QoE-Fair Video Streaming over DASH

Using Reinforcement Learning

uOttawa

Sa'di Altamimi

School of Electrical Engineering and Computer Science
University of Ottawa

Thesis submitted in partial fulfillment of the requirements for the
Master of Applied Science
degree in
Electrical & Computer Engineering

I would like to dedicate this thesis to
the Palestinian child
Handala!

Acknowledgements

I would like to express my gratitude to my supervisor Prof. Shervin Shirmohammadi for his continuous support, trust, flexibility, and immense knowledge. Without his guidance, this achievement would not have been possible. I would also like to thank the rest of my thesis committee: Prof. Amiya Nayak, and Prof. Peter Liu, for their insightful comments and constructive feedback.

Besides my supervisor, I would like to thank two inspiring professors: Prof. Martin Bouchard and Prof. Marcel Turcotte. Their dedication, organization, enthusiasm and hard work shaped my thoughts and allowed me to experience, discover, and understand how to pursue my dream in becoming a successful professor in the future.

Additionally, I would like to thank Kifah and his wife Ala' for being my "second" family, and my friends Shady, Rumeysa, and Hamideh who made Canada feels like home!

Last but not least, I am profoundly grateful to my dedicated parents Younes and Sabah, my in-laws Anas and Amal, my amazing wife Jana, and all my family including Mohammad, Samar, Basel and Azeeza for their strong and continuous support which propelled me to complete this work.

Abstract

Video streaming has become, and is expected to remain, the dominant type of traffic over the Internet. With this high demand for multimedia streaming, there is always a question on how to provide acceptable and fair Quality of Experience (QoE) for consumers of the over-the-top video services, despite the best-effort nature of the Internet and the limited network resources, shared by concurrent users. MPEG-DASH, as one of the most widely used standards of HTTP-based adaptive streaming, uses a client-side rate adaptation algorithms; which is known to suffer from two practical challenges: in one hand, clients use fixed heuristics that have been fine-tuned according to strict assumptions about deployment environments which limit its ability to generalize across network conditions. On the other hand, the absence of collaboration among DASH clients leads to unfair bandwidth allocation, and typically ends up in an unbalanced equilibrium point

We believe that augmenting a server-side rate adaptation significantly improves the fairness of network bandwidth allocation among concurrent users. We have formulated the problem as a Decentralized Partially Observable Markov Decision Process (Dec-POMDP) model, and used RL to train two neural networks to find an optimal solution to the proposed Dec-POMDP problem in a distributed way. We showed that our proposed client-server collaboration outperforms the state-of-the-art schemes in terms of QoE-efficiency, QoE-fairness, and social welfare by as much as 16%, 21%, and 24% respectively.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In the last 15 years, the data rates of both mobile and fixed networks have improved dramatically. This paved the way for the rise of video traffic, which has rapidly become the most predominant type of traffic over the Internet. Cisco Visual Networking Index [3] predicted that video content will account for 82% of all Internet traffic by 2021. As indicated in Fig. 1.1, live Internet video and video surveillance will account for 13% and 3.4% of Internet video traffic respectively. However, the major fraction of this video traffic comes from video-on-demand (VOD) and over-the-top (OTT) video services.

Examples of these services are YouTube and Netflix, where the goal is to stream videos in continuous manner while providing the customers with the highest quality of experience (QoE) possible. From a network perspective, this requires high bandwidth to minimize packet-loss and insure good quality of service (QoS). It is known that QoS and QoE are different concepts even though they are closely related [49]. Moreover,

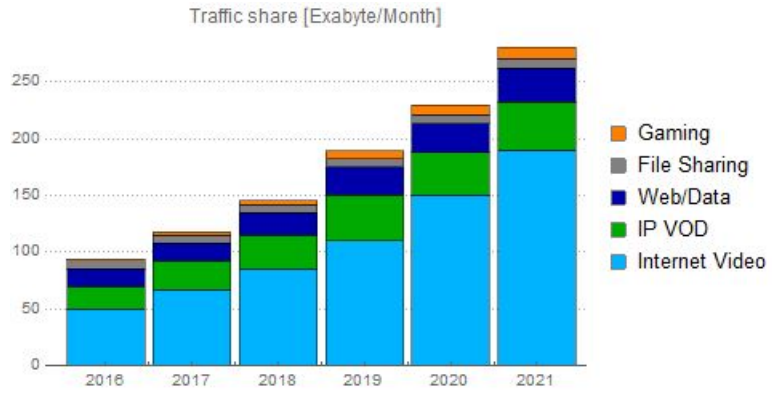


Fig. 1.1 Predicted Internet traffic by category (source [3])

when multiple video streams concurrently run on the network, potentially competing for the limited bandwidth, achieving the best QoE becomes a challenging task.

In 2011, Dynamic Adaptive Streaming over HTTP (DASH) standard has emerged. DASH is solely a client-side standard, where the client is the only agent that manages the video streaming process. However, since each DASH client has access only to local information about the network and unaware of actions taken by the other clients, a DASH client can suffer from three performance issues: (1) player instability, (2) unfairness between the players and (3) the under-utilization of the available bandwidth [34].

1.1 Motivations

Adaptive bitrate algorithms (ABR) are the primary tool that content providers use to optimize video quality. Traditionally, these algorithms are implemented in a distributed fashion in which they run on client-side video players and dynamically choose a bitrate for each video chunk (e.g., 4-second block) based on various observations; such as the estimated network throughput and playback buffer occupancy. However, maximizing the bandwidth utilization does not necessarily result in an optimum video quality. End-user's QoE, rather than QoS, should be the final goal of any network service.

In order to maximize the user's QoE, a solution for video rate adaptation is needed for congestion avoidance and bandwidth sharing among multiple video streams. Such solution can be distributed and end-node-driven, or centralized and network-assisted. In either case, it is important to take into account the multi-agent nature of the problem. For example, due to the dynamic characteristics of the network, any changes in users' current bitrates would affect the state of the network and all other users, which could in turn affect the QoE of the original user in subsequent times. However, most existing approaches use a stochastic passive model to represent network behavior as an aggregation of all other agents, neglecting the essence of interaction between different users of the network. This negatively affects the overall performance. Therefore, it is important to consider not only the instantaneous multimedia quality, but also how the immediate adaptation impacts the long-run expected quality in future.

Selecting the right bitrate for individual clients in a multi-agent setup is challenging in practice. A successful solution should consider the following challenges:

- Networks' states can be highly variable, unpredictable, and unobservable. This leads to inaccurate throughput predictions. Despite that, ABR algorithms must try to achieve maximum QoE possible.
- Due to dynamic characteristics of the network, bitrate decisions for one video chunk can have cascading effects on later chunks. As a result, ABR algorithms must take into account the long-term effects of its decisions by proactively plan for future rather than relaying on instantaneous rates.
- QoE objectives (e.g., high bitrates, minimal rebuffering, smoothness) are inherently at odds with one another. Moreover, users have different QoE preferences; some want the highest bitrate, while others are intolerant to rebuffering. ABR must deal with this conflicting nature of QoE requirements.
- Maximizing bandwidth utilization or avoiding network congestion does not necessarily result in an optimum video quality. ABR algorithms must consider optimizing QoE and QoS through joint actions at transport layer and application layer. Also, it is important to take into account the multi-agent nature of the problem, since fairness is as important as efficiency when it comes to customer satisfaction.
- Videos are typically encoded at a handful of bitrates. Thus, ABR algorithms are restricted to a small set of choices and cannot make fine-grained control decisions. Despite that, ABR algorithms should learn to behave optimally under this constraint.

1.2 Approach

We followed the same approach developed in [19], where our adaptive video streaming problem is formulated as a decision making problem with Markov property and partially observable information about the network state. We attempted to address all of the challenges mentioned in Section 1.1 by implementing a quality-driven fairness-aware end-to-end congestion control and bandwidth sharing mechanism. Our proposed RL-based algorithm learns optimal ABR rules automatically, without using any pre-programmed models or explicit assumptions about the operating environment. During training, the agent starts knowing nothing about the task at hand. It then gradually learns to make better ABR decisions through reinforcement, in the form of reward signals that reflect video QoE for past decisions. Our QoE model incorporate the

notion of a *social welfare* function that combines the main objectives of efficiency and fairness. By optimizing its control policy based on the actual performance of past choices, the agent discovers policies that are much more robust than those used by traditional algorithms which rely on accurate system models and fixed heuristics.

To train our agent, we used two datasets: (i) one contains a large corpus of network traces to faithfully emulate realistic network conditions. (ii) the other contains different genre to cover differences in scene characteristic. We then evaluated our algorithm using a full system implementation. Once the agent learned the optimal policy, we deployed the neural network model on an ABR server. The server collaborates with the clients to optimize the social welfare function. This design is compatible with the DASH standard since it doesn't require any modification on the client-side.

1.3 Contributions

In this work, we use the theoretic framework developed in [20] to implement a reinforcement learning (RL) based server-side adaptive video streaming algorithm. The goal is to achieve a client-server cooperation to maximize fairness and efficiency without modifying DASH components and algorithms on the client-side (i.e. maintain the compatibility with DASH standard). We summarize our contributions as follows:

- We design and implement an Asynchronous Advantage Actor-Critic (A3C) algorithm to find the optimal solution of the proposed DEC-POMDP [20] model. Our algorithm outperformed the proposed SARSA algorithm as well as all other state-of-the-art algorithms in terms of QoE-efficiency, QoE-fairness, and social welfare by as much as 16%, 21%, and 24%respectively.

- We extend the proposed framework from being a server-side algorithm to allow for better client-server cooperation. This is challenging since unfairness is mainly caused by the clients' quality adaptation algorithms themselves [5]. However, allowing clients to run their own ABR mechanism (under server "supervision") helps the system scale without overloading the server.

- We suggest an improvement to the reward signal to reflect users' QoE preferences (i.e. we implemented a personalized QoE model). According to [30], users can have different QoE preferences; some want the highest bitrate, while others are intolerant to rebuffering, or prefer smooth playback. The parameters of the reward model are learnable (tunable) by our agent.

- We resolve three important implementation issues: (1) We left restrictions on newly arriving and leaving flows; so that the number of clients is not required to stay fixed during the streaming session. (2) We improved the learning speed by using a parallel and distributed learning algorithm. (3) We used a more realistic asynchronous setting, where actions (rate selection) of different agents are not forced to occur at synchronized time instances.

1.4 Organization of Thesis

In the next chapter we review some theoretical background from areas related to the topic of this project. It includes some details about DASH standard, RL, and QoE metrics. The reviewed algorithms were divided into three groups based on whether they used fixed or learnable heuristic, and whether they were implemented on client-side or server-side. We devoted one section for RL-based algorithm and explained how they are related/different from our proposed algorithm.

In Chapter 3, we present a survey of the recent developments in adaptive video streaming. The related works have been reviewed in three categories based on the approach taken by the authors: traditional approach of using fixed heuristics on the client-side, fairness-aware server-side approach, and RL-based approach. Chapter 4 contains the system design and implementation which details all our contributions. A summary of the obtained results, and some directions for future research in this area appear in Chapter 5, and 6 respectively.

Chapter 2

Background

This chapter provides an overview of the systems and techniques underlying the whole work. It first presents the DASH standard, and the video bitrate adaptation mechanism it supports, with a review of the most important video QoE issues. Then, it introduces the basic theoretical concepts of RL, specifying the equations behind the most common RL algorithms as well as the motivations for using RL in adaptive video streaming context.

2.1 DASH Video Streaming

Adaptive video streaming is a technique to dynamically change the video bitrate, adapting to network conditions. The Dynamic Adaptive Streaming over HTTP (DASH) standard was published in 2011, and has rapidly become well-adopted: the newest version of the Hyper-Text Markup Language, HTML5, supports JavaScript-based DASH players as a way to embed video in webpages without using external plugins. Before we dive deep into the DASH standard itself in the following subsections, it's worth noting how video streaming was evolved overtime from TCP to UDP and back to TCP again thanks to DASH.

Due to the overhead issues of the Transmission Control Protocol (TCP), video streaming research in the late 1990s and early 2000s mostly focused on the User Datagram Protocol (UDP) [35]. However, this has changed and TCP-based streaming came back to the picture. The reason for this shift is mostly *compatibility*: the existing infrastructure of servers and network switches is based on TCP traffic since most of the Internet relies on the HyperText Transfer Protocol (HTTP) which runs over it.

Fig. 2.1 Freezing in TCP and error artifacts in UDP

One major advantage for TCP protocol over UDP-based video streaming is that TCP effectively eliminates any error artifacts, as the transmission is reliable. However, freezes are the major quality issue in HTTP streaming, as they are due to *delays* that are hard to control in HTTP/TCP. Fig. 2.1 illustrates both effects.

The *overhead* cost of HTTP-based streaming was an acceptable price for efficient caching, no firewall problems, and easier deployment of Content Delivery Networks (CDNs). These features allow copies of the media content to be stored efficiently in a geographically distributed manor on "local" cache servers that are close to the client. This greatly reduces time delay in TCP-based video streaming. To tackle overhead issues, a promising solution has emerged through the newly publish HTTP/2 protocol. A useful feature in this context is the server's ability to *push* multiple contents to the client without making individual request for each content [45]. This feature is helpful in the context of DASH streaming because DASH fragments media contents and configuration files into small segments and fetch them upon request.

2.1.1 DASH Data Model

In a conventional DASH application, several copies of every video at different bitrates are stored on the streaming server. The DASH client requests an XML configuration file called Media Presentation Description (MPD), containing all the metadata that the client needs in order to stream the video. Among these data is the list of available encoding bitrates supported for the media content being requested. The client observes the channel available bandwidth, and streams media content at a suitable bitrate to ensure smooth streaming and avoid rebuffering. Fig. 2.2 shows the general structure of an MPD file.

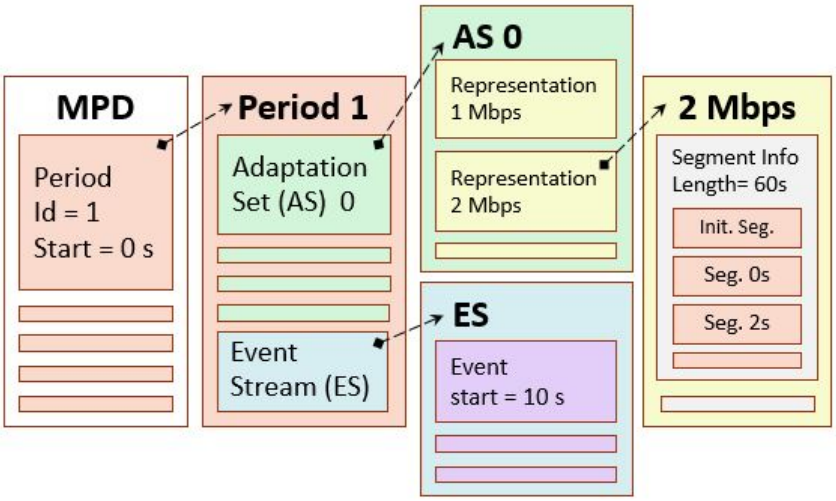


Fig. 2.2 MPD structure according to DASH.

According to Fig 2.2, the actual media content is divided into segments. Each segment is encoded into different representations (i.e. bitrates), whose number is constant throughout the video (or at least a video period). Clients can choose among different representations in the same adaptation set, as they effectively represent the same content. This allows the user to choose the compression level for each video segment and to adapt the requested video bitrate to the available channel rate [43]. Multimedia components that logically belong together are grouped into adaptation sets, for example, videos of different codecs or audios of different languages are separated into different adaptation sets.

Periods are usually used to separate the media content, e.g., for ad insertion. This process can be done dynamically through *Event Stream* (ES) element in the MPD file, which in turn contains *Event* elements that reference ranges in the timeline. When the playhead hits one of these ranges, the event fires, and delivers the payload of the underlying Event element. In the case of ad-insertion, the server separates the media content into two periods and adds the new ad content as a new period in between. The client, then, requests a new MPD file and starts streaming the new segments.

The different segments are identified by their URLs, and may be either in separated files or within the same file (using byte offsets to separate them); using separate files allows for more efficient caching, so it is often preferred. Each segment must begin with a Stream Access Point (SAP), allowing the client to decode and play it without any previous information. This is necessary to avoid errors during quality switches,

as the previous segment may be part of a different representation and have different characteristics. The length of these segments is a result of an application-dependent trade-off between reaction speed to changing network conditions and network overhead. The longer the segment the less the network overhead, and the shorter the segment the faster the switching; which is desired in highly variable bandwidth conditions.

In the case of live streaming, the MPD file can be fragmented and downloaded piece-wise as the new content appears on the web server. Given that the server supports live streaming, DASH standard allows the client to update the MPD file in the future by setting its *profile* type to "live" as opposed to "on-demand". Obviously, live streaming presents additional challenges, as both the information and the segments available to the client are limited by the real-time constraint.

2.1.2 DASH as Client-side standard

DASH provides a standard solution for the efficient and easy streaming of multimedia using existing available HTTP infrastructure (particularly servers and CDNs, but also proxies, caches, etc.). It enables the deployment of the streaming services using the existing low cost and wide-spread Internet infrastructure without any special provisions. It supports both on-demand and live streaming.

As DASH servers are basically standard HTTP servers, the video bitrate adaptation algorithm is entirely client-side; a client only needs to send the appropriate HTTP requests to implement any bitrate selection policy. This means that HTTP server does not need to cope with the additional load of stream adaptation management in large scale deployments. DASH also allows the client the choice of streaming from various CDNs, and therefore achieving further load-balancing of the network for the benefit of the client.

DASH clients are free to choose the quality for each segment from the available adaptation sets provided in their MPD file as shown in Fig. 2.3. Performing video bitrate adaptation aims at avoiding exceeding the available channel rate, which may otherwise lead to the most important source of QoE drops: rebuffering events. Whenever the video buffer empties, the client has to wait for the next segment to finish downloading, which results in stopping the video playback and annoying the user [29].

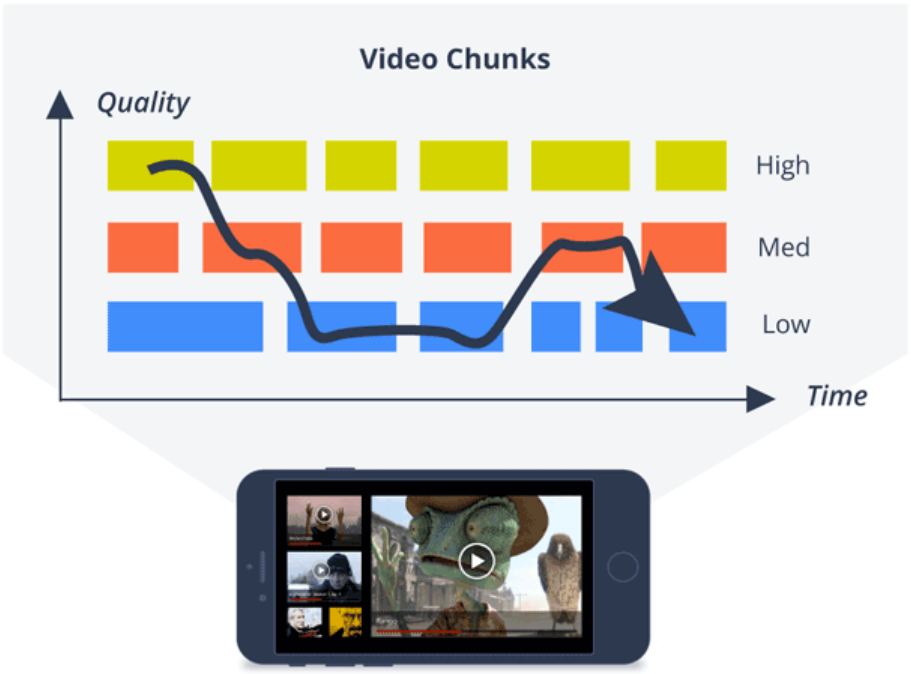


Fig. 2.3 Simplified DASH workflow [source: kaltura.com]

Most of the commercially available systems implement simple heuristics that try to stabilize the buffer level: intuitively, this avoids most rebuffering events by keeping a high buffer level, while increasing the quality every time the channel rate is high enough to fill the buffer over the desired level. These heuristics do not take QoE effects into account, and several studies have found them to be inefficient in terms of QoE optimization [6]. A possible performance enhancement to the DASH paradigm is *pipelined* DASH: a pipelined client can send more than one HTTP request at the same time, downloading either more than one segment or several parts of the same segment (using byte offset). Pipelining can be beneficial in mobile networks with high packet loss and latency [31].

Despite its wide popularity, DASH faces several challenges relevant to stability, fairness and efficiency, particularly when multiple clients compete for the same link, creating a bottleneck link scenario [21] [5]. This is mainly inherited from the fact that DASH is purely a client-side protocol. For example, to achieve *stability*, the client should detect the short term bandwidth variations; and accordingly avoid frequent short-term bitrate switches while maintaining the highest possible quality with respect to its available throughput (i.e. high *efficiency*) [32]. This is however very hard to

predict due to the limited information each client has about the environment. This becomes even harder when multiple clients compete on shared resources. In such scenarios, *fairness* among the connected clients is desired. All these challenges make the design and development of a robust DASH system highly difficult without any kind of client-server collaboration.

2.2 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm inspired by behaviorist psychology. The basic mechanism behind RL is to learn through trial and error, where an *agent* takes as input the *state* of the *environment*, and makes a choice about its *action*, which in turn results in some *reward* and a probabilistic *transition* to a new state of the environment. What distinguishes reinforcement learning from supervised learning is that only partial feedback is given to the learner about its actions as opposed to teach the agent through a *dataset* of state-action pairs. Furthermore, the actions may have long term effects through influencing the future state of the environment. Thus, time plays a special role, and learning is carried out through an *interactive* process over time.

The interaction between the agent and the environment is at the core of any RL problem. Finding a way to mathematically model both the environment and the agent is the first step to solve such problems. At one end, we need a mathematical model that captures the notions of states, and transition between states over time. On the other end, we need an algorithm that highlights the strategy the agent is following to interact with the environment (referred to *policy*), as well as a model to describe how to optimize this policy to achieve highest possible rewards. This is what the following two subsections are all about. When the agent learns the best (optimal) policy to follow in all states, we say that the RL problem has been solved.

2.2.1 Markov Decision Process

Markov Decision Process (MDP) [9], is the standard mathematical tool for decision making under uncertainty. It is a Markov process in a sense that the transition probability, from the current state to a next state, depends only on the current state and action regardless of all past (and future) states or actions. It is also a decision process in a sense that it is an extension of Markov chains where actions and rewards are added to the model to allow the agent to have partial control over environment

dynamics (i.e. take *decisions*). Formally speaking, a Markov Decision Process is defined as a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma >$ where

- $\mathcal{S}$ is the finite set of states s of the environment
- $\mathcal{A}$ is the finite set of actions available to the agent in state $s \in \mathcal{S}$
- $\mathcal{T}$ is the state transition function that provides $P(s_{t+1}|s_t, a_t)$; i.e. the probability of transition to a next state s_{t+1} given that the agent starts at state s_t and takes action $a_t \in \mathcal{A}$ at time step t.
- $\mathcal{R}$ is the immediate reward function which depends on the state of the environment and action of the agent.
- γ is an exponential discount factor on future rewards. In order to avoid divergence, it needs to be in the [0, 1] interval. For example, $\gamma = 0.99$ implies that current actions will be influenced by 100 future steps.

Fig. 2.4 illustrates the relationship between the agent and the environment. In this model, the next state and reward depend only on the current state and the action taken. The agent does not have a complete model of the environment or the effects of its actions, but it gradually learns how to maximize the reward by trying different actions at different states. A policy $\pi(s_t, a_t) : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is the function that links every state to an action by defining the probability of picking each action $a_t \in \mathcal{A}$ when being in state s_t. Different policies can be evaluated by calculating their associated *value functions* $V_\pi(s_t)$; a function represents how good it is to be in a state s_t and is equal to expected total reward starting from state s at time t and following the policy π to pick all actions in the future.

Once the environment has been modeled as an MDP, finding a solution means searching for an *optimal* policy. When the time window for the long-term reward is finite, and the MDP is relatively small and its states are fully available, standard dynamic programming techniques can find the optimal solution of MDPs. However, when the state space is not manageable by dynamic programming, RL algorithms can be used thanks to two key ideas: using *samples* to compactly represent the dynamics of the environment, and using *function approximation* to compactly represent the value functions $V_\pi(s)$. The first idea is important because it allows to deal with learning scenarios when the environment dynamics is unknown or very complex (i.e. model-free

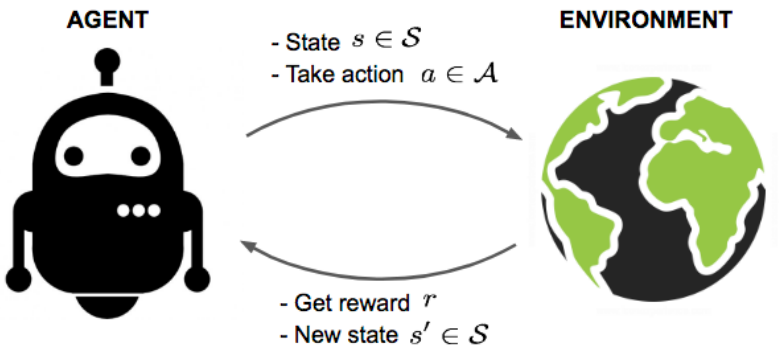


Fig. 2.4 The concept behind reinforcement learning.

algorithm). The significance of the second idea is that it allows dealing with large, high-dimensional state-spaces and action-spaces [42].

RL algorithms do not provide an exact solution, but they do not need complete knowledge of the MDP to converge. Partially Observable MDP (POMDP) framework is used to tackle decision problems with imperfect or unobservable state information [46]. When there are more than one active decision-maker in the environment, we are dealing with a multi-agent system. Decentralized Partially Observable Markov Decision Process (Dec-POMDP) framework is an extension of single-agent POMDP to a multi-agent cooperative setting. In a Dec-POMDP, the dynamics of the system and the objective function depend on the actions of all agents. However each agent needs to make decisions based on its local information and imperfect observation of the state. Formally speaking, a DEC-POMDP is defined by the following tuple $< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \mathcal{O}, O, \mathcal{I}, \gamma >$ where in addition to the standard element of an MDP we have:

- $\mathcal{O} = \mathcal{O}_1 \times \ldots \times \mathcal{O}_N$; is the finite set of joint observations $\boldsymbol{o} = < o_1, \ldots, o_N >$, where an individual observation of an agent n is denoted by $o_n \in \mathcal{O}_n$
- O is the observation function that provides $P(s_{t+1}|s_t, \boldsymbol{a})$, the probability that the agents receive joint observation $\boldsymbol{o}$ of the hidden state s_{t+1}, when they reach this state through joint action $\boldsymbol{a}$
- $\mathcal{I} \in \mathcal{P}(\mathcal{S})$ is the initial probability distribution of the state, where $\mathcal{P}(.)$ denotes the set of probability distributions over its arguments

Agent	Observation	
	Perfect	Imperfect
Single	MDP	POMDP
Multiple	MMDP	DEC-POMDP

Table 2.1 Different frameworks for decision making under uncertainty

Also, note that the original definition of MDP should be modified to adapt for joint nature of the multi-agent case. For example, the set of actions become $\mathcal{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_N$ which represent the finite set of joint actions of all agents $\boldsymbol{a} = < a_1, \ldots, a_N >$, where an individual action of an agent n is denoted by $a_n \in \mathcal{A}_n$. Table 2.1 captures the relation between different frameworks for decision making under uncertainty. Note that if the problem in hand has no decision making part, MDP becomes Markov Chain (MC), and POMDP becomes Hidden Markov Model (HMM).

Since environments in real world don't tend to follow MDP structures- where a given state conveys everything the agent needs to act optimally- it's essential to find a way to make sense of such limited, changing world. The key idea is to give the agent a capacity for *temporal integration* of observations. The intuition behind this is simple: if information at a single moment isn't enough to make a good decision, then enough varying information over time probably is. For example, if you see a picture with a thrown ball in it, a single image of a ball in motion tells us nothing about its direction, but two images in sequence allows us to infer the direction of movement. A longer sequence might even allow us to make sense of the speed of the ball. The same principle can be applied to problems defined under POMDP framework.

Within the context of RL, there are several possible ways to accomplish this temporal integration. One solution is to modify the environment and create some kind of buffer to store environment states over time, and then use this history to train our agent. This solution was applied by DeepMind in their original paper on Deep Q-Networks [28]; where they modified the Atari simulator. Instead of feeding the agent with a single frame at a time, they used an external frame buffer which kept the last four frames of the game in memory and fed this to the agent. This approach worked relatively well for the simple environment they employed, but it isn't ideal for a number of reasons. The first is that it isn't necessarily biologically plausible; when light hits our retinas, it does it at a single moment. There is no way for light to be stored up and passed

all at once to an eye. Secondly, by using blocks of multiple frames as their state, the state-space becomes larger and the training process requires a larger amount of potentially unnecessary memory. Lastly, we may simply need to keep things in mind that happened much earlier than would be feasible to capture with stacking frames. Sometimes an event hundreds of frames earlier might be essential to deciding what to do at the current moment.

A better solution can be achieved by moving the temporal integration into the agent itself allowing the agent to keep events in mind more robustly. This is accomplished by utilizing a recurrent neural network (RNN) in our agent architecture agent. By utilizing a recurrent block in our architecture, we can pass the agent a single frame of the environment, and the RNN will be able to change its output depending on the temporal pattern of observations it receives. It does this by maintaining a hidden state that it computes at every time-step. The recurrent block can feed the hidden state back into itself, thus acting as an augmentation which tells the network what has come before.

2.2.2 Learning Algorithm

Agents in RL follow some kind of a strategy called *policy* which tells the agent what action to do in each state. A policy $\pi(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ maps a state to probability distribution over actions; where actions that are believed to result in highest rewards will be picked most of the time. In theory, two types of policies can be found: *deterministic* policies and *stochastic* policies. However, we use the definition of stochastic policy to cover both cases by considering deterministic policies as a special case; where the probability of picking the best action at some state s_t is exactly 1. And the probability of picking all other actions at the same state is set to 0.

Among all possible policies, the goal is to learn the *optimal* policy $\pi_*(s_t, a_t)$ that outperforms all other policies or at least achieves "equal performance". This implies that the optimal policy doesn't have to be unique [12]. In practice, the agent starts with a random (arbitrary) policy, then gradually tries to improve it by observing reward signals until it converges to the optimal policy. The performance of different policies can be evaluated and compared by calculating their associated *value functions*. The main three functions available are:

- *action*-value function $Q(s_t, a_t)$: sometimes referred to as Q-function; indicates agent's belief of how good it is to take action a at state s at time t and then

follows policy π in the future.

$$Q(s_t, a_t) = \mathbb{E}_\pi[R_t \mid S = s_t, A = a_t] \tag{2.1}$$

- *state*-value function $V_\pi(s_t)$: indicates agent's belief of how good, on average, it is to be at state s at time t if it follows policy π in the future. It is given by

$$V_\pi(s_t) = \sum_{a \in \mathcal{A}} \pi(a_t|s_t)\, Q(s_t, a_t) \tag{2.2}$$

- *advantage* function $A_\pi(s_t, a_t)$: indicates agent's belief of how much more reward it expects should it pick action a_t at state s_t; compared to picking an action in random. This means $A_\pi(s_t, a_t)$ represents relative (rather than absolute) state-action values which is easier to learn (it is easier to learn that one action has better consequences than another, rather than to learn the actual return from taking that action). The advantage function is given by:

$$A_\pi(s_t, a_t) = Q(s_t, a_t) - V_\pi(s_t) \approx \Big(r_t + \gamma V_\pi(s_{t+1})\Big) - V_\pi(s_t) \tag{2.3}$$

For discrete state-action spaces with small number of states and actions, value functions can be stored as tables (where rows represent states and columns represent actions). However, for large state-action spaces (when either states or actions are continuous values), the size of value function becomes intractable. In this case, some *function approximation* techniques can be used. For example, the value function $V_\pi(s_t)$ can be parameterized by neural network (NN) weights yielding an approximated version $V_\pi(s_t\,;\omega)$. Similar approach can be applied to approximate $Q(s_t, a_t\,;\omega)$ and $A_\pi(s_t\,;\omega)$.

Learning the optimal policy π_* involves performing two complementary steps: *policy evaluation* and *policy improvement*. The first step aims at approximating the value function, while the second step enhances the policy. Three families of algorithms can be found in literature depends on how those two steps are implemented:

- *value*-based methods: take advantage of the Bellman equation and recursively approximate the value function (say $Q(s, a)$) as an intermediate step, and use an implicit policy $\pi(Q(s, a))$ defined by that function (such as ϵ-greedy policy). In this case, improving the value function implicitly improves the policy derived from it. Algorithms in this family include Q-learning and SARSA [40]. The

evaluation step in given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big((R_{t+1} + \gamma\, \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \Big) \tag{2.4}$$

where the only difference is in how $\hat{Q}(s_{t+1}, a_{t+1})$ is evaluated. In SARSA it is $Q(s_{t+1}, a_{t+1})$, and in Q-learning it is $\max_a Q(s_{t+1}, a)$.

- *policy*-based algorithms: use experience/samples to improve the policy without worrying about finding any intermediate representation in the form of value functions. In this case, the evaluation step is replaced by observing the sampled reward $\hat{v}_t$ along a trajectory $\tau = s_t, a_t, r_t, s_{t+1}, \ldots$ generated by Monte Carlo method. The improvement step involves updating policy parameters $\pi(a_t|s_t\, \omega)$ to maximize experienced return through gradient descent. An example of this family is REINFORCE algorithm [41] with the following update rule:

$$\omega \leftarrow \omega + \alpha\, \nabla_\omega\, log\Big(\pi(s_t, a_t\, ; \omega)\Big)\, \hat{v}_t \tag{2.5}$$

  The result is an improved stochastic policy from which we can directly sample actions; for continuous actions, this could be the mean and standard deviations of Gaussian distributions, whilst for discrete actions this could be the individual probabilities of a multinomial distribution. Policy-based methods are especially useful when the action space is continuous or when stochastic policy is needed.

- *actor-critic* algorithms: aim at combining the strong points of value-based and policy-based methods, and it works with parameterized policies. The critic uses an approximation architecture to learn a value function, which is then used to update the actor's policy parameters. In order to reduce variance, critic usually uses advantage function $A_\pi(s_t, a_t\, \omega')$ as a value function; where ω' is the weight matrix of critic NN. The role of the actor is to optimize its parameters ω, and come up with a new policy $\pi(s_t, a_t\, ; \omega)$ that results in higher value function. This can be done by enforcing actions that yielded higher rewards, and avoiding actions that resulted in bad effects.

Note that the role of critic doesn't go beyond helping the actor to learn the optimal policy. Once the optimal policy is found, only the actor network is needed to deploy the algorithm. Also, note that in the policy evaluation step, the agent (critic) needs not to attempt to compute or approximate the *exact* value function, which is a high-dimensional object. It should ideally compute a certain

projection of the value function onto a low-dimensional subspace spanned by a set of *basis functions* that are completely determined by the parameterization of the actor [22].

During the process of learning the optimal policy, the agent tries to explore the state-action space to learn more about the environment. However, pure exploration may waste much time and computing resources exploring task-irrelevant parts of the environment and performing actions that do not help the agent achieve its goals. As it turns out, it is often advantageous to find some suitable balance of both *exploration* and *exploitation*. If the agent can exploit its current knowledge of the environment, it may be able to identify the most worthwhile parts of the environment to explore. Further, minimizing the learning costs (i.e., maximizing the agent's performance during learning) cannot be done without some degree of exploration of the environment so that effective behaviours can be discovered.

In policy-based methods, stochastic policies allow the agent to pick action a_t at state s_t with certain probability. Taking random actions allows the agent to explore different actions even if they don't appear to be optimal. As the agent gains more knowledge, good actions will be assigned high probability while the probability of picking bad actions will be low. Note that the trade-off between exploration and exploitation is implicit and can't be controlled directly.

Also, in actor-critic frameworks, we pick the baseline as the expected cumulative reward $V_\pi(s_t)$ of the current state, which couples the two functions $V_\pi(s_t)$ and $\pi(s_t, a_t))$ together in the training; the two functions reinforce each other. A correct $\pi(s_t, a_t))$ gives high-rewarding trajectories which update $V_\pi(s_t)$ towards the right direction, and a correct $V_\pi(s_t)$ picks out the correct actions for $\pi(s_t, a_t))$ to reinforce. This mutual reinforcement behavior makes actor-critic model converge faster. However, it can be prone to converge to bad local minima since on-policy models follow the very recent policy to sample trajectory during training. If the experience received by the agent in consecutive batches is highly correlated and biased towards a particular subset of the environment, then both $\pi(s_t, a_t))$ and $V_\pi(s_t)$ will be updated towards a biased direction and the agent may never see the whole picture [47].

One common practice to prevent the agent from over-optimize on a small portion of the environment is to add an entropy regularization term $\beta \nabla_\omega H(\pi(s_t \,.; \omega))$ to the

loss [26]. This encourages exploration by pushing ω in the direction of higher entropy and discourage any premature convergence to sub-optimal deterministic policies.

In value-based methods, it is common to use ϵ-greedy policy or softmax policies [11] to explicitly control the exploration process through a decaying parameter. In the former case, the agent behaves greedily with probability ϵ and chooses a non-greedy action at random with probability $1 - \epsilon$. This strategy guarantees a certain degree of exploration, which can be changed by adjusting the ϵ parameter. However, this strategy does not distinguish between the non-greedy actions, and so it does not discard very damaging actions. On the other hand, softmax policy (Equation 2.6) takes into account how well each non-greedy action is by using the function $Q(s_t, a_t)$. Actions with high expected rewards Qt(a) have a higher probability to be chosen, so the exploration is directed towards the most promising directions.

$$\pi(s, a) = \frac{e^{Q(s,a)/\epsilon}}{\sum_a e^{Q(s,a)/\epsilon}} \tag{2.6}$$

2.3 Quality of Experience

Quality of Experience (QoE) is the perceptual quality of a service from the end-users' perspective. A key aspect in all forms of video delivery, including video streaming, is the need to optimize user-perceived QoE. Traditionally, QoE is obtained from *subjective* test, where human viewers evaluate the quality of test videos under a laboratory environment and report their perceived QoE. This measure is referred to as Mean Opinion Score (MOS), and is expressed as a single number in the range 1 to 5, where 1 corresponds to the lowest perceived quality, and 5 to the highest. Modeling QoE can be of two types of subjective scores: retrospective QoE and continuous-time QoE. The first provides a single score describing the overall QoE on each presented video sequence, while the second involves a real-time measurement of QoE, which may be triggered by changes in video quality or streaming and by short or long term memory effects.

Since subjective test is costly, time consuming, and can't be measured in an online-mode, *objective* quality models have been developed to predict QoE, in terms of MOS, based on available objective parameters, such as video coding and compression schemes and network QoS parameters [7] [13] . MOS is often used as a benchmark to compare the accuracy of objective metrics; the more strongly a metric correlates with MOS, the

Fig. 2.5 QoE: the way it is defined, and the way users perceive it.

closer it is to representing the actual user QoE. Fig. 2.5 shows MOS scale for subjective video quality assessment.

Improving users' QoE is crucial for sustaining the advertisement and subscription based revenue models that enable the growth of Internet video. Despite the rich literature on video and QoE measurement, our understanding of Internet video QoE is limited because of the shift from traditional methods of measuring video quality (e.g., Peak Signal-to-Noise Ratio, Structural Similarity Index) and user experience (e.g., opinion scores). These have been replaced by new quality metrics (e.g., rate of buffering, bitrate) and new engagement-centric measures of user experience (e.g., viewing time and number of visits).

While a higher video quality is expected to improve QoE, existing literature on QoE of video streaming provides evidence that frequent switches or sudden changes in video quality or bitrate result in dissatisfaction of users and reduces QoE [36]. Additionally, video freezing or stalling, which happens due to rebuffering of video player, is the most harmful factor in QoE degradation. [33] found that a single rebuffering event has three times the impact of a bitrate change, and incur abandonment rates six times higher than start-up latency.

2.3.1 QoE Metrics

A wide variety of video quality assessment models have been proposed. Even though these metrics are still widely used in the literature, they are not well correlated with

the perceived visual quality of the human visual system, which is non-linear. For example, consider a strongly compressed video that exhibits very annoying artifacts on a 60-in TV (especially when watched closely). The same video could appear to have fine quality when it is scaled to small size and viewed on a smartphone. Since metrics like SSIM are independent of viewing device, it will give the same score even if the perceptual QoE is clearly different.

One of the well-accepted QoE models for adaptive streaming provides a prediction or estimation of MOS through a linear combination of the following three parameters: the average quality of video segments over a streaming session, the variation of video qualities and the effect of rebuffering:

$$QoE = \alpha\mu + \beta\sigma + \gamma\phi + \delta \tag{2.7}$$

where μ is the average quality of video segments, σ is the standard deviation of video qualities, ϕ is a measure of rebuffering or freezes, and α, β, γ, and δ are tunable parameters [44].

When multiple players share network resources, QoE metrics need to incorporate new factors in addition to quality (defined as efficiency in utilizing bandwidth). Those include fairness and stability; and are described as follows [21]:

- Unfairness Index: is defined as $\sqrt{1 - J(\boldsymbol{r})}$. Where $J(\boldsymbol{r})$ is the well-known Jain's Index; a metric that rates the fairness of a set of N values r_n. The result ranges from $\frac{1}{n}$ (worst case) to 1 (best case), and it is maximum when all users receive the same bitrates.
$$J(\mathbf{x}) = \frac{\left[\sum_n x_n\right]^2}{N \sum_n x_n^2} \quad \in [\tfrac{1}{N}, 1]$$
- Instability Index: measures the variation of requested bitrates. The instability index at time step k for each player n is defined as:
$$\frac{\sum_{d=0}^{K-1} |r_{k-d}^n - r_{k-d-1}^n| \, w(d)}{\sum_{d=0}^{K-1} r_{k-d}^n \, w(d)}$$

Chapter 3

Related Work

In this chapter, we will review the state-of-the-art in adaptive video streaming applications. We group the most recent related work in this area into four groups. First, we consider traditional client-side adaptation schemes. Then, network-assisted rate adaptation mechanisms are discussed. Finally, RL-based algorithms are explored in details. We highlight the difference and similarities between our proposed algorithm and related works with a clear distinction between rate-fairness and QoE-fairness.

3.1 Traditional Methods

Many ABR algorithms have been proposed to address the challenges associated with video streaming. The earliest solutions can be primarily grouped into two classes: rate-based and buffer-based. Rate-based algorithms [51] [39] first estimate the available network bandwidth using past chunk downloads, and then request chunks at the highest bitrate that the network is predicted to support. However, these methods are hindered by the biases present when estimating available bandwidth on top of HTTP. Several systems aim to correct these throughput estimates using smoothing heuristics and data aggregation techniques, but accurate throughput prediction remains a challenge in practice.

In contrast, buffer-based approaches [38] solely consider the client's playback buffer occupancy when deciding the bitrates for future chunks. The goal of these algorithms is to keep the buffer occupancy at a pre-configured level which balances rebuffering and video quality. The state-of-the-art algorithms, like MPC [48], have investigated combining both throughput estimates and buffer occupancy information to select bitrates that are expected to maximize QoE over a horizon of several future chunks.

Combining rate-based approaches (which are best at start-up time and when link rates are stable), and buffer-based approaches (which are sufficient and more robust in steady state and in the presence of time-varying networks) improved ABR performance. However, MPC still relies heavily on accurate throughput estimates which are not always available. Also, tuning fixed heuristics to perform well in different environments is challenging.

3.2 Server-client Cooperation

Server-side ABR algorithms are considered to achieve an anticipatory and proactive behavior to the bitrate selection process which outperforms reactive and shortsighted adaptation schemes in client-based methods. However, this creates an execution overhead on the server-side. The server is responsible for the whole adaptation process even though the client can successfully handle part of it. To solve this problem, collaboration between the server and the client is required, and so the server should be activated only when necessary.

Traditional methods in dynamic adaptive streaming over HTTP scheme are limited in achieving efficiency, fairness, and stability. Solutions proposed in the recent literature target these objectives, but tackle them from either client or server side separately. For example, [24] employs feedback control theory to design a Quality Adaptation Controller for live streaming. The controller is located at the server side to avoid delays in the control loop and to reduce the workload on the client.

Akhshabi et al. [4], used a reactive approach to address the instability issue when multiple players share a bottleneck link. The proposed shaping mechanism aims at eliminating the root cause of instability which is the *off* behavior of the players. It also helps in preventing frequent oscillations between the different quality levels by implementing a server-side ABR algorithm. The server-algorithm is reactive in nature and is activated only when needed.

To further improve the server-client interaction mechanism, Wei et. al [45] relied on the newly published HTTP/2 protocol in order to reduce the start-up time as well as the end-to-end delay in HTTP communications. This is done by allowing the server to push multiple content to the client without making individual requests for each content.

As a result, the overhead caused by high numbers of HTTP requests generated in traditional protocol is avoided.

3.3 RL-Based ABR

Existing classic ABR algorithms, server-based or client-based, all rely on *fixed* heuristics that have been exhaustively tuned according to rigid assumptions about deployment environments. Further, each of these algorithms has been explicitly configured to optimize for a specific QoE metric. As a result, today's ABR algorithms fail to generalize, and experience degraded performance if any assumptions are violated. In these scenarios, algorithms must again be manually tuned to suit for the new environment. To overcome these issues, learning-based approach for generating ABR algorithms becomes the trend.

Unlike all previous works that implement fixed heuristics, RL in adaptive video streaming was used in [14] and [15]. In this case, an agent gradually improves its bitrate adaptation policy by interacting with the video streaming environment. However, these systems are largely simulation based, which only preliminary demonstrate the "learnability" of an RL policy. The promising results lead to more efforts in applying RL in more realistic scenarios.

For example, Bokani et al. used an MDP formulation of adaptive streaming in vehicular environment, and demonstrated its superior performance in terms of reducing playback deadline miss compared to non-MDP solutions [10]. Similarly, Zhou et al [50] aimed at maximizing the users' QoE under time-varying channel conditions by taking into account key factors impacting the visual quality including video playback quality, video rate switching frequency and amplitude, buffer overflow/underflow and buffer occupancy. Another online learning adaptation strategy for DASH clients was proposed in [15] based on an MDP optimization. The authors introduced a penalty function into the reward function to penalize the system for re-buffering events as well as moving away from a safe buffer level.

Despite the performance improvement gained by aforementioned algorithms, there is still more to consider. In particular, such algorithms didn't capture the multi-agent nature of the problem, even though the problem clearly involves more than one utility-maximizing decision-maker interacting with each other. Also, they dealt with the

Reference #	Key Features				
	Heuristic	Type	State	Fairness	Learning Algorithm
[38]	Fixed	Client-side	Buffer-level	No	N/A
[21]	Fixed	Client-side	Throughput	Yes	N/A
[19]	RL	Server-side	PLR	Yes	SARSA
[5]	RL	Server-side	Packetloss	Yes	Q-learning
[25]	RL	Server-side	Mix	No	A3C
Ours	RL	Server & Client	PLR, Bitrates	Yes	A3C

Table 3.1 Summary of Related Works.

problem as if the environment states are always fully-observable; which is clearly not the case.

To deal with the first issue, Habach et al. [18] took a partially observable approach in formulating the congestion control problem by using a POMDP model, whose state is defined to include the source rate, which is known by the agent, and the packet loss rate, which cannot be directly observed. Users need to infer the congestion status of the bottleneck links using observations and QoE feedback measurements, which are provided in terms of MOS. Once the problem is formulated, they used temporal difference ($TD - \lambda$) online learning algorithm to solve it. Hemmati [20] took similar approach by formulating the problem as POMDP, and proposed a decentralized solution for it. Their Dec-POMDP algorithm was trained to optimize specific QoE metrics (namely social welfare function) to achieve fairness among all agents in addition to efficiency and stability. The solution was obtained by using SARSA algorithm.

Recent developments in RL algorithms [47] [37] have significantly improved the classic algorithms used in these work. Mao et al. [25] integrated the modern learning system into an industry standard DASH player and comprehensively compared the performance with the state-of-the-art approaches. In particular, they used a server-side A3C algorithm, and trained it on both synthetic and actual network traces. However, fairness was not consider in this work.

Summary of most-related previous works are shown in table 3.1. Note that our method was inspired by, and is a combination of, the following ideas:

- We formulate the problem of adaptive video streaming using a POMDP-based framework; whose state is defined to include the source rate and the hidden packetloss rate (as in [18]). However, we use TFRC-feedback rather than TCP acknowledgement.
- Our QoE metrics were derived from [19], where the agents try to optimize QoE-fairness and QoE-efficiency rather than rate-fairness and bandwidth-efficiency.
- We consider a collaborative role between the server and the clients rather than having a pure server-side algorithm (similar to [4]). However, instead of being completely reactive to clients needs, our server looks for clients that are most likely to need help in the future and plans accordingly.
- We use the A3C algorithm with NN architectures similar to [25]. However, our work is different in all aforementioned four points.

Chapter 4

System Design

In this chapter, we describe the design and implementation of our work in a structured way that is aligned with typical RL problem setup. Section 4.1 explains the details of our environment; including the network model, and the effects packetloss has on QoE of end users. Section 4.2 highlights the training methodology and learning algorithms underlying our agent. Finally, we explain the implementation details and how to apply learned models to real streaming sessions in section 4.3.

4.1 Environment

Our environment consists of concurrent video streaming sessions sharing a single bottleneck. The rewards signal (namely QoE) is negatively affected by packetloss; caused by traffic congestion at the bottleneck link. In order to design a network that faithfully emulated real conditions, we used throughput traces from real datasets to model congestion dynamics at the bottleneck. This is important to help the agent perform well when tested in real-world environments.

4.1.1 Network Model

Consider N concurrent video streaming sessions over the Internet, sharing the bandwidth of the network. Each session is composed of a sender node (media streaming server) and a receiver node (client) that establish an end-to-end transport layer connection, equipped with TCP-Friendly Rate Control (TFRC) protocol [16] to stream a multimedia content. The sender-side of each session cannot observe the traffic generated by other sessions, and the major source of information about network

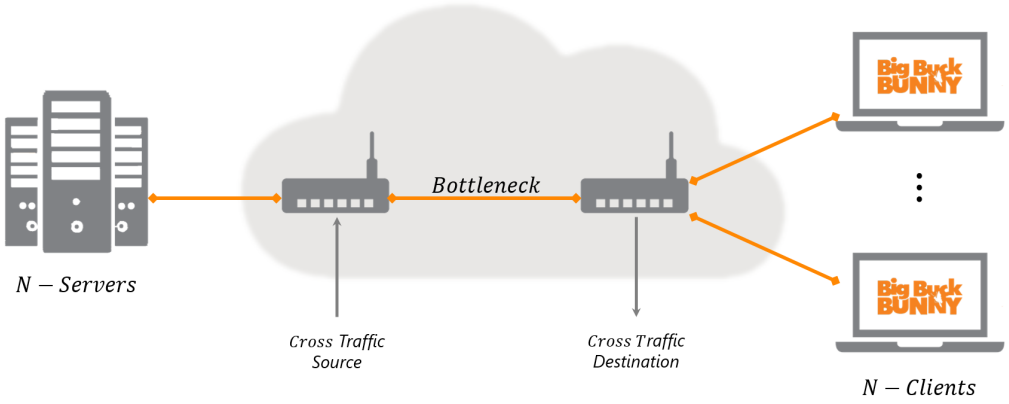


Fig. 4.1 Network model: N-concurrent streaming sessions competing on single bottleneck.

congestion level is the receiver's *estimate* of packetloss rate $\hat{p}$, which is included in TFRC feedback packets. Fig. 4.1 shows our network topology.

The choice of using TFRP feedback signal rather than typical TCP signal to indicate the packetloss is based on [20]. According to this paper, the use of TCP acknowledgement as an observation to train our RL-agent has many drawbacks. On on hand, the agent suffers from large state-space dimension that drastically increases the computational complexity of the problem. On the other hand, the binary observations in TCP make it difficult for the learning agents to converge to the optimal policy. By replacing TCP with TFRC, we take advantage of the observation of packetloss ratio $\hat{p} \in [0, 1]$ in order to improve the model's capability for inferring the unobservable network congestion.

We assume that the network has a single bottleneck link with a capacity of $C_b = C_x + C_s$; where C_x represents the portion of bottleneck capacity occupied by a time-varying cross-traffic r_x, and C_s is the slack left for the traffic generated by video streaming sessions r_n. The congestion level, denoted by C_g, takes values in the interval of [0, 1] and is defined as the ratio of total traffic on the bottleneck link to its capacity:

$$C_g = min\left\{\frac{r_x + \sum_n r_n}{C_b}, 1\right\} \tag{4.1}$$

As mentioned before, each user observes neither the traffic generated by other users, nor the congestion level of the bottleneck link. The only feedback available from the

network is the TFCP feedback signal. As the total traffic on bottleneck link comes close to its capacity (i.e. $\sum_n r_n \approx C_s$), all sessions would experience higher rates of packet loss and a degradation of QoE. Thus, all agents should consider reducing their traffic by streaming at a lower bitrates. To avoid dealing with trivial environment; where bottleneck link can't handle any video traffic, or when no congestion happens: we assume that $0 < C_s < \sum_n r_n$. To help train our RL-agent, we define the *critical-capacity* C_ϵ to be minimum bandwidth required at which ϵ-percentage of clients succeed to achieve acceptable level of QoE. We will call this metric the percentage of satisfied clients (PSC).

To emulate a realistic network condition, we used a corpus of real network traces to model our time-varying cross-traffic. The traces are used to shape the bandwidth of the bottleneck link for which the agents are required to predict. Being able to predict the environment dynamics helps the agents to plan their actions in the future (i.e. what video quality to stream in different network conditions). It is important to note that the quality of the dataset greatly affects the overall performance of learning agents. This is true because these traces are the only chance for agents to experience the magnitude and speed of bandwidth fluctuation in real networks. Using over-simplified random function to generate random traffic could limit the agent ability to perform well in real-world environments.

From network point of view, encoding bit rate (r) and packetloss rate (p) among many other factors (e.g. codec type and specifications, spatial resolution, key frame interval, delay, frame rate) can affect the perceived quality of a video. Among these, we assume that all are fixed (or have negligible impact on video quality) during a streaming session except the first two. Under such conditions, according to G.1070 standard, the perceived video quality by the end user is described by:

$$\Psi(r,p) = 1 + \left[\xi - \frac{\xi}{1+(r/\eta)^\lambda}\right] e^{-p/\mathcal{D}} \ \in [1,5] \tag{4.2}$$

where ξ; η; and λ are codec-dependent constants and $\mathcal{D}$ is the degree of robustness against packetloss p and is itself a function of bit rate r, frame rate, and a number of codec-dependent constants. As we can see in figure 4.2, it is clearly observed that a significant degradation of video quality occurs at packet loss rates above 0.05%, even with a pretty high source bit rate. Therefore, when implementing a QoE-aware rate adaptation algorithm, the agents should not only try to maximize their bandwidth

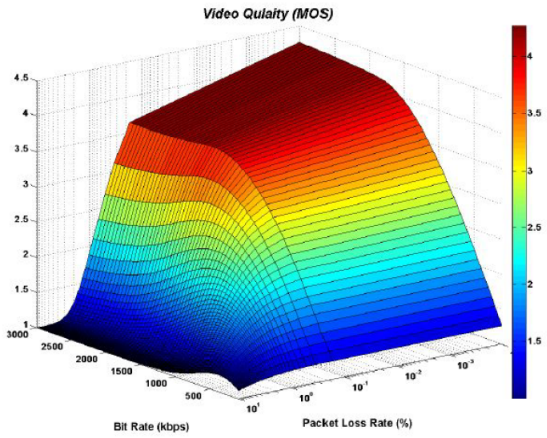


Fig. 4.2 Packetloss effect over video quality (Equation 4.2)

utilization, but also be able to pro-actively avoid congestion leading to higher packet loss rates.

4.1.2 Agent Model

The dynamic interaction is taking place among a finite number of video streaming sessions, regarded as agents. Each streaming session comprises a *server* (sender) and a DASH *client* (receiver) in the network. Being a client-side standard, DASH suffers from three main performance issues: instability, unfairness, and under-utilization of the available bandwidth. The main reason is that DASH clients are dependant on local information only. To overcome this limitation, we consider a server-side adaptive streaming solution; where the agents of the model are actually the sending entities of the video streaming process.

Following standard actor-critic architecture, our agent (the server-side) consists of two neural networks. The actor NN decides the suitable parameters to use in MPD (bitrate list and MPD update time). The client helps provide the observations to the server about (1) network; by sending back TFRC-like feedback packets, and (2) its own ABR algorithm; by reporting actual bitrate it selects. The critic network (on the server) collects the observation and tries to improve the actor performance. The server then uses its new policy to control the clients by utilizing the MPD file. This solution has a desired feature of being compatible with DASH standard; since no modification on the client-side is required.

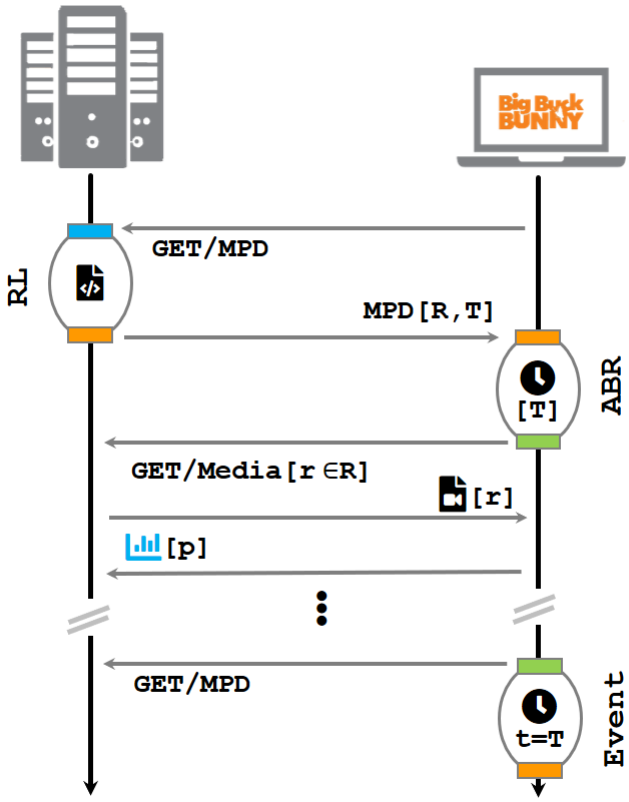


Fig. 4.3 Proposed approach for video delivery optimization. Server (left) use RL to forces client (right) to update its MPD every Ts. Client help server by send its TFRC-feedback.

The ability to (1) dynamically slice various live and on-demand content into periods, (2) fragment and deliver MPD files in parts to reduce the session start-up delay, and (3) update MPD file during the streaming session; motivate our solution in using MPD to include the server in the ABR selection process. According to [1], the standard allows to use *events* to insert ads inside a new *period* and forces the client to update its MPD with the new content. We use similar approach but for a different use case: the server updates MDP with new list of bitrates whenever a major change in network conditions occurs. It then sets expiration time T for the MPD file. The clients streams media content as normal; ABR algorithm will select suitable bitrates from the updated list in the MPD, and when playhead hits time T it requests an updated version of MPD. Figure 4.3 illustrates this procedure.

The value of MPD update-time T determines the level of involvement the server have in the decisions made by the clients. For example, setting $T \geq$ *video length* and

Fig. 4.4 Simplified MPD file showing how EventSteam element is defined.

$r = r_{MAX}$, represents the conventional client-side DASH streaming with no server interference (the server allows the client to choose from the full range of supported bitrates all the time). On the other hand, setting $T = L$ (where L is segment length) and setting bitrate to some particular value r_0 represents the pure server-side ABR case; where server forces the client to choose one particular bitrate every time step. In practice, it is desired to keep T as large as possible to reduce the overhead on the server and network. This implies that the server should only intervene when clients' local ABR algorithms fail to maintain overall quality and fairness across concurrent streaming sessions.

It is important to note that once T has been specified and MPD file is sent, the server has no control over the client during this period. Only when a timeout event occurs at client-side, the client will ask for new MPD file with new list of bitrates. Also, the MPD update is done through an *EventStream* element which is calculated relative to the current playing time t. This means that if the client skips watching part of the video and seeks to $t' \geq t + T$, it will never encounter the update event and will not update its MPD in the future. To solve this issue, we include a list of static events. If the first coming event -which was calculated dynamically- has been skipped, the next static event will force the client to do the update. Otherwise, the static events will be ignored as shown in Fig. 4.4).

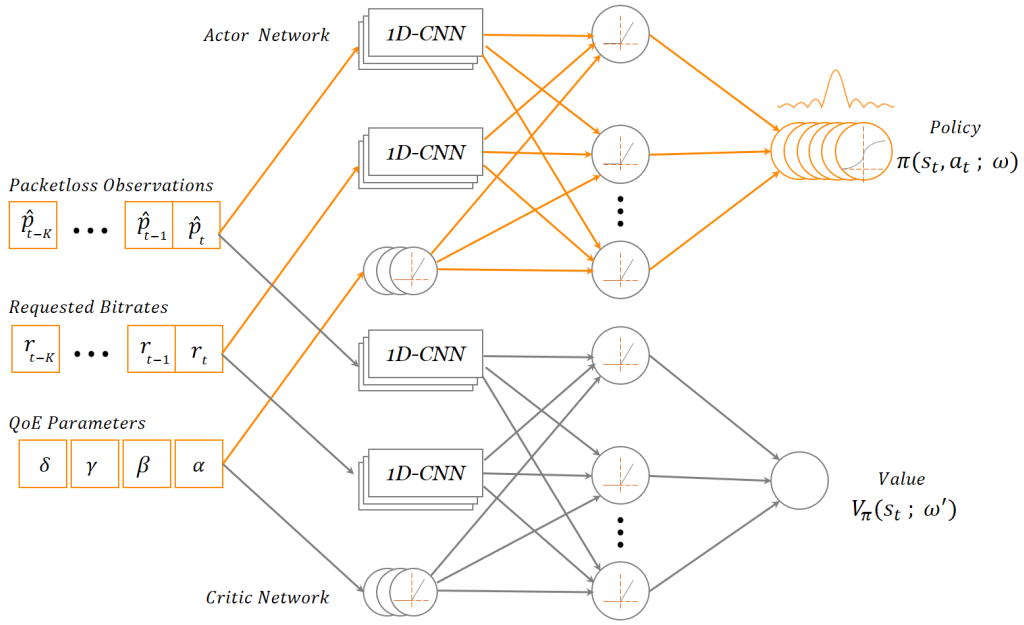


Fig. 4.5 Agents Neural Network Architecture

There is another useful idea behind using T which is to help the server calculate the length of video segments. As mentioned before, shorter segments are preferred in highly variable network conditions because it allows for faster switching. In this case, the server will not only choose shorter time interval T; to be able to update MPD more frequently, but also order the client to request media content that are shorter in length.

If the local ABR algorithm on the client is disabled (i.e. client is assumed to strictly follow the maximum bitrate found in the MPD), it will be enough to train the agent to learn the underlying dynamics of the environment using packetloss observations. However, we want to allow DASH clients to have an active role in selecting the suitable bitrate (under the guidance of the server). Therefore, we included additional piece of information to help the server better understand the client ABR logic. In this case, we include the history of *actual* bitrates used by the client in the NN inputs. Fig. 4.5 illustrates the network architecture of our agent.

Both the actor and critic networks share similar architecture that consists of three components: a 1D-CNN network (to pre-process the time-series observation vectors),

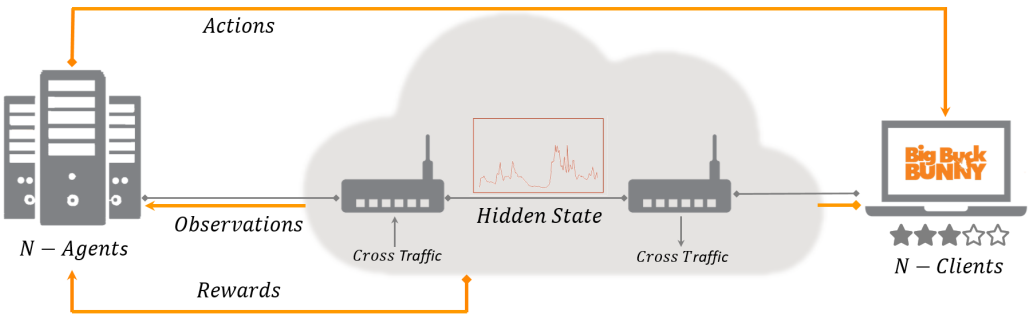


Fig. 4.6 Applying reinforcement learning to bitrate adaptation.

an optional[1] RNN Long Short-Term Memory block (to help the agent infers the hidden states of the partially observable environment), and fully connected layers (that serve as general function approximators of policy and value functions). Note that the activation function in the final neuron of the critic network is linear (i.e. no activation), since the output is not probability distribution. QoE-related parameters are used to better model a personalized QoE for each user (see section).

4.2 Learning Algorithm

Most existing rate adaptation methods [21][48], whether it's buffer-based or throughput-based, develop a *deterministic* control rules for making bitrate decisions. These schemes require significant tuning and do not generalize to different network conditions or QoE objectives. However, the decision policy guiding our ABR algorithm, as shown in Fig. 4.6, is not handcrafted. Instead, it is derived from training a neural network. In particular, we implement an actor-critic algorithm on the server to collect network packetloss values sent by DASH clients and use it to infer congestion level which represents the network state. The agent then feeds these values to its actor neural network, which outputs an action, i.e., the maximum bitrate to use for the next T seconds. The resulting QoE is then observed and passed back to the ABR agent as a reward. The agent uses the reward information to train and improve its critic neural network model.

Since multiple sessions are streaming concurrently, an asynchronous parallel training framework is required to support online training in which many users concurrently

[1]To simplify our analysis, we followed the same approach used by DeepMind and replaced the RNN component with a buffer that stores environment's past states. For more about training Actor-critic algorithms with RNNs, refer to [8]

send their experience feedback to the agent. This is why we adopted the *asynchronous* variation of advantage actor-critic algorithm (known as A3C). In this case, each agent (on the sending-end) will interact with environment using the available packetloss observations it has and will continually send their {state, action, reward} tuples to a *central* agent, which aggregates them to generate a single ABR algorithm model. For each sequence of tuples that it receives, the central agent uses the actor-critic algorithm to compute a gradient and perform a gradient descent step as we'll describe shortly. Once timeout event occurs after T seconds, the central agent updates the actor network and pushes out the new model to the agent which issued the MPD timeout event.

4.2.1 Problem Formulation

Our implementation follows the formulation developed in [20], where the dynamic interaction is taking place among a finite number of concurrent video streaming sessions. The agents of the model are the *sending* entities of the video streaming process (i.e. the servers). The receiver nodes (clients) help provide the observations to the senders by sending back TFRC-like feedback packets. The clients have their own local ABR algorithms which are considered to be part of the environment dynamics from other agents' perspective. This multi-agent decision process runs over the course of a sequence of discrete time steps indexed by $t = 0; 1; 2; \ldots$. We set the time step of Dec-POMDP to be equal to one RTT.

The proposed Dec-POMDP model is specified by the following components $< \mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{I}\ \gamma >$, where:

Actions $\mathcal{A}$ & Transitions $\mathcal{T}$ The action $a_t \in \mathcal{A}$ taken by an agent at a given time step t is to choose the appropriate sending rate from a set of *discrete* pre-defined sending rates available to the user. Since the joint actions taken by users affect the next state the environment will transition to, the agents should plan their strategy (*policy*) to avoid congestion at bottleneck. This common goal will result in an indirect collaboration since a congestion event will hurt everyone. It is important to note that the action taken by the agents (servers) are the maximum available bitrates for each client and not the actual bitrate that the client will choose (even though they are closely related). The actual bitrate a client is requesting depends on its local ABR algorithm.

States $\mathcal{S}$ & Observations $\mathcal{O}$ Our states are set to be the *unobservable* congestion level of the *single* bottleneck link within the network (continuous state-space). In our POMDP, individual users can't observe the traffic generated by other users, nor the congestion level of the network. The only available piece of information about the network condition is receiver's estimate of the Packetloss Rate, denoted by $\hat{p}$. We assume that the observation of each agent is statistically independent of others' observations.

Since the actual requested bitrate depends on client's local ABR algorithm, we can consider the client to be part of the environment and its selected bitrate to be part of the environment's hidden dynamics. However, since we aim at achieving maximum cooperation between the server and the client [2], we defined our agent to include both sides. Thus, we included the actual bitrate into the states vector. This helps the server guide different clients differently according to their local ABR algorithm.

Initial State Distribution $\mathcal{I}$ We assume that network congestion is not at the extreme low or extreme high levels at the beginning of the video streaming service, and is uniformly distributed over the rest of the interval.

Reward $\mathcal{R}$ The common goal of all agents is to achieve and maintain an optimal allocation of the network bandwidth leading to an *efficient* and *fair* distribution of end-users' QoE. This is done through the notion of maximizing a *social welfare* function Φ defined by a linear combination of both measures; where for N users downloading a video of K segments:

$$\begin{aligned} \Phi(\mathbf{QoE}) &= log\Big(\sum_n qoe_n\Big) + 0.5\, log\Big(\ J(\mathbf{QoE})\ \Big) \\ &= log\left(\Big(\sum_n qoe_n\Big)^2 \Big/ \sqrt{N\sum_n qoe_n^2}\right) \end{aligned} \tag{4.3}$$

Where $J(\{x_1, \ldots, x_n\}) \in [\frac{1}{n}, 1]$ is the well-known *Jain's Index*. And for QoE, we use the well-accepted model for adaptive streaming which provides a prediction or estimation of MOS through a linear combination of three factors: the *average quality*

[2] the original formulation in [20], as well as most server-side ABR algorithms assume greedy client; the client always select the maximum available bitrate.

of video segments over a streaming session, variation of video qualities, and the total re-buffering time.

$$\begin{aligned}\mathbf{QoE} = \alpha \sum_{k} \frac{q(r_k)}{K} - \beta \sum_{k} \Big| q(r_{k+1}) - q(r_k) \Big| \\ - \gamma \sum_{k} \left(\frac{d(r_k)}{C_k} - B_k \right)^{+} + \delta\end{aligned} \tag{4.4}$$

Where $d(r_k)$ represents the size of k-th segment, B_k is the current buffer level, and $q(r_k)$ is a nondecreasing function mapping bitrate to video quality perceived by user. The constants α, β, γ, and δ are non-negative weighting parameters representing the fact that different users are affected differently by the three mentioned factors. In our case, we used RL to tune these parameters to help better capture user's perception and improve the QoE model.

Once QoE is calculated for all users and social welfare function is computed, the expected cumulative discounted reward can be calculated over the horizon h using:

$$\mathcal{R} = \mathbb{E}\left[\sum_{i=1}^{h} \gamma^i \; \Phi(\; \mathbf{QoE}_i \;) \right] \tag{4.5}$$

4.2.2 A3C Algorithm

After applying each action a_t, the environment provides the learning agent with a reward for the action he took. Recall that the primary goal of the RL agent is to maximize the expected cumulative (discounted) reward that it receives from the environment. Thus, the reward is set to reflect the performance of each chunk download according to the specific QoE metric we wish to optimize. The agent then uses the sampled rewards to evaluate and improve its policy according to the A3C learning algorithm [27].

The *actor* uses policy gradient method [41] to train its policy. The key idea in policy gradient methods is to estimate the gradient of the expected total reward by observing the trajectories of executions obtained by following the policy, and to tune the parameters of our current policy $\pi(s_t, a_t \; ; \omega)$ in the direction that increases the probability of any "good" action a_t at state s_t. The size of the step depends on the value of the *advantage* ("how good") of action a_t in state s_t as provided by *critic*. Thus,

the net effect is to reinforce actions that empirically lead to better returns. In order to prevent the actor from over-optimize on a small portion of the environment, we add an entropy regularization term $H(\pi(s_t\ ;\omega))$ to the loss to encourage exploration (by pushing ω in the direction of higher entropy), and to discourage any premature convergence to sub-optimal deterministic policies. Concretely, The gradient of the cost function $J(\omega)$ with respect to the policy parameters ω, can be computed as

$$\begin{aligned} \nabla_\omega J(\omega) &= \nabla_\omega \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \\ &= \mathbb{E}_\pi \Big[\nabla_\omega \ log\Big(\pi(s_t, a_t\ ;\omega)\Big)\ A(s_t, a_t) \Big] \end{aligned} \tag{4.6}$$

and the update rule for actor NN parameters is given by

$$\omega \leftarrow \omega + \alpha\ \nabla_\omega log\Big(\pi(s_t, a_t\ ;\omega)\Big)\ A(s_t, a_t) \quad + \quad \beta\ \nabla_\omega H(\pi(s_t\ ;\omega)) \tag{4.7}$$

where the parameter β controls the strength of the entropy regularization term, and is set to a large value at the beginning to encourage exploration and then decreases over time to emphasize improving rewards.

For each experience, the advantage $A(s_t, a_t)$ is given by $R(s_t, a_t) - V_\pi(s_t)$, where $R(s_t, a_t)$ is the total reward obtained in the experience and $V_\pi(s_t)$ is the expected total reward started at state s_t and following the policy π. The role of the critic network is to learn $V_\pi(s_t)$ from empirically observed rewards. We follow the standard Temporal Difference method [40] to train the critic network parameters ω' to reduce $\nabla_{\omega'}(R(s) - V_\pi(s_t))^2$; where $R(s)$ is the actual reward sampled in the trajectory following state s_t. Equivalently, we can rewrite the whole equation in terms of estimated value function $V_\pi(s_t\ \omega')$ only, and perform updates to the critic network to reduce mean-squared error as follows:

$$\omega' \leftarrow \omega' - \alpha'\ \nabla_{\omega'} \left[\Big(r_t + \gamma V_{\pi_\omega}(s_{t+1}\ ;\omega')\Big) - V_{\pi_\omega}(s_t\ ;\omega') \right]^2 \tag{4.8}$$

where α' is the learning rate of the critic. In practice, two important observations can be stated about the role of the critic. First, the critic network merely helps to train the actor network. Once trained, only the actor network is required to execute the ABR algorithm and make bitrate decisions. Second, the critic needs not to attempt to

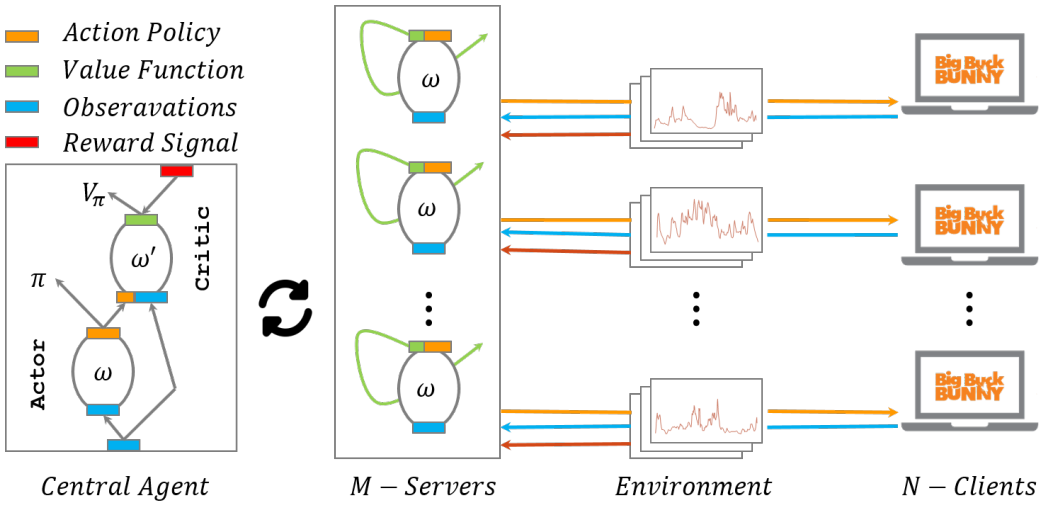


Fig. 4.7 Policy-evaluation (critic) and policy-improvement (actor) in A3C Algorithm. Agents experience different environment conditions and update their NN asynchronously with the central agent.

compute or approximate the *exact* value function, which is a high-dimensional object. The critic should ideally compute a certain *projection* of the value function onto a low-dimensional subspace spanned by a set of *basis functions* that are completely determined by the parameterization of the actor [22].

Fig. 4.7 summarizes how the actor and the critic perform policy evaluation and policy improvement. In the first step, the agent tries to evaluate its policy by calculating the associated value-function from the observed collected rewards. The agent doesn't necessarily know the actual value-function (it may over/underestimate the value-function during the learning process). The goal of policy-improvement step is to come up with a new policy that produces higher value-function. This can be done by enforcing actions that yield higher rewards and avoiding actions that resulted in bad effects. The agent repeats these steps until it converges to the optimal policy.

To further enhance and speed up the training, we ran multiple learning agents in parallel as suggested by the A3C paper. In this setup, each agent is configured to experience a different set of input parameters (e.g., network traces). However, the agents continually send their {state, action, reward} tuples to a central agent, which aggregates them to generate a single ABR algorithm model. For each sequence of tuples that it receives, the central agent uses the actor-critic algorithm to compute a gradient and perform a gradient descent step. The central agent then updates the

DASH Parameters	
Parameters	Values
Buffer size	20 s
(at max bitrate)	30 s
BW safety factor	0.9

Server Specifications	
Specifications	Values
Processor	i7, 6 cores, 12 Threads
RAM	32 GB
GPU	11 GB NVIDIA GeForce-GTX 1080 Ti

Table 4.1 Experiment Parameters Settings.

actor network and pushes out the new model to the agent which sent that tuple. Note that this can happen asynchronously among all agents, i.e., there is no locking between agents.

Despite having an N concurrent streaming sessions, and thus an N agents to train, the final learning model in our A3C algorithm is represented by the central agent. The role of the parallel agents is to explore different parts of the environment regardless of which agent serves which streaming session. In particular, the number of parallel agents can be fixed (determined by the number servers, and number of supported cores per server, etc), while number of clients (and thus number of streaming sessions) can be any arbitrary number. In this case, all servers collect observations from their own clients in parallel and perform an asynchronous update to/from central network whenever they need.

4.3 Implementation

Server & Client. In our setup, the client video player was a Google Chrome browser (version 67). We used ABR algorithms implemented in dash.js (akamai version 2.7) [2] without modification. For Pensieve [25], dash.js was configured to fetch bitrate selection decisions from the server that implemented the corresponding algorithm. For our proposed algorithm, we kept ABR rules on dash.js untouched but gave ABR servers the authority to limit available list of bitrates the client can request. The video server is a simple Node.js application (version 6.0) ran on Alienware Aurora machine. The server used Redis database to store clients observation as well as RL parameters. The A3C algorithm implemented in python using Tensorflow and ran as a child process in Node.js server. Multiple agents were implemented as multiple instances of Node.js on the same machine but on different ports. Table 4.1 summarizes the specifications of devices.

Network Topology. We used three machines to build our network: a server and two clients connected with two routers via a single bottleneck link. Each machine runs multiple instances of its type (client run dash.js instances and server runs node.js instances) to stream videos over the single bottleneck link that connects the two routers. iPerf3 is used as a service to periodical report packet-loss every L seconds (where L is video segment length and is selected by the server). The number of streaming sessions is not fixed. Any client can join or drop at any time.

RL-Agent. To train our agents, we used two arrays of size 8 to store past packetloss observations and actual past bitrates. Each array is passed through a 1D convolution layer with 32 filters, each of size 4 and stride 1. Results from these layers are then aggregated with other 4 inputs that represents QoE parameter. All are then connected by a fully connected hidden layer of 128 neurons followed by a softmax activation function. The critic network uses the same NN structure, but its final output has linear activation function. This architecture was inspired by [25], where we use *convolutional* neural networks (CNN) instead of *recurrent* neural networks (RNN) to process time-series observations. For the hyper-parameters: we use a discount factor $\gamma = 0.99$, learning rates for the actor and critic are $\alpha = 10^4$ and $\alpha' = 10^3$ respectively, and the entropy factor β is controlled to decay from 1 to 0.1.

Dataset We used two datasets to train our agent:

- 3G/HSDPA throughput traces: was used to emulate realistic network conditions as described in 4.1.1. We used a corpus of network traces from 3G/HSDPA mobile dataset collected in Norway [17] to model our time-varying cross-traffic. The traces are used to traffic-shape the capacity of bottleneck. To avoid scenarios where bitrate selection is *trivial*, i.e., situations where picking the maximum bitrate is always the optimal solution, or where the network cannot support any available bitrate for an extended period, we only considered original traces whose average throughput is less than 6 Mbps, and whose minimum throughput is above 0.6 Mbps. Furthermore, we bound $\sum_n r_n(min) < C_s < \sum_n r_n(max)$ where min/max refers to the case when all sessions are streaming at minimum/maximum available bitrate. When testing on variable number of users, we scaled the values from a randomly picked ensemble of this available bandwidth dataset by the number of the users sharing the bottleneck to determine the bandwidth of the bottleneck link.

- ITEC-dataset [23]: was used for video content. We compiled a dataset of three genre: animation, sport, and movie with 20 minutes long each to cover differences in scene characteristics (high motion in sport, low detail in Animation, etc.). All videos were encoded by the H.264/MPEG-4 codec at bitrates in {200, 300, 400, 500, 700, 900, 1200, 1500, 2000, 2500 3000} kbps. Additionally, the segment length L was chosen according to the value of T (in particular $L = round(T/4)$ where $T \in \{10, 25, 60\}$ seconds). The options were {2, 6, 15} seconds.

Chapter 5

System Evaluation

In this section, we provide the evaluation metrics and results compared to related works together with comments on results.

5.1 QoE Evaluation

In order to evaluate the performance of our proposed algorithm, we use the same three QoE-based metrics defined in [19]:

- *Average QoE*: Using Equation 4.4, we calculate the average QoE of all players, which is a measure of satisfaction level of users. Note that this metric is actually representing both efficiency and stability.
- *Fairness of QoE*: we use the Jain's index $J(.)$ as a measure of fairness of distribution of QoE across all users.
- *Social Welfare*: this metric *explicitly* combines the efficiency, stability and fairness measures into one single social welfare metric, as defined in equation 4.3 to evaluate the overall performance of the rate adaptation methods.

While it is common to define inefficiency, unfairness, and instability as a *rate*-based metrics (i.e. metrics that are defined based on the video bitrate fetched) as in [21]. we preferred to use the aforementioned *QoE*-based metrics. The reasons is that when fairness, for example, is considered in the literature, it is about fair distribution of throughput, while the end users' QoE fairness is more desirable. This observation was clear in [19] where all tested methods perform more or less equally well in terms of rate-based metrics, there were a significant difference in all QoE-based metrics.

We compare our method to the following algorithms which collectively represent all most directions in bitrate adaptation:

- **BOLA** [38]: represents a traditional ABR algorithm that uses fixed heuristic that uses Lyapunov optimization to select bitrates solely considering buffer occupancy observations. We use the BOLA implementation in dash.js
- **FESTIVE** [21]: is a client-side ABR algorithm that tries to strike an appropriate balance among three performance metrics for video streaming: fairness, efficiency, and stability. However, the notion of fairness employed in their work is all about equal share of bandwidth and enjoying the same amount of bitrate for different flows rather than fairness in QoE.
- **SARSA** [20]: represents a server-side RL-based algorithm. This is the original proposed algorithm in the theoretic framework we are building our solution on. They incorporate a social welfare function that explicitly integrate efficiency and fairness into a single performance index; which is used as reward signal to train the RL-agent.
- **Pensieve** [25]: represents a server-side RL-based approach that uses A3C algorithm to learn ABR logic. Like FESTIVE, it aims at achieving a balance between fairness, efficiency and stability. However, the algorithm is not optimized for fairness.

5.2 Test scenarios

For the sake of illustration and ease of discussion, we first provide the results of our performance evaluation scenarios of three clients to show the behaviour of different solutions. We then focus on testing our system for larger number of users.

5.2.1 Bandwidth Efficiency

In this experiment, the behaviour of a single client is tested against various network conditions. The goal is to compare the efficiency of the different algorithms under realistic network conditions. Fig. 5.1 shows a sample cross-traffic used to emulate one realistic environment. The average bandwidth of the channel is 2M*bps*; which is less than the maximum bitrate available to the client. The minimum and maximum bounds are 300K*bps* and 3.2M*bps*; to avoid the trivial case; the client can't simply

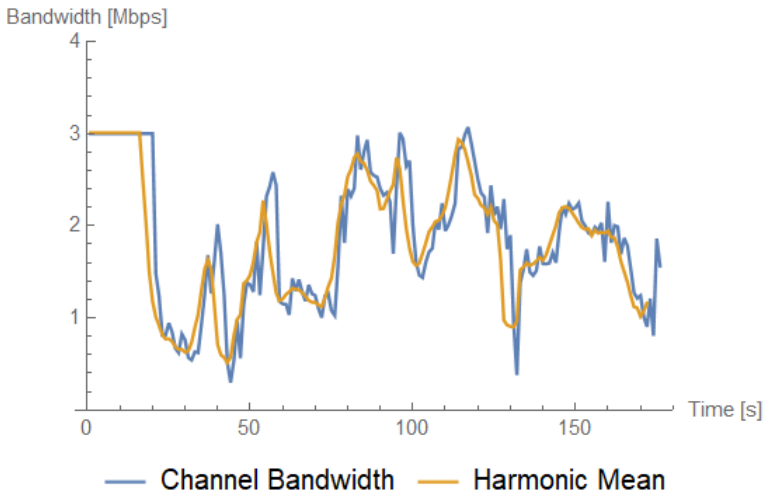


Fig. 5.1 A sample network trace used to test different algorithms.

choose the highest bitrate all the time. The channel also has high fluctuations; where careful planning is required to handle situations where a temporary increase in available bandwidth can be followed by sudden drop. The harmonic mean estimation is also plotted to show a rough prediction similar to what FESTIVE is using to perform its adaptation. Note that this is not the actual estimation because of off-period phenomena as we'll describe later.

According to Fig. 5.2, we note that at the beginning, when no cross-traffic is presented, all considered adaptation methods increased the video bitrate up to the maximum video bitrate available in the representation set at comparable time interval. However, during the presence of cross-traffic, the strengths and weakness of each algorithm can be observed.

The reduction in the available bandwidth caused a sudden drop in the buffer level for all players. As expected, the largest decrease in buffer was observed for BOLA since this method is sluggish to react to changes in the available bandwidth. For example, Fig. 5.2 shows that BOLA players reacted slowly against available bandwidth variations as they were waiting for the buffer level to go below the predefined threshold. For instance, the cross-traffic started at the $t = 20s$ while BOLA player started responding to this variation after about $10s$. In order to restore the buffer level to the reliable threshold and to prevent from probable video freezes, BOLA had to select bitrates lower than the available bandwidth. Consequently, it suffered lower efficiency compared

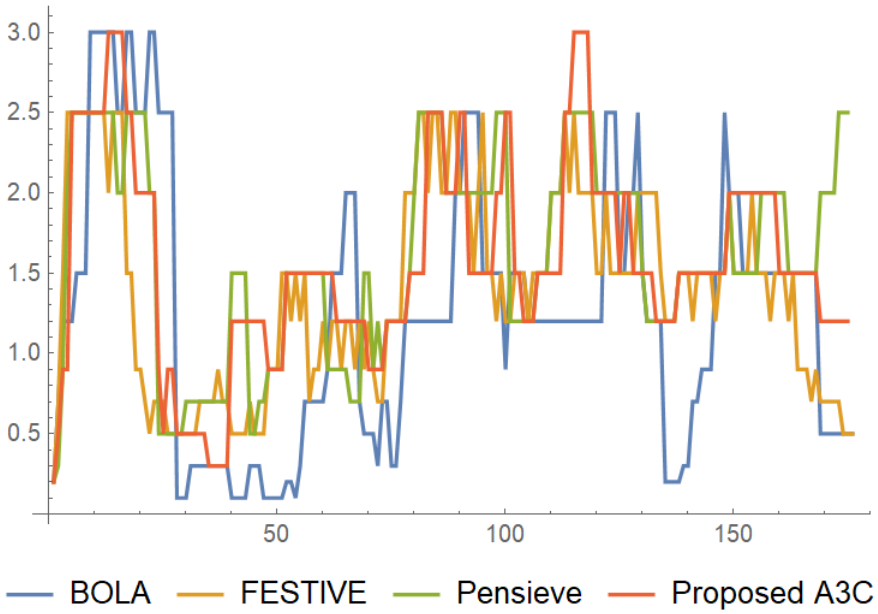


Fig. 5.2 The played representation levels of a single agent under variable network conditions.

to other algorithms. In fact, this can also lead to instability when multiple BOLA clients compete on limited bandwidth resources.

FESTIVE, on the other hand, showed fast reaction to bandwidth changes since it is a rate-based method which uses harmonic mean of last five chunks to estimate the bandwidth. However, it produced considerable fluctuations in the experienced segment throughput and subsequently instability in the adapted video bitrate. Both Pensieve and our A3C algorithm showed better utilization of the network bandwidth since they don't use fixed heuristics. additionally, because they are server-side algorithms, they benefited from the available observations about the environment collected by all clients.

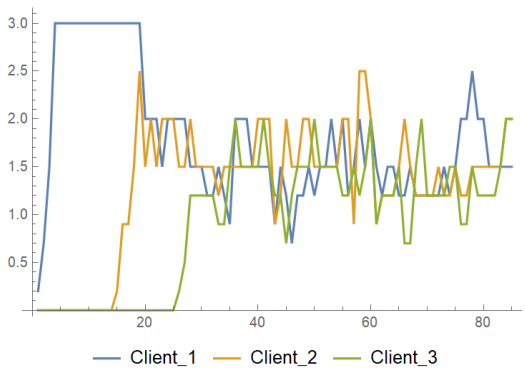
We can also observe that when buffer occupancy level reaches high values, the clients enter an *off* period where they stop downloading new chunks to avoid buffer overflow. This effect prevented BOLA from taking advantage of the improved network conditions around $t = 50s$ which resulted in low efficiency. Similarly, when FESTIVE entered an off-period, its estimate of the available bandwidth became largely inaccurate; specially when dramatic network changes occur during that off-period. This explained the rebuffering event around $t = 120s$.

Despite using similar architecture, our algorithm showed different behaviour than Pensieve. This is expected because both algorithms were trained to observe different representation of the environment. For example, when no cross-traffic was presented, Pensieve took a practical decision and never tried to request segments at bitrate higher than 2.5M*bps*. This reflects its belief that, in practice, the available bandwidth is likely to drop in the future bellow its maximum level. In contrast, our algorithm tried to take advantage of the stable network conditions at the beginning, or the high available bandwidth at around $t = 115s$. Thus, it managed to streamed videos at 3M*bps*. We can also notice that towards the end of the streaming session, Pensieve tends to be aggressive to avoid building large buffer when no more chunks are there to download. Even though this approach resulted in better bandwidth utilization, it negatively affect fairness when multiple clients are streaming.

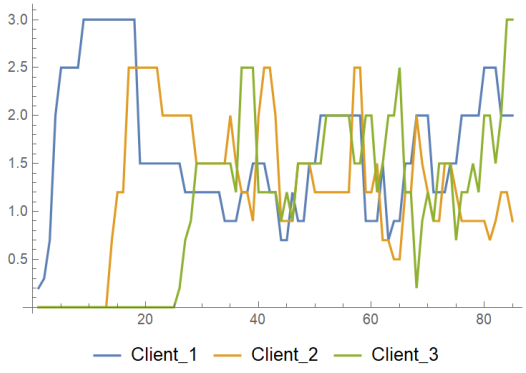
5.2.2 Fairness & Stability

In this experiment we compare our A3C algorithm in terms of fairness and stability when multiple agents are competing on the available bandwidth. To better illustrate players behaviour, we plot the requested bitrates of 3 homogeneous players under same channel conditions used in Section 5.2.1. The only difference between the clients is the length of the streaming session; client 1 is streaming one long video while client 2 & 3 are streaming three shorter videos. The bandwidth of the bottleneck link was set to 7 M*bps*. When the aggressive cross-traffic started, it took on average third of the available bandwidth at 2 M*bps*. This phenomenon can be justified by the fact that the cross-traffic uses persistent connections while the players use non-persistent connections. Hence, in the absence of cross-traffic, the fair share of available bandwidth for each client was around 2.3 M*bps*; which is less than the maximum available bitrates. And in the presence of cross-traffic, the fair share for each player dropped to 1.6 M*bps*.

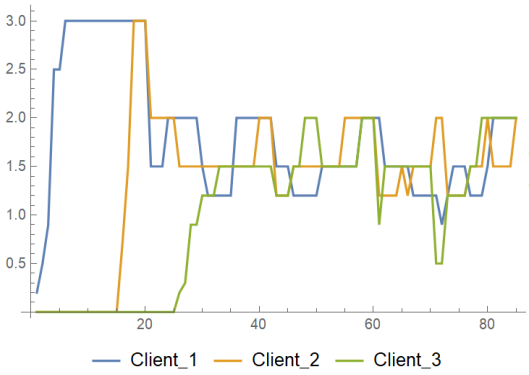
Fig. 5.3 shows the played representation levels of the different clients for FESTIVE, Penseive, and our A3C algorithms. As we can see, client 1 connected alone to the server at $t = 0$ and benefited from the whole bandwidth; where 3M*bps* is the highest level lower than its maximum bandwidth capacity. Client 2 joins shortly and tried to improve its acquired representation level until achieving the highest level it can visualize at $t = 15s$. This didn't affect client 1 as the shared bandwidth is sufficient to satisfy the requests of both clients and keep them stable (3.5M$bps > 3$M*bps*). The cross-traffic started to affect the bottleneck at $t = 20s$, caused both clients to drop their bitrates. In the case of Pensieve, since client 2 was streaming at lower bitrate it



(a) FESTIVE Algorithm.



(b) Pensieve Algorithm.



(c) Our A3C Algorithm.

Fig. 5.3 Evaluation of adaptation algorithms when multiple players compete on single bottleneck. (bitrate [M*bps*] vs time [s])

maintained higher buffer level. Consequently, it managed to keep its bitrate at higher level than client 1.

When client 3 connected at $t = 25s$, it started competing with other clients on the available 4.15M*bps*. Fig. 5.3b shows that both Pensieve clients were affected even though client 1 is not utilizing all its fair share. This is a direct result of not taking fairness into account when optimizing QoE metric. Note also that the overall utilization of the link is dropped in the case of Pensieve. This is justified by the fact that the increase in the quality level of client 3 caused a decrease in the played level of client 1 & 2 but the decreased amount is greater by far than the increased bandwidth amount. Note also that despite being able to achieve a level of fairness among the 3 players, Pensieve can maintained it for short amount of time only. This is because fairness is not inherited in Pensieve's metric, it is rather an indirect result of the balance each individual client tries to maintain between being aggressive and conservative.

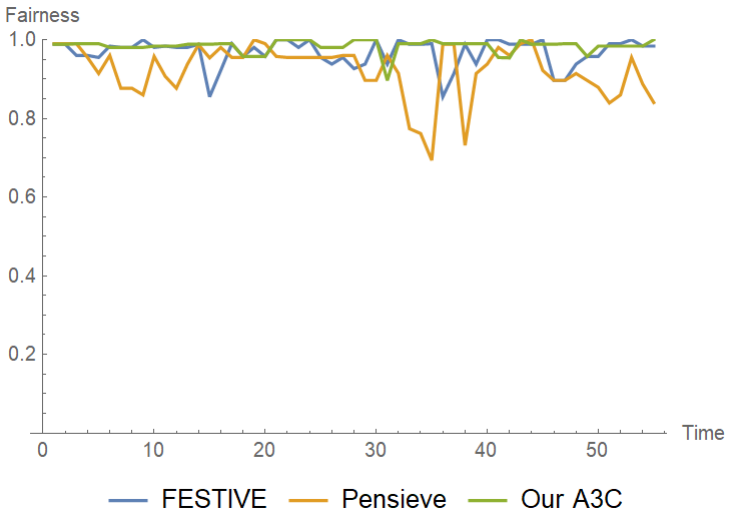


Fig. 5.4 Fairness of different algorithms when 3 clients competes on single bottleneck.

To better understand how our algorithm performs against FESTIVE, we plot fairness metric (defined by Jain's Index) in Fig. 5.4. As we can see, the players running our proposed method fairly share the bottleneck bandwidth when operating along with the cross-traffic. At one hand, the 3 players streamed video at similar quality levels. On the other hand, when more bandwidth is available, the server allowed client 1 to take the initiative and improve its bitrate at (see Fig. 5.3 $t = 40s$). However, when more bandwidth became available, the server prevented client 1 from grabbing that bandwidth and allowed client 2 to enjoy slightly higher bandwidth. It is interesting to note that this strategy is completely self-taught, since the server started knowing nothing about time-division techniques and has not been configured to use any heuristics.

Fig. 5.3b shows a subtle observation about Pensieve ABR algorithm. Recall that Pensieve uses the "number of chunks left" in its training. Consequently, Pensieve clients tend to be aggressive at the end of a streaming session (there is no need to keep large buffer when no more content is available to download). Despite being helpful in improving the efficiency, this behaviour harms other streaming sessions that are still in progress (see the two spikes at $t = 40$ and $t = 60$ for client 2 in Fig. 5.3b). This behaviour is not observed in our algorithm, since fairness is explicitly included in the QoE model that defined the reward signal.

Algorithm	Average Metric		Average Bitrate M*bps*		
	QoE	Fairness	Client 1	client 2	Client 3
FESTIVE	3.91	0.96	1.48	1.54	1.36
Pensieve	4.27	0.92	1.51	1.32	1.58
Our A3C	4.45	0.99	1.53	1.54	1.55

Table 5.1 Summary of the results when 3 clients compete on single bottleneck.

5.2.3 QoE-based Metrics

In previous subsections we examined the behaviour of our algorithm and compared it to other related works in terms of rate-based metrics. This is useful in better understanding how they will behave under variable network conditions, and when multiple streaming sessions are competing on network resources. However, the ultimate goal is not to optimize the bitrate nor the bandwidth fairness. It's the average QoE and fairness of QoE is what the end users care about. Also, in order to better evaluate our system, we use larger number of clients (20 clients) and carried out 20 runs of experiments and calculated the QoE achieved by each algorithm over the course of a video streaming session.

The normalized QoE-based metrics are depicted in Fig. 5.7, with higher values indicating better performance. We observe significant improvements in all these metrics, compared to existing methods. Specifically, our A3C algorithm achieves an average QoE which is 23%, 20%, and 6% higher than that of BOLA, FESTIVE, and Pensieve respectively. Note that while our algorithm didn't greatly outperform Pensieve on this metric, it achieves close performance despite being trained on smaller dataset. The reason is that Pensieve suffers from rebuffering in the bandwidth range bellow 3 M*bps* as it was not able to learn to be more conservative. We believe that this is due to the way they represent environment state; Pensieve was trained on all type of measurements even though they are not directly related to the QoE-metrics. An example of that is the aggressive behaviour at the end of streaming session Pensieve has learned from the number of chunks left.

In terms of QoE-fairness and social welfare, Fig. 5.6 showed that our proposed client-server collaboration outperforms the state-of-the-art schemes by as much as 21%, and 24% respectively. This is mainly due to our unique QoE model that explicitly includes fairness in the reward signal, and combines all the metrics in a single number:

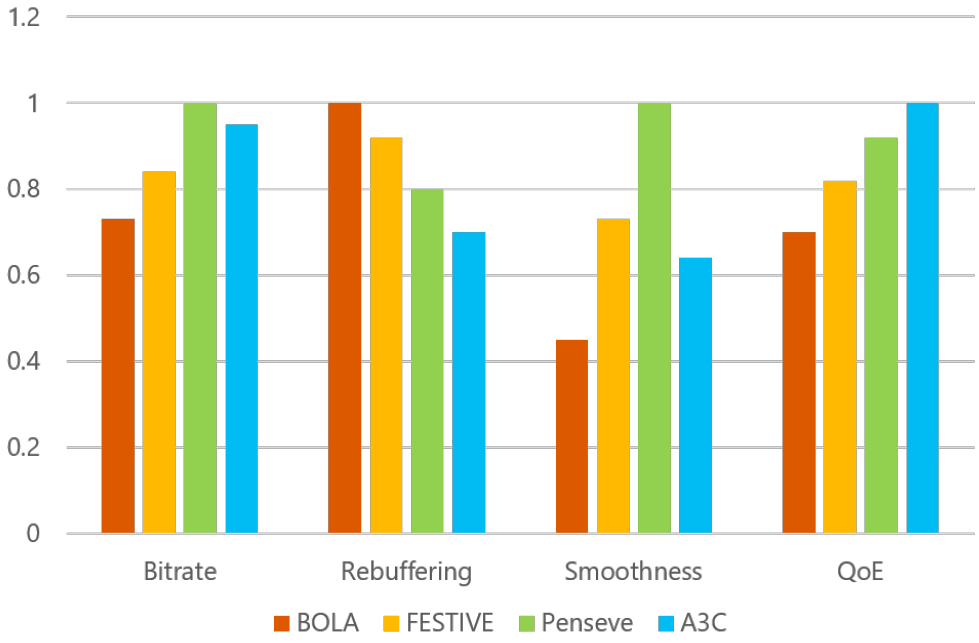


Fig. 5.5 Individual components of QoE

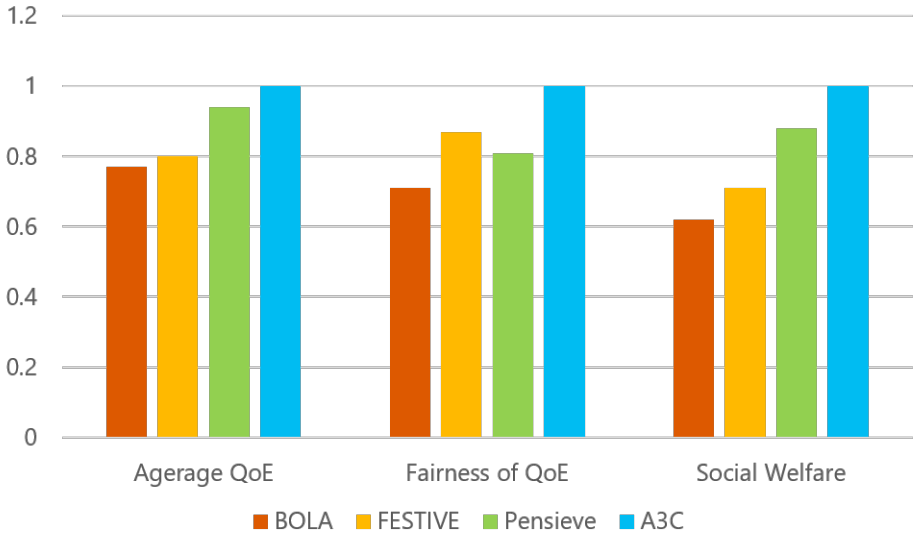


Fig. 5.6 QoE-based metrics

Fig. 5.7 Normalized metrics of 20 DASH clients sharing a bottleneck link.

social welfare function. Note that our proposed algorithm doesn't only outperform other algorithms in terms of QoE-fairness. It also maintains a stable fairness index even for relatively large number of clients. This achieves better stability over the

original proposed SARSA algorithm. The main advantage is due to the powerful A3C algorithm.

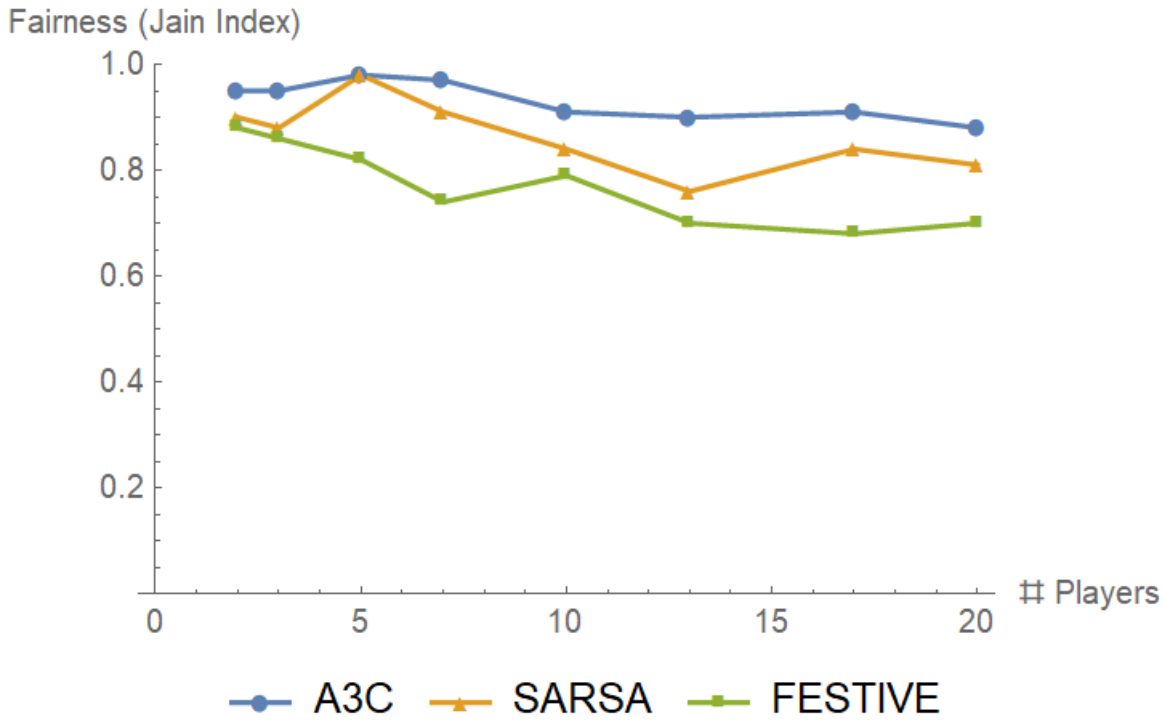


Fig. 5.8 Robustness of FESTIVE, original SARSA, and our A3C algorithms to varying number of active agents.

Chapter 6

Conclusion

We studied the problem of fair dynamic bandwidth allocation among concurrent video streaming users and proposed a new method that achieves some form of fairness across the users' QoE. In this concluding chapter, we summarize the contribution and achieved results and show possible directions for future research in this area.

6.1 Summary of the Results

In this work, we use the theoretic framework developed in [20] to implement a RL based server-side adaptive video streaming algorithm. The goal is to achieve a client-server cooperation to maximize fairness and efficiency without modifying DASH components and algorithms on the client-side (i.e. maintain the compatibility with DASH standard). We summarize our contributions as follows:

- We design and implement A3C algorithm to find the optimal solution of the proposed DEC-POMDP model. Our algorithm outperformed the proposed SARSA algorithm as well as all other state-of-the-art algorithms in terms of QoE-efficiency, QoE-fairness, and social welfare by as much as 16%, 21%, and 24% respectively.

- We extend the proposed framework from being a server-side algorithm to allow better client-server cooperation. This is challenging since unfairness is mainly caused by the clients' quality adaptation algorithms themselves [5]. However, allowing clients to run their own ABR mechanism (under server "supervision") helps the system scale without overloading the server.

- We suggest an improvement to the reward signal to reflect users QoE preferences (i.e. we implemented a personalize QoE model). According to [30], users can have different QoE preferences; some want the highest bitrate, while others are intolerant to rebuffering, or prefer smooth playback. The parameter of the reward model are learnable (tunable) by our agent.
- We resolve three important implementation issues: (1) we left restrictions on newly arriving and leaving flows so that the number of clients is not required to stay fixed during the streaming session. (2) we improved the learning speed by using a parallel and distributed learning algorithm. (3) we used a more realistic asynchronous setting, where actions (rate selection) of different agents are not forced to occur at synchronized time instances.

6.2 Future Works

For future research in the area RL-based DASH streaming, a number of extensions can be further studied. One is to use RNNs and LSTM networks (which are designed to remember larger amounts of history) to develop an online version of our algorithm. Further more, we can benefit from deploying the algorithm on a CDN to benefit from its fully observable nature. This will help the agents achieve better collaboration and converge to the optimal policy in shorter time.

Another area of research could be using RL to identify the root cause of QoE issues in real-time. In our case, the agents react to any high congestion event by reducing their bitrate without questioning the cause of this congestion. However, it is important to pinpoint the cause of such a problem and try to eliminate it in order to maintain the highest QoE all the time.

References

[1] (2004). Information technology — dynamic adaptive streaming over http (dash), https://www.iso.org/obp/ui/iso:std:iso-iec:23009:-1:ed-2:v1:cor:1:v1:en, part 1.

[2] (2018a). Akamai. dash.js, https://github.com/dash-industry-forum/dash.js/.

[3] (2018b). Cisco visual networking index: Forecast and methodology, 2016-2021.

[4] Akhshabi, S., Anantakrishnan, L., Dovrolis, C., and Begen, A. (2013). Server-based traffic shaping for stabilizing oscillating adaptive streaming players. In *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '13, pages 19–24. ACM.

[5] Akhshabi, S., Anantakrishnan, L., Dovrolis, C., and Begen, A. C. (2012). What happens when http adaptive streaming players compete for bandwidth? *Proceedings of the 22Nd International Workshop on NOSSDAV*, (12):9–14.

[6] Akhshabi, S., Begen, A., and Dovrolis, C. (2011). An experimental evaluation of rateadaptation algorithms in adaptive streaming over http. *Proceedings of the second annual ACM conference on Multimedia systems*, pages 157–168.

[7] Alreshoodi, M. and J.Woods (2013). Survey on qoe/qos correlation models for multimedia services. *International Journal of Distributed and Parallel Systems*, 4(3):53.

[8] Bakker, B. (2007). Reinforcement learning by backpropagation through an lstm model/critic. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 127–134.

[9] Bertsekas, D. (1995). *Dynamic programming and optimal control*. Athena Scientific Belmont, MA.

[10] Bokani, A., Hassan, M., Kanhere, S., and Zhu, X. (2015). Optimizing http-based adaptive streaming in vehicular environment using markov decision process. *IEEE Trans. on Multimedia*, 17(12):2297–2309.

[11] Bridle, J. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. *Neurocomputing NATO ASI Series (Series F: Computer and Systems Sciences)*, 68(1126-1165):227–236.

[12] Bäuerle, N. and Rieder, U. (2011). *Markov Decision Processes with Applications to Finance*. Springer.

References

[13] Chen, Y., Wu, K., and Zhang, Q. (2015). From qos to qoe: A tutorial on video quality assessment. *IEEE Communications Surveys and Tutorials*, 17(2).

[14] Chiariotti, F. (2014). Reinforcement learning algorithms for dash video streaming. Master's thesis, University of Padova.

[15] Chiariotti, F., D'Aronco, S., Toni, L., and Frossard, P. (2016). Online learning adaptation strategy for dash clients. *Proceedings of the 7th International Conference on Multimedia Systems*.

[16] Floyd, S., Handley, M., Padhye, J., and Widmer, J. (2008). Tcp friendly rate control (tfrc): Protocol specification (proposed standard).

[17] Haakon, R., Paul, V., Carsten, G., and Halvorsen, P. (2013). Commute path bandwidth traces from 3g networks: Analysis and applications. In *Proceedings of the 4th ACM Multimedia Systems Conference*, MMSys '13, pages 114–118, New York, NY, USA. ACM.

[18] Habachi, O., Hu, Y., van der Schaar, M., Hayel, Y., and Wu, F. (2012). Mos-based congestion control for conversational services in wireless environments. *IEEE Journal on Selected Areas in Communications*, 30(7):1225–1236.

[19] Hemmati, M. (2017). *New Bandwidth Allocation Methods to Provide Quality-of-Experience Fairness for Video Streaming Services*. PhD thesis, University of Ottawa, School of Electrical Engineering & Computer Science.

[20] Hemmati, M., Yassine, A., and Shirmohammadi, S. (2015). A dec-pomdp model for congestion avoidance and fair allocation of network bandwidth in rate-adaptive video streaming. *IEEE Symposium Series on Computational Intelligence*.

[21] Jiang, J., Sekar, V., and Zhang, H. (2014). Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transaction Network*, 22(1):326–340.

[22] Konda, V. and Tsitsiklis, J. (2003). On actor-critic algorithms. *SIAM J. Control Optim*, 42(4):1143–1166.

[23] Lederer, S., Müller, C., and Timmerer, C. (2012). Dynamic adaptive streaming over http dataset. *Proceedings of the ACM Multimedia Systems Conference*.

[24] Luca, D. C., Mascolo, S., and Palmisano, V. (2011). Feedback control for adaptive live video streaming. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pages 145–156. ACM.

[25] Mao, H., Netravali, R., and Alizadeh, M. (2017). Neural adaptive video streaming with pensieve. *Proceedings of the ACM SIGCOMM Conference*.

[26] Mnih, V., Badia, A., Mirza, M., and Silver, D. (2016). Asynchronous methods for deep reinforcement learning. *International Conference on Machine Learning*, page 928–1937.

[27] Mnih, V., K, K., and Silver, D. (2015). Human-level control through deep reinforcement learning. volume 25, page 529–533.

[28] Mnih, V., Kavukcuoglu, K., and Silver, D. (2013). Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*.

[29] Mok, R., Chan, E., and Chang, R. (2011a). Measuring the quality of experience of http video streaming. *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 485–492.

[30] Mok, R., Chan, E., Luo, X., and Chang, R. (2011b). Inferring the qoe of http video streaming from user-viewing activities. *Proceedings of the First ACM SIGCOMM Workshop on Measurements*.

[31] Muller, C., Lederer, S., and Timmerer, C. (2012). An evaluation of dynamic adaptive streaming over http in vehicular environments. *Proceedings of the 4th Workshop on Mobile Video*, pages 37–42.

[32] Nadembega, A., Hafid, A., and Taleb, T. (2014). An integrated predictive mobile-oriented bandwidth-reservation framework to support mobile multimedia streaming. *IEEE Trans. Wireless Commun.*, 13(12):6863–6875.

[33] Nam, H., Kim, K.-H., and Schulzrinne, H. (2016). Qoe matters more than qos: Why people stop watching cat videos. pages 1–9.

[34] Ozfatura, E., Ercetin, O., and Inaltekin, H. (2018). Optimal network-assisted multiuser dash video streaming. *IEEE Transactions on Broadcasting*, 64(1):1–19.

[35] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. (2003). Rtp: a transport protocol for real-time applications. 7.

[36] Seufert, M., Egger, S., Slanina, M., Zinner, T., Hossfeld, T., and Tran-gia, P. (2015). A survey on quality of experience of http adaptive streaming. *IEEE Communication Surveys & Tutorials*, 17(1):469–492.

[37] Silver, D., Huang, A., Maddison, C., Guez, A., and Sifre, L. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, (529):484–503.

[38] Spiteri, K., Urgaonkar, R., and Sitaraman, R. (2016). Bola: Near-optimal bitrate adaptation for online videos. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9.

[39] Sun, Y., Yin, X., and Jiang (2016). Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 272–285. ACM.

[40] Sutton, R. and Barto, A. (1998). *Reinforcement learning: An introduction*. MIT Press.

[41] Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA. MIT Press.

[42] Szepesvari, C. (2010). *Algorithms for Reinforcement Learning*. Morgan & Claypool Publishers.

[43] Thang, T., Ho, Q.-D., Kang, J. W., and Pham, A. T. (2012). Adaptive streaming of audiovisual content using mpeg dash. *IEEE Transactions on Consumer Electronics*, 58(1):78–85.

[44] Vriendt, J. D., Vleeschauwer, D. D., and Robinson, D. (2013). Model for estimating qoe of video delivered using http adaptive streaming. *Proc. IFIP/IEEE Int. Symposium on Integrated Network Management*, pages 1288–1293.

[45] Wei, S. and Swaminathan, V. (2014). Cost effective video streaming using server push over http 2.0. *Proc. IEEE 16th Int. Workshop Multimedia Signal Process. (MMSP)*, page 1–5.

[46] Wiering, Marco, van Otterlo, and Martijn (2012). *Reinforcement Learning: State-of-the-Art*. Springer.

[47] Wu, Y. and Tian, Y. (2017). Training agent for first-person shooter game with actor-critic curriculum learning. *In Submitted to Intl Conference on Learning Representations*.

[48] Yin, X., Jindal, A., Sekar, V., and Sinopoli, B. (2015). A control-theoretic approach for dynamic adaptive video streaming over http. *In SIGCOMM. ACM*.

[49] Zhang, J. and Ansari, N. (2011). On assuring end-to-end qoe in next generation networks: challenges and a possible solution. *IEEE Communications Magazine*, 49(7):185 – 191.

[50] Zhou, C., Lin, C.-W., , and Guo, Z. (2016). mdash: A markov decision based rate adaptation approach for dynamic http streaming. *IEEE Transactions on Multimedia*, 18(4).

[51] Zou, X. K. (2015). Can accurate predictions improve video streaming in cellular networks? *HotMobile ACM*.