Q-learning for Waiting Time Control in CDN/V2V Live streaming

Zhejiayu Ma, Frédéric Giroire, Guillaume Urvoy-Keller, Soufiane Roubia

To cite this version:

Zhejiayu Ma, Frédéric Giroire, Guillaume Urvoy-Keller, Soufiane Roubia. Q-learning for Waiting Time Control in CDN/V2V Live streaming. 2023 IFIP Networking Conference (IFIP Networking), Jun 2023, Barcelona, Spain. pp.1-9, 10.23919/IFIPNetworking57963.2023.10186429 . hal-04309215

HAL Id: hal-04309215
https://inria.hal.science/hal-04309215

Submitted on 27 Nov 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Q-learning for Waiting Time Control in CDN/V2V Live streaming

Zhejiayu Ma, Soufiane Roubia
EasyBroadcast
Nantes, France
{ma.zhejiayu, rouibia}@easybroadcast.fr

Frédéric Giroire and Guillaume Urvoy-Keller
Université Côte d'Azur, CNRS
Sophia-Antipolis, France
{frederic.giroire, guillaume.urvoy-keller}@univ-cotedazur.fr

***Abstract*—HTTP-based streaming has become the dominant technology for streaming due to the widespread adoption of the HTTP protocol. Many streaming providers use a combination of Content Delivery Network (CDN) and Viewer-to-Viewer (V2V) technology, known as Hybrid CDN/V2V live streaming, for both efficiency and cost-effectiveness. V2V technology allows for offloading streaming traffic from the CDN and reducing operational costs, and WebRTC technology facilitates direct V2V transfer, as it is natively supported by all browsers. In a WebRTC-based V2V network, some viewers cache the video chunks on their devices, while others wait and fetch chunks from their neighbors. A common strategy used to determine when a viewer should stop waiting for chunk delivery and revert to the CDN is called Random Waiting Time Control (RWC). However, due to the complex dynamics in the V2V system, RWC is far from optimal. In this work, we have formulated the Waiting Time Control determination problem as a reinforcement learning problem and proposed a Q-learning-based Waiting Time Control (QWC) solution. We conducted offline experiments in the Grid5000 [1] testbed and validated our results through a 14-day A/B testing in the wild. Our findings showed that QWC improves overall streaming Quality-of-Experience (QoE) in rebuffering (-29% fewer events), video quality (+17% higher), and buffer length (+5% longer), with a slightly improved V2V ratio (+5% more).**

***Index Terms*—hybrid P2P, live streaming, q-learning, machine learning**

I. Introduction

The HTTP-based streaming has become a standard way to effortlessly provide live streaming and VoD service thanks to the universality of the HTTP protocol. Popular protocols are Apple HLS (HTTP Live Streaming) and Google MPEG-DASH (Dynamic Adaptive Streaming over HTTP). The Content Delivery Network (CDN) providers, such as Akamai [2], Cloudfront [3], and Cloudflare [4] provide robust streaming delivery over multiple regions. However, the downside of CDNs is their cost which surges as the audience increases. As a solution, the V2V (viewer-to-viewer) technology based on WebRTC (Web Real-Time Communication) is used to reduce the operational cost of streaming services. It enables the browsers to open data channels with each other when they are watching the same stream so that the audience forms an active edge-device resource cache. This translates into fewer requests to the CDN.

This work has been supported by the French government through the UCA JEDI (ANR-15-IDEX-01) and EUR DS4H (ANR-17-EURE-004) Investments in the Future projects, by the European Project dAIEDGE, and by Inria associated team EfDyNet.

Video players, such as HLS, can be easily used with an open-source JavaScript library. This simplicity facilitates the deployment of Web-based streaming services. Technically, the HLS player controls how the video chunks are fetched. The default behavior is straightforward in HLS players: they issue an HTTP request to get the current manifest listing all the available video chunks, and they start fetching the first one. The manifest file is updated periodically in live streaming, and leads the player to continue playing forward.

The HLS player that has been enhanced with V2V technology operates in the following manner: Initially, the viewers form a V2V overlay network that comprises a maximum of 10 other viewers. Subsequently, the manifest file is retrieved in the same manner as before. An external JavaScript library manages the task of chunk fetching, where a viewer with a new chunk notifies its neighbors about the availability of this chunk, which they can then request. In this scenario, a crucial question is: "How long should a viewer wait for a chunk to be accessible in its neighborhood before resorting to the CDN?" The answer to this question is complex in the context of live streaming, where new chunks must be swiftly distributed to maintain all clients in sync. Additionally, there are challenges associated with peer dynamics, such as peer churn [5] (the departure of peers, which disrupts the V2V networks) and flash crowds [6] (a sudden surge of audience joining the live stream). A random waiting time is often employed to prevent synchronization among viewers, as reported in [7]. However, its influence on the V2V ratio[1] is difficult to predict. We advocate an adaptive strategy to adjust the waiting time.

In this work, we make the following contributions: (1) We formulate the problem of selecting the waiting time as a reinforcement learning task and use a Q-learning algorithm for the agent to learn to make optimal waiting time decisions. The agent is rewarded for a short V2V download time and penalized for an inappropriate waiting time that leads to a rebuffering event (such as when the buffer is less than the average chunk transfer time). (2) Through an A/B testing deployment in a real-world scenario, we demonstrate that our Q-learning agent can enhance the waiting time decisions and increase various QoE metrics for the users without sacrificing the V2V ratio.

[1]The percent of data downloaded from V2V instead of CDN. It is one of the KPIs (Key Performance Index) for Hybrid CDN/V2V Live Streaming systems

The paper is organized as follows: Section II frames the waiting control problem for a Hybrid CDN/V2V live streaming system and discusses the related work. In Section III, we introduce the Q-learning reinforcement learning algorithm and our design of state space, action space, and reward functions. We validate the design with an offline experiment in a large experimental platform, Grid5000 [1]. Finally, in Section IV, we analyze the statistics collected from an A/B testing campaign in order to compare the Random Waiting Time Control (RWC) and the Q-learning-based Waiting Time Control (QWC).

II. Background

To understand the Waiting Time Control problem in Hybrid CDN/V2V live streaming, we first present the player module of our HTTP-based hybrid CDN/V2V system. Second, we formulate the Waiting Time Control problem. Lastly, we present the related work by including other P2P or hybrid CDN/P2P systems.

Our hybrid CDN/V2V live streaming system relies on WebRTC for building the data channels. Our system architecture involves a tracker for matching the viewers and a JavaScript library which enables the connections among Web users. Similar approach is used in many other hybrid CDN/V2V systems [2, 8]–[10].

A. Hybrid CDN/V2V chunk fetching module

Before diving into the hybrid CDN/V2V chunk fetching module (CFM), we first present the two important modules of a pure-CDN HTTP streaming player. A typical pure CDN HTTP streaming player, such as the one provided by *HLS.js* [11] and *Dash.js* [12] has a CFM. In *HLS.js*, the module is called Fragment Loader (fLoader) while in *Dash.js* the module is named as XHRLoader (XMLHttpRequest Loader). The function of this module is relatively straightforward: it makes HTTP requests and returns the fetched resource to the player. Note that the player module is a built-in module that controls the fetching of manifest files, quality changes, playing, pausing, stopping, seeking, and downloading of the video chunks. The player module and the CFM are in a master-slave relationship where the CFM takes orders from the player module, as shown in Figure 1.

The process of incorporating V2V capability into the pure-CDN *HLS.js* and *Dash.js* involves a re-implementation of the CFM. This implementation entails: (1) Whenever the V2V neighbors have cached the chunks, one should be obtain the chunks from its neighbors; (2) If there is a cache miss or if the V2V fetching operation times out, traditional HTTP requests are sent to retrieve the chunks from the CDN.

In our platform, the player module remains unaltered, and we integrate a V2V-enabled CFM. This ”plug-and-play” feature enables the V2V service to be seamlessly integrated without the risk of disrupting other built-in functionalities of HLS or DASH, such as ABR (Adaptive Bitrate Streaming) algorithms.

B. Performance Metrics

To evaluate the system performance, we use different Quality-of-Service (QoS) and Quality-of-Experience (QoE) metrics:

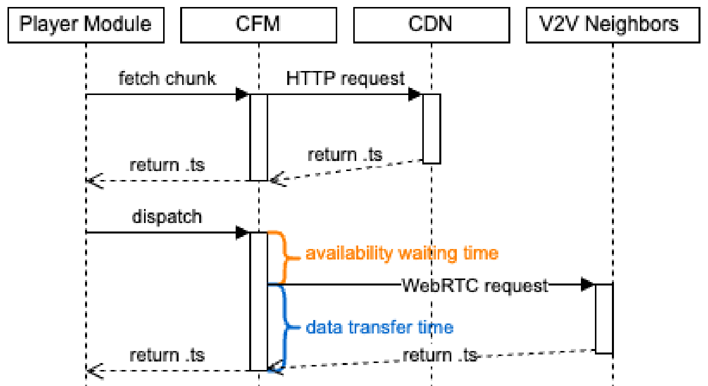


Fig. 1: The sequence diagram shows the player module, the chunk fetching module (CFM), and the CDN/V2V data fetching scheme. The total chunk fetching time in V2V mode comprises availability waiting time and data transfer time.

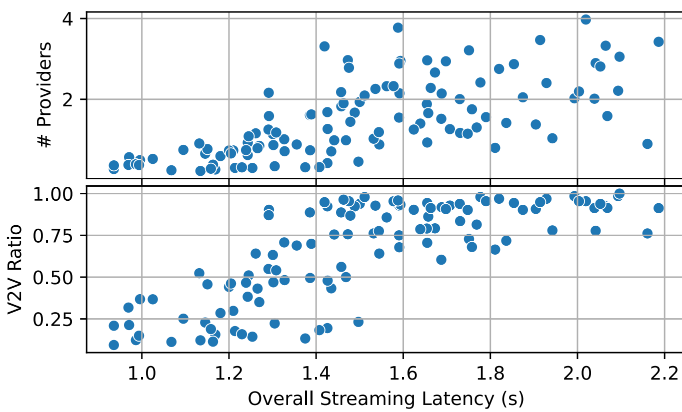
- The *V2V ratio* measures the percentage of data downloaded in V2V mode.
- The *Rebuffering Count (RebC)* measures the number of rebuffering events due to buffer starvation.
- The *Rebuffering Time (RebT)* measures the duration of the rebufferings.
- The *Buffer Length* measures, in seconds, how long the players' buffer is filled. If the player's buffer is low, there is a higher risk of rebuffering.
- The *Average Chunk Size* measures the stream quality in terms of the chunk size whose unit is bytes. For instance, in a multi-quality HLS streaming, each quality has a separate manifest file that lists the corresponding links for *.ts* objects. Those *.ts* objects have different file sizes. It is thus convenient to use the average chunk size to indicate the overall streaming quality.

C. Waiting Time Control Problem

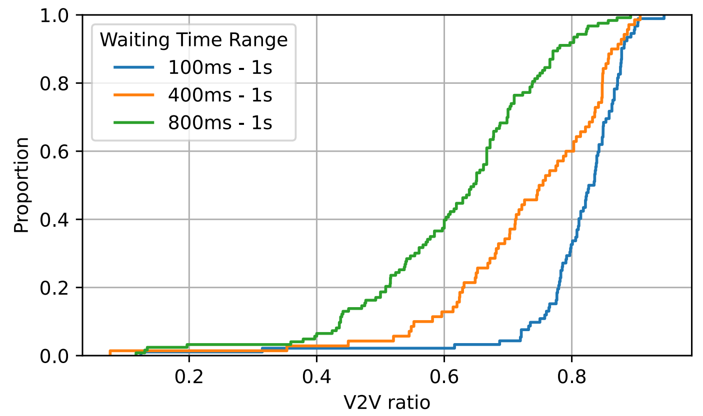
As mentioned earlier, the player module in the platform calls the CFM sequentially to fetch chunks. The CFM of the hybrid CDN/V2V system then decides whether to fetch the chunks from other neighbors or from the CDN. It's important to note that there is a waiting time that can be added between the moment the player requests a chunk and the moment the chunk is cached, as shown in Figure 1. This waiting time, known as the availability waiting time T_{av}, provides a window for the chunk to be downloaded by a neighboring viewer and advertised within the network, allowing the actual V2V chunk fetching to occur after T_{av} has passed.

In this section, we address the question of how the availability waiting time T_{av} influences the system performance. We do so from both the perspective of individual viewers and of the performance of the entire system. For individual viewers, we examine how the availability waiting time affects their QoE. For the performance of the entire system, we examine the impact of T_{av} on the V2V ratio.

1) Viewers having higher latency can achieve higher V2V ratio: If a peer selects a longer waiting time before fetching



(a) The Availability Versus Delay



(b) CDF of V2V ratio by waiting time

Fig. 2: In Figure 2a, we see that the peers that fetch chunks late enough experience a higher chunk availability from neighbors (given by the number of providers) during their session. Furthermore, the availability is often translated into higher V2V ratio. The late peers retrieve their chunks more frequently from other peers. As demonstrated in Figure 2b, the wider the range of the random waiting time, the higher the overall V2V ratio becomes.

a chunk, it will increase its latency in relation to the latest chunk of the stream. This extra waiting time provides an opportunity for other neighbors to download the next needed chunk and become providers. As a result, the peer will benefit from increased chunk availability and be able to download the chunk using V2V mode. This phenomenon is demonstrated in Figure 2a which reports results from an experiment in the Grid5000 platform involving hundreds of simulated viewers. We observe that the peers with a larger delay (larger overall streaming latency) from the live streaming playback head have more chunk providers (top plot) and, thus, are more likely to fetch the chunks from other users (bottom plot). However, the selection of waiting time is a trade-off and must balance various factors. If the waiting time is set too high, it may negatively impact the viewing experience, particularly for live events such as sports where users want to watch the content as soon as it is produced.

2) Desynchronization of users promotes V2V ratio.: From a system-wide viewpoint, availability is strongly tied to the level of desynchronization among the users. As the viewers become more desynchronized, the availability of chunks increases. The reasoning behind this is that if there is minimal desynchronization, the viewers are likely to be watching the same chunk, leaving little room for V2V exchanges. Another experiment was conducted on the Grid5000 platform where different groups of viewers were configured to take a random waiting time from different ranges. As shown in Figure 2b, the group with the widest range of waiting time (100ms - 1s) achieved the highest V2V ratio.

To conclude, the waiting time problem reveals a trade-off between a higher V2V ratio and the Quality of Experience.

D. Related work

In this section, we examine previous studies that have explored the chunk diffusion strategy in both pure P2P and hybrid CDN/P2P[2] networks. To emphasize the involvement of actual viewers, we use the term V2V throughout this work. One such study [13] models the system with a large window of chunks that can be downloaded between the playback head and the latest chunk. The system makes a decision every w seconds to schedule the chunks for downloading and to determine the seeders to which the requests are sent. This approach leads to a significant latency between the source of the streaming and the playback head, which is generally not ideal for live streaming. In contrast, in our work, the latest chunk is requested as soon as it is listed in the manifest file, reducing the latency between the source and the playback head.

While previous studies, such as LiveSky [14], have focused on the architecture of the V2V network and its impact on performance, our approach prioritizes QoE metrics, such as low latency and fewer rebuffering events. The LiveSky system proposed a hybrid mesh-pull and tree-push multiple substream approach, in which a parent peer can continuously push content to its children, as a solution to the long delays observed in pure pull approached in P2P live streaming networks [15]. However, this complex tree-based and mesh-based structure still involves at least three round-trip times (RTTs) in the download process, making it less efficient.

In contrast, our work aims to achieve better QoE metrics by relying on a CDN to reduce the complexity of the V2V network and remove the drawback of pure-pull in P2P systems with respect to delay. By utilizing a CDN, we can ensure that the latest chunk is requested soon after it is listed in the manifest file, minimizing latency and improving the overall streaming experience for the users. This approach aligns with the current goals of streaming service providers, which prioritize delivering high-quality, low-latency streams to their users.

PRIME [16] organizes the peers in different levels, implicitly

[2]In this work, we use the term V2V to emphasize the participation of actual viewers.

assuming that the source has low uploading capacity. Peers on the first level fetch the chunks available at the source, and then the peers on the second level pull from their parents and so on. Additional random links are used to form a mesh network, improving the resilience to churn. While they put significant effort in adjusting the number of children with the uplink capacity and content rate, such multi-level architecture creates latency which increases linearly with the level. Consequently, the users on the leaf nodes can experience a significant latency which degrades users' viewing experiences, especially in real-time events such as a sport event. In addition, the evaluation is done through simulations only. In contrast, we rely both on controlled experiments in a testbed and experiment in the wild to evaluate our proposal.

A recent closely related work is [13], where the authors consider also a hybrid CDN/P2P network for video live streaming. Their focus is on the interplay between the ABR algorithm and the CFM module since the bandwidth estimation done at the ABR level is difficult due the difference of downloading rate between a CDN server and a regular peer. In our work, we address a different problem, namely the trade-off between the V2V ratio and QoE metrics, using reinforcement learning to decide how long to wait for the neighbors before falling back to the CDN.

III. Design and Offline Experiments

This section details the Reinforcement Learning (RL) agent and the Q-learning [17] algorithm we use. Furthermore, we explain the challenges encountered during an offline experiment in Grid5000 to motivate the discussion on reward shaping and an interpretation of the learned policies.

A. Reinforcement Learning Formulation

The training setup follows the standard Reinforcement Learning framework to train a Q-learning model. In Q-learning, a tabular structure known as the Q-table is maintained, which stores the Q-values $Q(s,a)$ for all $s \in S$ and $a \in A$, where S represents the set of discrete states and A represents the set of discrete actions. The Q-value $Q(s,a)$ estimates the expected reward for the agent if it is in state s and takes action a, and then continues to interact with the environment according to some policy π until the end of the episode. A policy π maps the current state s to an action a. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the current value and the new information:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha[r_t + \gamma \max_{a \in A} Q(s_{t+1},a) - Q(s_t,a)]. \tag{1}$$

More details about Q-learning can be found in [17].

In our waiting time control problem, the Q-learning algorithm is part of the CFM as follows. Prior to each download request from the player module at each step t, the RL agent observes the current *state* $s_t = (b_t, n_t, p_t)$, encoding *Low-on-buffer*, *Low-on-neighbor*, and the *previous role*, respectively. The agent takes an *action* a_t which is to choose a range of waiting time. As a feedback for the *action* a_t taken, the agent receives a *reward* r_t based on the agent's contribution to V2V transfers and a *penalty* received if an undesired rebuffering event occurs.

1) State $s_t = (b_t, n_t, p_t)$: *Low-on-buffer* is determined by a threshold b_{low} (in seconds) so that $b_t = 1$ when the buffer is below b_{low}, otherwise $b_t = 0$. The low-on-buffer state is crucial in helping the agent to learn that a low buffer level could cause a rebuffering event. Similarly, *Low-on-neighbor* is determined by a threshold n_{low}. The rationale is that an agent might be able to identify the correct waiting time according to the number of connected neighbors, e.g., wait shorter when abundant resources exist. *Previous role* captures the previous behavior of an agent and the dynamics within the neighborhood.The possible values of the previous role include: (1) *CDN-only*, when the previous chunk was only downloaded from the CDN and no further exchange took place; (2) *CDN-seeder*, when the previous chunk was downloaded from the CDN and subsequently sent to neighbors; (3) *leecher-only*, when the previous chunk was obtained from a neighbor and no uploading occurred; (4) *leecher-seeder*, when the previous chunk was obtained from a neighbor and then sent to another peer.

2) Action $a_t \in \{1, 2, ..., k\}$: The Random Waiting Time Control (RWC) takes a random waiting time (milliseconds) value in the range $[0, 4000)$. This is the legacy behavior of our library in our operational system. In the Q-learning-based Waiting Time Control (QWC), we subdivide the bounding ranges into k ranges. For instance, suppose $k = 4$: we get four different actions $\{0, 1, 2, 3\}$ with the corresponding waiting time range $[0, 1000)$, $[1000, 2000)$, $[2000, 3000)$, and $[3000, 4000)$. When the agent takes action $a_t = 0$, it samples a value from the range $[0, 1000)$ as its waiting time.

3) Reward $r_t \in \mathbb{R}$: The reward function provides feedback to the agent to shape its behavior in fulfilling three distinct requirements: (1) promoting more V2V chunk exchanges, (2) reducing the waiting time, and (3) preventing rebuffering events. Thus, the reward function is composed of three distinct parts, as illustrated in Equation (2). The first two requirements are satisfied by the first line of the equation, where v_{t+1} is measured at time $t+1$ and indicates whether the chunk was downloaded from V2V after the agent took action a_t at time t. Similarly, u_{t+1} represents the number of times the chunk at time t was sent to the agent's neighbors. The discount factor $\eta < 1$ discounts the reward for longer wait, i.e., a longer wait results in a smaller reward. Note that the reward for chunk uploads (u_{t+1}) is not discounted in the reward function because uploading is a desirable behavior, even when the waiting time is long. The second line of the equation fulfills the last requirement by giving the agent a negative reward $-p$ in case the buffer is too low.

$$r_t = \begin{cases} v_{t+1} \times \eta^{a_t} + u_{t+1}, & \text{if buffer}_{t+1} > b_{danger} \\ -p, & \text{otherwise} \end{cases} \tag{2}$$

This reward function rewards both uploading and downloading from V2V. Similarly, we are also interested in testing different reward functions such as *download* which rewards

only downloading from V2V, and *upload* which rewards only uploading V2V chunk. We will detail the impact of choosing *download-upload*, *download* and *upload* as the reward function in Section III-B1.

4) Policy $\pi(s_t) = a_t$: We employ the $\epsilon-$greedy policy where the agent takes a random action with probability ϵ and a greedy action $\arg\max_a Q(s,a)$ with probability $1-\epsilon$. $\epsilon-$greedy policy helps to balance the exploration and exploitation in case the agent gets stuck on a sub-optimal policy [17, Ch. 2.3]. In a stationary environment, in the beginning, the agent starts interacting with the environment with a relatively high ϵ. As time progresses, the agent is supposed to switch from exploring to exploiting by shrinking ϵ. A common way to shrink ϵ is by using a decay parameter: after each action, $\epsilon := \max(\epsilon \times \delta, \epsilon_{\min})$.

B. Offline experiments on Grid5000

We conducted an offline experiment campaign on Grid5000 [1], a large-scale academic grid testbed. We set up a hybrid CDN/V2V live streaming system using a Kubernetes cluster with containerized headless Chrome browsers, which serve as viewers. Traffic control, using *tc* [18], can be added to limit the bandwidth of WebRTC. The configuration of the containerized Chrome browser and traffic control settings were done through a Docker file.

The experiments allow us to gain insight into (1) the influence of the reward function and (2) the advantages of using a blacklisting algorithm.

1) Reward function: The choice of reward function has a significant effect on the definition of roles within a neighborhood. As demonstrated in Figure 3a, three reward functions were tested: *download*, *download-upload*, and *upload*. The results show that *download-upload* and *upload* functions lead to improved performance over time, while the *download* function results in a gradual decline in the V2V ratio. The reason why a download-only reward fails is that by rewarding only V2V downloads, viewers are incentivized to make riskier decisions, such as waiting longer (see Figure 3b), which increases the chances of reverting to CDN.

2) Blacklisting: The previous experiments on the reward function were conducted under the assumption of equal download and upload bandwidths for all agents. However, in reality, agents have varying bandwidth capacities, particularly in terms of uplink. Therefore, we considered a scenario with 60 agents divided into three different bandwidth categories (4, 8, and 12 Mbps), each category representing 1, 2, and 3 times the streaming rate, each with 20 agents.

As shown in Figure 4a, the sliding average index over time for actions taken by agents in the three categories was analyzed. The shortest waiting time corresponds to action 0 while the longest waiting time corresponds to action 3. We expected that high-bandwidth agents would act as seeders and low-bandwidth ones as leechers. However, this was not the case, even with the use of a "download-upload" reward function, as agents across all three bandwidth categories took similar actions.

The reason for this behavior is a constraint in WebRTC. WebRTC uses UDP for low latency, but connections are unreliable and senders are not informed when a chunk is lost. As a result, low-bandwidth peers send chunks and receive rewards even if the chunk is lost, leading them to also choose a short waiting time like high-bandwidth peers.

To mitigate this, we designed a *blacklisting algorithm*. This prevents a peer from requesting chunks from neighbors who have sent lost chunks, and instead requests chunks from other neighbors. This reduces future chunk losses and reduces rewards for uploading among low-bandwidth peers, who will then wait longer to receive rewards from downloading in V2V instead. In contrast, high-bandwidth peers are not affected and continue acting as seeders.

As demonstrated in Figure 4b, with the blacklisting algorithm activated, we observe a distinct average action time series compared to before. There is improved separation between the bandwidth categories. Low-bandwidth peers tend to take larger actions and wait longer to download from V2V, while high-bandwidth peers take smaller actions and act as seeders.

IV. Experiment and Results

We compare QWC and RWC in production to address the following inquiries: (1) Does QWC enhance the QoE metrics compared to RWC? (2) How does QWC affect various user subgroups, particularly those with longer and shorter sessions? (3) How QWC performs during peak or non-peak hours?

A. Deployment in the wild

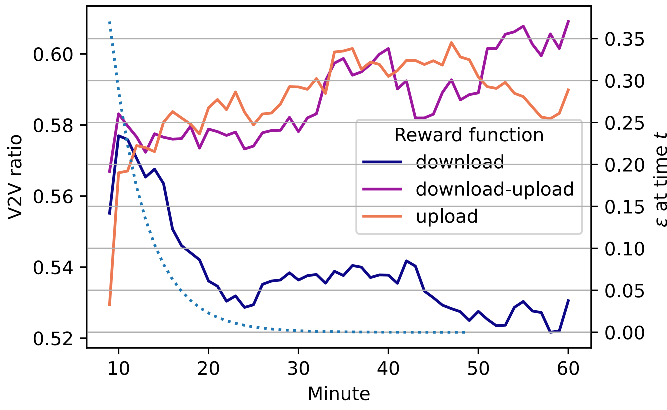
We used A/B testing campaigns for comparing QWC and RWC. The reason is that audience varies daily, so comparing two algorithms simultaneously removes the time-related variance. In A/B testing, the peers are assigned uniformly at random to one bucket. Then, they can only find and connect to other ones within the same bucket, and each bucket uses a different algorithm (RWC or QWC).

RWC and QWC are compared during a 14-day A/B testing deployment in the wild. RWC randomly chooses a waiting time between two bounding values; QWC learns through interactions how to choose a more specific range as discussed in Section III-A2.

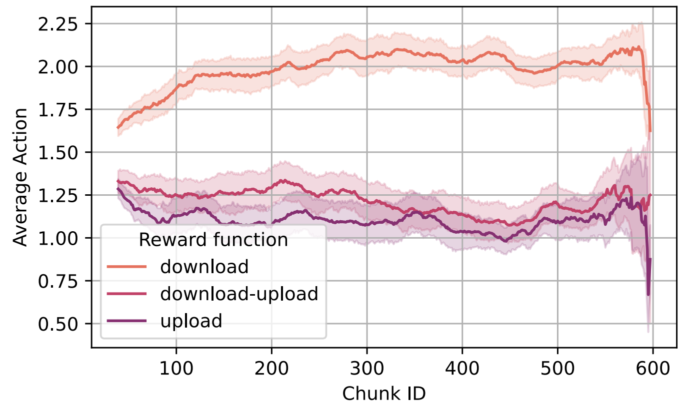
For each client, statistical data is sent by the JavaScript library every 30 seconds. The library aggregates the data downloaded from the CDN and V2V, the number of rebuffering events, and the video quality. We analyzed the metrics for each one-hour interval, resulting in 336 (14 x 24) data points for the online A/B testing. Each day typically has one audience peak for a given channel, and we further analyze the performance during peak and non-peak hours.

We tested QWC on two Moroccan Over-The-Top (OTT) TV channels, which are HLS channels. Each chunk (.ts file) consists of 6 seconds of video with three different video qualities: 500KB, 900KB, and 1.3MB per chunk. The default HLS Adaptive Bitrate algorithm controls the quality switches. The majority of users are from Morocco, and most use a PC with Windows or a mobile phone to watch the live stream.

As shown in Figure 5, during the fourteen days of A/B testing, we observed a strong periodicity of the number of viewers at different hours of the day. On some specific days,

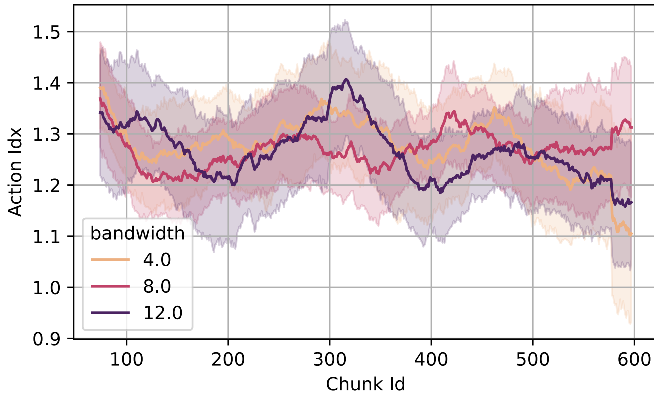


(a) *Upload* and *download-upload* groups do not suffer from performance drop as in *download* group ($\epsilon_{min} = 0.1$). These experiments are carried out in a homogeneous 8Mbps environment.

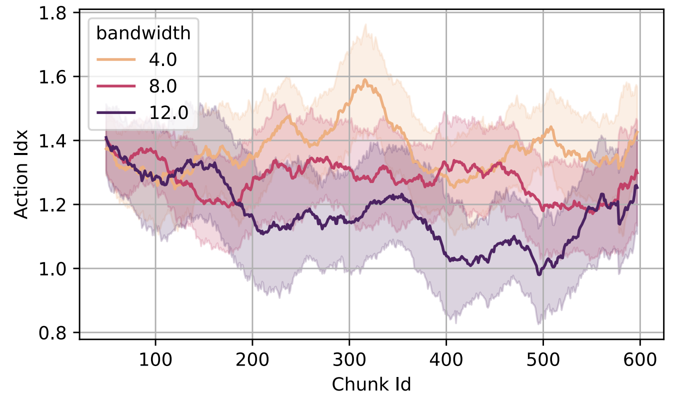


(b) The *download* reward function drives the users to taken a larger waiting time to get rewards whereas the *upload* and *download-upload* encourage the users to be a seeder.

Fig. 3: Reward functions and their impact on the V2V ratio ($\epsilon_{min} = 0.1$)



(a) Without blacklisting: all bandwidth categories have similar average index of actions.



(b) With blacklisting: low-bandwidth agents tend to wait more and high-bandwidth agents tend to wait less.

Fig. 4: Time-series of the average index of actions per peer bandwidth category without (a) or with (b) blacklisting. The figures demonstrate that the high-bandwidth peers automatically take the role of seeder when using the blacklisting algorithm.

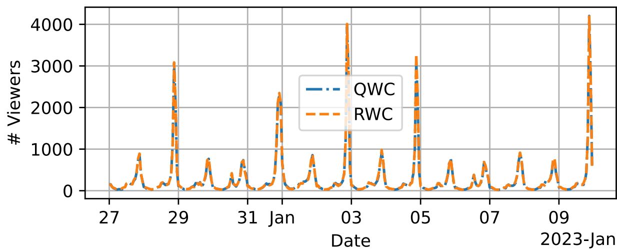


Fig. 5: Number of viewers **per bucket** demonstrates a strong periodicity with the peak time coming at night. The maximum reaches more than 4 000 viewers per bucket.

we saw larger peaks corresponding to highly popular events. The largest peak was on January 9, 2023, when each bucket had more than 4,000 viewers during the peak.

B. Reward function and Hyperparameters

For our A/B testing campaign, we use the *download-upload* reward function combined with the blacklisting algorithm (see Section III-B).

The values of the hyperparameters of the Q-learning algorithm are displayed in Table I. They were chosen based on the offline experiments. We set $\epsilon_{min} = 0.1$ to keep a minimum of 10% of exploration rate throughout the agent's lifetime. We use $\alpha = 0.7$ as the learning rate. Because of the high churn of such a live streaming system, a very large number of peers stay a very short time in the system, less than 10 minutes. These peers only take 100 actions (1 chunk per action, 10 chunks per minute). Thus, they should learn as fast as possible good moves as they do not have the time to learn optimal actions. For this reason, we chose a high value of the learning rate to speed up the learning.

The hyperparameter γ, taken from the range $[0, 1]$, represents the discount factor for future rewards. When $\gamma = 0$, the agent only considers short-term rewards, since future rewards are zeroed out. Conversely, when $\gamma = 1$, the agent values future rewards equally to current rewards and may be willing to delay them. However, in our problem, we do not want rewards to

name	α	γ	ϵ_{max}	ϵ_{min}	δ	k	b_{low}	n_{low}	b_{danger}	η	p
value	0.700	0.500	0.900	0.1	0.935	4	10	2	3	0.9	4

TABLE I: Hyperparameters settings

be delayed, as the sessions are short and agents will only experience a limited number of rewards.

Lastly, δ is the decay rate of ϵ. We set δ to be 0.935 so that after approximately 33 chunks (3.3 minutes), the agents will reach their minimum exploration rate. This is because we desire the users to spend one-third of their time learning and two-thirds of their time exploiting, as the average viewing time of viewers is 10 minutes.

C. Overall streaming performance

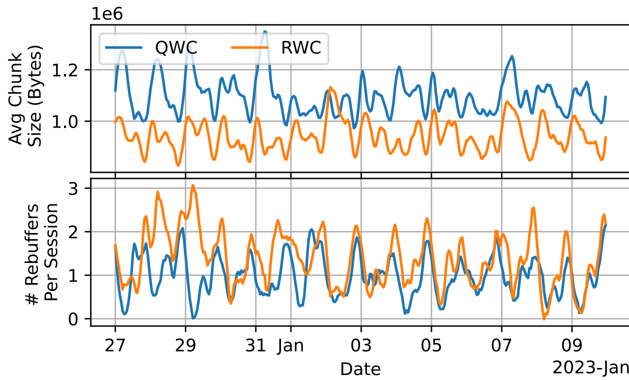


Fig. 6: Timeseries of the one-week long A/B testing. The top and bottom figures display the average chunk size and the rebuffering count per user. The average chunk size in the QWC bucket is significantly higher than viewers in the RWC bucket most of the time. Furthermore, the rebuffering is significantly reduced in QWC.

The main performance metrics are reported in Table II. They are all improved with the use of our Q-learning algorithm. Indeed, during the 14-day A/B testing comparing RWC and QWC, we see a 5% increase in the V2V ratio in QWC. Furthermore, there are 29% fewer rebuffering events in QWC than in RWC. QWC reduced the number of rebuffering per user from 1.399 to 0.981. The average duration of the rebuffering events is reduced by 8%. In terms of buffer length, the average buffer level increases from 126.73 to 133.01 seconds, a 5% of improvement. And finally, QWC offers a 17% improvement to the average chunk size. Figure 6 shows the timeseries of the two metrics which are significantly improved by QWC.

1) QWC encourages a higher bitrate: Adaptive HTTP Live Streaming is widely adopted and the quality is adapted according to the user's bandwidth and buffer occupation. The player issues an up-switch when the ABR (Adaptive Bitrate Algorithm) determines that a higher quality is more appropriate. There are several reasons for quality up-switches: (1) an abundance of buffer, (2) a shorter chunk fetching time, which leads to a higher bandwidth estimation, and (3) reduced rebuffering events, resulting in fewer quality down-switches.

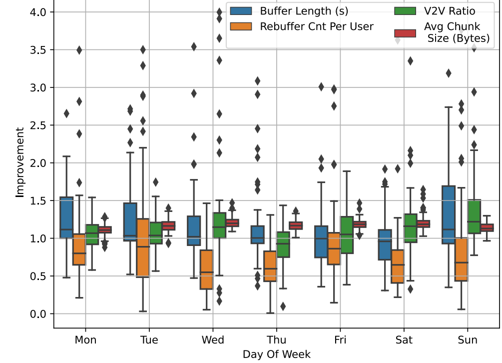


Fig. 7: Improvement of QWC over RWC grouped by Day of Week. Each instance represents the improvement for a one-hour duration. Note that, the median rebuffer count per user is smaller than 1.0 indicating a smaller number of rebuffering events when using our Q-learning algorithm.

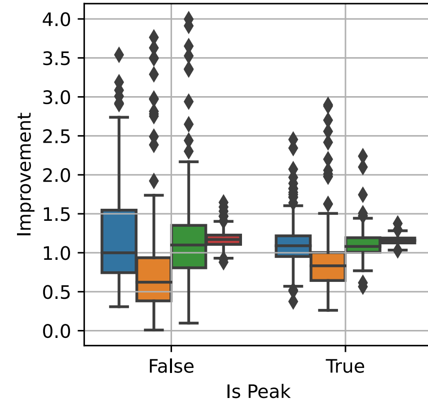


Fig. 8: Improvement of QWC over RWC grouped by peak (IsPeak=True) vs non peak (IsPeak=False) hours. This figure shares the same legend as Figure 7. Each point is labeled as Peak or Non-peak based on the number of users. When the number of users is over 200, we label the hour as a peak hour.

We now detail the analysis in terms of **buffer length**, **chunk fetching time**, and **rebufferings**

Firstly, the **buffer length** is larger in QWC. Figure 7 reports the above metrics over the 14 days of the experiments. The blue boxplots show that the buffer is more abundant in QWC, as the median improvement is mostly above 1, except on Saturday. However, this could be due to the high variance of non-peak hours. To prove this point, we show in Figure 8 that the variance of the buffer length improvement is significantly higher during non-peak hours. During peak hours, we mostly observe improvements in terms of buffer length in the QWC bucket.

Secondly, the **chunk fetching time is shorter in QWC.** As shown in Figure 1, the chunk fetching time is a composition

Metric	V2VRatio	RebC/User	RebT/User	BufLen	AvgChunkSize
RWC	0.410	1.399	2.36	126.73	943KB
QWC	0.431	0.981	2.16	133.01	1100KB
Improvement	+5%	-29%	-8%	+5%	+17%

TABLE II: Summary of the A/B testing.

of waiting time and data transfer time. The chunk fetching time should be reduced in QWC. As previously mentioned in Section III-A, QWC rewards the agent to minimize the waiting time for chunks. The reward function specifically rewards the agent for waiting a shorter period of time to score a V2V chunk, while penalizing it for waiting longer. This leads to the agent learning the strategies: (1) when it lags behind its neighbors and a longer waiting time is not required, it waits for a shorter period to obtain a V2V chunk, (2) when it is ahead of its neighbors, it acts as a seeder for the next chunk to increase its rewards, and (3) when there are no active cached chunks from other viewers and waiting would result in the risk of rebuffering, the agent selects the CDN as its streaming source.

Thirdly, in terms of **rebuffering count**, QWC has shown a significant improvement as depicted in Figure 7. The highest improvement can be seen on Wednesday and Thursday, with a median improvement of 48% and 40% respectively. Nevertheless, Figure 8 reveals that the improvement in rebuffering count is less pronounced during peak hours, with a median decrease of 25%, compared to non-peak hours where the median decrease is 40%.

2) QWC slightly increases V2V ratio: One of the key factors to consider in determining the suitability of using QWC in a production environment is whether the reduced waiting time results in a negative impact on the V2V ratio. The results of our experiments, as depicted in Figure 7, show that the V2V ratio is improved on all days except Thursday, where a slight degradation is observed. However, on Wednesday, Saturday, and Sunday, the improvement is significant. It is important to note that the improvement occurs both during peak and non-peak hours, although the variance is higher during non-peak hours, leading to some periods of time with a slight degradation. Nevertheless, the V2V ratio is increased by 5% globally.

D. Per-viewer QoE analysis

The overall performance of the streaming system is improved with the use of the QWC method, as evidenced by the improvement of Quality of Experience (QoE) metrics. However, in a V2V streaming system, there is a high rate of churn, with a significant number of users entering and exiting the system. To better understand the impact of churn on the user's QoE, we examine the performance metrics based on the duration of the user's viewing session. Specifically, we analyze the performance metrics for users who stay in the system for more than 10 minutes and those who have an enter-and-leave behavior.

The results of this analysis are shown in Figure 9, which presents empirical Cumulative Distribution Function (CDF) plots comparing QWC and RWC for different metrics. The results for long sessions are shown at the bottom of the figure, while the results for short sessions are shown at the top. From this comparison, we can make several key observations.

For example, the comparison of the V2V ratio in the first column of Figure 9 reveals that 40% of the short sessions do not download any V2V chunks. Additionally, there is a slight difference between the V2V ratios of most agents in the short sessions, with the top 25% of users having a slightly higher V2V ratio. However, the V2V ratio becomes more polarized for the long sessions, with more mass accumulating at two bounding values. This indicates that the blacklisting algorithm effectively assigns roles to agents based on their uploading capacity.

Rebuffering events are not common in short viewing sessions, with approximately 75% of these sessions being rebuffering-free. In comparison, only about 50% of the long viewing sessions are rebuffering-free. The QWC method leads to fewer rebuffering events than the RWC method, with only around 25% of users experiencing more than three rebuffering events in QWC compared to more than four rebuffering events in RWC. This improvement is particularly evident for long sessions, which correspond to users who are more engaged with the content.

The last column of the results shows that QWC also leads to an improvement in the chunk size for both short and long sessions. In QWC, more than 50% of sessions have a chunk size of 1.2MB, compared to only around 30% in RWC. As expected, the improvement in chunk size is more pronounced for long sessions, which represent the actual viewers, than for short sessions.

V. Conclusion and Future research

This work addresses the waiting time control problem using a Q-learning algorithm in a commercial hybrid CDN/V2V live streaming system. We validated our design with a 14-day A/B testing campaign that demonstrated that our proposed solution can reduce the waiting time and improve buffer length without affecting the V2V ratio, significantly enhancing the viewers' QoE metrics, such as video quality and rebuffering events.

In our work, we utilized the Q-learning algorithm as the method of choice for solving the problem at hand. However, there are other approaches that could potentially be used for this problem as well. One such alternative is Deep Q-Networks (DQN) [19], which utilizes a neural network to approximate the Q-values. This allows for the system to consider a larger number of states, making it a more robust solution. Another approach that could be considered is the Deep Deterministic Policy Gradient (DDPG) [20]. This method allows for the agents to operate in a continuous action space, which eliminates the need for manually dividing the random waiting time range.

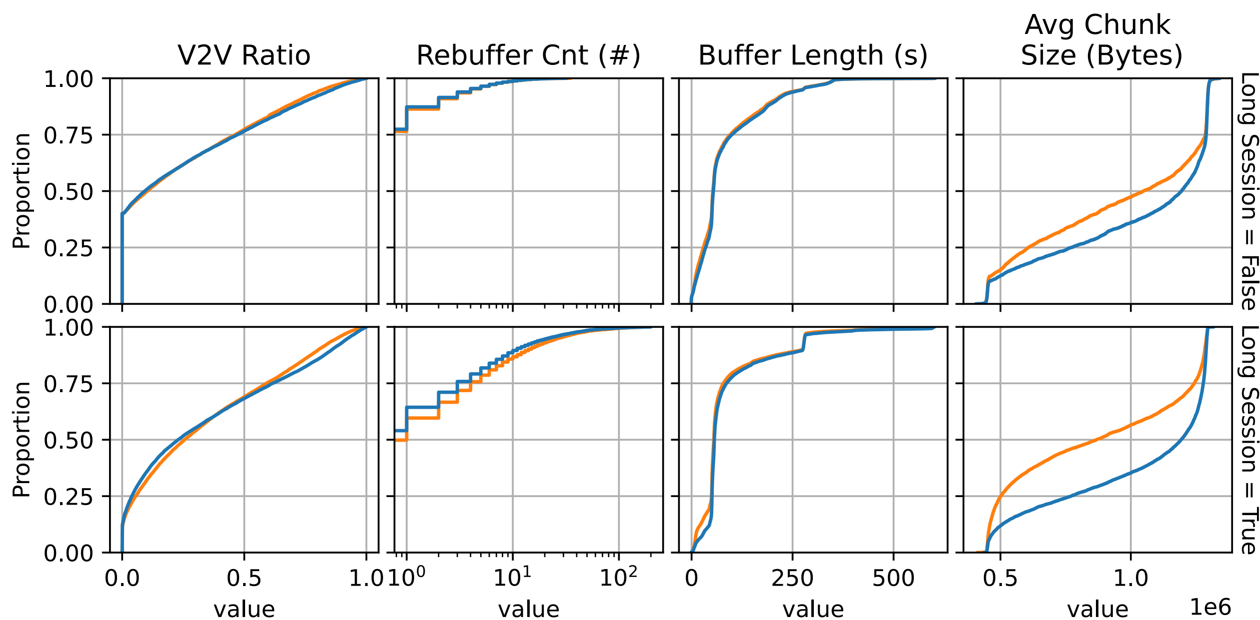


Fig. 9: The Empirical Cumulative Distribution Function plots of different metrics per session. The blue line represents QWC, and the orange line represents RWC. The first (second, resp.) row represents the short (long, resp.) sessions that last less (more, resp.) than 10 minutes. Improvements due to the Q-learning algorithm are more pronounced for users really watching the stream (long sessions).

In addition to modeling independent agents in our Multi-agent Reinforcement Learning approach [21], it is important to consider the risk of these agents adopting a self-serving behavior and locking into specific patterns of action. To mitigate this risk, a potential avenue for future research is to explore the use of cooperative agents that share information with one another. The idea behind this is that by working together, the individual actions of the agents can be combined to form a joint action, which can lead to faster convergence and a more optimal solution. This cooperative Multi-agent Reinforcement Learning approach aligns all agents towards a shared goal of achieving a higher V2V ratio, promoting collaboration and coordination among the agents. In this way, the use of cooperative agents can result in a more efficient and effective solution, improving the overall performance of the system.

References

[1] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[2] "Akamai," 2020. [Online]. Available: https://www.akamai.com/fr/fr/

[3] "Cloudfront," 2022. [Online]. Available: https://aws.amazon.com/cloudfront/

[4] "Clouflare," 2022. [Online]. Available: https://www.cloudflare.com/

[5] J. Liu, S. G. Rao, B. Li, and H. Zhang, "Opportunities and challenges of peer-to-peer internet video broadcast," *Proceedings of the IEEE*, vol. 96, no. 1, pp. 11–24, 2007.

[6] A. Stavrou, D. Rubenstein, and S. Sahu, "A lightweight, robust p2p system to handle flash crowds," in *10th IEEE International Conference on Network Protocols, 2002. Proceedings.* IEEE, 2002, pp. 226–235.

[7] F. R. N. Barbosa and G. L. de Souza Filho, "Bemtv: Hybrid cdn/peer-to-peer architecture for live video distribution over the internet."

[8] Z. Ma, S. Rouibia, F. Giroire, and G. Urvoy-Keller, "Neighbor selection strategies in the wild for cdn/v2v webrtc live streaming: Can we learn what a good neighbor is?" in *2022 IEEE 47th Conference on Local Computer Networks (LCN).* IEEE, 2022, pp. 295–298.

[9] (2020) Cdnetworks. [Online]. Available: https://www.cdnetworks.com/

[10] Z. Ma, S. Roubia, F. Giroire, and G. Urvoy-Keller, "When locality is not enough: Boosting peer selection of hybrid cdn-p2p live streaming systems using machine learning," in *Network Traffic Measurement and Analysis Conference (IFIP TMA 2021)*, 2021.

[11] video dev, "Hls.js," 2022. [Online]. Available: https://github.com/video-dev/hls.js

[12] Dash-Industry-Forum, "Dash.js," 2022. [Online]. Available: https://github.com/Dash-Industry-Forum/dash.js

[13] H. Yousef, J. Le Feuvre, P.-L. Ageneau, and A. Storelli, "Enabling adaptive bitrate algorithms in hybrid cdn/p2p networks," in *Proceedings of the 11th ACM Multimedia Systems Conference*, 2020, pp. 54–65.

[14] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li, "Design and deployment of a hybrid cdn-p2p system for live video streaming: experiences with livesky," in *Proceedings of the 17th ACM international conference on Multimedia*, 2009, pp. 25–34.

[15] M. Zhang, J.-G. Luo, L. Zhao, and S.-Q. Yang, "A peer-to-peer network for live media streaming using a push-pull approach," in *Proc. of the 13th annual ACM international conference on Multimedia*, 2005, pp. 287–290.

[16] N. Magharei and R. Rejaie, "Prime: Peer-to-peer receiver-driven mesh-based streaming," *IEEE/ACM transactions on networking*, 2009.

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[18] B. Hubert, "tc: Manipulate traffic control settings," 2001. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[21] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proceedings of the tenth international conference on machine learning*, 1993, pp. 330–337.