Pytheas: Enabling Data-Driven Quality of Experience Optimization Using Group-Based Exploration-Exploitation

Junchen Jiang, *Carnegie Mellon University;* Shijie Sun, *Tsinghua University;*
Vyas Sekar, *Carnegie Mellon University;*
Hui Zhang, *Carnegie Mellon University and Conviva Inc.*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/jiang

This paper is included in the Proceedings of the
14th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Pytheas: Enabling Data-Driven Quality of Experience Optimization Using Group-Based Exploration-Exploitation

Junchen Jiang[†], *Shijie Sun*[°], *Vyas Sekar*[†], *Hui Zhang*[†⋆]
[†]*CMU,* [°]*Tsinghua University,* [⋆]*Conviva Inc.*

Abstract

Content providers are increasingly using data-driven mechanisms to optimize quality of experience (QoE). Many existing approaches formulate this process as a *prediction* problem of learning optimal decisions (e.g., server, bitrate, relay) based on observed QoE of recent sessions. While prediction-based mechanisms have shown promising QoE improvements, they are necessarily incomplete as they: (1) suffer from many known biases (e.g., incomplete visibility) and (2) cannot respond to sudden changes (e.g., load changes). Drawing a parallel from machine learning, we argue that data-driven QoE optimization should instead be cast as a *real-time exploration and exploitation (E2)* process rather than as a prediction problem. Adopting E2 in network applications, however, introduces key architectural (e.g., how to update decisions in real time with fresh data) and algorithmic (e.g., capturing complex interactions between session features vs. QoE) challenges. We present *Pytheas*, a framework which addresses these challenges using a *group-based E2* mechanism. The insight is that application sessions sharing the same features (e.g., IP prefix, location) can be grouped so that we can run E2 algorithms at a per-group granularity. This naturally captures the complex interactions and is amenable to real-time control with fresh measurements. Using an end-to-end implementation and a proof-of-concept deployment in CloudLab, we show that Pytheas improves video QoE over a state-of-the-art prediction-based system by up to 31% on average and 78% on 90th percentile of per-session QoE.

1 Introduction

We observe an increasing trend toward the adoption of data-driven approaches to optimize application-level quality of experience (QoE) (e.g., [26, 19, 36]). At a high level, these approaches use the observed QoE of recent application sessions to proactively improve the QoE for future sessions. Many previous and ongoing efforts have demonstrated the potential of such data-driven optimization for applications such as video streaming [25, 19], Internet telephony [24] and web performance [30, 39].

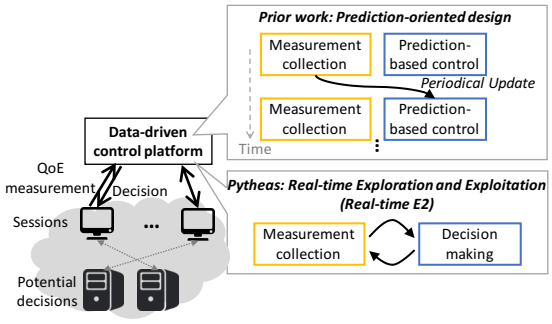


Figure 1: ***Prediction-oriented designs vs. Real-time E2.***

Existing frameworks typically formulate data-driven optimization as a *prediction* problem (e.g., [25, 39]). As depicted in Figure 1, they use passive measurements from application sessions to periodically update a prediction model, which captures the interaction between possible "decisions" (e.g., server, relay, CDN, or bitrate), different network- and application-layer features (e.g., client AS, content type), and QoE metrics. The prediction model is then used to make decisions for future sessions that yield the best (predicted) QoE.

While previous work has shown considerable QoE improvement using this prediction-based workflow, it has key limitations. First, QoE measurements are collected passively by a process independently of the prediction process. As a result, quality predictions could easily be biased if the measurement collection does not present the representative view across sessions. Furthermore, predictions could be inaccurate if quality measurements have high variance [39]. Second, the predictions and decisions are updated periodically on coarse timescales of minutes or even hours. This means that they cannot detect and adapt to sudden events such as load-induced quality degradation.

To address these concerns, we argue that data-driven optimization should instead be viewed through the lens of *real-time exploration and exploitation (E2)* frameworks [43] from the machine learning community rather than as prediction problems. At a high level (Figure 1), E2 formulates measurement collection and decision making as a *joint* process with real-time QoE mea-

surements. The intuition is to continuously strike a dynamic balance between exploring suboptimal decisions and exploiting currently optimal decisions, thus fundamentally addressing the aforementioned shortcomings of prediction-based approaches.

While E2 is arguably the right abstraction for data-driven QoE optimization, realizing it in practice in a networking context is challenging on two fronts:

- *Challenge 1: How to run real-time E2 over millions of globally distributed sessions with fresh data?* E2 techniques require a fresh and global view of QoE measurements. Unfortunately, existing solutions pose fundamental tradeoffs between data freshness and global views. While some (e.g., [19, 25]) only update the global view on timescales of minutes, others (e.g., [16]) assume that a fresh global view can be maintained by a single cluster in a centralized manner, which may not be possible for geo-distributed services [34, 33].
- *Challenge 2: How to capture complex relations between sessions in E2?* Traditional E2 techniques assume that the outcome of using some decision follows a static (but unknown) distribution, but application QoE often varies across sessions, depending on complex network- and application-specific "contexts" (e.g., last-mile link, client devices) and over time.

We observe an opportunity to address them in a networking context by leveraging the notion of **group-based E2**. This is driven by the domain-specific insight that network sessions sharing the same network- and application-level features intuitively will exhibit the same QoE behavior across different possible decisions [25, 24, 19]. This enables us to: (1) *decompose* the global E2 process into separate per-group E2 processes, which can then be independently run in geo-distributed clusters (offered by many application providers [33]); and (2) reuse well-known E2 algorithms (e.g., [14]) and run them at a coarser *per-group* granularity.

Building on this insight, we have designed and implemented *Pytheas*[1], a framework for enabling E2-based data-driven QoE optimization of networked applications. Our implementation [9] synthesizes and extends industry-standard data processing platforms (e.g., Spark [10], Kafka [2]). It provides interfaces through which application sessions can send QoE measurement to update the real-time global view and receive control decisions (e.g., CDN and bitrate selection) made by Pytheas. We demonstrate the practical benefit of Pytheas by using video streaming as a concrete use case. We extended an open source video player [5], and used a trace-driven evaluation methodology in a CloudLab-based deployment [4]. We show that (1) Pytheas improves video QoE over a state-of-the-art prediction-based baseline by 6-31% on average, and 24-78% on 90th percentile, and (2) Pytheas is horizontally scalable and resilient to various failure modes.

[1] Pytheas was an early explorer who is claimed to be the first person to record seeing the Midnight Sun.

2 Background

Drawing on prior work, we discuss several applications that have benefited (and could benefit) from data-driven QoE optimization (§2.1). Then, we highlight key limitations of existing prediction-based approaches (§2.2).

2.1 Use cases

We begin by describing several application settings to highlight the types of "decisions" and the improvements that data-driven QoE optimization can provide.

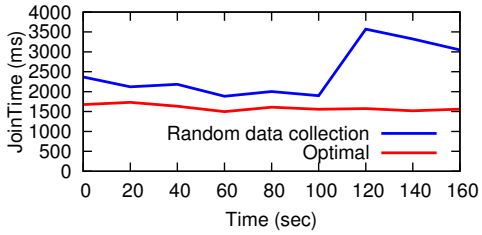
Video on demand and live streaming: VoD providers (e.g., YouTube, Vevo) have the flexibility to stream content from multiple servers or CDNs [41] and in different bitrates [23], and already have real-time visibility into video quality through client-side instrumentation [17]. Prior work has shown that compared with traditional approaches that operate on a single-session basis, data-driven approaches for bitrate and CDN selection could improve video quality (e.g., 50% less buffering time) [19, 25, 40]. Similarly, the QoE of live streaming (e.g., ESPN, twitch.tv) depends on the overlay path between source and edge servers [27]. There could be room for improvement when these overlay paths are dynamically chosen to cope with workload and quality fluctuation [32].

Internet telephony: Today, VoIP applications (e.g., Hangouts and Skype) use managed relays to optimize network performance between clients [24]. Prior work shows that compared with Anycast-based relay selection, a data-driven relay selection algorithm can reduce the number of poor-quality calls (e.g., 42% fewer calls with over 1.2% packet loss) [24, 21].

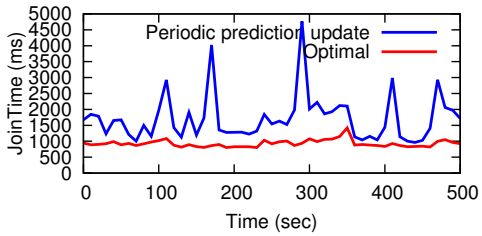
File sharing: File sharing applications (e.g., Dropbox) have the flexibility to allow each client to fetch files [18] from a chosen server or data center. By using data-driven approaches to predict the throughput between a client and a server [40, 39, 44], we could potentially improve the QoE for these applications.

Social, search, and web services: Social network applications try to optimize server selection, so that relevant information (e.g., friends' feeds) is co-located. Prior work shows that optimal caching and server selection can reduce Facebook query time by 50% [37]. Online services (e.g., search) can also benefit from data-driven techniques. For instance, data-driven edge sever selection can reduce search latency by 60% compared to Anycast, while serving 2× more queries [30].

Drawing on these use cases, we see that data-driven QoE optimization can be applied to a broad class of net-



(a) Example A: Suboptimal quality due to fixed random data collection.



(b) Example B: Overload and oscillations between decisions due to periodic prediction.

Figure 2: ***Limitations of prediction-oriented abstraction (e.g., CFA [25]) manifested in two real examples.***

worked applications that meet two requirements: (i) application providers can make decisions (e.g., server, bitrate, relay) that impact QoE of individual application sessions; and (ii) each session can measure its QoE and report it to application provider in near real time. Note that this decision-making process can either be run by the application provider itself (e.g., Skype [24]) or by some third-party services (e.g., Conviva [19]).

2.2 Limitations of predictive approaches

Many prior approaches (e.g., [19, 25, 40, 24]) for data-driven QoE optimization use a *prediction-based* workflow. That is, they periodically train a quality prediction model based on passive measurements to inform decisions for future sessions; e.g., using history data to decide what will be the best relay server for a Skype call or the best CDN for a video session? While such prediction-based approaches have proved useful, they suffer from well-known limitations, namely, *prediction bias* and *slow reaction* [22, 39]. Next, we highlight these issues using CDN selection in video streaming as a concrete use case.

2.2.1 Limitation 1: Prediction bias

A well-known problem of prediction-based workflows is that the prediction can be biased by prior decisions. Because the input measurement data are based on previous set of best decisions, we will not have a reliable way to estimate the potential quality improvements of other decisions in the future [39]. A simple solution is to use a fixed percentage of sessions to explore different decisions. This could eliminate the above prediction bias. However, it can still be suboptimal, since it might either let too many sessions use suboptimal decisions when quality is stable, or collect insufficient data in presence of higher variance.

Example A: Figure 2a shows a trace-driven evaluation to highlight such prediction biases. We use a trace of one of the major video providers in US. As a baseline, we consider prior work called CFA [25], which uses a fixed fraction of 10% sessions to randomly explore suboptimal decisions.[2] We see that it leads to worse average video startup latency, or join time, than an optimal strategy that always picks the CDN with the best average quality in each minute. Each video session can pick CDN1 or CDN2, and in the hindsight, CDN1 is on average better CDN2, except between t=40 and t=120, when CDN2 has a large variance. Even when CDN1 is consistently better than CDN2, CFA is worse than optimal, since it always assigns 10% of sessions to use CDN2. At the same time, when CDN2 becomes a better choice, CFA cannot detect this change in a timely fashion as 10% is too small a fraction to reliably estimate quality of CDN2.

2.2.2 Limitation 2: Slow reaction

Due to the time taken to aggregate sufficient data for model building, today's prediction-based systems update quality predictions periodically on coarse timescales; e.g., CFA updates models every tens of seconds [25], and VIA updates its models every several hours [24]. This means that they cannot quickly adapt to changes in operating conditions which can cause *model drifts*. First, if there are sudden quality changes (e.g., network congestion and service outage), prediction-based approaches might result in suboptimal quality due to its slow reaction. Furthermore, such model shifts might indeed be a consequence of the slow periodic predictions; e.g., the best predicted server or CDN will receive more requests and its performance may degrade as its load increases.

Example B: We consider an AS and two CDNs. For each CDN, if it receives most sessions from the AS, it will be overloaded, and the sessions served by it will have bad quality. Figure 2b shows that CFA, which always picks the CDN that has the best quality in the last minute, has worse quality than another strategy which assigns half of sessions to each CDN. This is because CFA always overloads the CDN that has the best historical performance by assigning most sessions to it, and CFA will switch decisions only *after* quality degradation occurs, leading to oscillations and suboptimal quality.

At a high level, these limitations of prediction-based approaches arise from the logical separation between measurement collection and decision making. Next, we

[2]The process begins by assigning sessions uniformly at random to all decisions in the first minute, and after that, it assigns 90% sessions to the optimal decisions based on the last minute.

discuss what the right abstraction for data-driven QoE optimization should be to avoid these limitations.

3 QoE optimization as an Exploration-Exploitation process

To avoid these aforementioned limitations of prediction-based approaches, ideally we want a framework where decisions are updated in concert with measurement collection in real time. There is indeed a well-known abstraction in the machine learning community that captures this—exploration and exploitation (E2) processes [43]. Drawing on this parallel, we argue why data-driven QoE optimization should be cast instead as a *real-time E2* process rather than a prediction-based workflow.

Background on exploration and exploitation: An intuitive way to visualize the exploration and exploitation (E2) process is through the lens of a multi-armed bandit problem [43]. Here, a gambler pulls several slot machines, each associated with an unknown reward distribution. The goal of the gambler is to optimize the mean rewards over a sequence of pulls. Thus, there is some intrinsic *exploration* phase where the gambler tries to learn these hidden reward functions, and subsequent *exploitation* phase to maximize the reward. Note that the reward functions could change over time, and thus this is a continuous process rather than a one-time shot.

QoE optimization as E2 (Figure 3): Given this framework, we can see a natural mapping between E2 and data-driven QoE optimization. Like E2, data-driven QoE optimization observes the QoE (i.e., reward) of a decision every time the decision is used (i.e., pulled) by a session. Our goal is to maximize the overall QoE after a sequence of sessions.

Casting data-driven optimization as E2 not only provides a systematic framework for data-driven QoE optimization, but also allows us to leverage well-known algorithms (e.g., [14]) from the machine learning literature. As E2 integrates the measurement collection (exploration) and decision making (exploitation) in a joint process, we can dynamically explore decisions whose quality estimation has high uncertainty, and exploit decisions that are clearly better. For instance, in Example A, an E2 process could reduce traffic for exploration when QoE is stable (before 40 second and after 120 seconds), and raise it when QoE changes (between 40 second and 120 second). By running E2 in real time with the most up-to-date measurement data, we could detect QoE drift as soon as some sessions have experienced them, and adapt to them faster than prediction-based approaches. For instance, in Example B, real-time E2 would detect load-induced QoE degradation on CDN1 as soon as its QoE is worse than CDN2, and start switching sessions to CDN2 *before* overloading CDN1.[3]

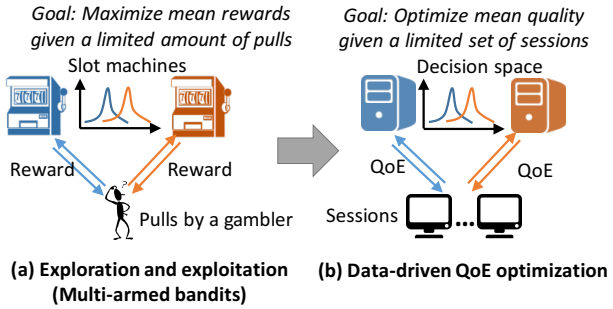


Figure 3: ***Casting data-driven QoE optimization into formulation of exploration and exploitation (E2).***

Challenges: While E2 offers the right abstraction in contrast to prediction-based approaches, applying it in network applications raises practical challenges:

- Traditional E2 techniques (e.g., [43, 29, 16]) need fresh measurements of all sessions, but getting such a fresh and global view is challenging, because application providers store fresh data in geo-distributed clusters, called *frontend clusters*, which only have a partial view across sessions. Existing analytics framework for such geo-distributed infrastructure, however, either trade global view for data freshness (e.g., [19]), or target query patterns of a traditional database (e.g., [33]), not millions of concurrent queries from geo-distributed clients, as in our case.
- Traditional E2 techniques also make strong assumptions about the context that affects the reward of a decisions, but they may not hold in network settings. For instance, they often assume some notion of continuity in context (e.g., [38]), but even when some video sessions match on all aspects of ISP, location, CDN resource availability, they may still see very different QoE, if they differ on certain key feature (e.g., last-hop connection) [25].

4 Pytheas system overview

To address the practical challenges of applying E2 in network applications, we observe a key domain-specific insight in networked applications that enables us to address both challenges in practice. We highlight the intuition behind our insight, which we refer to as group-based E2, and then provide an overview of the Pytheas system which builds on this insight.

Insight of Group-based E2: Our insight is that the "network context" of application sessions is often aligned with their "network locality". That is, if two sessions share the context that determines their E2 decisions, they will be likely to match on some network-specific fea-

[3] Note that we do not need to know the capacity of each CDN, which is often unknown to content providers.

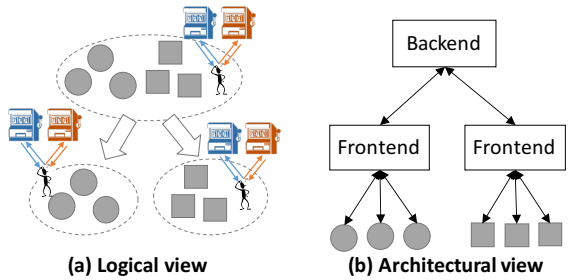


Figure 4: ***Illustration of group-based E2.***

tures. We see manifestations of this insight in many settings. For instance, video sessions with similar QoE from the same CDN/server tend to match on client IP prefix [25, 40]. Similarly, VoIP calls between the same ASes are likely to share the best relays [24], and clients from same /24 IP prefix will have similar web load time from the same edge proxy [30]. In §6.2, we validate this insight with a real-world dataset.

This insight inspires the notion of *group-based E2*, which can address the above challenges by enabling an effective decomposition of the E2 process (Figure 4a). Specifically, instead of a global E2 process over all sessions, we group together sessions with similar context by network locality and other key features (such as device and location), and use one E2 process for each group. Since sessions within a group share network locality (e.g., in the same locations and IP prefixes), they are likely to be mapped to the same frontend cluster. By running the per-group E2 logic in this frontend cluster, we can update decisions with fresh data from other sessions in the group received by this frontend cluster. Furthermore, as each group consists of sessions with similar context, it is sufficient to use traditional E2 techniques based on the data of sessions in one group, without needing a global view of all sessions. It is important to note that sessions are not grouped entirely based on IP prefixes. The sessions in the same network locality could have very different QoE, depending on the device, last-hop connectivity, and other features. Therefore, we group sessions on a finer granularity than IP prefix.

System overview: Figure 4b shows how the group-based E2 is realized in the Pytheas architecture. Each session group is managed by one per-group E2 process run by one frontend cluster. When a session comes in, it sends a request for its control decisions, which includes its features, to the Pytheas system. The request will be received by a frontend cluster, which maps the session to a group based on its features, then gets the most up-to-date decision from the local per-group E2 process, and returns the decision to the session. Each session measures its QoE and reports it to the same frontend cluster. When this frontend receives the QoE measurement, it again maps the session to a group, and updates the E2 logic of the group with the new measurement. In most cases, the E2 logic of a group is run by the same cluster that receives the requests and measurements of its sessions, so E2 logic can be updated in real time.

The backend cluster has a global, but slightly stale, view of QoE of all sessions, and it determines the session groups – which group each session belongs to and which frontend should run the per-group logic for each group. Normally, such grouping is updated periodically on a timescale of minutes. During sudden changes such as frontend cluster failures, it can also be triggered on demand to re-assign groups to frontend clusters.

The following sections will present the algorithms (§5) and system design (§6) of Pytheas, and how we implemented it (§7) in more details.

5 Pytheas algorithms

Using group-based E2, Pytheas decouples real-time E2 into two parts: a *session-grouping* logic to partition sessions into groups, and a *per-group E2* logic that makes per-session decisions. This section presents the design of these two core algorithmic pieces and how we address two issues:[4] (i) Grouping drift: the session-grouping logic should dynamically regroup sessions based on the context that determines their QoE; and (ii) QoE drift: the per-group control logic should switch decisions when QoE of some decisions change.

5.1 Session-grouping logic

Recall that sessions of the same group share the same factors on which their QoE and best decisions depend. As a concrete example, let us consider CDN selection for video. Video sessions in the same AS whose QoE depends on the local servers of different CDNs should be in the same group. However, video sessions whose QoE is bottlenecked by home wireless and thus is independent to CDNs should not be in the same group. In other words, sessions in the same group share not only the best decision, but also the factors that determine the best decisions.

A natural starting point for this grouping decision is using the notion of critical features proposed in prior work [25]. At a high level, if session A and B have the same values of critical features, they will have similar QoE. Let $S(s,F,\Delta)$ denote the set of sessions that occur within the last Δ time interval and share the same feature values as s on the set of features F, and let $Q(X)$ denote the QoE distribution of a session set X. Then, the critical feature set F^* of a session s:

$$\text{argmin}_{F\subseteq F^{all},|S(s,F,\delta)|>n}|Q(S(s,F^{all},\Delta))-Q(S(s,F,\Delta))|$$

[4] We assume in this section that the per-group control logic is updated in real time (which will be made possible in the next section).

That is, the historical session who match values on critical features F^* with s have very similar QoE distribution to those matching on all features with s on a long timescale of Δ (say last hour). The clause $|S(s,F,\delta)| > n$ ensures that there is sufficient mass in that set to get a statistically significant estimate even on small timescales δ (e.g., minutes). Such a notion of critical features has also been (implicitly) used in many other applications; e.g., AS pairs in VoIP [24] and /24 prefixes for web page load time [30]. Thus, a natural strawman for grouping algorithm is to groups sessions who match on their critical features, i.e., they have similar QoE.

However, we observe two problems inherent to critical features, which make it unsuitable to directly group sessions based on critical features: (1) First, grouping sessions based on critical features may result in groups that consist of only sessions using similar decisions, so their measurement will be biased towards a subset of decisions. (2) Second, grouping sessions based on critical features will also create overlaps between groups, so E2 logic of different groups could make conflicting decisions on these overlapping sessions. For instance, consider two Comcast sessions, s_1 and s_2, if the critical feature of s_1 is ISP, and the critical feature of s_2 is its local WiFi connection, s_2 will be in both the "WiFi" group and the "Comcast" group.

To address these issues, we formulate the goal of session grouping as following. Given a session set, the session-grouping logic should output any non-overlapping partition of sessions so that if two sessions s_1 and s_2 are in the same group, s_1 and s_2 should match values on s_1 or s_2's non-decision-specific critical features. Non-decision-specific features are the features independent of decisions; e.g., "device" is a feature independent of decisions, since video sessions of the same device can make any decisions regarding CDN and bitrate.

Operationally, we use the following approach to achieve such a grouping. First, for each session, we learn its critical features, and then ignore decision-specific features from the set of critical features of each session. Then, we recursively group sessions based on the remaining critical features in a way that avoids overlaps between groups. We start with any session s_1, and create a group consisting of all sessions that match with s_1 on s_1's critical features. We then recursively do the two following steps until every session is in some group. We find a session s_2, who is not included in any existing group, and create a new group of all sessions that match with s_2 on s_2's critical features. If the new group does not overlap with any existing group, it will be a new individual group, otherwise, we will add it to the existing groups in the way illustrated in Figure 5. We organize the existing groups in a graph, where each node is split by values of a certain feature, and each group includes

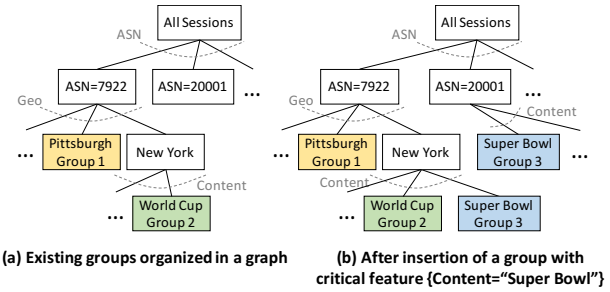


Figure 5: ***An illustrative example of session groups organized in a graph and how to a new group is added.***

multiple leaf nodes. For instance, if we want to add a new group that consists of sessions whose "content" is "Super Bowl" to a graph of existing groups as shown in Figure 5a, we will fork a path to create a new leaf node whenever the new group overlap with a existing group. Note that, this means multiple leaf nodes may be belong to the same group (e.g., "Group 3" in Figure 5b contains two different leaf nodes).

5.2 Per-group E2 logic

To run E2 in presence of QoE drift, we use Discounted UCB algorithm [20], a variant of the UCB algorithm [14], as the per-group E2 logic. UCB (Upper Confidence Bound) algorithms [14] are a family of algorithms to solve the multi-armed bandits problem. The core idea is to always opportunistically choose the arm that has the highest upper confidence bound of reward, and therefore, it will naturally tend to use arms with high expected rewards or high uncertainty. Note that the UCB algorithms do not explicitly assign sessions for "exploration" and "exploitation".

We use Discounted UCB algorithm to adapt to QoE drift, because it automatically gives more weight to more recent measurements by exponentially discounting historical measurements. Therefore, unlike other UCB algorithms which will (almost) converge to one decision, Discounted UCB is more likely to revisit suboptimal decisions to retain visibility across all decisions. We refer readers to [20] for more details. Given a session s, it returns a decision that has not been tried, if there is any. Otherwise, it calculates a score for each potential decision d by adding up an exponentially weighted moving average of d's history QoE and an estimation on the uncertainty of reward of d, and picks the decision with highest score.

6 Pytheas system architecture

Given the algorithmic pieces from the previous section, next we discuss how we map them into a system architecture. At a high level, the E2 logic of each group is independently run by frontend clusters, while the session-to-group mapping is continuously updated by the backend.

6.1 Requirements

The Pytheas system design must meet four goals:

1. *Fresh data:* The per-group E2 logic should be updated every second with newest QoE measurements.
2. *Global scale:* It should handle millions of geo-distributed sessions per second.
3. *Responsiveness:* It should respond to requests for decisions from sessions within a few milliseconds.
4. *Fault tolerance:* QoE should not be significantly impacted when parts of the system fail.

A natural starting point to achieve this goals might be to adopt a "split control plane" approach advocated by prior work for prediction-based approaches [19, 25]. At a high level, this split control plane has two parts: (1) a backend cluster that generates centralized predictions based on global but stale data, and (2) a set of geodistributed frontend servers that use these predictions from the backend to make decisions on a per-session basis. This split control architecture achieves global scale and high responsiveness, but fundamentally sacrifices data freshness.

Pytheas preserves the scale and responsiveness of the split control approach, but extends in two key ways to run group-based E2 with fresh data. First, each frontend cluster runs an active E2 algorithm rather than merely executing the (stale) prediction decisions as in prior work. Second, the frontend clusters now run per-group logic, not per-session logic. This is inspired by the insight that sessions in the same group are very likely to be received by the same frontend cluster. Thus, group-based E2 could achieve high data freshness on the session group granularity, while having the same scale and responsiveness to split control. Next, we discuss the detailed design of the frontend and backend systems.

6.2 Per-group control by frontends

The best case for group-based E2 is when all sessions of the same group are received by the same frontend cluster. When this is true, we can run the per-group E2 logic (§5.2) in real time with fresh measurements of the same group. In fact, this also is the common case. To show this, we ran session-grouping logic (§5.1) on 8.5 million video sessions in a real-world trace, and found around 200 groups each minute. Among these groups, we found that (in Figure 6) for 95% of groups, all sessions are in the same AS, and for 88% of groups, all sessions are even in the same AS *and* same city. Since existing session-to-frontend mappings (e.g., DNS or Anycast-based mechnisms) are often based on AS and geographical location, this means that for most groups, their sessions will be verly likely to be received in the same frontend clusters.

In practice, however, it is possible that sessions of one group are spread across frontend clusters. We have two

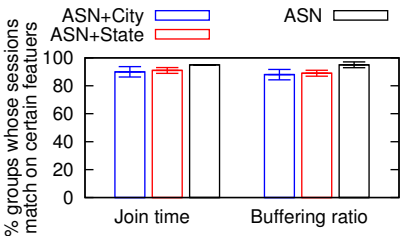


Figure 6: ***For most groups, the sessions are in the same ASN and even same city.***

options in this case:

1. Pick one cluster as the *leader* cluster of this group and let it run the E2 logic of the group based on the measurements received by this cluster. Meanwhile, other clusters, called *proxy* clusters of the group, simply receive decisions periodically from the leader cluster.
2. Keep the leader cluster and proxy clusters, but let proxy clusters not only receive decisions from the leader, but also forward QoE measurements to the leader cluster.

We see a tradeoff between the two options. While Option 1 is less complex to implement than Option 2, the leader proxy in Option 1 runs per-group logic based on only a subset of sessions, especially when the sessions of a group are evenly spread across many frontend clusters. We pick Option 2, because it is cleaner in that the per-group logic is based on all sessions in a group. In fact, implementing Option 2 does not add much complexity. Finally, Option 2 can easily fall back to Option 1 by stop forwarding measurements from proxy clusters to the leader cluster.

6.3 Updating session groups in backend

The backend cluster uses a global, stale view of measurements to update two tables, which are sent to the frontend to regroup sessions.

- First, the backend runs the session-grouping logic (§5.1) to decide which group each session belongs to, and outputs a *session-to-group table*.
- Second, it decides which frontend should be the leader cluster of each group and outputs a *group-to-leader table*. For each group, we select the frontend cluster that receives most sessions in the group as the leader.

The backend periodically (by default, every ten minutes) updates the frontend clusters with these two maps. The only exception for the maps to be updated in near real time is when one or more frontend clusters fail, which we discuss next.

6.4 Fault tolerance

As we rely on fault-tolerant components for the individual components of Pytheas within each cluster (see §7), the residual failure mode of Pytheas is when some clusters are not available. Next, we discuss how we tackle

three potential concerns in this setting.

First, if a failed frontend is the leader cluster of a group, the states of the E2 logic of the group will be lost, and we will not be able to update decisions for sessions of the group. To detect frontend failures and minimize their impact, each frontend sends frequent heartbeat messages through a "fast channel" every few seconds (by default, every five seconds) to the backend, so backend can detect frontend failures based on these heartbeat messages. Once the backend detects a frontend failure, it will select a new leader clusters for any group whose leader cluster has been the failed one, and recover the per-group logic in the new leader cluster. To recover the per-group states, each leader always shares the per-group states with its proxy clusters in the decision update messages, so that when a proxy cluster is selected as the new leader, it can recover the per-group states as they are cached locally. Note that even without a leader cluster, a proxy cluster can still respond requests with the cached decisions made by the leader cluster before it fails.

Second, the sessions who are assigned to the failed frontend will not receive control decisions. To minimize this impact, Pytheas will fall back to the native control logic. Take video streaming as an example, when Pytheas is not available, the client-side video player can fall back to the control logic built into the client-side application (e.g., local bitrate adaptation) to achieve graceful QoE degradation, rather than crash [19].

Finally, if the backend cluster is not available, Pytheas will not be able to update groups. However, Pytheas does not rely on backend to makde decisions, so clients will still receive (albeit suboptimal) decisions made by Pytheas's frontend clusters.

7 Implementation and optimization

Pytheas is open source ($\approx$ 10K lines of code across Java, python, and PHP) and can be accessed at [9]. Next, we describe the APIs for applications to integrate with Pytheas, and then describe the implementation of frontend and backend, as well as optimizations we used to remove Pytheas's performance bottlenecks.

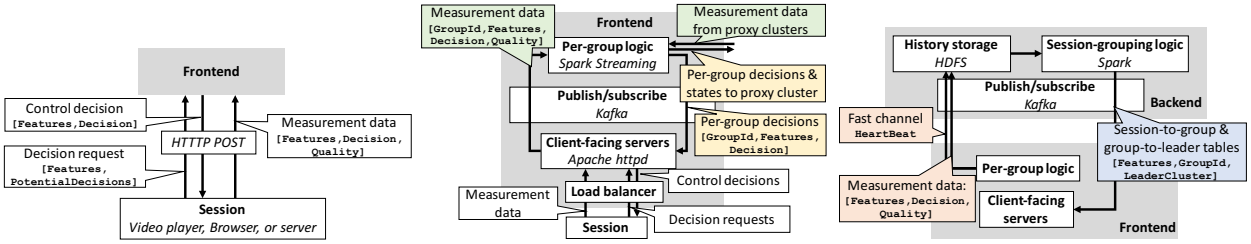
Pytheas APIs: Application sessions communicate with Pytheas through two APIs (Figure 7b): One for requesting control decisions, one for uploading measurement data. Both are implemented as standard HTTP POST messages. The session features are encoded in the data field of the POST message. Pytheas also needs content providers to provide the schema of the message that sessions send to the Pytheas frontend, as well as a list of potential decisions. Content providers may also provide QoE models that compute QoE metrics from the raw quality measurements sent by sessions.

Frontend: Figure 7b shows the key components and interfaces of a frontend cluster. When a session sends a control request to Pytheas, the request will be received by one of the client-facing servers run by Apache httpd [1]. The server processes the request with a PHP script, which first maps the session to a group and its leader cluster by matching the session features with the session-to-group and group-to-leader tables. Then the server queries the E2 logic of the group (§5.2), for the most up-to-date decision of the group, and finally returns the decision to the session. The script to process the measurement data uploaded by a session is similar; a client-facing server maps it to a group, and then forwards it to the per-group E2 logic. The per-group E2 logic is a Spark Streaming [3] program. It maintains a key-value map (a Spark RDD) between group identification and the per-group E2 states (e.g., most recent decisions), and updates the states every second by the most recent measurement data in one MapReduce operation. The communication between these processes is through Kafka [2], a distributed publish/subscribe service. We used Apache httpd, Spark Streaming, and Kafka, mainly because they are horizontally scalable and highly resilient to failures of individual machines.

While the above implementation is functionally sufficient, we observed that the frontend throughput if implemented as-is is low. Next, we discuss the optimizations to overcome the performance bottlenecks.

- *Separating logic from client-facing servers:* When a client-facing server queries the per-group control logic, the client-facing server is effectively blocked, which significantly reduces the throughput of client-facing servers. To remove this bottleneck, we add an intermediate process in each client-facing server to decouple querying control logic from responding requests. It frequently (by default every half second) pulls the fresh decision of each group from per-group logic and writes the decision in a local file of the client-facing server. Thus, the client-facing server can find the most up-to-date decisions from local cache without directly querying the control logic.
- *Replacing features with group identifier:* We found that when the number of session features increases, Spark Streaming has significantly lower throughput as it takes too long to copy the new measurement data from client-facing servers to Kafka and from Kafka to RDDs of Spark Streaming. This is avoidable, because once the features are mapped to a group by the client-facing servers, the remaining operations (updating and querying per-group logic) are completely feature-agnostic, so we can use group ID as the group identifier and remove all features from messages.

Backend: Figure 7c shows the key components and interfaces of the backend cluster. Once client-facing servers receive the measurement data from sessions, they will forward the measurement data to the backend clus-



(a) API to application sessions (b) Frontend (c) Backend

Figure 7: ***Key components and interfaces of Pytheas implementation.***

ter. On receiving these measurement data, the backend stores them in an HDFS for history data, and periodically (by default every 10 minutes) runs the session-grouping logic (§5.1) as a Spark [10] job to learn the session-group mapping and group-cluster mapping from the stored history data. These tables are sent to each frontend through Kafka, so that the future messages (requests and measurement data) from sessions will be matched against new tables. In addition, to detect failures of frontend clusters, each frontend cluster sends a small heartbeat message to the backend cluster every 5 seconds.

8 Evaluation

To evaluate Pytheas, we run our prototype [9] across multiple instances in CloudLab [4]. Each instance is a physical machine that has 8 cores (2.4 GHz) and 64GB RAM. These instances are grouped to form two frontend clusters and one backend cluster (each includes 5 to 35 instances). This testbed is an end-to-end implementation of Pytheas described in §7[5].

By running trace-driven evaluation and microbenchmarks on this testbed deployment, we show that:

- In the use case of video streaming, Pytheas improves the mean QoE by up to 6-31% and the 90th percentile QoE by 24-78%, compared to a prediction-based baseline (§8.1).
- Pytheas is horizontally scalable and has similar low response delay to existing prediction-based systems (§8.2).
- Pytheas can tolerate failures on frontend clusters by letting clients fall back to local logic and rapidly recovering lost states from other frontends (§8.3).

8.1 End-to-end evaluation

Methodology: To demonstrate the benefit of Pytheas on improving QoE, we use a real-world trace of 8.5 million video sessions collected from a major video streaming sites in US over a 24-hour period. Each video session can choose one of two CDNs. The sessions are replayed in the same chronological order as in the trace. We call a group of sessions a *unit* if they match values on AS, city, connection type, player type and content name.[6] We assume that when a video session is assigned to a CDN, its QoE would be the same to the QoE of a session that is randomly sampled from the same unit who use the same CDN in the same one-minute time window. For statistical confidence, we focus on the units which have at least 10 sessions on each CDN in each one-minute time windows for at least ten minutes. We acknowledge that our trace-driven evaluation has constraints similar to the related work, such as the assumption that QoE in a small time window is relatively stable in each unit (e.g., [24]) and a small decision space (e.g., [25]).

For each video session in the dataset, we run a DASH.js video player [5], which communicates with Pytheas using the API described in Figure 7a. To estimate the QoE of a video session, the video player does not stream video content from the selected CDN. Instead, it gets the QoE measurement from the dataset as described above.

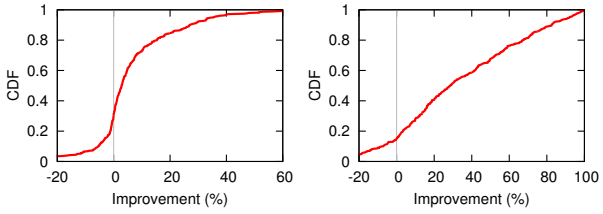
We use CFA [25] as the prediction-based baseline. It is implemented based on [25]. CFA updates QoE prediction in the backend every minute, and trains critical features every hour. The frontend clusters run a simple decision-making algorithm – for 90% sessions, it picks the decision that has the best predicted QoE, and for the rest sessions, it randomly assigns them to a CDN.

We consider two widely used video QoE metrics [17, 25]: join time (the start-up delay of a video session), and buffering ratio (the fraction of session duration spent on rebuffering). We define improvement of Pytheas for a particular unit at t-th minute by $Improve_{Pytheas}(t) = \frac{Q_{CFA}(t) - Q_{Pytheas}(t)}{Q_{CFA}(t)}$, where $Q_{Pytheas}(t)$ and $Q_{CFA}(t)$ are the average QoE of Pytheas in t-th minute and that of the baseline, respectively. Since we prefer smaller values on both metrics, a positive value means Pytheas has better QoE.

Overall improvement: Figure 8 shows the distribution of improvement of Pytheas across all sessions. We can

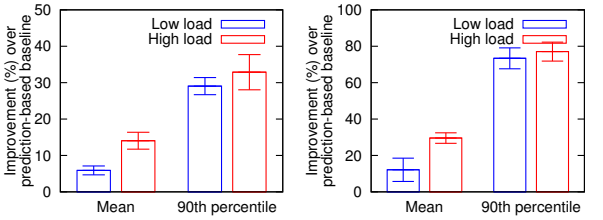
[5]Pytheas can use standard solutions such as DNS redirection to map clients to frontend clusters, and existing load balancing mechanisms provided by the host cloud service to select a frontend server instance for each client.

[6]The notion of unit is used to ensure statistical confidence of QoE evaluation, and is not used in Pytheas.



(a) Join time (b) Buffering ratio

Figure 8: ***Distribution of improvement of Pytheas over the prediction-based baseline.***



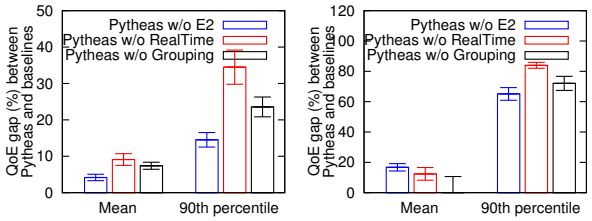
(a) Join time (b) Buffering ratio

Figure 9: ***Improvement in presence of load effect.***

see that the mean improvement is 6% for join time and 31% for buffering ratio, and the 90th percentile improvement is 24% for join time and 78% for buffering ratio. To put these numbers in context, prior studies show a 1% decrease in buffering can lead to more than a 3-minute increase in expected viewing time [17]. Note that Pytheas is not better than the baseline on every single session, because the E2 process inherently uses a (dynamic) fraction of traffic to explore suboptimal decisions.

Impact of load-induced QoE degradation: We consider the units where QoE of a CDN could significantly degrade when most sessions of the unit are assigned to use the same CDN. We assume that the QoE of a session when using a CDN under a given load (defined by the number of sessions assigned to the CDN in one minute) is the same to the QoE of a session randomly chosen in the dataset which uses the same CDN when the CDN is under a similar load. Figure 9 shows that the improvement of Pytheas when the number of sessions is large enough to overload a CDN is greater than the improvement when the number of sessions is not large enough to overload any CDN. This is because the prediction-based baseline could overload a CDN when too many sessions are assigned to the same CDN before CFA updates the QoE prediction, whereas Pytheas avoids overloading CDNs by updating its decisions in real time.

Contribution of Pytheas ideas: Having shown the overall benefit of Pytheas, we now evaluate the contribution of different components of Pytheas: (1) E2; (2) real-time update; and (3) grouping. We replace certain pieces of Pytheas by baseline solutions and compare their QoE with Pytheas's QoE. Specifically, for (1), we replace the



(a) Join time (b) Buffering ratio

Figure 10: ***Factor analysis of Pytheas ideas***

per-group E2 logic by CFA's decision-making logic; for (2), we run Pytheas with data of one-minute staleness; and for (3), we run the same E2 process over all sessions, rather than on per-group basis. Figure 10 shows the improvement of Pytheas over each baseline, and it shows that each of these ideas contributes a nontrivial improvement to Pytheas; about 10-20% improvement on average QoE and 15-80% on the 90th percentiles.

8.2 Microbenchmarks

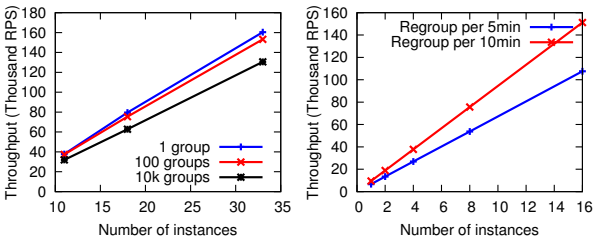
We create micro-benchmarks to evaluate the scalability and bandwidth consumption of Pytheas, as well as the benefits of various performance optimizations (§7).

8.2.1 Scalability

Frontend: Figure 11a shows the maximum number of sessions that can be served in one second, while keeping the update interval of per-group logic to be one second. Each session makes one control request and uploads QoE measurement once. Each group has the same amount of sessions. The size of control request message is 100B. We run Apache Benchmark [6] for 20 times and report the average throughput. We can see that the throughput is almost horizontally scalable to more frontend server instances. While the number of groups does impact the performance of frontend cluster, it is only to a limited extent; throughput of handling 10K groups is about 10% worse than that of handling one group.

Next, we evaluate the performance optimizations described in §7. We use a frontend cluster of 32 instances. Figure 12 shows that by separating E2 logic from client-facing servers, we can achieve 8x higher throughput, because each request reads cached decisions, which are still frequently updated. By replacing features by group identifiers, we can further increase throughput by 120%, because we can copy less data from client-facing servers to the servers running E2 logic. Note these results do not merely show the scalability of Spark Streaming or web servers; they show that our implementation of Pytheas introduces minimal additional cost, and can fully utilize existing data analytics platforms.

Backend: Figure 11b shows the maximum number of sessions that can be served in each second by a backend cluster, while keeping the completion time of session-



(a) Frontend (b) Backend

Figure 11: ***Pytheas throughput is horizontally scalable.***

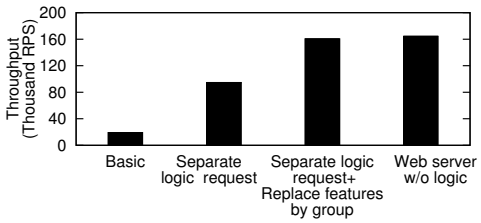


Figure 12: ***Optimizations of frontend throughput.***

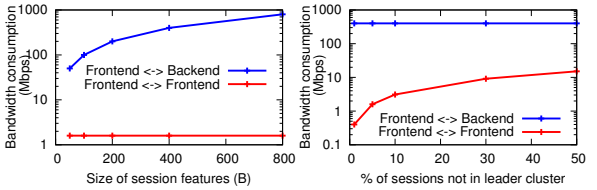
grouping logic within 5 minutes or 10 minutes. We see that the throughput is also horizontally scalable with more instances in the backend cluster.

To put the scalability numbers of both frontend and backend in context, let us consider a content provider like YouTube which has 5 billion sessions per day [11] (i.e., 57K sessions per second). Pytheas can achieve this throughput using one frontend cluster of 18 instances and a backend cluster of 8 instances, which is a tiny portion compared to the sheer number of video servers (at least on the magnitude of hundreds of thousands [7]). This might make Spark Streaming and Kafka an overkill for Pytheas, but the scale of data rate can easily increase by one to two magnitudes in real world, e.g., tens of GB/s; for instance, each video session can request tens of mid-stream decisions during an hour-long video, instead of an initial request.

8.2.2 Bandwidth consumption

Since the inter-cluster bandwidth could be costly [42, 33], we now evaluate the inter-cluster bandwidth consumption of Pytheas. We consider one session group that has one proxy cluster and one leader cluster.

First, we evaluate the impact of message size. We set the fraction of sessions received by the proxy cluster to be 5% of the group, and increase the request message size by adding more features. Figure 13a shows that the bandwidth consumption between the frontend clusters does not grow with larger message size, because the session features are replaced by group identifiers by the client-facing servers. Only the bandwidth consumption between frontend and backend grows proportionally with the message size but such overhead is inherent in existing data collection systems and is not caused by Pytheas.



(a) Impact of message size (b) Impact of % sessions not in the leader cluster

Figure 13: ***Bandwidth consumption between clusters.***

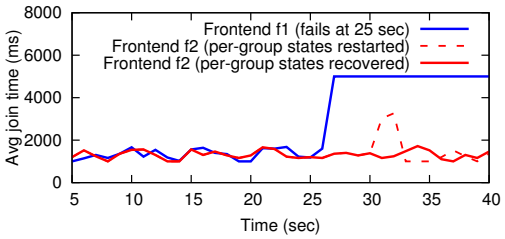


Figure 14: ***Pytheas can tolerate loss of a frontend cluster by falling back to player native logic gracefully, and recovering the logic states in a new cluster.***

Next, we evaluate the impact of fraction of sessions received by the proxy cluster. We set the message size to be 400B, and change the fraction of sessions received by each proxy cluster. Figure 13a shows that the bandwidth consumption between frontend clusters raises as more measurement data need to be forwarded from proxy to the leader cluster, but it is much smaller than the bandwidth consumption between frontend and backend.

8.3 Fault tolerance

Finally, we stress test the prototype under the condition that a leader frontend cluster fails. We set up 10 video players, each of which can stream content from two CDNs. CDN1 has 5000ms join time and CDN2 has 1000ms join time. By default, the player's native logic chooses CDN1. There are two frontend clusters, f1 and f2. The experiment begins with f1 being the leader cluster, and it loses connection at $t = 25$.

Figure 14 shows the time-series of QoE of sessions that are mapped to each frontend clusters. First, we see that the sessions mapped to f1 can fall back to the CDN chosen by the player's native logic, rather than crashing. Second, right after f1 fails, f2 should still be able to give cached decision (made by f1 before it fails) to its sessions. At $t = 30$, the backend selects f2 as the new leader for the group. At the point, a naive way to restart per-group logic in the new leader is to start it from scratch, but this will lead to suboptimal QoE at the beginning (the dotted line between $t = 30$ and $t = 35$). Pytheas avoids this cold-start problem by keeping a copy of the per-group states in the proxy cluster. This allows the proxy cluster to recover the per-group control states

without QoE degradation.

9 Related work

Data-driven QoE optimization: There is a large literature on using data-driven techniques to optimize QoE for a variety of applications, such as video streaming (e.g., [19, 25]), web service (e.g., [30, 39]), Internet telephony [21, 24], cloud services (e.g., [28]), and resource allocation (e.g., [15]). Some recent work also shows the possibility of using measurement traces to extrapolate the outcome of new system configurations [12]. Unlike these prediction-based approaches, we formulate QoE optimization as an real-time E2 process, and show that by avoiding measurement bias and enabling real-time updates, this new formulation achieves better QoE than prediction-based approaches.

Related machine learning techniques: E2 is closely related to reinforcement learning [43], where most techniques, including the per-group E2 logic used in Pytheas, are variants of the UCB1 algorithm [14], though other approaches (e.g., [13]) have been studied as well. Besides E2, Pytheas also shares the similar idea of clustering with linear mixed models [31], where a separate model is trained for each cluster of data points. While we borrow techniques from this rich literature [20, 35], our contribution is to shed light on the link between QoE optimization and the techniques of E2 and clustering, to highlight the practical challenges of adopting E2 in network applications, and to show group-based E2 as a practical way to solve these challenges. Though there have been prior attempts to cast data-driven optimization as multi-armed bandit processes in specific applications (e.g., [24]), they fall short of a practical system design.

Geo-distributed data analytics: Like Pytheas, recent work [33, 34, 42] also observes that for cost and legal considerations, many geo-distributed applications store client-generated data in globally distributed data centers. However, they focus on geo-distributed data analytics platforms that can handle general-purpose queries received by the centralized backend cluster. In contrast, Pytheas targets a different workload: data-driven QoE optimization uses a specific type of logic (i.e., E2), but has to handle requests from millions of geo-distributed sessions in real time.

10 Discussion

Handling flash crowds: Flash crowds happen when many sessions join at the same time and cause part of the resources (decisions) to be overloaded. While Pytheas can handle load-induced QoE fluctuations that occur in individual groups, overloads caused by flash crowds are different, in that they could affect sessions in multiple groups. Therefore, those affected sessions need to be regrouped immediately, but Pytheas does not support such real-time group learning. To handle flash crowds, Pytheas needs a mechanism to detect flash crowds and create groups for the affected sessions in real time.

Cost of switching decisions: The current design of Pytheas assumes there is little cost to switch the decision during the course of a session. While such assumption applies to today's DASH-based video streaming protocols [8], other applications (e.g., VoIP) may have significant cost when switching decisions in the middle of a session, so the control logic should not too sensitive to QoE fluctuations. Moreover, a content provider pays CDNs by 95th percentile traffic, so Pytheas must carefully take the traffic distribution into account as well. We intend to explore decision-making logic that is aware of these costs of switching decisions in the future.

11 Conclusions

With increasing demands of user QoE and diverse operating conditions, application providers are on an inevitable trajectory to adopt data-driven techniques. However, existing prediction-based approaches have key limitations that curtail the potential of data-driven optimization. Drawing on a parallel from machine learning, we argue that real-time exploration and exploitation is a better abstraction for this domain. In designing Pytheas, we addressed key practical challenges in applying real-time E2 to network applications. Our key insight is a *group-based E2* mechanism, where application sessions sharing the same features can be grouped so that we can run E2 at a coarser per-group granularity. Using an end-to-end implementation and proof-of-concept deployment of Pytheas in CloudLab, we showed that Pytheas improves video quality over state-of-the-art prediction-based system by 6-31% on mean, and 24-78% on tail QoE.

Acknowledgments

We appreciate the feedback by the anonymous NSDI reviewers, and thank Anirudh Badam for the caring and diligent shepherding of the paper. Junchen Jiang was supported in part by NSF award CNS-1345305 and a Juniper Networks Fellowship.

References

[1] Apache HTTP Server Project. `https://httpd.apache.org/`.

[2] Apache Kafka. `https://kafka.apache.org/`.

[3] Apache Spark Streaming. `http://spark.apache.org/streaming/`.

[4] CloudLab. `https://www.cloudlab.us/`.

[5] dash.js. `https://github.com/Dash-Industry-Forum/dash.js/wiki`.

[6] How a/b testing works. http://goo.gl/SMTT9N.

[7] How many servers does youtube. https://atkinsbookshelf.wordpress.com/tag/how-many-servers-does-youtube/.

[8] Overview of mpeg-dash standard. http://dashif.org/mpeg-dash/.

[9] Source code of Pytheas. https://github.com/nsdi2017-ddn/ddn.

[10] Spark. http://spark.incubator.apache.org/.

[11] Youtube statistics. http://fortunelords.com/youtube-statistics/.

[12] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 2016.

[13] A. Agarwal, D. Hsu, S. Kale, J. Langford, L. Li, and R. Schapire. Taming the monster: A fast and simple algorithm for contextual bandits. In *Proceedings of The 31st International Conference on Machine Learning*, pages 1638–1646, 2014.

[14] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3), 2002.

[15] Y. Bao, X. Liu, and A. Pande. Data-guided approach for learning and improving user experience in computer networks. In *Proceedings of The 7th Asian Conference on Machine Learning*, pages 127–142, 2015.

[16] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[17] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. A. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. SIGCOMM*, 2011.

[18] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[19] A. Ganjam, F. Siddiqi, J. Zhan, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-scale control plane for video quality optimization. In *NSDI*. USENIX, 2015.

[20] A. Garivier and E. Moulines. On upper-confidence bound policies for non-stationary bandit problems. *arXiv preprint arXiv:0805.3415*, 2008.

[21] O. Haq and F. R. Dogar. Leveraging the Power of Cloud for Reliable Wide Area Communication. In *ACM Workshop on Hot Topics in Networks*, 2015.

[22] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[23] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: evidence from a large video streaming service. In *ACM SIGCOMM 2014*.

[24] J. Jiang, R. Das, G. Anathanarayanan, P. Chou, V. Padmanabhan, V. Sekar, E. Dominique, M. Goliszewski, D. Kukoleca, R. Vafin, and H. Zhang. Via: Improving Internet Telephony Call Quality Using Predictive Relay Selection. In *ACM SIGCOMM*, 2016.

[25] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang. CFA: a practical prediction system for video QoE optimization. In *Proc. NSDI*, 2016.

[26] J. Jiang, V. Sekar, I. Stoica, and H. Zhang. Unleashing the potential of data-driven networking. In *Proceedings of 9th International Conference on COMmunication Systems & NETworkS (COMSNET)*, 2017.

[27] L. Kontothanassis, R. Sitaraman, J. Wein, D. Hong, R. Kleinberg, B. Mancuso, D. Shaw, and D. Stodolsky. A transport layer for live streaming in a content delivery network. *Proceedings of the IEEE*, 92(9):1408–1419, 2004.

[28] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.

[29] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670. ACM, 2010.

[30] H. H. Liu, R. Viswanathan, M. Calder, A. Akella, R. Mahajan, J. Padhye, and M. Zhang. Efficiently delivering online services over integrated infrastructure. In *Proc. NSDI*, 2016.

[31] C. E. McCulloch and J. M. Neuhaus. *Generalized linear mixed models*. Wiley Online Library, 2001.

[32] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang. Practical, real-time centralized control for cdn-based live video delivery. In *ACM SIGCOMM*, pages 311–324. ACM, 2015.

[33] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *Proc. SIGCOMM*, 2015.

[34] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proc. NSDI*, 2014.

[35] P. Rigollet and A. Zeevi. Nonparametric bandits with covariates. In *Proc. Conference on Learning Theory*, 2010.

[36] S. Seshan, M. Stemm, and R. H. Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, pages 1–13, 1997.

[37] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *Proc. NSDI*, 2016.

[38] A. Slivkins. Contextual bandits with similarity information. *The Journal of Machine Learning Research*, 15(1):2533–2568, 2014.

[39] M. Stemm, R. Katz, and S. Seshan. A network measurement architecture for adaptive applications. In *INFOCOM 2000*. IEEE.

[40] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli. CS2P: Improving Video Bitrate Selection and Adaptation with Data-Driven Throughput Prediction. In *to appear in Proc. SIGCOMM*, 2016.

[41] R. Torres, A. Finamore, J. R. Kim, M. Mellia, M. M. Munafo, and S. Rao. Dissecting video server selection strategies in the youtube cdn. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 248–257. IEEE, 2011.

[42] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 323–336, 2015.

[43] R. Weber et al. On the gittins index for multiarmed bandits. *The Annals of Applied Probability*, 2(4):1024–1033, 1992.

[44] Y. Zhang and N. Duffield. On the constancy of internet path properties. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 197–211. ACM, 2001.