled as a directed acyclic graph (DAG) of computations. In addition to the normal operators from existing systems, I propose a novel set of `maybe` operators to express the specification of degradations. The proposed APIs do not require developers to be exact on the quantity, making effort to integrate with existing applications minimal.

Automatic multi-dimensional profiling (Section 2.4): The system automatically explores the parameter space to learn a Pareto-optimal degradation strategy in a application-specific manner. This process frees application developers from a tedious and repetitive process.

Runtime Adaptation (Section 2.5): Finally the streaming application is deployed with a wide-area orchestration manager. At runtime, AdaptiveStream provides all the necessary modules that act as the control plane and adapt the application execution. At a high level, it performs bandwidth estimation, congestion monitoring and adaptation. It uses the profile learned from the second stage to guide the level of degradation.

2.3 Programming Abstraction

Applications in AdaptiveStream are composed by connecting a set of operators to form a dataflow graph. The system provides basic APIs such as `map`, `filter`, `window` (Table 2.1) that are similar to existing stream processing systems. The core contribution of this thesis is the `maybe` APIs that allow the specification of degradation operations for bandwidth-accuracy trade-off.

Normal Operators	*map*(f: I ⇒ O)	Stream<I> ⇒ Stream<O>
	filter(f: I ⇒ bool)	Stream<I> ⇒ Stream<I>
	skip(i: Int)	Stream<I> ⇒ Stream<I>
	sliding_window(count: Int, f: Vec<I> ⇒ O)	Stream<I> ⇒ Stream<O>
	tumbling_window(count: Int, f: Vec<I> ⇒ O)	Stream<I> ⇒ Stream<O>
	timed_window(time: Duration, f: Vec<I> ⇒ O)	Stream<I> ⇒ Stream<O>
	...	...
Degradation Operators	*maybe*(knobs: Vec<T>, f: (T, I) ⇒ I)	Stream<I> ⇒ Stream<I>
	maybe_skip(knobs: Vec<T>)	Stream<I> ⇒ Stream<I>
	maybe_downsample(knobs: Vec<(Int, Int)>)	Stream<Image> ⇒ Stream<Image>
	...	...

Table 2.1: comparison between normal stream processing operators and our degradation operators. Vec<T> represents a list of elements of type T. Notice the type constrain on the second argument passed to `maybe`.

Degradation Operators

To design the degradation operator, let's first consider a strawman solution: manual policies for degradation. JetStream [43] offers an example: "if bandwidth is insufficient, switch to sending images at 75% fidelity, then 50% if there still isn't enough bandwidth. Beyond that point, reduce the frame rate, but keep the images at 50% fidelity." This manual policy specification has the following issues:

Lack of precision: These policies are developer heuristics and rarely backed up by measurements. First, there is no direct association of the application accuracy with the 75% fidelity configuration. Besides, the effect of each rule on the data size is not trivially available. While it seems intuitive that the level of degradation will change the data size, the precise effect is not always straightforward. For example, one might think that reducing the frame rate by 50% will half the data rate. When video encoding is employed, the inter-frame difference will increased (P-frame size) when the frame rate is reduced. This leads to a larger data size for each frame. Fig. 2.2 illustrates this complex relationship with an example of H.264 encoding under four different frame rates.

Not scalable: The strawman solution quickly leads to too many policies when multiple degradation operations are involved or a fine-grained control is desired. This manual process becomes tedious and error-prone. When too few rules are provided, the application may oscillate between two rules: one that's too aggressive (always faces insufficient bandwidth) and one that's too conservative (a suboptimal strategy).

I argue that when using the above strawman solution, developers are forced to manually study and measure the effect of individual degradation policy, prohibiting its wide adoption in practice.

On the other extreme of the design spectrum lies a completely developer-free solution. It is not practical, either. While static analysis has been shown to optimize application execution adaptively

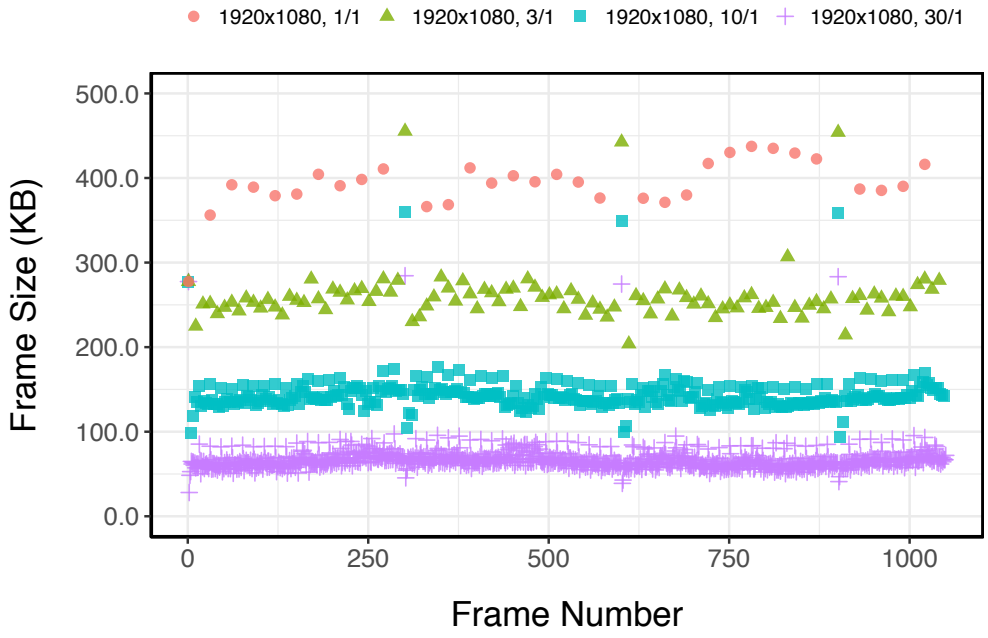

Figure 2.2: H.264 requires more information per frame when the frame rate is reduced. The horizontal axis is the frame number with a video clip; the verticle axis is their size after H.264 encoding. All measurements are for videos with 1920x1080 resolution and the same H.264 configuration. 1/1, 3/1, 10/1 and 30/1 in the legend mean that the frame rates are 1 FPS, 3 FPS, 10 FPS and 30 FPS. While 1 FPS will have one-third total points in comparison to 3 FPS, the frame size is much larger because the frame difference increases.

in a mobile-cloud context [17], it only concerns with computation placement (on the mobile or in the cloud) rather than tuning application parameters. For our dataflow programming model, static analysis is prone to false positives: exploring wrong or unnecessary parameters. For example, when the application is configured to generates statistics with a `timed_window` operation, static analysis may falsely detect the duration parameter and alter the behavior of the application in an unexpected way. Also, as we will illustrate in Section 2.4, with each introduced parameter, the profiling time increases drastically as all parameters pose a combinatorial space. The explosion of search space prohibits explorations of unnecessary parameters.

AdaptiveStream takes a middle ground between these two extremes: developers use a novel `maybe` API to annotate degradation operations without being exact on the values. Think of these APIs as hints from developers: this operation, when in use, will likely reduce the data size and affect the data fidelity; however the exact quantity is not clear.

The basic form of `maybe` operator takes two arguments: a knob and a degradation function (see Table 2.1). The knob indicates different degradation levels; the function performs the actual degradation operation with a configurable level. There is one restriction on the type signature of the function argument: $f(T, I) \Rightarrow I$. That is, the degradation function should not alter the type of

the stream. While this seems a strong restriction, when combined with the `map` operator, the API set is still expressive enough. I describe the implementation and usage of the APIs in Chapter 3.

Based on the `maybe` primitive, one can implement wrappers for common degradation operations. For example, `maybe_skip` will optionally subsample a stream; `maybe_downsample` can adjust the image resolution to a configured target. Using the `maybe` API, the example mentioned earlier can now be implemented as follows:

```
let app = Camera::new((1920, 1080, 30))
  .maybe_downsample(vec![(1600, 900), (1280, 720)])
  .maybe_skip(vec![2, 5])
  .map(|frame| frame.show())
  .compose();
```

This snippet first instantiates a `Camera` source, which has the type `Stream<Image>`. It's configured to produce images with 1920x1080 resolution and 30 FPS. Two degradation operations are chained after the source: one that downsamples the resolution to either 1600x900 or 1280x720; the other skips frames with a parameter of 2 or 5, resulting in $30/(2+1) = 10$ FPS or $30/(5+1) = 5$ FPS. After the degradation, images are shown on the display; in practice, further processing operators can be chained after the degradation.

While the API itself has simplified the specification of degradation, the exact amount has to be known for precise rate adjustment at runtime. We then turn to the second stage of our system that performs the automatic profiling.

2.4 Offline Profiling

The goal of the profiling stage is to explore the bandwidth-accuracy trade-off and learn a Pareto-optimal *profile* for a specific application and its target scenario. The profile consists a set of rules that determines how each degradation operations will configured; each rule is a *(bandwidth, configuration)* tuple.

Before we discuss properties of profile, we define the terms and notations (Table 2.2). Each `maybe` operator within an application corresponds to a knob k. Suppose the application has n knobs, their combination forms a configuration $c = [k_1, k_2, ...k_n]$. The set of all configurations $\mathbb{C}$ is the space that the profiling system need to explore.

There are two mappings that we are interested: a mapping from a particular configuration to its bandwidth requirement $B(c)$ and the accuracy measure $A(c)$. The Pareto-optimal set $\mathbb{P}$ can then be defined (Eq. 2.1): for all $c \in \mathbb{P}$, there is no alternative configuration c' that requires less bandwidth while giving a higher accuracy.

$$\mathbb{P} = \{c \in \mathbb{C} : \{c' \in \mathbb{C} : B(c') < B(c), A(c') > A(c)\} = \varnothing\} \tag{2.1}$$

Since there is often no close-form relation for $B(c)$ and $A(c)$, for arbitrary degradation operations, AdaptiveStream takes a data-driven approach. With a representative dataset and an application-specific utility function, the system evaluates each configuration for the bandwidth

Symbol	Description
n	number of degradation operations
k_i	the i-th degradation knob
$c = [k_1, k_2, ...k_n]$	one specific configuration
$\mathbb{C}$	the set of all configurations
$B(c)$	bandwidth requirement for c
$A(c)$	accuracy measure for c
$\mathbb{P}$	Pareto-optimal set

Table 2.2: Notations used in profiling.

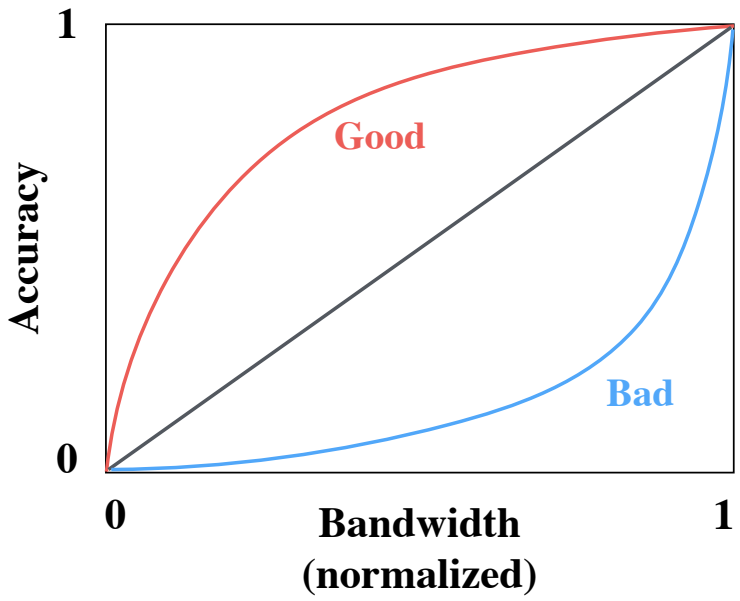

Figure 2.3: Illustration on the behavior of different degradation operations.

demand and the application utility. The utility could either be measured against the groundtruth; or in the case when labelled dataset is not available, the system uses the reference results when all degradations are turned off.

When only one knob is involved, we can represent its impact using bandwidth-accuracy curve (Fig. 2.3 left). The point on the curve has a coordinate $(B(c), A(c))$ for configuration c. The straight line from (0, 0) to (1, 1) splits the space into two parts. Along this line, the amount of bandwidth saving leads to an equal amount of accuracy drop (when both quantities are normalized to 1). If a degradation's profile lies above this line, it indicates an effective degradation that should be employed as it offers little accuracy drop with more bandwidth savings. For degradation operations below the straight line, data information loss is more severe than the savings when the degradation is in effect. One way to capture the relative power of different degradation strategy is

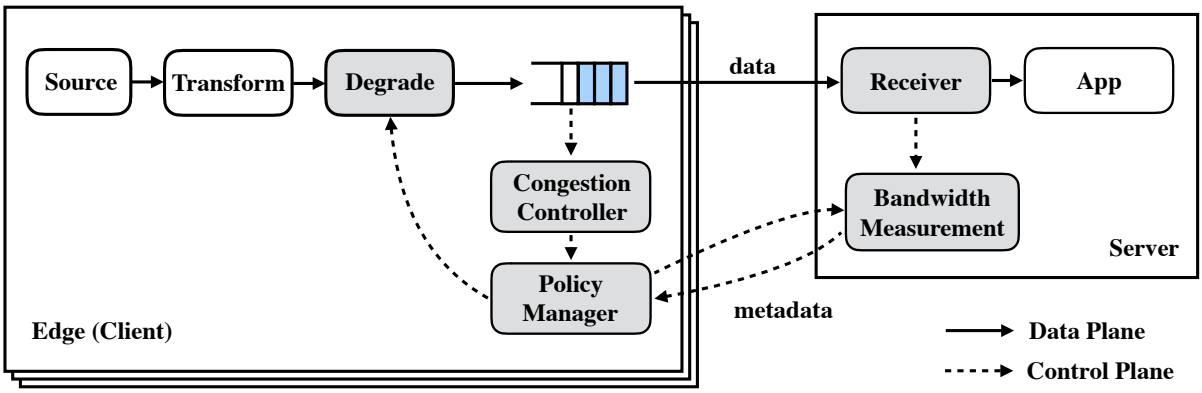

Figure 2.4: Runtime adaptation system architecture. AdaptiveStream's provided components are grey: they form the control plane to compensate the application's data plane.

to use the area under the curve (AUC). With this setup, the Pareto-optimal strategy can be define as the curve that has a maximal AUC. We will show some concrete profile curves in Section 3.3. While $B(c)$ and $A(c)$ are usually monotonic along individual dimension k_i, when multiple degradations combined, each curve may have a distinct shape.

Although the complexity of degradation operations prohibits further optimizations to reduce the profiling time, there are several techniques in practice that can make the problem tractable: (i) Exploring these configurations is an offline task; there is no direct dependencies among configurations, resulting in an embarrassingly parallel task. Executed in a cluster, it scales out perfectly with provided compute resources. (ii) When the degradation level increases or when multiple degradation operations are in effect, the amount of data to be analyzed becomes smaller; also the computational complexity for each data item can also be dramatically reduced. (iii) Developers can specify a lower bound on the accuracy and the profiling can stop exploring worse configurations. Once there is a known configuration that is worse than the configured threshold, configurations with a larger level of degradation do not need to be profiled.

2.5 Runtime Adaptation

At runtime, AdaptiveStream provides the necessary modules for an automatic adaptation. Fig. 2.4 shows the architecture of AdaptiveStream runtime. Applications are split into a client half and server half and they interact with each other under the control of AdaptiveStream. We detail the functionality of each module as follows:

Object-level queue: The queue bridges data generation from the application and handles its mismatch with the network resource. When the network capacity is sufficient, the queue will at most have one object in the process of being transmitted. Network congestion creates backlogged data

that are placed inside this queue. The queue is being monitored by the congestion controller to detect congestion.

Congestion controller: Whenever a new object enqueues, the congestion controller keeps track of the number of objects as well as the queued data size (in bytes). Using the queue length alone is not enough to determine congestion status because some degradation operations (such as `maybe_skip`) will change the rate of data generation. Using the data size alone also doesn't work because degradation operations such as `maybe_downsamples` will alter the data size of individual object. In AdaptiveStream design, we've adopted an approach with an adaptive size for congestion signaling: developers configure a tolerance on the communication delay and the threshold is derived based on the the tolerance and current configuration.

Bandwidth Measurement: The receiver delivers the received data from the network to the application. In the meantime, it measures the effective throughput between each client-server pair as an indication of current available bandwidth [27]. To avoid spikes in the bandwidth measurement, exponential smoothing is employed. The receiver performs bandwidth estimation every second; but it does not send the information back to each client. In this way, we avoid unnecessary communication overhead. When the client detects congestion, it will query the latency with a remote procedure call (RPC).

Policy Manager: Upon receiving signals from congestion controller, it performs an RPC request to the server for current bandwidth measurement. Using the measurement, together with the learned profile from the last stage, it determines the degradation level. In the case of congestion, the policy manager will take a conservative approach: using a constant modifier that's smaller than one to adjust the available bandwidth. On the other hand, when the congestion is resolved, the policy manager gradually reduce the degradation level. This is similar to the additive increase phase in TCP congestion control.

Degrade: Achieving the actual degradation operation is in fact quite simple. Operators based on the `maybe` APIs support a `set` function that would change the internals of the operator. The control plane will invoke the `set` function with the appropriate parameter to adjust the degradation level.

Chapter 3

Implementation and Evaluation

In this chapter, I will present a prototype system that realizes AdaptiveStream design and three non-trivial wide-area streaming applications with evaluations.

3.1 Framework

The proposed `maybe` APIs are not language specific. Although the prototype is built on top of Rust, it can be easily implemented in other existing frameworks. I chose Rust mainly because of the memory safety guarantee and the efficient runtime. First, Rust's memory safety guarantee ensures applications to run continuously for an extended period of time. This eliminates a large body of bugs caused by memory corruption; these bugs often require careful programming analysis for traditional system programming languages such as C or C++. Besides, the zero-cost abstraction offers runtime efficiency. It removes the possibilities of tail latency that will happen in garbage-collection (GC) languages, especially when uncoordinated GC kicks in [35]. There are some additional features about Rust that made the prototype appealing, such as the type enforcement on the function argument passed to the `maybe` APIs.

All operators implement the `Stream` trait that has an associate type `Item` and a core function `next`. The `next` function, when invoked, will return a `Datum` that is a union type of `Stream::Item` or `Error`. Imagine a source operator that implements the `Stream` trait, applications can keep calling `next` function and will receive the data under normal conditions. When an error occur, the `next` function will return an `Error` that contains the cause of the error.

The `maybe` function is one operator that transforms a stream (the first argument `self`) into an object of `Maybe` type. The `self` argument means that this function can be invoked on an object of this type. Imagine a data structure that implements the `Stream` trait and we have an instantiated object called `foo`, then you can use `foo.maybe(...)` where `foo` will be the `self`.

Two other arguments taken by `maybe` are almost a direct translation of the API specification in the design (Table 2.1). While the design uses a vector for knobs, the Rust implementation is more general: any type (including vector) that implements `IntoKnob` trait can be used as the knob. The function argument here is `FnMut`, which is one type of the call operators (often a closure) that

is allowed to have side effects—it takes mutable references to the closed environment. Interested readers should refer to Rust documentations for Rust closures.

```
pub trait Stream {
   type Item;
   fn next(&mut self) -> Datum<Self::Item, Error>;

   fn maybe<K, F>(self, opts: K, f: F) -> Maybe<Self, F>
      where Self: Sized,
            K: IntoKnob,
            F: FnMut(K::Item, Self::Item) -> Self::Item {

       // omitted
   }
}

pub trait IntoKnob {
   fn into_knob(self) -> Knob;
}
```

Developers can then use the above API with arbitrary user-defined functions.

```
let quantized_stream = vec![1, 2, 3, 4]
   .into_stream()
   .maybe(vec![2, 3], |knob, d| d / knob);
   .collect();
```

The snippet above shows how a quantization degradation can be implemented. First, a vector of values is converted into a stream object. Then a `maybe` operator is chained after the stream; it's configured with a knob value (2 or 3) and a function (closure in this example) that does an integer division (quantization). The function `collect` will run this stream and output the result as a vector. In this example, depending on the degradation level, the output stream could either be [1, 2, 3, 4] (no degradation), [0, 1, 1, 2] (quantized by 2), or [0, 0, 1, 1] (quantized by 3).

In the implementation, we've also extended the basic API set for common operations. For building video processing applications, we've built a specialized `maybe_downsample` operator can that wraps `downsample` function:

```
fn downsample(res:  (usize, usize), image:  Mat) -> Mat;
```

In our current prototype, an application built with AdaptiveStream runs as a single process and the entire processing pipeline—without explicit separation of client and server—is often specified in a single main file. The execution mode (profiling, runtime as client or runtime as server) is configured with command line arguments or environment variables. Our deployment manager is currently a shell script using Docker container.

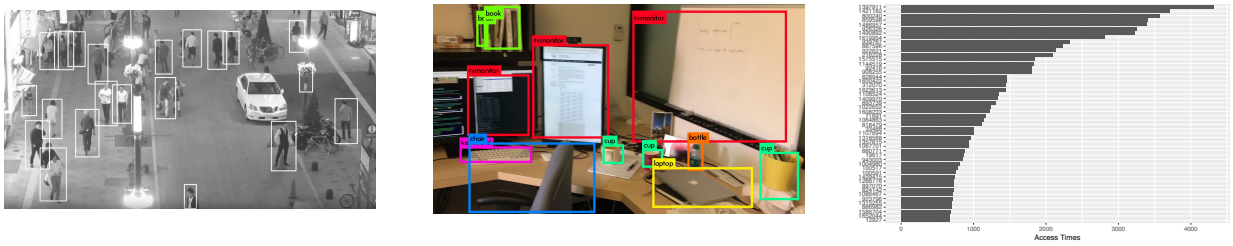

Figure 3.1: AdaptiveStream applications

Application	Knobs	Utility	Dataset
Pedestrian Detection	resolution, framerate, quantizer	F1 score	MOT16-04 (training) MOT16-03 (testing)
Augmented Reality	resolution, framerate, quantizer	F1 score	Video clips with iPhone office (training) home (testing)
Top-K	head (N), local threshold (T)	Kendall's W	`https://sec.gov` access log 4 days (training) 12 days (testing)

Table 3.1: AdaptiveStream Applications

3.2 Building AdaptiveStream Applications

Using AdaptiveStream, we've built three applications: pedestrian detection surveillance, an augmented reality and a distributed Top-k.

Fig. 3.1 shows an illustration of these applications and the application-specific part, including knobs, utility function and data set, is in Table 3.1. Below we describe each application in details.

Pedestrian detection: This application analyzes video streams from installed CCTV cameras and detect pedestrians inside. The detection result is a list of bounding boxes (rectangles) representing pedestrian's relative location within the view. Variant of this application can be used for safety monitoring, anomaly detection or waiting line counting.

We implement most of the image-related operations with OpenCV 3.1 [12]. Pedestrians are detected using histogram of oriented gradients (HOG) [20] with the default linear SVM classifier. To ensure real-time processing of frames, GPU-accelerated implementation is used in favor of the CPU-based implementation.

For video encoding, H.264 scheme is chosen for its prevalence in existing systems. Our implementation is based on GStreamer [51], using `x264enc` plugin. To integrate with AdaptiveStream, we first create a pipeline that exposes `appsrc` (to feed raw image data) and `appsink` (to get encoded bytes). The GStreamer main loop is managed in a separate thread and AdaptiveStream communicates with it via Rust's channel. The `x264enc` is configured with `zerolatency` present

and runs using four threads. It uses constant quality encoding and the quantizer is exported as a parameter that can be tuned: increasing the quantizer will degrade the image quality.

Overall, this application has three degradation operations: reducing image resolution, dropping frame rate or lower video encoding quality by adjusting the quantizer.

For application accuracy measure, the detection results over degraded streams are compared against a reference result where no degradation is in effect. A successful detection is defined when the intersection over union (IOU), between the detection and the reference results, is greater than 50% [23]. All the detection are combined and we use use F1 score (%) as the final accuracy. It represents the harmonic mean of precision and recall, and has been used commonly in computer vision tasks [46].

Augmented Reality: We target at mobile augmented reality applications which offload the heavy computation to resources elsewhere. Although local computation is gaining attraction [47, 57], wireless communication link is also susceptible to capacity variation; our adaptation techniques can be applied to the wireless domain.

We use a similar setup (OpenCV + GStreamer) as the pedestrian detection application except the actual function that analyzes the stream. To recognize objects, a pre-trained neural network model [44] is used. The model has been trained with ImageNet [31] and is able to recognize everyday objects. Similar to our first application, we use the GPU-accelerated implementation for real-time processing.

Although the utility function here is also F1 score, the criterion for a successful detection is more strict: true-positive depends not only on the IOU criteria, but also the type of objects must be matched.

Distributed Top-K: Many distributed system monitoring applications require to answer the "top-k" question [10], such as the top-k most popular URLs or the top-k most access files. Naive methods of transmitting all the raw log entries to the aggregation point is not feasible as popular servers typically have millions of requests per second. Local worker nodes can first perform a window-based transformation that generates data summary, such as key-value pairs of `<item, count>`. However, even after this operation, the data size could still be too large given most real-world access patterns follow a long-tail distribution. There is a large-but-irrelevant tail that is unnecessary to send.

We consider two degradation operations that individual worker node can perform: (1) a local Top-`N` operation that shortens the list first; (2) a local threshold `T` that further filters small entries. Obviously, these two operations are not orthogonal to each other. Their impact on data size reduction and quality degradation depends on the distribution of the actual data. Fig. 3.2 illustrates one concrete example of the entire pipeline and the effect of two degradations. The final results are evaluated using Kendall's W. The Kendall's W is a distance measure of the concordance between two ranked list. It outputs a statistic measure ranging from 0 to 1, representing no agreement to complete agreement, respectively [2].

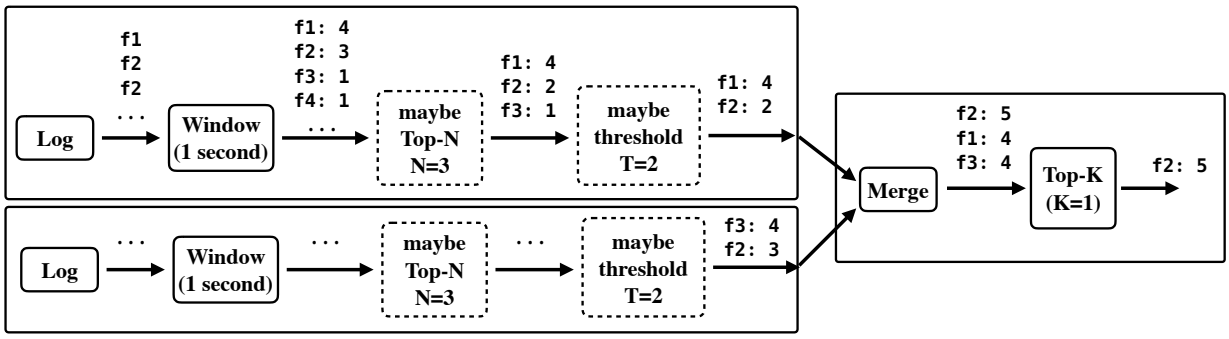

Figure 3.2: A distributed Top-K application has two tunable parameters: a local Top-N (N) and a local threshold (T).

3.3 Evaluation

In this section, we first show the learned profile for each application. These profiles serve as empirical evidence to back up the system design. Then for each application, We show how they adapt the behavior at runtime. Under a controlled experiment, even with only transient network capacity drop, our system is able to maintain an end-to-end delay for 10 seconds in the wide-area and accuracy level above 80%. Application-agnostic protocols create significant backlogged data (TCP for about 100 seconds) or unusable accuracy (UDP).

Degradation Performance

We describe the dataset we used for each application and then discuss what we've learned from each result.

Pedestrian Detection: We use MOT16 dataset [37] to evaluate this application. Specifically we used MOT16-04 as the training dataset. The video feeds capture a busy pedestrian street at night with an elevated viewpoint. The original resolution is 1920x1080, with frame rate 30. The training dataset has 1050 frames in total, amounting to 35-second monitoring. On averge there are 45.3 people per frame.

There are three knobs in this application: resolution, frame rate and encoding quality. To maintain the same 16:9 aspect ratio with the original 1920x1080 resolution, the resolution degradation only uses common 16:9 settings: 1600x900, 1280x720, 960x540 and 640x320. For the framerate, integer values are chosen in favor of fraction values. The original frame rate is 30, and our degradation explores 10, 5, 3, 2, 1. To adjust the video encoding quality of H.264, we vary the encoding quantizer. The quantizer has a range from 0 (lossless) to 51 (worst possible), and just for reference, 18 is the visually lossless setting [11]. In our experiment, we use 10, 20, 30, 40, 50 as degradation parameters.

The generated profile is shown in Fig. 3.3 with x-axis being the required bandwidth and the y-axis being the accuracy (F1 score). Each point in the scatter plot represents one configuration

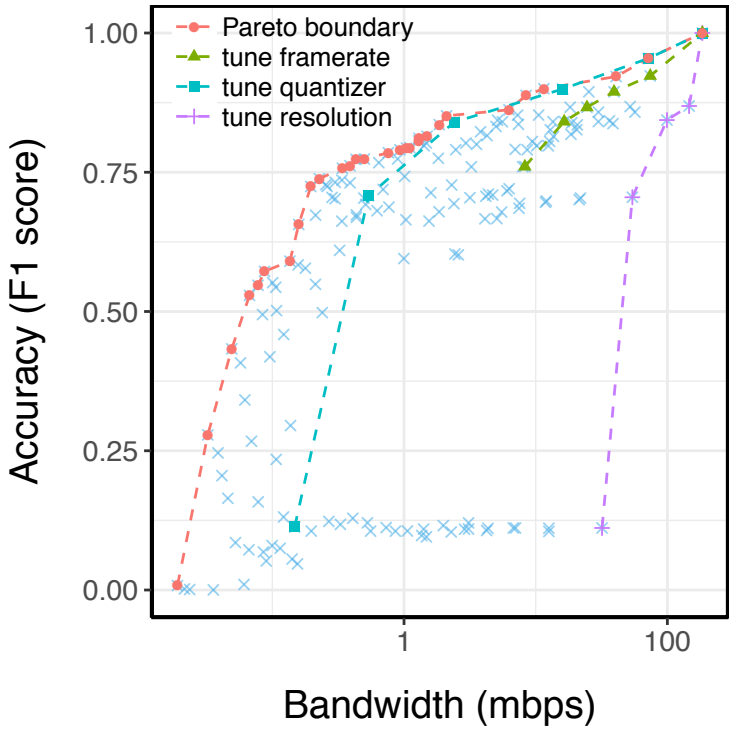

Figure 3.3: Bandwidth-accuracy tradeoff curve for pedestrian detection. Every point has a coordinate $(B(c), A(c))$ where c represents one specific degradation configuration. The goal of the profiling is to obtain the Pareto boundary as an optimal degradation strategy. We also show the profile if only one degradation operation is involved, e.g. only tune the frame rate, only tune the quantization or only tune the resolution. These profiles with one dimension tunable have a sub-optimal performance in comparison to strategies that involve multiple dimensions.

that our offline profiling has evaluated. Notice the vast spread in bandwidth requirement among configurations with similar accuracy as well as the wide spread in accuracy among configurations that consumes similar bandwidth.

We annotate three lines with configurations that only have one degradation operation in effect. Notice the distinct behavior of these three lines: reducing the resolution has the most penalty because the HOG detector has a minimal 128 pixels by 64 pixels window. The camera is deployed in a far-field context such that scaling down the image will quickly affect the detection rate. Tuning frame rate doesn't harm the accuracy too much. In fact, even with 1 FPS, the accuracy is still relatively high (75%). However, reducing frame rate alone doesn't bring much bandwidth saving as the data size of each frame increases (as we have mentioned in Fig. 2.2). The most effective way that reduces the bandwidth while preserving the accuracy is to adjust the quantizer because it has an effect on every pixel. But adjusting the quantizer alone quickly leads to accuracy drop because edge information is lost with an aggressive quantization.

The Pareto boundary, or *profile*, is the most important curve. Starting from the right side, we

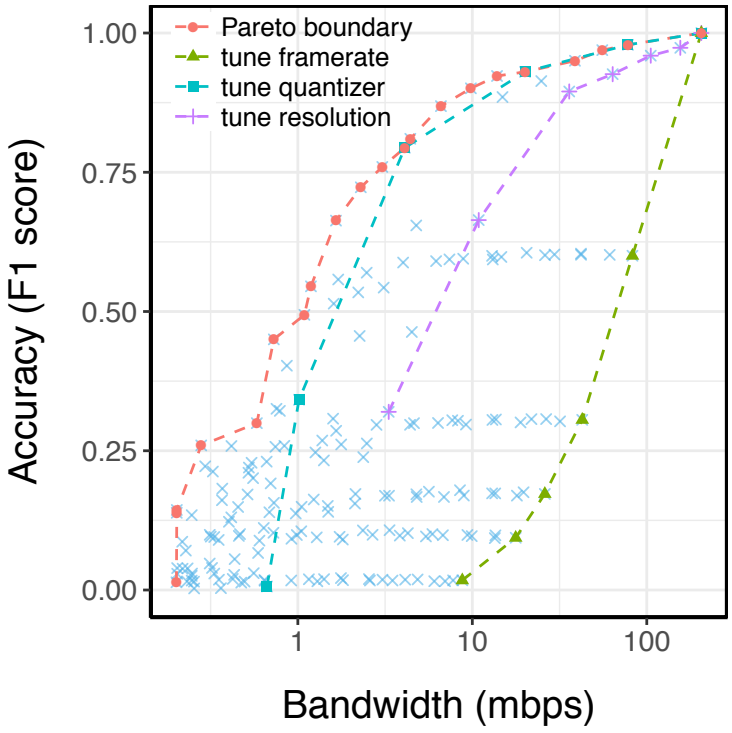

Figure 3.4: Bandwidth-accuracy tradeoff curve for Augmented Reality. This figure has a similar setup as Fig. 3.3 except the degradation profile has a different behavior.

see how the profile is close to the curve when only quantizer is tuned; and for the case of more limited bandwidth, the most optimal strategy is alone: it's only achievable with a combination of the three operations.

Augmented Reality: For this application, we've collected training data and test data by ourselves. The training data set is a 23-second video clip with 1920x1080 resolution and 30 FPS. It's taken on a mobile phone in an office environment. The test data is 37-second in a home environment. During the capture, we change the camera view in a slow pace to emulate how a real user would look around the environment. Because target objects are relatively close to the camera, we hypothesize that the profile for this application will has a less accuracy drop if frame resolution is degraded; instead, due to the movement of the camera, reducing frame rate will have a detrimental effect.

The generate profile is shown in Fig. 3.4. We do see a quick degradation when the frame rate is tuned. The resolution degradation has a modest effect and still, tuning the quantizer is mostly optimal up to a point. The Pareto boundary is only achieved when multiple degradations are in effect.

Top-K: To evalute the top-k application, we generate synthetic dataset based on real-world access logs (EDGAR log file dataset, the access log of `https://sec.gov` server). The original log file contains CSV-format data extracted from their Apache web servers [22]. Because the original

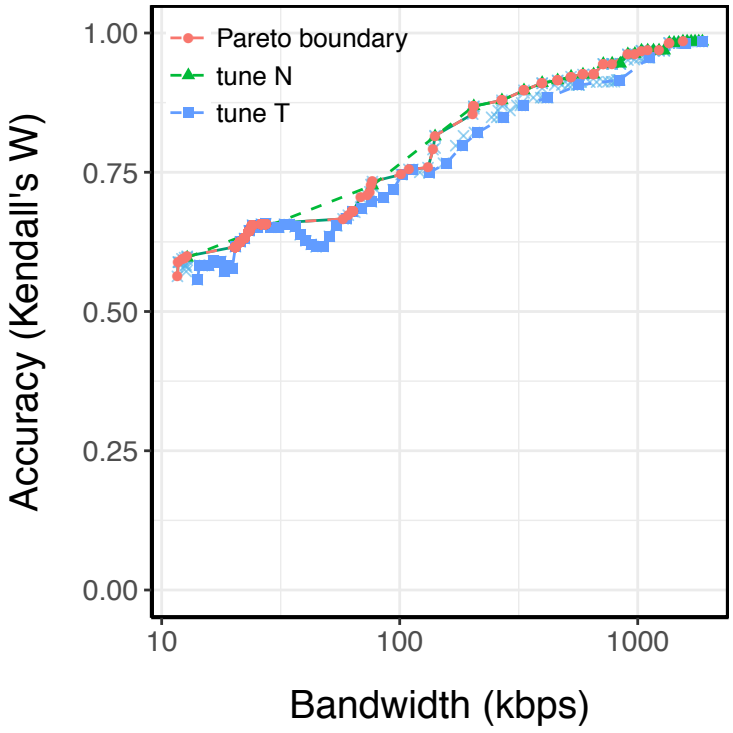

Figure 3.5: Top-k

log has only 500k access per hour, which is rather small in comparison to today's CDN log, we condensed each hour-long data into one second as our training and test data. After performing the local aggregation, the data size is reduced from 500k entries per second to 50k key-value pairs (10x reduction). Further reductions are made possible with the two degradation operations. The parameter `N` ranges from 100 to 15000 while `T` is from 0 to 500.

Fig. 3.5 shows the generated profile. As we can see, most configurations are very close to the Pareto boundary. The Pareto boundary is mostly achieved by tuning `N`. If applied for a different data set, this result may be different.

Runtime Performance

To evaluate the runtime behavior of AdaptiveStream, we conduct controlled experiments using four geo-distributed worker nodes from Amazon EC2 (*t2.large* instances) and an aggregation server from our institute. For each experiment, worker nodes transmit test data for about 10 mins. During each session, we use Linux `tc` utility to adjust outgoing bandwidth to experiment with network resource variation.

We compare our system with baseline systems that directly uses TCP and UDP. In all three applications, the raw data streams are orders of magnitude larger. While our system can adapt the rate, it could be unfair to baseline solutions. We've adjusted the default degradation operation

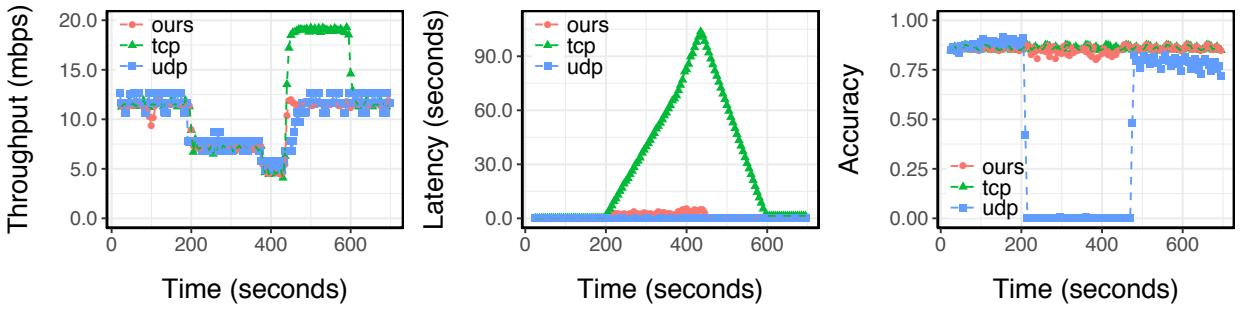

Figure 3.6: Runtime adaptation for pedestrian detection. The leftmost figure shows the throughput according to the receiver before and after we apply traffic shaping; note how TCP is catching up at 440s when the traffic shaping is removed. The middle figure shows the latency using three strategies, especially the increased latency for TCP when network capacity drops. The rightmost figure shows the accuracy. During the traffic shaping, UDP is experiencing a severe packet loss that many frames failed to be recovered.

so that TCP and UDP would work just fine under the normal conditions; in this way, we make a fair comparison and study how the adaptation behaves. In the case of UDP, shaping at the source doesn't emulate the packet loss behavior with out-of-order delivery. We use `netem` to control packet loss rate to match the desired shaping characteristics in the wide-area.

The results are shown in Fig. 3.6 and Fig. 3.7.[1] In our experiments, we see long delays in TCP when the network bandwidth is limited. The latency increases linearly after the traffic shaping starts. When the bandwidth shaping stops, TCP quickly fills the connection to recover. Depending on the queued size, the recovery could take a few minutes or tens of seconds. For UDP, the latency has been consistently small (mostly below 1 second) because there is no queue building up. But when traffic shaping starts and packet loss occurs, the accuracy drop is catastrophic.

Applications built with AdaptiveStream perform with a middle-ground behavior between the two extremes: a bounded latency with a reasonable accuracy drop. We notice that the top-k application still has about ten seconds delay. The reason for the slow adaptation in this application is two-folds: (1) our current implementation only requests for bandwidth information when congestion is detected, the delay of getting bandwidth estimation can be large in the case of network capacity drop; (2) we perform the bandwidth estimation in a conservative way with exponential smoothing to avoid sudden changes. While more improvements are possible (and we plan to achieve it), the current results have demonstrated the effectiveness of trading data fidelity for data freshness well.

[1]The augmented reality application exhibits a similar behavior as the pedestrian detection. However, due to time constrain, the entire experiment is not finished.

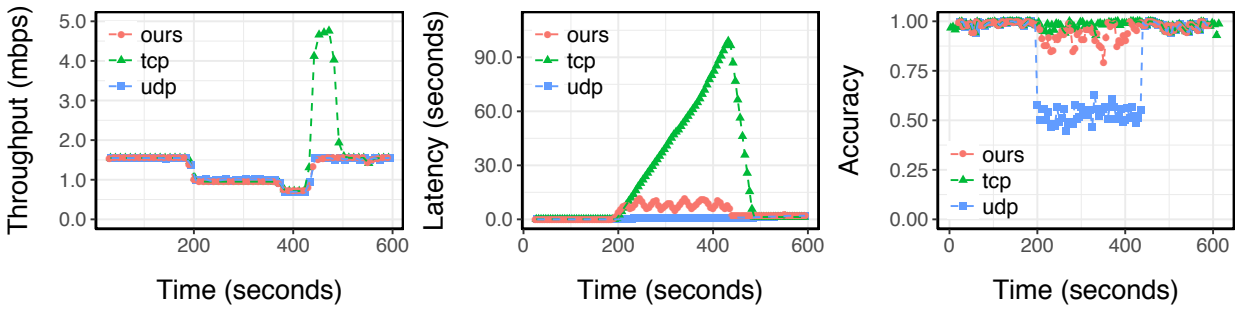

Figure 3.7: Runtime Adaptation of Top-K

Chapter 4

Thesis Plan

This thesis aims to make the following statement: a system-level design that empowers adaptation in an automatic way will enable the resilient execution of diverse streaming applications.

The current research results presented in this manuscript, i.e. the proposal, demonstrate how such a design achieves a balance between delay and application utility in the presence of network resource variation.

The complete thesis will extend the current work to a more holistic framework. Specifically, I plan to work on the following two directions (see Table 4.1 for the concrete timeline):

Multitask resource allocation: Many streaming applications will co-exist and share the infrastructure. When uncoordinated, they will compete for resources such as the network bandwidth. While the adaptation strategy may be optimal for each application, taking them collectively, the system may not be at an optimal state. In this direction, I will study the multitask resource allocation and how it can adapt each streaming applications for an overall maximal utility.

Adaptation to computing resources: In addition to network resources, the computing resources also have a large variation across the heterogeneous platforms. The available platforms range from embedded devices to powerful rack-level machines. Also, the availability of the platforms is not always guaranteed: nodes join and leave; the connectivity may be completely cut off. In this direction, I will study how to adapt applications to different configurations of the computing resources.

Time	Research Topic
Spring 2017	Resource allocation among multiple streaming applications
Summer 2017	Preliminary study on adaptation to computing resources
Fall 2017	Combine network and compute resources for a holistic control plane
Spring 2018	Finish the thesis

Table 4.1: Thesis Timeline

Bibliography

[1] Daniel J Abadi et al. “The Design of the Borealis Stream Processing Engine.” In: *CIDR*. Vol. 5. 2005, pp. 277–289.

[2] Hervé Abdi. “The Kendall Rank Correlation Coefficient”. In: *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA* (2007), pp. 508–510.

[3] Sameer Agarwal et al. “BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 29–42.

[4] Tyler Akidau et al. “MillWheel: Fault-tolerant Stream Processing at Internet Scale”. In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.

[5] John G Allen, Richard YD Xu, and Jesse S Jin. “Object Tracking Using Camshift Algorithm and Multiple Quantized Feature Spaces”. In: *Proceedings of the Pan-Sydney area workshop on Visual information processing*. Australian Computer Society, Inc. 2004, pp. 3–7.

[6] Sara Alspaugh et al. “Analyzing Log Analysis: An Empirical Study of User Log Mining.” In: *LISA*. 2014, pp. 53–68.

[7] Ganesh Ananthanarayanan et al. “GRASS: Trimming Stragglers in Approximation Analytics.” In: *NSDI*. 2014, pp. 289–302.

[8] Michael P Andersen and David E Culler. “BTrDB: Optimizing Storage System Design for Timeseries Processing”. In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 39–52.

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. “The Internet of Things: A Survey”. In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[10] Brian Babcock and Chris Olston. “Distributed Top-K Monitoring”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 28–39.

[11] Fabrice Bellard, M Niedermayer, et al. “FFmpeg”. In: *Availabel from: http://ffmpeg. org* (2012).

[12] G. Bradski. “The Opencv Library”. In: *Doctor Dobbs Journal* (2000).

[13] Matt Calder et al. “Mapping the Expansion of Google’s Serving Infrastructure”. In: *Proceedings of the 2013 conference on Internet measurement conference*. ACM. 2013, pp. 313–326.

[14] John Canny. “A Computational Approach to Edge Detection”. In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), pp. 679–698.

[15] Paris Carbone et al. “Apache Flink: Stream and Batch Processing in a Single Engine”. In: *Data Engineering* (2015), p. 28.

[16] Sirish Chandrasekaran et al. “TelegraphCQ: Continuous Dataflow Processing”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 668–668.

[17] Byung-Gon Chun et al. “Clonecloud: Elastic Execution Between Mobile Device and Cloud”. In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 301–314.

[18] Benjamin Coifman et al. “A Real-time Computer Vision System for Vehicle Tracking and Traffic Surveillance”. In: *Transportation Research Part C: Emerging Technologies* 6.4 (1998), pp. 271–288.

[19] Graham Cormode et al. “Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches”. In: *Foundations and Trends in Databases* 4.1–3 (2012), pp. 1–294.

[20] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Vol. 1. IEEE. 2005, pp. 886–893.

[21] Piotr Dollar et al. “Pedestrian Detection: An Evaluation of the State of the Art”. In: *IEEE transactions on pattern analysis and machine intelligence* 34.4 (2012), pp. 743–761.

[22] The Division of Economic and Risk Analysis (DERA). *EDGAR Log File Data Set*. `https://www.sec.gov/data/edgar-log-file-data-set`. Accessed: 2017-01-25.

[23] Mark Everingham et al. “The pascal visual object classes (VOC) challenge”. In: *International journal of computer vision* 88.2 (2010), pp. 303–338.

[24] Adrian Grange, Peter de Rivaz, and Jonathan Hunt. “VP9 Bitstream & Decoding Process Specification”. In: *Google, March* (2016).

[25] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. “Online aggregation”. In: *ACM SIGMOD Record*. Vol. 26. 2. ACM. 1997, pp. 171–182.

[26] Cisco Visual Networking Index. “The Zettabyte Era: Trends and Analysis”. In: *Cisco white paper* (2013).

[27] “iPerf: The TCP/UDP bandwidth measurement tool”. In: (). Accessed: 2017-01-17.

[28] Xin Jin et al. “Optimizing Bulk Transfers with Software-defined Optical WAN”. In: *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM. 2016, pp. 87–100.

[29] Michael D JOHAS TEENER et al. “Heterogeneous Networks for Audio and Video: Using IEEE 802.1 Audio Video Bridging”. In: *Proceedings of the IEEE* 101.11 (2013), pp. 2339–2354.

[30] Andrew Krioukov et al. “Building application stack (BAS)”. In: *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM. 2012, pp. 72–79.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet Classification with Deep Convolutional Neural Networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[32] Sanjeev Kulkarni et al. “Twitter Heron: Stream Processing at Scale”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 239–250.

[33] David G Lowe. “Distinctive image features from scale-invariant keypoints”. In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[34] Chaochao Lu and Xiaoou Tang. “Surpassing Human-level Face Verification Performance on LFW with Gaussian Face”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI’15. Austin, Texas: AAAI Press, 2015, pp. 3811–3819. ISBN: 0-262-51129-0. URL: http://dl.acm.org/citation.cfm?id=2888116.2888245.

[35] Martin Maas et al. “Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications”. In: *ACM SIGOPS Operating Systems Review* 50.2 (2016), pp. 457–471.

[36] MG Michalos, SP Kessanidis, and SL Nalmpantis. “Dynamic Adaptive Streaming over HTTP”. In: *Journal of Engineering Science and Technology Review* 5.2 (2012), pp. 30–34.

[37] Anton Milan et al. “MOT16: A Benchmark for Multi-Object Tracking”. In: *arXiv preprint arXiv:1603.00831* (2016).

[38] Sangmin Oh et al. “A Large-scale Benchmark Dataset for Event Recognition in Surveillance Video”. In: *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE. 2011, pp. 3153–3160.

[39] *One nation under CCTV: the future of automated surveillance*. http://www.wired.co.uk/article/one-nation-under-cctv. Accessed: 2017-01-27.

[40] Roger Pantos and William May. “HTTP Live Streaming”. In: (2016).

[41] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. “Deep Face Recognition.” In: *BMVC*. Vol. 1. 3. 2015, p. 6.

[42] Qifan Pu et al. “Low Latency Geo-Distributed Data Analytics”. In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 421–434.

[43] Ariel Rabkin et al. “Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area”. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 275–288.

[44] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. `http://pjreddie.com/darknet/`. 2013–2016.

[45] Iain E Richardson. *The H.264 Advanced Video Compression Standard*. John Wiley & Sons, 2011.

[46] C. J. Van Rijsbergen. *Information Retrieval*. 2nd. Newton, MA, USA: Butterworth-Heinemann, 1979. ISBN: 0408709294.

[47] Mahadev Satyanarayanan et al. "The Case for VM-based Cloudlets in Mobile Computing". In: *IEEE pervasive Computing* 8.4 (2009).

[48] Henning Schulzrinne. "Real time streaming protocol (RTSP)". In: (1998).

[49] *Skynet achieved: Beijing is 100% covered by surveillance cameras, and nobody noticed*. `https://www.techinasia.com/skynet-achieved-beijing-100-covered-surveillance-cameras-noticed`. Accessed: 2017-01-27.

[50] Yi Sun et al. "CS2P: Improving Video Bitrate Selection and Adaptation with Data-driven Throughput Prediction". In: *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM. 2016, pp. 272–285.

[51] GStreamer Team. *GStreamer: Open Source Multimedia Framework*.

[52] Ankit Toshniwal et al. "Storm@ twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156.

[53] Paul Viola and Michael Jones. "Rapid Object Detection Using a Boosted Cascade of Simple Features". In: *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*. Vol. 1. IEEE. 2001, pp. I–I.

[54] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. "Clarinet: WAN-Aware Optimization for Analytics Queries". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association. 2016, pp. 435–450.

[55] Xiaoqi Yin et al. "A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP". In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 325–338.

[56] Matei Zaharia et al. "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters". In: *Presented as part of the*. 2012.

[57] Ben Zhang et al. "The Cloud is Not Enough: Saving IoT from the Cloud." In: *HotCloud*. 2015.

[58] Hong Zhang et al. "Guaranteeing Deadlines for Inter-Data Center Transfers". In: *IEEE/ACM Transactions on Networking* (2016).