SOFTWARE DEFINED RESOURCE ALLOCATION FOR ATTAINING QOS AND QOE GUARANTEES AT THE WIRELESS EDGE

A Dissertation

by

RAJARSHI BHATTACHARYYA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2019

Major Subject: Computer Engineering

ABSTRACT

Wireless Internet access has brought legions of heterogeneous applications all sharing the same resources. However, current wireless edge networks that provide Quality of Service (QoS) guarantees that only cater to worst or average case performance lack the agility to best serve these diverse sessions. Simultaneously, software reconfigurable infrastructure has become increasingly mainstream to the point that dynamic per packet and per flow decisions are possible at multiple layers of the communications stack. In this dissertation, we explore several problems in the space of cross-layer optimization of reconfigurable infrastructure with the objective of maximizing user-perceived Quality of Experience (QoE) under the resource constraints of the Wireless Edge.

We first model the adaptive reconfiguration of system infrastructure as a Markov Decision Process with a goal of satisfying application requirements, and whose transition kernel is discovered using a reinforcement learning approach. Our context is that of reconfigurable (priority) queueing, and we use the popular application of video streaming as our use case. Self declaration of states by all participating applications is necessary for the success of the approach. This need motivates us to design an open market-based system which promotes the truthful declaration of value (state). We show in an experimental setup that the benefits of such an approach are similar to those of the learning approach. Implementations of these techniques are conducted on off-the-shelf hardware, which have inherent restrictions on reconfigurability across different layers of the network stack. Consequently, we exploit a custom hardware platform to achieve finer grained reconfiguration capabilities like per packet scheduling and develop a platform for implementation and testing of scheduling protocols with ultra-low latency requirements. Finally, we study a distributed approach for satisfying strict application requirements by leveraging end user devices interested in a shared objective. Such a system enables us to attain the necessary performance goals with minimal use of centralized infrastructure.

DEDICATION

To my family.

ACKNOWLEDGMENTS

I would first like to thank my advisor, Dr. Srinivas Shakkottai, who has guided, inspired and motivated me throughout the duration of my studies. He has been very patient and understanding with me, for which I am ever grateful.

I am also thankful to my committee members, Dr. A.L.N. Reddy, Dr. Jean-Francois Chamberland Tremblay and Dr. Radu Stoleru for their guidance. Their insights have been invaluable towards the completion of my dissertation.

Finally, I would like to thank my lab mates over the years for their collaboration on several projects. Their friendship and support has made this journey very enjoyable and fufilling.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of advisor Dr. Srinivas Shakkottai, Dr. A.L. Narasimha Reddy and Dr. Jean-Francois Chamberland-Tremblay of the Department of Electrical and Computer Engineering and Dr. Radu Stoleru of the Department of Computer Science and Engineering.

The collection of data for the experiments described in this dissertation, and the subsequent analysis, was conducted in collaboration with Bainan Xia, KiYeob Lee, Suman Paul, Simon Yau, Ping-Chun Hsieh, Desik Rengarajan, Archana Bura and Mason Rumuly. All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by a research assistantship from Texas A&M University.

NOMENCLATURE

AP	Access Point
CDF	Cumulative Distrbution Function
DQN	Deep Q Network
DP	Decision Period
DQS	Delivery Quality Score
HD	High Definition
MAC	Medium Access Control Layer
MDP	Markov Decision Process
NQB	Number of QoE bins
NSB	Number of Stall bins
NUC	Next Unit of Computing
PHY	Physical Layer
QoS	Quality of Service
QoE	Quality of Experience
RL	Reinforcement Learning
RSSI	Received Signal Strength Indicator
RTT	Round Trip Time
SDN	Software Defined Networks
SDR	Software Defined Radio
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Rapid growth of Wireless networks has led to the proliferation of heterogeneous applications. Although these applications all share the same reources, they have varied requirements for optimal performance. Current wireless edge networks that provide Quality of Service (QoS) guarantees that only cater to worst or average case performance lack the agility to best serve these diverse sessions. Simultaneously, software reconfigurable infrastructure has become increasingly mainstream to the point that dynamic per packet and per flow decisions are possible at multiple layers of the communications stack. We explore several problems in the space of cross layer optimizations in reconfigurable infrastructure with the objective of maximizing user-perceived Quality of Experience (QoE) under the resource constraints of the Wireless Edge.

In the first chapter, we design and develop a system which is able to exploit such reconfigurability to enable different configurations, measure the impact of a particular configuration on the application performance (QoE), and adaptively select a new configuration based on its influence/effect on application performance. This feedback loop is modelled as a Markov Decision Process and we present QFlow, which instantiates this feedback loop as an application of reinforcement learning (RL). Our context is that of reconfigurable (priority) queueing, and we consider the popular application of video streaming as our use case. We develop both model-free and model-based RL approaches that are tailored to the problem of determining which clients should be assigned to which queue at each decision period. Through experimental validation, we show how the RL-based control policies on QFlow are able to schedule the right clients for prioritization in a high-load scenario to outperform the status quo, as well as the best known solutions with over 25% improvement in QoE, and a perfect QoE score of 5 over 85% of the time.

The success of the learning approach presented in Chapter 1 is dependent on the truthful declaration of state by all clients participating in the system. This motivates us to develop a system which promotes the clients to reveal their true value (state) in the next chapter. We demonstrate Flow-Bazaar, an open market-based approach to create a value chain from the end-user of an application on one side, to algorithms operating over reconfigurable infrastructure on the other. The ecosystem enables disparate applications to obtain necessary resources for optimal performance. We compare the performance of this approach with the learning approach on an experimental testbed and show that the market-based approach exhibits similar application performance guarantees.

Working with off-the-shelf hardware (Access Points) during the course of the first two projects exposed us to the inherent restrictions present on reconfigurability across different layers of the network stack. Therefore, we explore a custom hardware platform to achieve finer grained reconfiguration capabilities like per-packet scheduling in Chapter 3. We present PULS, a processor-supported scheduling implementation for testing of downlink scheduling protocols with ultra-low latency requirements. Based on our decoupling architecture, programmability of delay sensitive scheduling protocols is done on a host machine, with low latency mechanisms being deployed on hardware. This enables flexible scheduling policies on software and high hardware function re-usability, while meeting the timing requirements of the Medium Access Control layer. We performed extensive tests on the platform to verify the latencies experienced for per packet scheduling, and present results that show packets can be scheduled and transmitted under 5 ms in PULS. Using PULS, we implemented four different scheduling policies and provide detailed performance comparisons under various traffic loads and real-time requirements. We show that in certain scenarios, the optimal policy can maintain a loss ratio of less than 1% for packets with deadlines, while other protocols experience loss ratios of up to 65%.

The techniques for resource allocation discussed hitherto are based on a Server (Access Point)-Client architecture. In the last chapter, we design a distributed approach for meeting application performance requirements by leveraging end user devices interested in a shared objective. Such a system enables us to achieve the performance guarantees with minimal use of centralized infrastructure. We consider a group of co-located wireless peer devices that desire to synchronously receive a live content stream. The devices are each equipped with an expensive unicast base-station-to-device (B2D) interface, as well as a broadcast device-to-device (D2D) interface over a shared medium. The stream is divided into blocks, which must be played out soon after their initial creation. We transform the problem into the two questions of (i) deciding which peer should broadcast on the D2D channel at each time, and (ii) how long B2D transmissions should take place for each block. We analytically develop a provably-minimum-cost algorithm that can ensure that QoS targets can be met for each device. We study its performance via simulations, and present an overview of our implementation on a testbed consisting of Android phones.

2. QFLOW: A REINFORCEMENT LEARNING APPROACH TO HIGH QoE VIDEO STREAMING OVER WIRELESS NETWORKS*

2.1 Introduction

Growth in wireless networks is being fueled by a multitude of new applications that require a diverse set of link characteristics for optimal operation. However, current algorithms on the wireless edge (last-hop link) are geared towards an average or worst case performance in an application-agnostic manner. They are thus ill-equipped to adapt in order to optimize the user Quality of Experience (QoE) in real time. QoE is a number in the interval $[1, 5]$ that indicates end-user satisfaction, with a QoE of 5 being the best. However, such optimization is needed in upcoming dense, small cell deployments in WiFi and 5G networks that are expected to support high and diverse loads. This disconnect raises the question of whether it is possible to develop a framework for network reconfiguration that can ensure high QoE for users in a fair manner?

From software defined networking (SDN) at the network layer on commercial routers, to different sub-layers of PHY/MAC on software defined radio (SDR) platforms, it is becoming increasingly easier to reconfigure networking equipment at all layers of the networking stack. Among these sub-layers, a fundamental entity that impacts per-packet and per-flow performance is the behavior of queues at the router. How many queues exist on the router, at what rate they obtain service, and which flows are assigned to these queues all impact statistical QoS performance metrics, such as throughput, RTT, jitter and loss rate that flows experience. Indeed, the fundamental nature of queueing is the reason for much effort on the design and evaluation of throughput or delay optimal scheduling mechanisms [1, 2, 3, 4].

Even when differentiated queueing mechanisms are available, exploiting them for maximizing system-wide benefit requires a feedback control loop of the kind shown in Figure 2.1. First, we need to *configure* the system in terms of assigning flows to queues. Second, we need to *measure* the

*This chapter has been submitted to MobiHoc, 2019 for publication and is under review. It is also available at https://arxiv.org/abs/1901.00959.

impact of the configuration on QoE and relevant application state at the end-user. Third, we need to *learn* what is the relation between realized QoE and the configuration used (using a combination of offline and online learning). Finally, we need to *adapt* the policy used for configuration as we learn in order to maximize performance goals. Note that such a control loop applies to problem of choosing optimal reconfigurations at all layers of network stack. However, the fundamental nature of queueing implies that first order gains might be best attained through such adaptive queue control.

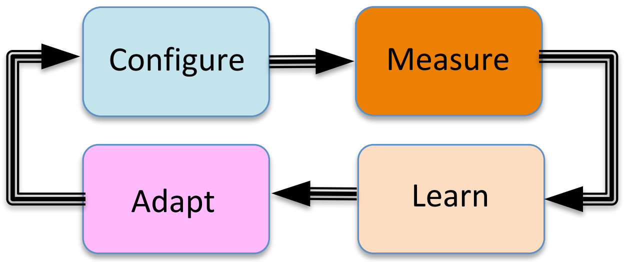

Figure 2.1: Feedback loop for configuration selection.

Posed in this manner, the application QoE and other measurable application-specific parameters (such as buffered seconds of video) is the observable application state of the system, whose evolution is mediated through the assignment of flows to queues. The network QoS statistics of each queue are hidden variables that cause transitions to the application state, potentially in a stochastic manner. The decision of which flows to assign to what queue determines the state transitions that a particular application is exposed to, and must be done in a manner that maximizes QoE over all applications. Thus, the control loop in Figure 2.1 can be interpreted as a Markov Decision Process (MDP) whose transition kernel is unknown, and which could potentially be discovered using reinforcement learning.

In this work, our goal is to design, implement, and evaluate QFlow, a platform for reinforcement learning that instantiates the feedback control loop described above. Here, we choose video streaming as the application of interest using the case study of YouTube, since video has stringent network requirements and occupies a majority of Internet packets today [5]. Our context is that of a WiFi access point that faces a high demand situation. Performance over high capacity wired backhaul links is near-deterministic, and hence the resource constraint usually applies to the last hop wireless link to a mobile device.

2.2 Main Results

The main innovations in QFlow that address the elements shown in Figure 2.1, and our experimental results are as follows.

Queue Configuration: We enable reliable delivery of configuration commands to hardware that can support re-configuration. We extend the OpenFlow protocol (currently limited to the network layer) in a generic manner that enables us to use it reconfigure queueing mechanisms. We select commercially available WiFi routers with Gigabit ethernet backhaul as the wireless edge hardware. Reconfigurable queueing is attained by leveraging differentiated queueing mechanisms available in the Traffic Controller (tc) package by installing OpenWRT (a stripped-down Linux version). Here, we can choose between queueing disciplines and set filters to assign flows to queues. Details are presented in Section 2.5.

Measurement of Application State and QoE: We enable continuous monitoring of client-specific application state consisting of buffered seconds of video and stall duration (when the video re-buffers). These monitors at the WiFi router and the mobile station, are compatible with our OpenFlow extensions, and use the protocol to periodically send statistics to the OpenFlow controller for processing. We continuously predict the QoE of the ongoing application (video streaming) flows as a function of the application state using existing maps of the relationship between video events (such as stalls) and QoE. Details are presented in sections 2.4, 2.5.

Model-Free Reinforcement Learning: We develop a model-free reinforcement learning (RL) method that enables adaptation to the current QoE and application state over all users to maximize the discounted sum of QoEs. We design a simulator that approximates the evolution of the underlying system, and its impact on application state and QoE. We use the simulator to train a

Q-Learning approach in an offline manner, with non-linear function approximation using a neural network. This so-called Deep Q Network (DQN) is able to account for state space explosion across the users and provides a Q-function approximation for all states. Details are presented in section 2.6.

Model-Based Reinforcement Learning: We next develop a model-based RL approach based on the observation that the state evolution of an individual client is independent of others given the action (queue assignment). We first use measurements conducted over the system using a range of control policies to empirically determine the transition probabilities on a per-client basis, and then use the independence observation to construct the system transition kernel (this applies to the vector of all client states taken together). While doing so, we reduce the system state space by discretization and aggregation to a subset of frequently observed system states. Finally, we solve the MDP numerically to obtain the model-based policy. Details are presented in section 2.7.

Experimental Results: The experimental configuration consists of a single queue in the base (vanilla) case, and two reconfigurable queues in the adaptive case. We conducted experiments in both a static scenario of 6 clients, as well as a dynamic one in which anywhere between 4 and 6 clients are in the system at a given time. Apart from model-free and model-based RL, we also implemented round-robin assignment, greedy maximization of expected QoE, and greedy selection of the clients with lowest video buffers (this policy has been shown to ensure low probability of stalling [6]). Our results on adaptive flow assignment (Section 2.8) reveal that the vanilla approach of treating all flows identically has significantly worse average QoE than adaptive approaches.

More interestingly, both the model-based and model-free approaches manage to ensure that any given client experiences a perfect QoE of 5 over 85% of the time, whereas the best that any other policy is able to achieve is only about 60%, while vanilla manages even less at about 50%. This impressive performance improvement of about 25-30% indicates that by selecting flows in need of QoE improvement (due to high likelihood of stalls in the near future), RL-based adaptive flow assignment improves QoE for the majority of clients.

2.3 Related Work

Our work brings together several different areas ranging from SDN, QoS, QoE and machine learning.

OpenFlow and Configuration: There has been much recent interest in extending the SDN idea to other layers. For example, CrossFlow [7, 8] uses SDN OpenFlow principles to control networks of Software Defined Radios. In ÆtherFlow [9], the SDN/OpenFlow framework is used to bring programmability to the Wireless LAN setting. They show that this type of system can handle hand-offs better than the traditional 802.11 protocol. These SD-X extensions (X being the MAC layer in this case) focus on centralized configuration of the hardware and do not provide sample statistics on performance that we desire. Closer to our theme, systems such as AeroFlux [10] and OpenSDWN [11] develop a wireless SDN framework for enabling prioritization rules for flows belonging to selected applications (such as video streaming) via middle-boxes using packet inspection. However, they do not tie such prioritization to the impact on application QoE or end-user value across competing applications from multiple clients. Nor do they use measured QoS statistics as feedback.

QoE Maps: The map between QoS and QoE has been studied recently, particularly on the wired network. The work in this space attempts to determine the QoS properties of a network, and then based on data obtained directly from an application, match the observed QoS to the corresponding QoE. Mok et al. [12] describe a method for determining the QoE for HTTP Streaming, focusing on the choice of initial streaming rate for maximizing QoE. Other work focuses on different applications, such as Skype [13] or general Web services [13], to identify conditions that are sufficient to meet the average QoE targets for those applications. Different from these, we desire a continuous estimate of QoE as a function of player state (such as buffered video seconds) and network state (QoS statistics).

SDN-based Video Streaming: A number of systems have been proposed to improve the performance and QoE of video streaming with SDN. One direction is to assign video streaming flows to different network links according to various path selection schemes [14] or the location of bottlenecks detected in the WAN [15]. In the home network environment, the problem shifts from managing the paths of video traffic to sharing the same network (link) with multiple devices or flows. VQOA [16] and QFF [17] employ SDN to monitor the traffic and change the bandwidth assignment of each video flow to achieve better streaming performance. However, without an accurate map of action to QoE, the controller can only react to QoE degradation passively.

Reinforcement Learning: An RL approach is natural for the control of systems with measurable feedback under each configuration. The idea of using model-free RL in the context of video streaming rate selection was explored in [18]. The work can be seen as the complement of our own. Whereas we are interested in allocating network resources (at the wireless edge) to suit concurrent video streams, their goal is to choose the streaming rate to suit the realized network characteristics.

2.4 System Model

We consider a system in which clients are connected to an wireless Access Point (AP) in a high demand situation. We choose video streaming as the application of interest using the case study of YouTube, since video has stringent network requirements and occupies a majority of Internet packets today [5]. Our goal is to maximize the overall QoE of all the clients in this resource constrained situation.

The AP has a high priority and low priority queue. Clients assigned to the high priority queue typically experience a better QoS (higher bandwidth, lower latency etc.) when compared to the clients assigned to the low priority queue. The controller assigns clients to each of these queues at every decision period (DP; 10 seconds in our implementation). Determining the optimal strategy is complex, since the controller does not have prior knowledge of the system model. Hence, the controller must *learn* the system model and/or control law.

2.4.1 Markov Decision Process

We consider a discrete time system where time is indexed by $t \in \{0, 1, ...\}$. At each DP ($t = 0, 1, 2..$) the controller makes an assignment of clients to queues, and observes the system. Based on its observation and previous assignment, the controller makes an assignment in the next DP, eventually learning the system model empirically. This class of problem falls within the Reinforcement Learning (RL) paradigm, and thus can be abstracted to a general RL framework consisting of an *Environment* that produces *states* and *rewards* and an *Agent* that takes *actions*.

Environment: The environment is composed of clients and the AP. Let $\mathcal{C}$ denote the set of clients.

State: Each client keeps track of its state which consists of its current buffer (the number of seconds of video that it has buffered up), the number of stalls it has experienced (i.e., the number of times that it has experienced a break in playout and consequent re-buffering), and its current QoE (a number in $[1, 5]$ that represents user satisfaction, with 5 being the best). The state of the system is the union of the states of all clients. Let s_t^c denote the state of client c at time t and s_t denote the state of the system,

$$s_t^c = \textit{[Current Buffer State, Stall Information, Current QoE]}\ \forall c \in \mathcal{C}$$

$$s_t = [\cup_{\forall c \in \mathcal{C}} s_t^c]$$

Agent: The controller is the agent, which takes an action $a_t \in \mathcal{A}$ (queue assignment) in every decision period in order to maximize its *expected discounted reward.* Let a_t^c denote the action taken on client c at time t,

$$a_t = [\cup_{\forall c \in \mathcal{C}} a_t^c]$$

Reward: The reward $R(s_t, a_t)$ obtained by taking action a_t at state s_t is the average QoE of all clients in state s_{t+1}.

The goal of the agent is to maximize the overall QoE of the system. This goal can be formulated as maximizing the expected discounted reward over an infinite horizon. Let $\pi(a_t|s_t)$ denote the probability of taking action a_t given the current state (called the policy) and γ denote the discount factor. Then the goal is to find π^*, the policy that maximizes the expected discounted reward,

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) | s_0 = s, a_t \sim \pi(\cdot|s_t)\right].$$

2.4.2 Measuring QoE for Video Streaming

Considerable progress has been made in identifying the relation between video events such as stalling, and subjective user perception (QoE) [19, 20, 21] via laboratory studies. However, these studies are insufficient in our context, since they do not consider the network conditions (QoS statistics) that gave rise to the video events in the first place. Nevertheless, we can leverage these studies by using them as models of human perception of objectively measurable video events. We considered three models in this context, namely Delivery Quality Score (DQS) [19], generalized DQS [20], and Time-Varying QoE (TV-QoE) [21]. All of the three models are based on the same features (stall event information) if there is no rate adaptation. Since our goal is to support high resolution video without degradation, we fix the resolution so as to prevent video rate adaptation. Under this scenario, all three models are fundamentally similar, and we choose DQS as our candidate. Note that DQS has been validated using 183 videos and 53 human subjects [19], and we do not repeat the user validation experiments. .

The DQS model weights the impact according to duration of the impairments to better model the human perception. For example, the impact of stall events during playback is greater on the QoE than that of initial buffering. Similarly, the first stalling event is looked at with less dissatisfaction than repeated stalling. The state diagram of the model is shown in Figure 2.2. The increases and the decreases in perceived QoE are captured by a function which is a combination of raised cosine and ramp functions. This enables it to model greater or lesser changes in the perceived QoE according to the time it spends in a particular state. The behavior of the predicted QoE by the model in the presence of a particular stalling pattern can be seen in Figure 2.3, where the two stall events result in degradation of QoE. Recovery of QoE from each stall event becomes progressively harder.

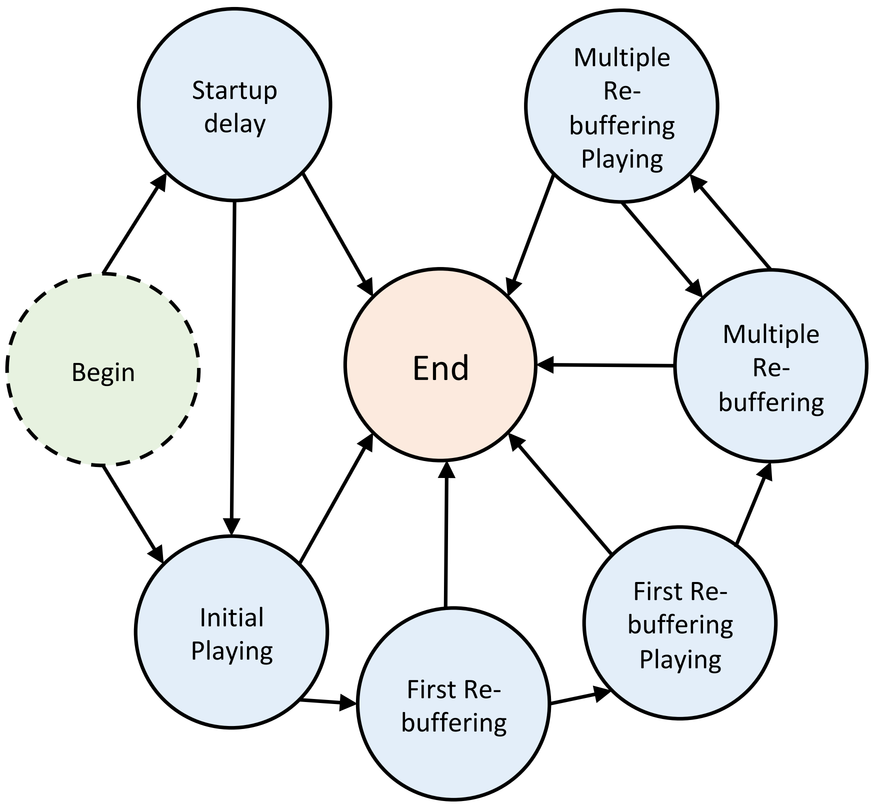

Figure 2.2: DQS state machine

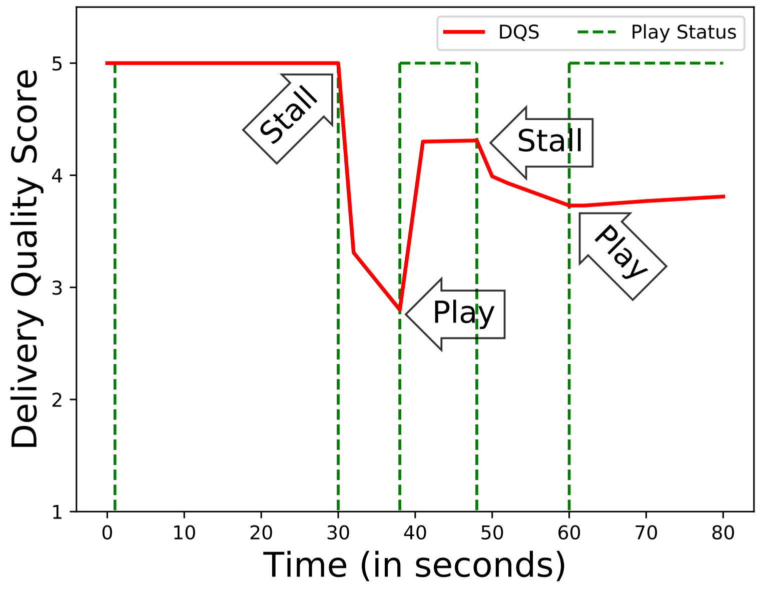

Figure 2.3: Sample DQS evolution.

2.4.3 System Architecture

The system architecture of QFlow is illustrated in Figure 2.4. The three main units of our system are, (i) an off-the-shelf WiFi access point running the OpenWRT operating system, (ii) a centralized controller hosted on a Linux workstation, and (iii) multiple wireless stations (Windows/Linux/Android supported). We denote each software functionality with both a color and a circled number. These functionalities pertain to ① queueing mechanisms, ② QoS policy (configuration selection), ③ Reinforcement Learning, and ④ Policy Adaptation, which we overview below. Tying together the units are ⑤ Databases at the Controller (to log all events), and at each station (that obtains a subset of the data for local decision making). The final components are ⑥ Network Interfaces and ⑦ User Application, which are unaware of our system. We refer to the user application as a client or session, which is composed of one or more flows that are treated identically.

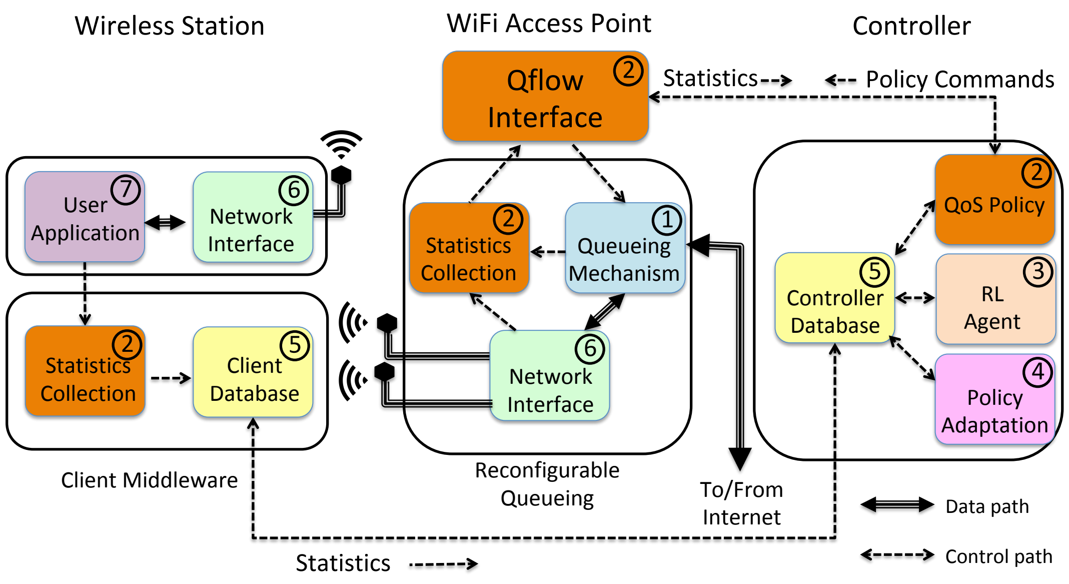

Figure 2.4: The system architecture of QFlow.

① **Per-Packet Queueing Mechanisms:** At the level of data packets, we utilize the MAC layer of software defined infrastructure, namely, reconfigurable queueing. Multiple Layer 2 queues can be created, and different per-packet scheduling mechanisms can be applied over them. When such mechanisms are applied to aggregates of flows, the resulting QoS statistics at the queue level can be varied, with higher priority queues getting improved performances. In turn, this results in state and QoE changes at the application.

② **QoS Policy and Statistics:** Policy decisions are used to select configurations (which clients are assigned to which queue) that result in different QoS vectors. These are made at a centralized controller that communicates using the OpenFlow protocol. We create a custom set of OpenFlow messages for QFlow. The Access Point runs QFlow, an application that interprets these messages and instantiates the queueing mechanisms and configurations selected by the controller. The access point periodically collects statistics related to QoS, including signal strengths, throughput, and RTT and returns those back to the controller (these statistics are for a sanity check and are not directly used for learning).

A smart middleware layer at clients is used to interface with QFlow in a manner that is transparent to the applications (such as YouTube) and the end-user. The middleware determines the foreground application, and samples the application to determine its state (stalls, and buffered seconds on YouTube). QoE is calculated using the DQS model. The client middleware contacts the Controller Database to periodically send the application state and QoE.

③ **Reinforcement Learning Agent:** Application state and configuration decisions (state-action pairs) are used to train RL agents. in the case of the Model-Free approach, a simulation environment duplicating the QFlow setup is used for offline training, and online training continues on the actual system. In the case of Model-Based RL, state-action pairs (resulting from various different policies) stored in the controller database are used for learning the model.

④ **Policy Adaptation:** Policy Adaptation has to do with implementing the policy as empirical data accumulates. An assignment algorithm (policy) matches sessions to queues every 10 seconds, and obtains a sample of client state each time it does so. This state-action pair is captured in the database, and a new action is obtained form the database (as determined by the RL agent). The assignment algorithm is geared towards discounted QoE maximization.

Interactions: The chronological order of small timescale events is shown in Figure 2.5. The Client Middleware at each wireless client captures the state and calculates the corresponding QoE values specific to the foreground application. These realized QoE and state values from all participating clients are sent to the Controller, which performs a policy decision for flow assignment. These policy decisions are sent to the Access Point using OpenFlow Experimenter messages. QFlow interprets and implements these policy decisions. These steps are executed once every 10 seconds.

2.5 QFlow Implementation Details

In this section, we describe our design decisions and implementation of QFlow, in which we extend the OpenFlow protocol using experimenter messages. We exploit the separation of control and data planes of OpenFlow to implement policy decisions using QFlow. Further, our choice of using experimenter messages to send QFlow commands ensures that we do not require implementation of specific changes at the controller. We use an off-the-shelf TP-Link WR1043ND v3 router with OpenWRT Chaos Calmer as the firmware for our implementation. We choose OpenWRT because of its support for Linux based utilities like `tc` (Traffic Control) for implementing per packet mechanisms. Since OpenWRT does not natively support SDN, we use CPqD SoftSwitch [22], an OpenFlow 1.3 compatible user-space software switch implementation.

We next extend SoftSwitch to include QFlow capabilities. Such capabilities include the ability to modify packet-handling mechanisms. Our goal is to enable configuration changes, in addition to the collection of statistics related to the implemented per packet mechanisms and the connected clients. We construct two types of QFlow commands for implementing the described capabilities, *Policy commands* and *Statistics commands*. The rationale behind this separation is to differentiate policy decisions from statistics collection. The controller uses Experimenter messages to communicate these commands to the Access Point using OpenFlow.

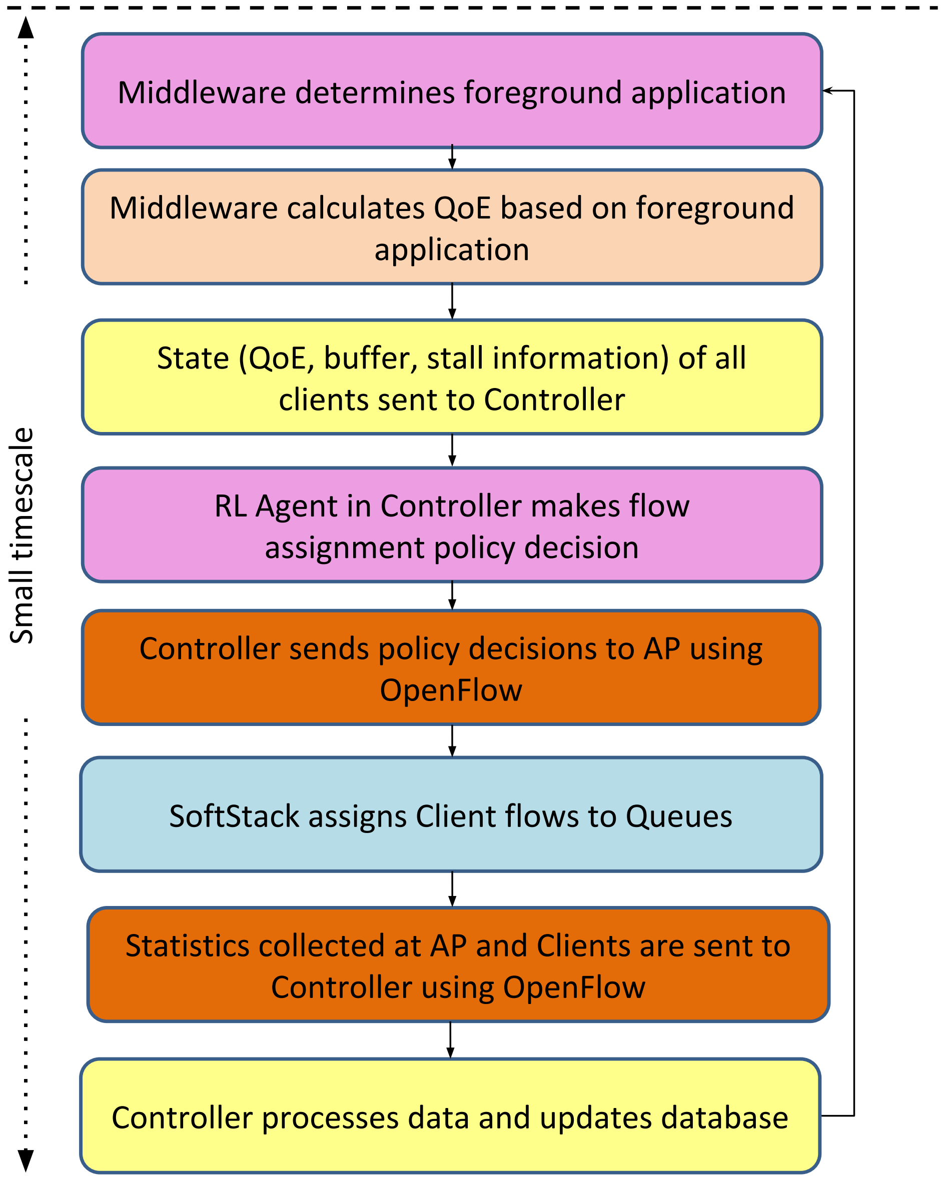
Small timescale
Middleware determines foreground application
Middleware calculates QoE based on foreground application
State (QoE, buffer, stall information) of all clients sent to Controller
RL Agent in Controller makes flow assignment policy decision
Controller sends policy decisions to AP using OpenFlow
SoftStack assigns Client flows to Queues
Statistics collected at AP and Clients are sent to Controller using OpenFlow
Controller processes data and updates database

Figure 2.5: Interaction among the different components of QFlow.

2.5.1 Policy Commands

We design Policy commands to allow us to choose between available mechanisms at different layers. Every time a Policy command is sent, it is paired with a *Solicited response* that is generated by the receiver and sent to the controller using an experimenter message. A Solicited response message thus provides us with the means of retransmission of a failed Policy command, thereby guaranteeing reliability. We define the format of the policy experimenter messages as shown in Figure 2.6. The Controller packs a policy command in the format, and sends it to the Access Point using OpenFlow. On receiving the message, QFlow unpacks it, identifies the specific policy command using the type field, and performs the corresponding operation. Using this framework, we implemented policy commands for the MAC layer.

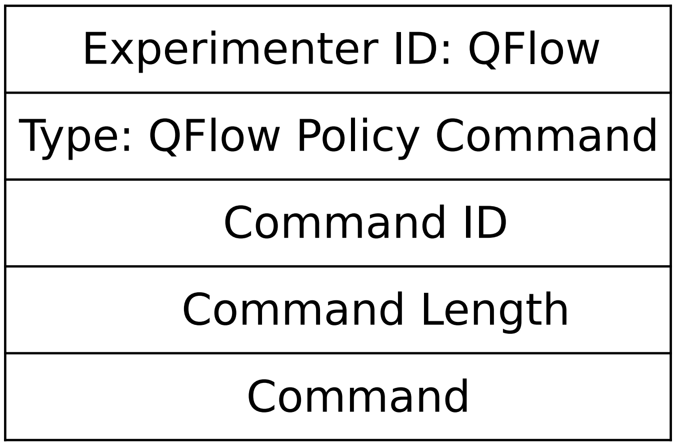

Figure 2.6: Policy packet format

Data Link Layer Queue Command: At the data link layer, we need a means of providing variable queueing schemes. Traffic control (`tc`) is a Linux utility that enables us to configure the settings of the kernel packet scheduler by allowing us to *Shape* (control the rate of transmission and smooth out bursts) and *Schedule* (prioritize) traffic. Each network interface is associated with a *qdisc* (Queueing discipline) which receives packets destined for the interface. We selected *Hierarchical Token Bucket* (`htb`) for our experiments because of the versatility of the scheme. It performs shaping by specifying *rate* (guaranteed bandwith) and *ceil* (maximum bandwidth) for a class, with sharing of available bandwidth between children of the same parent class, and can also prioritize classes. Finally, we use *Filter*s to classify and enqueue packets into classes.

In our experiments, we create queues with different token rates using `htb`. Tokens may be borrowed between queues, meaning that queues will share tokens if they have no traffic. We also create a default queue that handles any background traffic. Decisions at the data link layer include assigning flows to queues, setting admission limits, changing the throughput caps queues, and enabling or disabling sharing of excess (unused) throughput between them.

2.5.2 Statistics Commands

Policy commands result in changes to the QoS statistics of the queues. We define Statistics commands to collect these results and send them back to the controller for analysis. Queue statistics include cumulative counts of downlink packets, bytes and dropped packets. Client-specific statistics consist of average Round Trip Times (RTT; which includes both the RTT from the base station to the client as well as the RTT from the base station to the wide-area destinations with which the client communicates), signal strength (RSSI) and Application specific statistics like buffer state, stall information and video bitrate. Since statistics are sent periodically (once every second) to the controller, we label such messages as *Unsolicited response messages*.

Similar to Policy commands, we define the structure of both Queue and Client-specific Statistics messages. After collecting the respective statistics, QFlow packs the data and sends them to the Controller using OpenFlow. On receiving these messages, the Controller unpacks them, identifies the type from the header information and then saves the extracted data to the database. The packet formats of the Client Statistics messages is shown in Figure 2.7.

QFlow thus is capable of generating state-action, and measuring the resultant rewards in terms of QoE. The details of using the system for RL will be described in the next two sections.

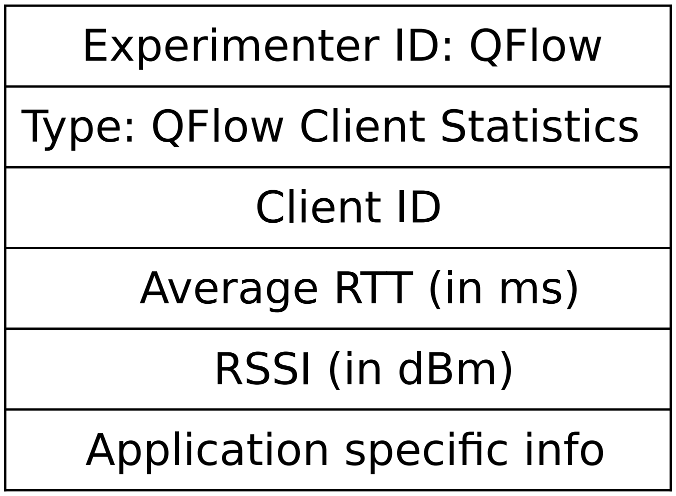

Figure 2.7: Client statistics packet format

The client-specific statistics, together with statistics of the queue it is placed in, constitute the Quality of Service (QoS) for a client.

2.6 Model-Free RL

In this section, we describe a model-free reinforcement learning based approach for learning a control algorithm for the system described in Section 2.4. More specifically, the objective is to learn a control policy for the MDP when the system model (transition probability kernel of the MDP) is unknown. Model-free RL algorithms learn the optimal control policy directly via the interactions with the system, without explicitly estimating the system model. The interaction of the RL agent with the system is modeled as a set of tuples $(s_t, a_t, R_{t+1}, s_{t+1})$ over time and the goal of the RL agent is to learn a policy π that recommends an action to take given a state, in order to maximize its long term expected cumulative reward. We will employ one specific model-free RL algorithm known as Q-learning algorithm.

2.6.1 Q-Learning

Each state-action pair (s, a) under a policy π can be mapped to a scalar value, using a Q-function. $Q^{\pi}(s, a)$ is the expected cumulative reward of taking an action a in a state s and following the policy π from there on. Q^{π} is specified as

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) | s_0 = s, a_0 = a\right],$$

where $\gamma \in (0, 1)$ is the discount factor. Thus, maximizing the cumulative reward is equivalent to finding a policy that maximizes the Q-function. The optimal Q-function, Q^*, satisfies the Bellman equation,

$$Q^*(s, a) = R(s, a) + \gamma \mathbb{E}_{s'}[\max_b Q^*(s', b)], \forall s, a.$$

The objective of the Q-learning algorithm is to learn this optimal Q^* from the sequence of observations $(s_t, a_t, R_{t+1}, s_{t+1})$. The optimal policy π^* can be computed from Q^* as,

$$\pi^*(s) = \arg\max_a Q^*(s, a).$$

Q-learning algorithm is implemented as follows. At each time step k, the RL agent updates the Q-function Q_k as

$$Q_{k+1}(s, a) = \begin{cases} (1 - \alpha_k) Q_k(s, a) + \alpha_k \left(R_k + \gamma \max_b Q_k(s_{k+1}, b)\right) \\ \qquad\qquad\qquad\qquad \text{if } s = s_k, a = a_k \\ Q_k(s, a) \qquad\qquad\quad \text{otherwise} \end{cases}$$

where α_k is the step size (learning rate). It is known that if each-state action pairs is sampled infinitely often and under some suitable conditions on the step size, Q_k will converge to the optimal Q-function Q^* [23].

2.6.2 Deep Q-Learning

Using a standard tabular Q-learning algorithm as described above to solve our problem is infeasible due to the huge state space associated with it. Figure 2.8 depicts our learning problem. The individual client states are combined to form a joint state. The aggregate reward is the reward of all clients combined. The learning agent observes the states and rewards, and outputs an action. The environment then moves to the next state, yielding a reward.

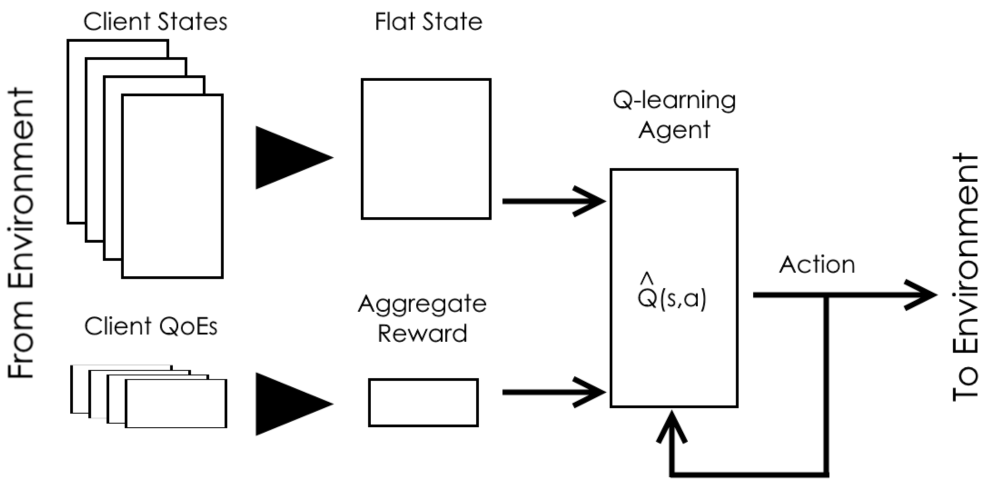

Figure 2.8: RL framework

The state of each client is a tuple consisting of its buffer state, stall information, and its QoE at t. Buffer state and QoE are considered to be real numbers, and thus can take an uncountable number of values. Even if we quantize, the number of states increases exponentially with the dimension and the number of clients. Tabular Q-learning approaches fails in such scenarios.

To overcome this issue due to the curse of dimensionality, we address this problem through the framework of deep reinforcement learning. In particular, we use the double DQN algorithm in [24] that achieved the state of the art performance in many tasks including Atari games. This approach is a clever combination of three main ideas: Q-function approximation with neural network, experience replay, and target network. We give a brief description below.

Q-function approximation with Neural Network: To address the problem of large and continuous state space, we approximate the Q-function using a multi-layer neural network, i.e., $Q(s,a) \approx Q_w(s,a)$ where w is the parameter of the neural network. Deep neural networks have achieved tremendous success in both supervised learning (image recognition, speech processing) and reinforcement learning (AlphaGo games) tasks. They can approximate arbitrary functions without explicitly designing the features like in classical approximation techniques. The parameter of the neural network can be updated using a (stochastic) gradient descent with step size α as

$$w = w + \alpha \nabla Q_w(s_t, a_t)(R_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t)) \tag{2.1}$$

Experience Replay: Unlike supervised learning algorithm, the data samples $\{s_t, a_t, R_t, s_{t+1}\}$ obtained by an RL algorithm is correlated in time due to the underlying system dynamics. This often leads to a very slow convergence or non-convergence of the gradient descent algorithms like (2.1). The idea of experience replay is to break this temporal correlation by randomly sampling some data points from a buffer of previously observed (experienced) data points to perform the gradient step in (2.1) [25]. New observation are then added to the replay buffer and the process is repeated.

Target Network: In (2.1), the target $R_t + \gamma \max_b Q_w(s_{t+1}, b)$ depends on the neural network parameter w, unlike the targets used for supervised learning which are fixed before learning begins. This often leads to poor convergence in RL algorithms. To addresses this issue, deep RL algorithms maintain a separate neural network for the target. The target network is kept fixed for multiple steps. The update equation with target network is given below.

$$\begin{aligned} w &= w + \alpha \nabla Q_w(s_t, a_t)(R_t + \gamma \max_b Q_{w^-}(s_{t+1}, b) - Q_w(s_t, a_t)) \\ w^- &= w \quad \text{after every } N \text{ steps.} \end{aligned}$$

The combination of neural networks, experience replay and target network forms the core of the DQN algorithm [25]. However, it is known that DQN algorithm suffers from overestimation

Table 2.1: Selected hyperparameters for RL agent

Hyperparameter	*Chosen Value*
Discount	0.9999
Network Hidden Layers	$(64, 32)$
Network Optimizer	Adam, Learning Rate 0.001
Replay Buffer	500000
Replay Batch	32
Target Sync Period	100000
Huber Loss	1.0
Double Learning	On
Control Policy	ϵ-greedy, Decay ϵ from 1.0 to 0.01 over 1000000 steps

of Q values. Double DQN algorithm [24] overcomes this problem using slightly modified updated equation as

$$w = w + \alpha \nabla Q_w(s_t, a_t)(R_t + \gamma Q_{w^-}(s_{t+1}, arg\, max_b Q_w(s_{t+1}, b)) - Q_w(s_t, a_t)).$$

The target network is updated after every N steps as before.

2.6.3 Training the RL Algorithm

We implemented the double DQN algorithm using the TensorForce library [26]. Hyperparameters are selected via random search. The final configuration and hyperparameter of the RL algorithm is specified in Table 2.1.

For the faster training of our RL algorithm, we first implement a simulation environment which closely mimics the dynamics of the physical testbed. The environment simulates each video including its bitrate, buffer, length, and QoE. The bitrate and length of each video is generated according to a normal distribution; buffer is stored in terms of time, rather than bits. Each client continuously plays one video after another, stalling where its buffer runs out and building up a buffer of 10 seconds before attempting playing again. Queues are serviced with a constant total bandwith, but the fairness of queue's service among flows assigned to that queue is chosen in each decision period (DP) according to a Dirichlet distribution. Each DP is of duration 10 seconds. The simulation environment uses a high-priority queue with 11 Mbps bandwidth and a low-priority queue with 4.3 Mbps. In the static network configuration, six clients are specified that draw video bit-rates from a truncated $\mathcal{N}(2.9, 10)$ distribution in Mbps, and draw video lengths from a truncated $\mathcal{N}(600, 50)$ distribution in seconds.

For hyperparameter search, the system was simulated for 200 DP per episode for 1000 episodes. Note that increasing the number of units or layers in the network used for value estimation after $(64, 32)$ does not significantly affect the convergence curve; however, the magnitude of the learning rate creates large differences in the performance to which the agent ultimately converges. Further, a single layer is incapable of learning to the performance achieved by the two-layer network. We therefore choose the $(64, 32)$ configuration for our agent. The evolution of value during the training process for the different network configurations and learning rates is shown in Figure 2.9. As is seen, the trained controller achieves a high QoE.

Next, we compare the performance of different policies in the simulation environment. Figure 2.10 shows the average QoE attained by different policies, which suggests that perhaps the model-free approach, while best, may not give substantial performance improvements. The QoE CDFs in Figure 2.11, however, indicate that model-free RL achieves a higher QoE for a larger fraction of clients, suggesting that it might be more robust to resource constraints. Indeed, we will see in experiments in Section 2.8 that it attains quite substantial gains over the other approaches in practice under a bandwidth constrained environment.

2.6.4 Dynamic Number of Clients

In the above description, we assumed that the number of clients in the system is static. The timescale at which the number of clients change is very large (several tens of minutes; this models human mobility as users connect and disconnect to different access points) when compared to the decision period (10 seconds). Hence, including a dynamic number of clients into training would require augmenting the state space with the number of connected clients, and a Markov model of transitions in this value. Since this increases the state space and training duration still further, we instead obtain the optimal static policy for the system with 4 to 6 clients using the model-free

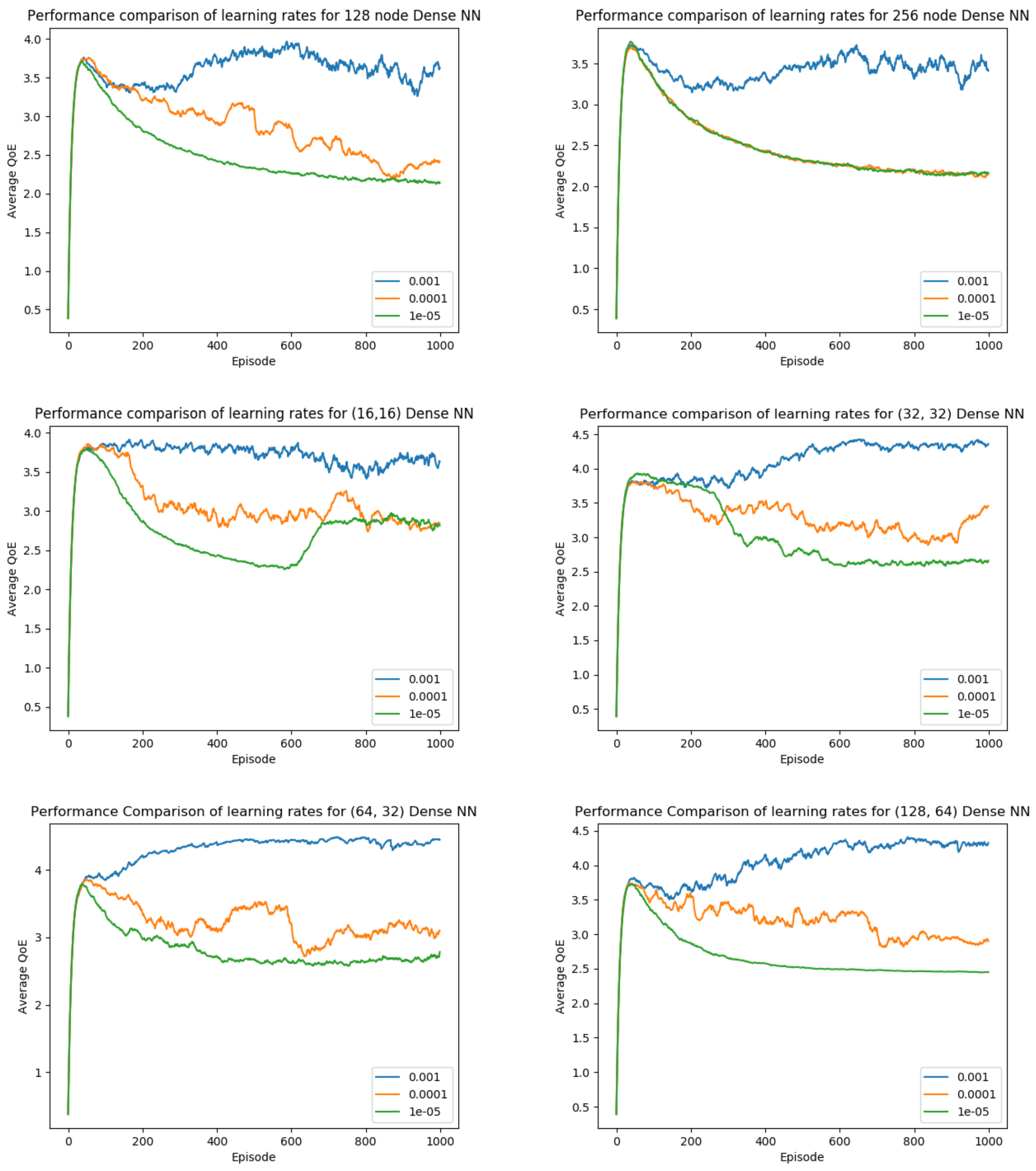

Figure 2.9: RL agent training with various network specifications and learning rates

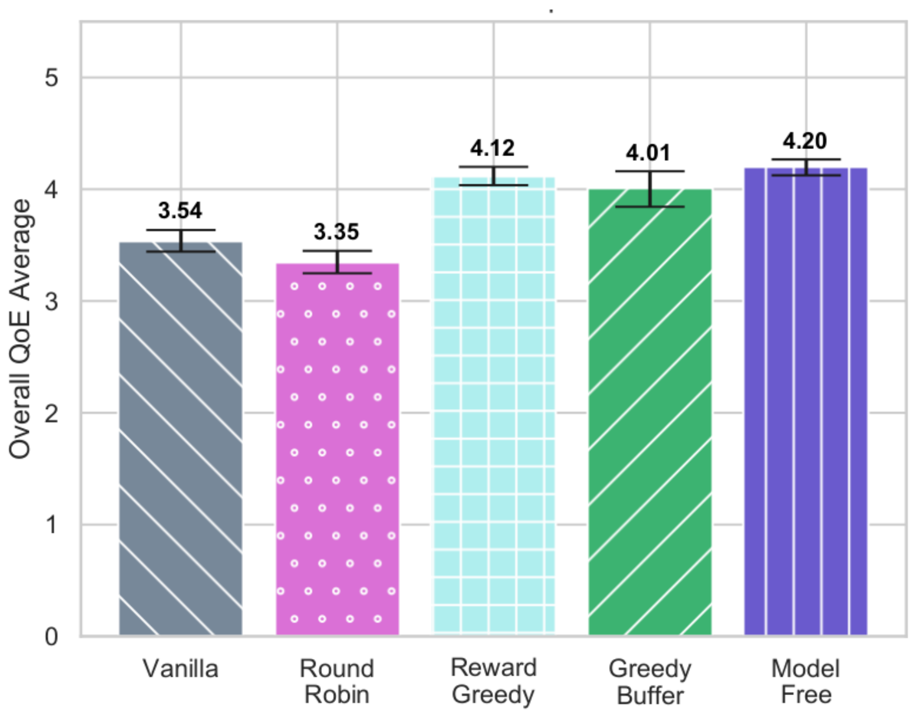

Figure 2.10: Simulated performance comparison for 6 clients

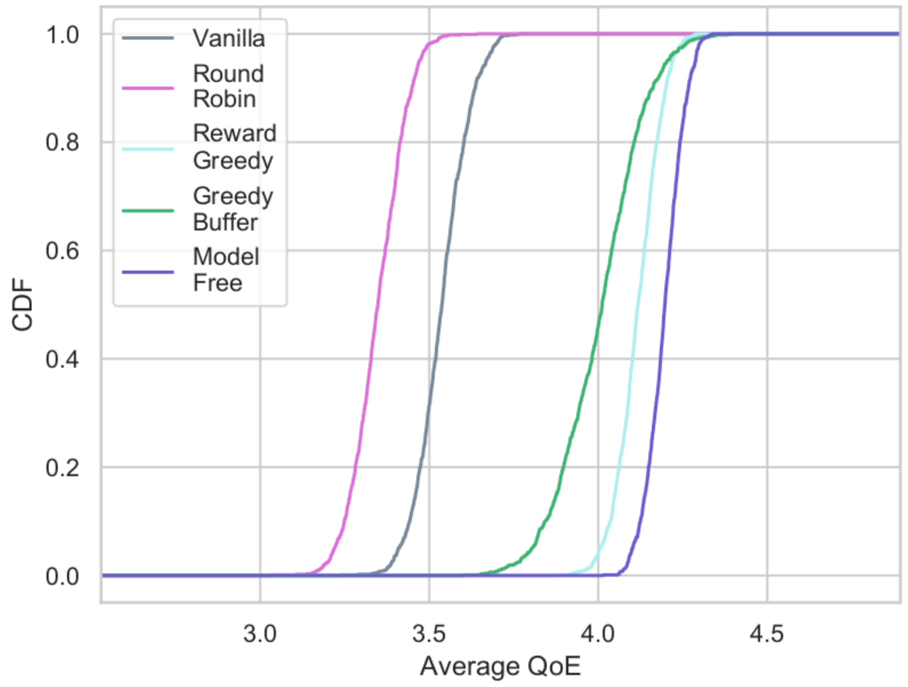

Figure 2.11: Simulated performance distribution for 6 clients

approach. Training for each policy can happen in parallel. Figure 2.12 shows the evolution of value over the training process over the different cases. As expected, the case of 6 clients produces the lowest average QoE. We create a composite controller using the individual static policies by simply choosing the right policy based on number of clients in the system at the time. Note that the composite controller is only slightly sub-optimal, due to the infrequent changes in the number of connected clients.

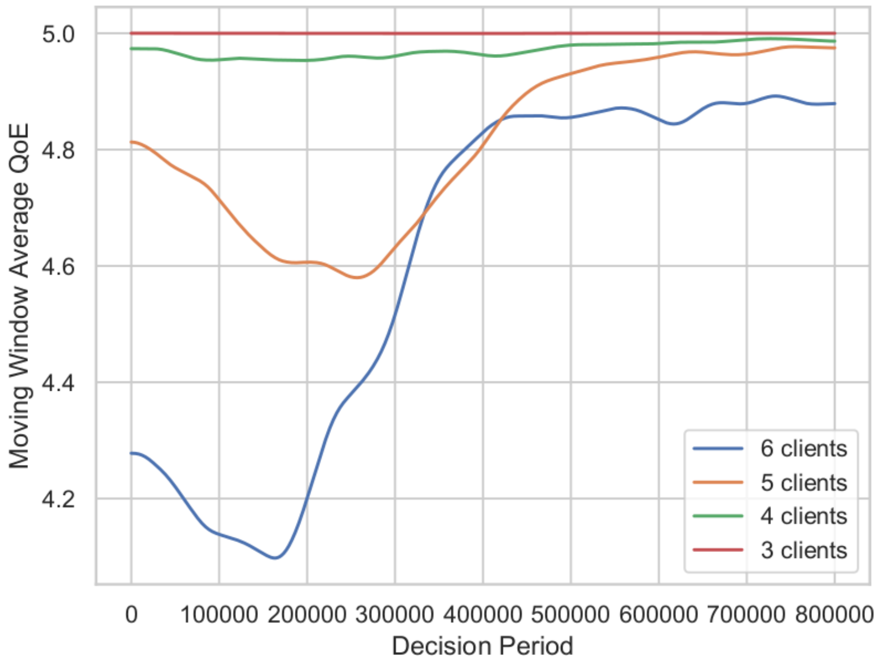

Figure 2.12: Evolution of QoE for dynamic clients

2.7 Model-Based RL

In this section, we discuss the scenario in which the dynamics of the system (transition kernel) are first determined, i.e., given the current state s_t of the system and the action taken a_t, we find the transition probabilities to the next states s_{t+1}. Given the transition kernel of the system $\mathcal{P}$, we can use policy or value iteration to solve for the optimal policy π^*. The model-based approach is particularly interesting because of its special structure, since the state transitions of a client given

Table 2.2: Client state space discretization

Parameter	Range	Bins
Buffer	[0,20]	21
Stalls	[0,5]	5
QoE	[1,5]	9

its current state and action are independent of the states and actions of other clients in the system. In other words,

$$\mathbb{P}(s_{t+1}|a_t, s_t) = \prod_{\forall c \in \mathcal{C}} \mathbb{P}(s^c_{t+1}|a^c_t, s^c_t)$$

It must also be noted that the state transitions of all clients in the system given their current states and actions are identical. Thus, we can determine the transition kernel of the system using the transition kernel of each individual client.

2.7.1 Static Model

In what follows, we determine the transition kernel of the system with a fixed number of clients, and obtain the optimal policy.

Experimental Traces: We generate state (s^c_t), action (a^c_t) and next state (s^c_{t+1}) tuples for all clients by running the system under Round Robin, Greedy Buffer, Random, Model Free and Vanilla policy for a duration of 10 hours.

Discretizing the State Space: The state of each individual client s^c_t and hence the state of the system s_t have elements that are (non-negative) real numbers. In order to calculate the transition kernel of the client in atractable manner, we discretize the state space of the client according to table 2.2. Since the state of a client is 3 dimensional (Buffer, Stall, QoE) we encode it to obtain a label for each client state as follows, Let NSB and NQB denote the number of stall and QoE bins respectively,

$$s^c_t = \text{Buffer} \times \text{NSB} \times \text{NQB } + \text{QoE} \times \text{NSB } + \text{Stall}$$

The discretized and encoded state space of a client $\mathcal{S}_c$ has a cardinality of 945.

Determining the Transition Kernel of a client: We determine the transition kernel of a single client by fitting an empirical distribution over the state, action, and next state tuples collected from experimental traces, i.e., we empirical determine,

$$\mathbb{P}(s^c_{t+1}|a^c_t, s^c_t) \quad \forall s^c_{t+1}, s^c_t \in \mathcal{S}_c \quad \forall a^c_t \in \mathcal{A}_c$$

from experimental traces. Here, $\mathcal{A}_c$ is the set of all actions for a client c.

Identifying Frequent States of the System: The state of the system (s_t) is the union of states of all clients (s^c_t) in the system. If there are N clients in the system, the state of the system is a N dimensional vector, where each dimension corresponds to the state of a client. Let $\mathcal{S}$ denote the discretized state space of the system. The cardinality of $\mathcal{S}$ is of the order of 945^N. Solving an MDP with 945^N states is intractable. Hence, based on experimental traces, we identify the most frequent states $\mathcal{S}_p$ of the system, and approximate all other states to these popular states using the L^2 norm, i.e., given a state in $\mathcal{S}$, we approximate it by a state in $\mathcal{S}_p$ with the least Euclidean distance.

Calculating the Transition Kernel of the System: The state space of our system has now reduced from $\mathcal{S}$ to $\mathcal{S}_p$. To obtain the transition kernel of this system, we empirically sample one hundred state transitions for each state in $\mathcal{S}_p$ under each action in $\mathcal{A}$ using the transition kernel of individual clients. If the generated state transitions are outside $\mathcal{S}_p$, we approximate it with the state in $\mathcal{S}_p$ which is closest in Euclidean distance. Thus, we obtain state, action, next state tuples for the system with state space $\mathcal{S}_p$. We fit an empirical distribution over these tuples to obtain the transition kernel of the system. Hence, we empirically determine

$$\mathbb{P}(s_{t+1}|a_t, s_t) \quad \forall s_{t+1}, s_t \in \mathcal{S}_p \quad \forall a_t \in \mathcal{A}.$$

Obtaining the Optimal Policy: We obtain the optimal policy π^* by running value iteration over the transition kernel generated for $\mathcal{S}_p$. It must be noted that the reward obtained by taking action a_t in state s_t is the average QoE of state s_{t+1} which is a part of s_{t+1} and hence need not be calculated explicitly.

2.7.2 Dynamic Number of Clients

In the previous section, we assumed that the number of clients in the system are static. To deal with a dynamic number of clients, we follow an approach similar to the one described in section 2.6. We obtain the optimal policy for the system with 4-6 clients using the static model approach described in the previous section. Since the time scale in which the number of clients change is large when compared to the decision period, the controller swaps between static policies based on number of clients in the system.

2.8 Evaluation

An off-the-shelf WiFi router installed with QFlow is used as the Access Point and three Intel NUCs are used to instantiate up to 6 clients (YouTube sessions) for our experiments. Note that each such session can be associated with multiple TCP flows, and we treat all the flows associated with a particular YouTube session identically. The three NUCs are equipped with 5th generation i7 processors with 8 GB of memory, each capable of running multiple traffic intensive sessions simultaneously. Relevant session information such as ports used by an application, play/load progress, bitrate and stall information for YouTube sessions is collected every second and written to the database.

We setup a scenario with two downlink queues, one with a higher bandwidth allocation than the other using token bucket queueing. A default queue is used for any background traffic. Two clients may be allowed into the high priority queue. The throughput limit for the high and the low priority queues are set such that clients in the high priority queue experience better QoS than those in the low priority queue. For the no differentiation case, we just set up a single queue with the same total throughput limit as that of the two queues in the previous scenarios. Our control problem is to determine which sessions to assign to which queues.

2.8.1 Policies

In addition to described Model-based and Model-free policies, we consider four additional policies for choosing these assignments.

Vanilla: This is the base case with a single queue that is allocated the full bandwidth, and with no differentiation between clients.

Round Robin: As the name suggests, we assign clients to the high priority queue in turn. Although it is computationally inexpensive, work-conserving and prevents starvation, it might lead to the wrong clients (those who have no hope of significantly increasing their QoE) being considered for the high-quality service instead of clients who might benefit much more from the service.

Reward Greedy: This policy computes the expected one-step reward on a per-client basis, and assigns clients so as to maximize the sum of rewards. We can think of this as a myopic version of model-based RL. This might starve sessions that were unlucky and stalled at some point, since QoE growth rates reduce after stalls.

Greedy Buffer: The smooth playout of a video depends on the size of the playback buffer. When the buffer is empty, the client experiences a stall and the perceived QoE drops. This approach promotes the clients with the lowest buffered video to the high priority queue to prevent this from occurring. This policy might promote the wrong agents who have low buffers because they are at the end of their videos, or those that have stalled multiple times and can never recover high QoE.

2.8.2 Basic Performance Study

Given the experimental setup, we first study the performance experienced by clients assigned to the queues with different configurations. We setup a scenario with two downlink queues, one with a higher bandwidth allocation than the other, and statically assigned one client to each queue. Each client ran a single YouTube session in HD (1080p). The evolution of realized QoS metrics and the predicted QoE for the clients is shown in Figure 2.13. The client in the high priority queue experiences better Throughput and RTT than that in the low priority queue, and as a result, witnesses no stalls, has buffered video (in seconds), and hence the predicted QoE is always higher. Another observation is that although the measured throughput is less than the recommended value (3Mbps) for HD streaming, the buffered video makes up for the deficiency and QoE remains high.

The evolution of the QoE with Buffer state and Throughput of a particular client as it moves

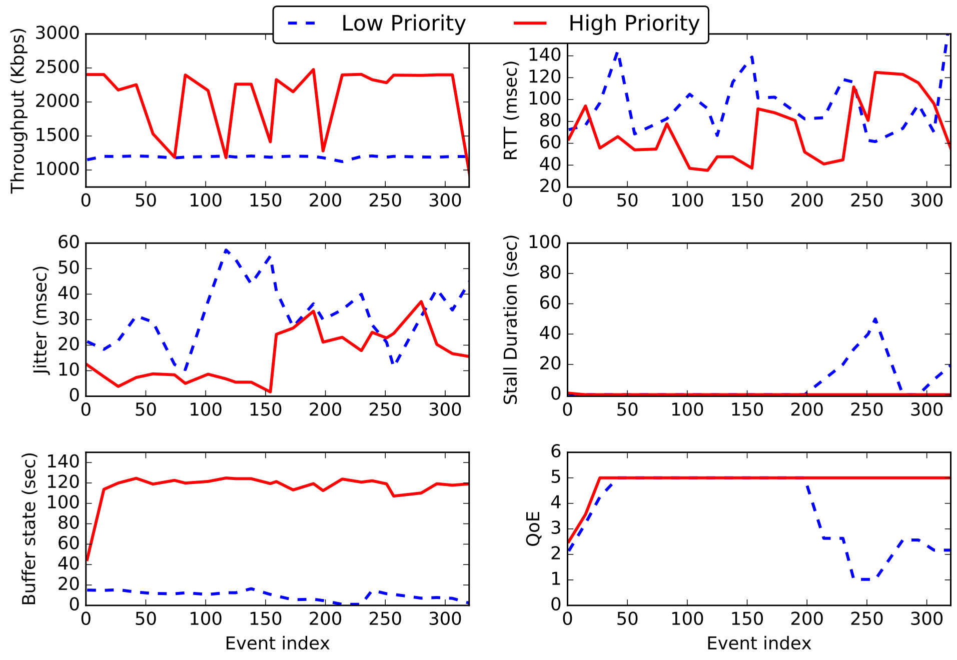

Figure 2.13: Comparison of QoS and QoE in high and low priority queues

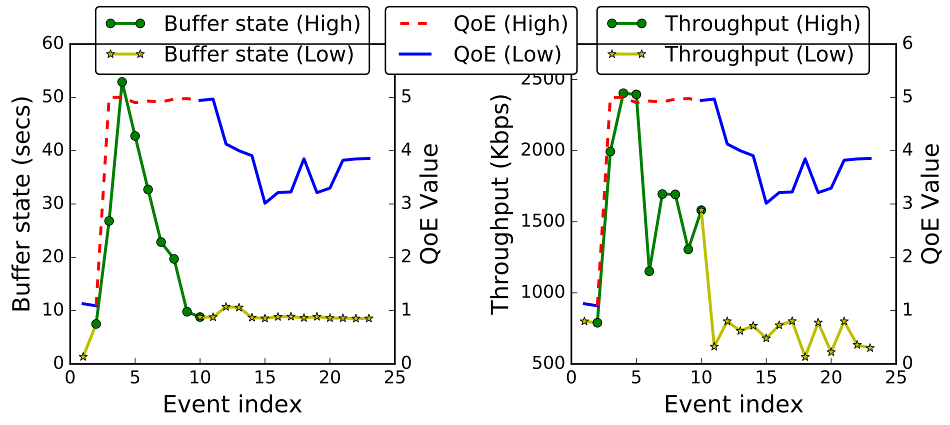

Figure 2.14: Evolution of QoE with buffer state and throughput

through the high and the low priority queues is seen in Figure 2.14. As the buffer of the client is depleted (seen in the beginning of each subplot), any intelligent scheduling policy moves it to the high priority queue where it is able to replenish its buffer and experience good QoE. When it is moved down to the low priority queue again, it depletes its buffer but is still able to experience good QoE. This behavior supports our claim that such policies are able to dynamically promote the flows which require better service to maintain a high average QoE.

2.8.3 Static Network Configuration

In our static configuration, each NUC hosts two YouTube sessions to simulate a total of 6 clients. The QoE performance comparison of the different policies is shown in Figures 2.15, 2.16 and 2.17. We first compare the average QoE of the various policies in the first figure. It is clear that the Model-based and the Model-free policies outperform the other policies. This gap in performance becomes even more evident when we compare the CDFs of the individual and the average QoE of the different policies in Figures 2.16 and 2.17. For example, we can observe from Figure 2.16 that the Model-based and the Model-free policies are able to provide a QoE of 5 for almost 90 and 85% of the time for all clients, whereas it is only about 65% of the time for the next best policy. Similarly, it can be deduced from Figure 2.17 that the Model-based and the Model-free policies are able to achieve an average QoE of 5 for all participating clients in the system about 55 and 40% respectively. The value of this metric for the next best policy is about 5%.

The QoE experienced by a client is affected by the buffer state of the client and the stalls experienced during video playback. Hence, we study the buffer state and the stall durations experienced by the clients under the different policies. Similar to the QoE plots, we compared the averages, the CDFs of the individual and the average values for both these features in Figures 2.18 to 2.23. Again, it is evident from the figures that the Model-based and the Model-free policies ensure better buffer state and lower stall durations (both individual and average) than the other policies under consideration.

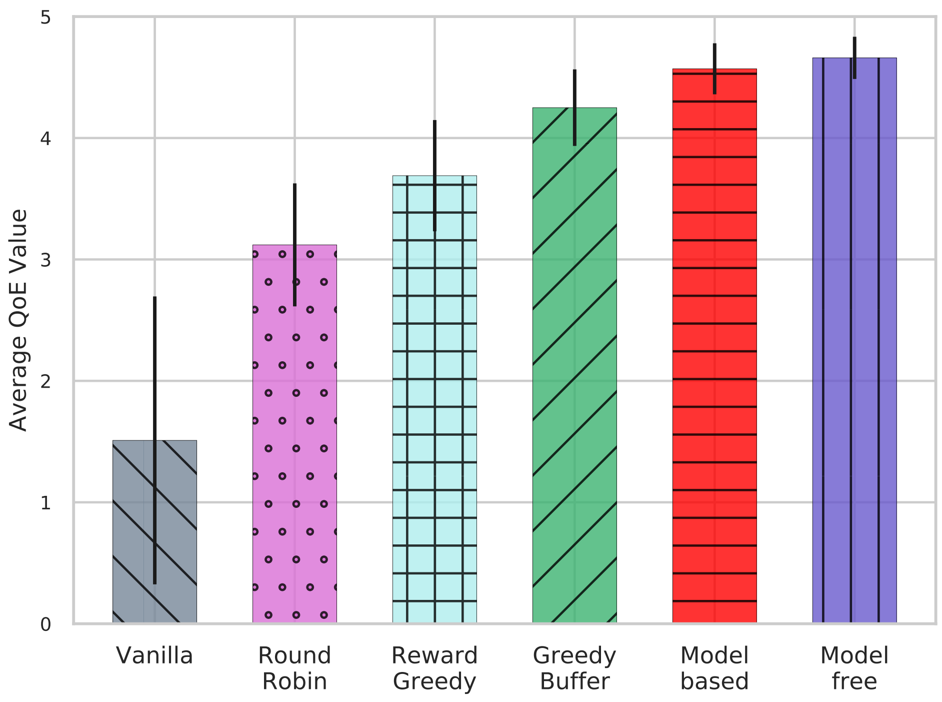

Figure 2.15: Comparison of average QoE

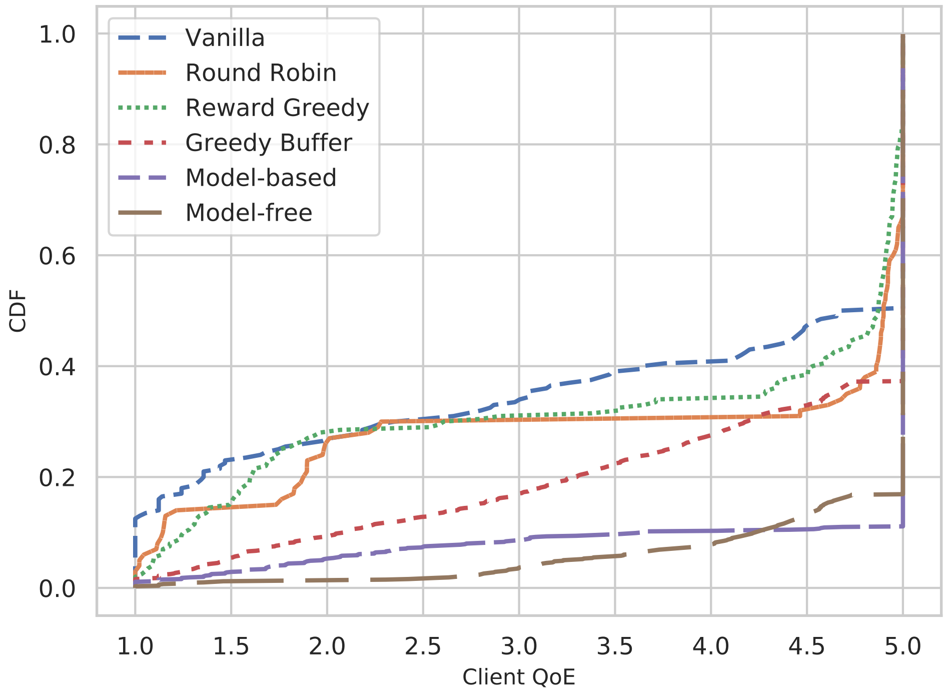

Figure 2.16: Comparison of client QoE CDF

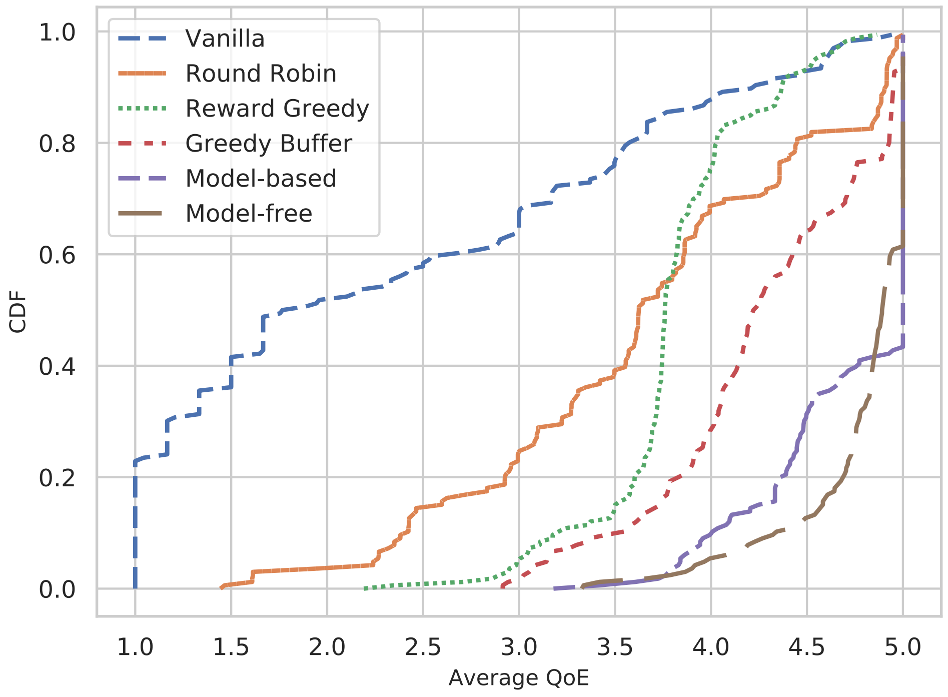

Figure 2.17: Comparison of average QoE CDF

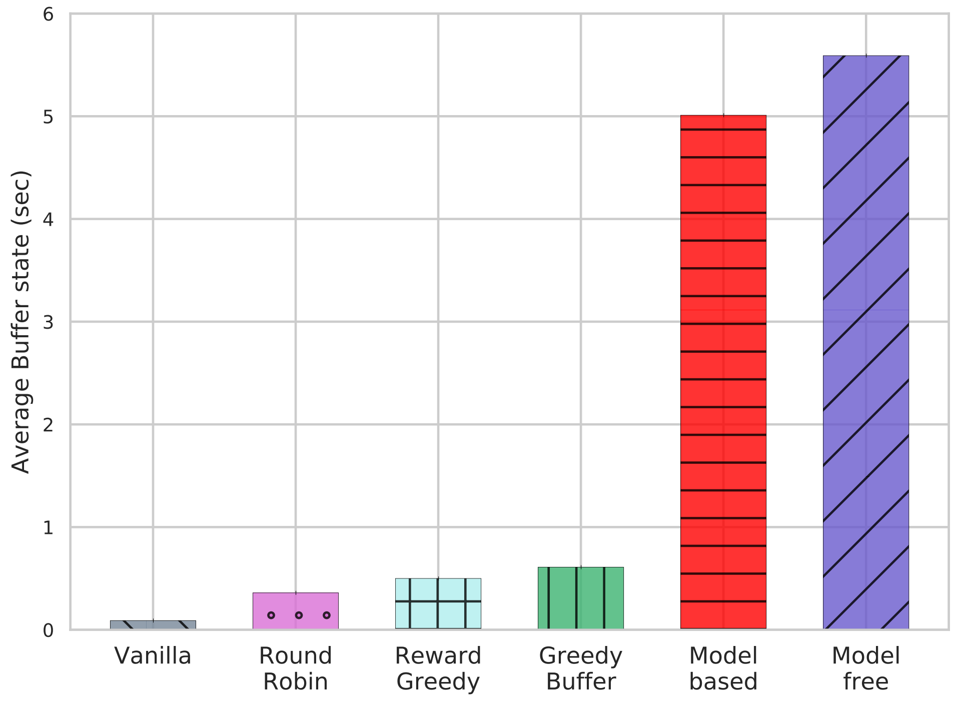

Figure 2.18: Comparison of average buffer state

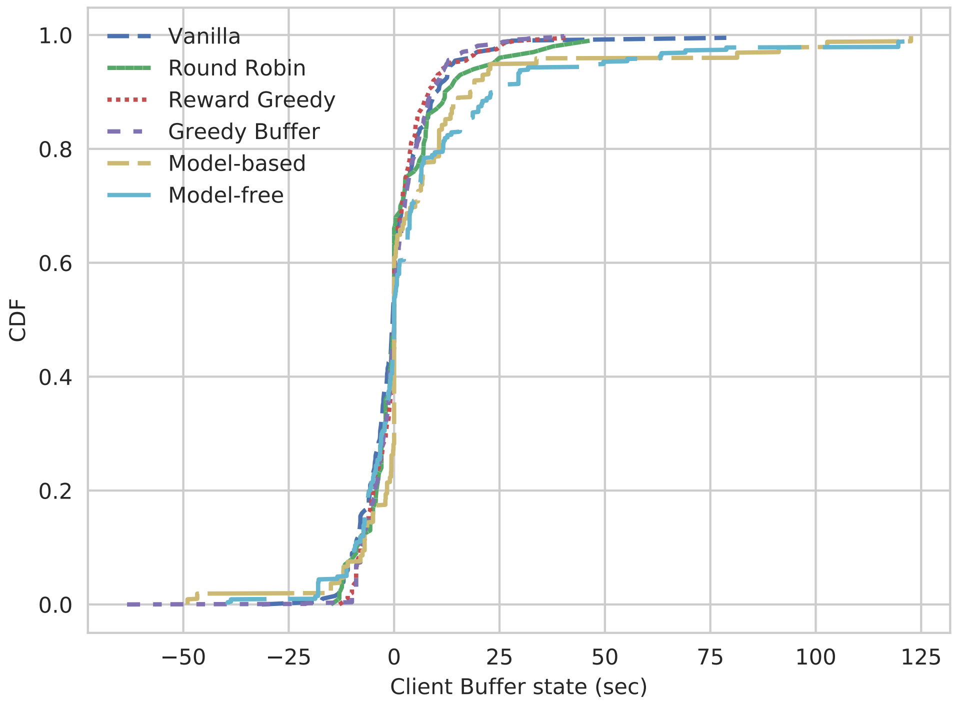

Figure 2.19: Comparison of client buffer state CDF

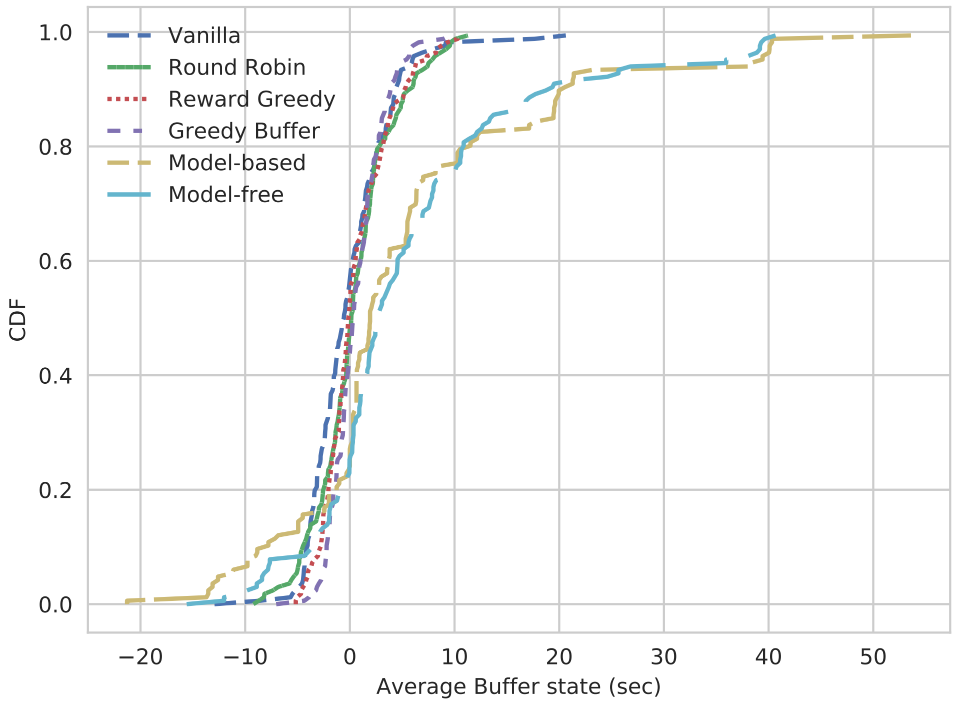

Figure 2.20: Comparison of average buffer state CDF

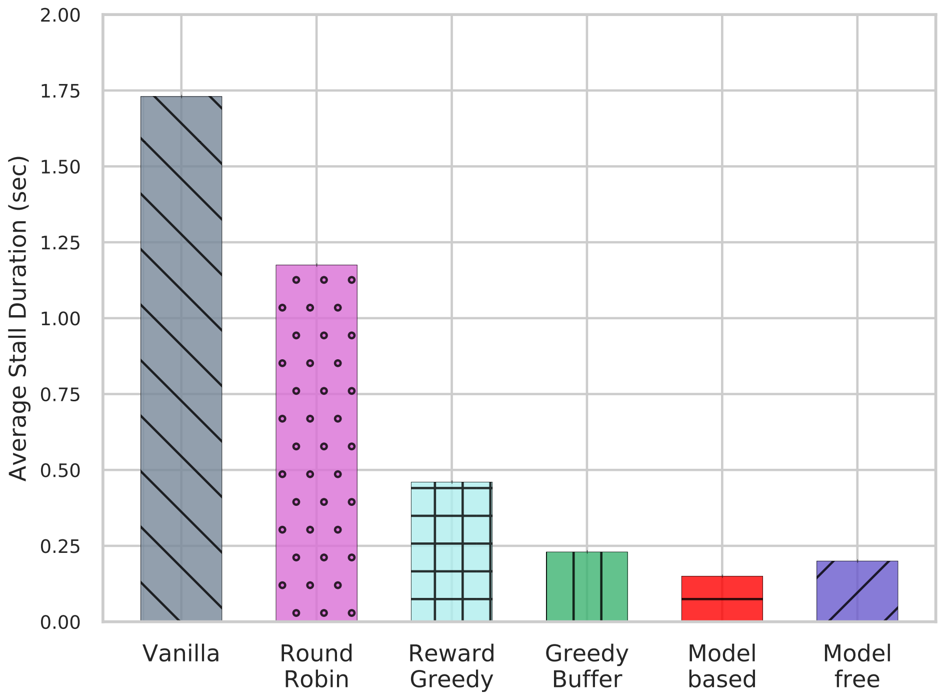

Figure 2.21: Comparison of average stall duration

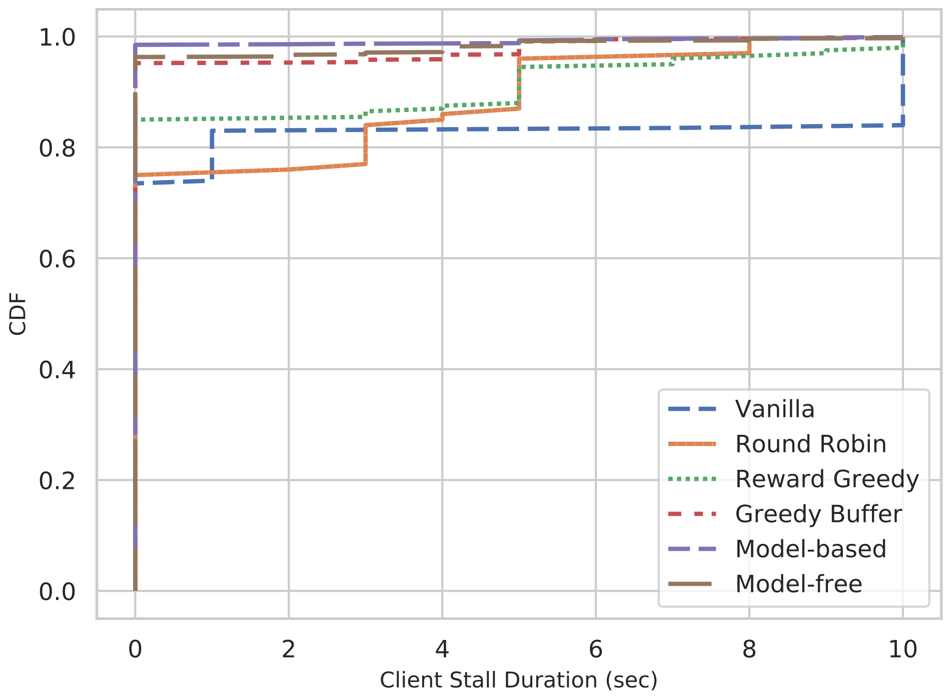

Figure 2.22: Comparison of client stall duration CDF

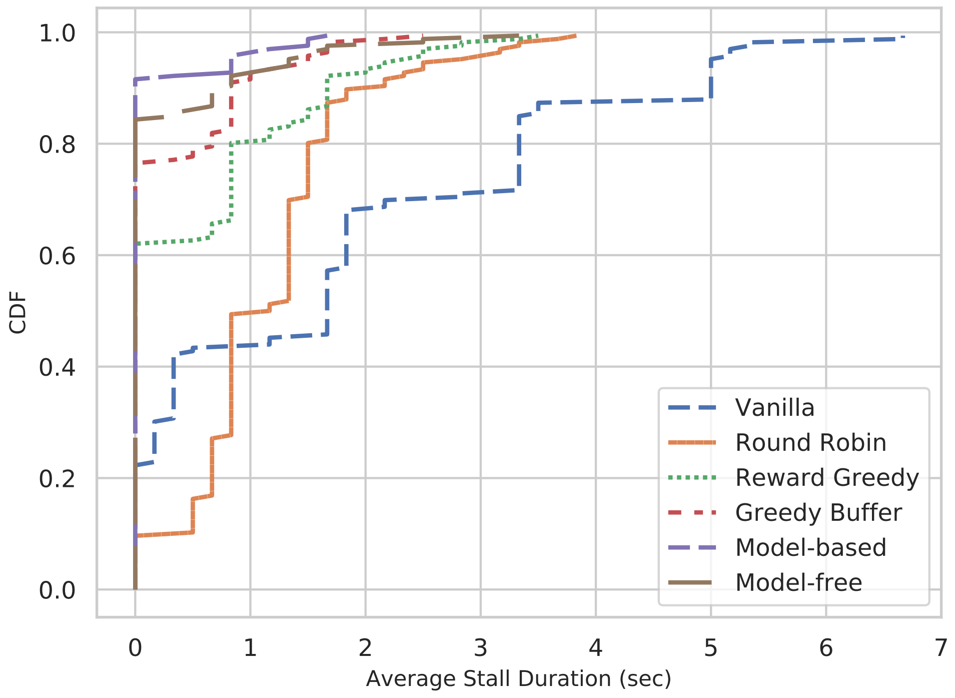

Figure 2.23: Comparison of average stall duration CDF

2.8.4 Dynamic Number of Clients

We next study the performance of the policies in a scenario with a varying number of clients. We choose the number of active clients in the system to vary between 4 and 6, while keeping the bandwidth allocation same as that of the static configuration.

We consider a larger timescale of 30 minutes for changing the number of clietns participating in the system. We start with 6 clients in the system and then remove 1 client each for the next two time periods. At the end of the third period, we introduce two more clients in the system. The evolution of the average QoE for each of the policies for the above scenario is shown in Figure 2.24. It is seen that Model-based and Model-free policies perform well irrespective of the number of users in the system, whereas other policies only do well when there are relatively fewer clients in the system.

Since the bandwidth allocation is the same, reducing the number of clients implies relaxation

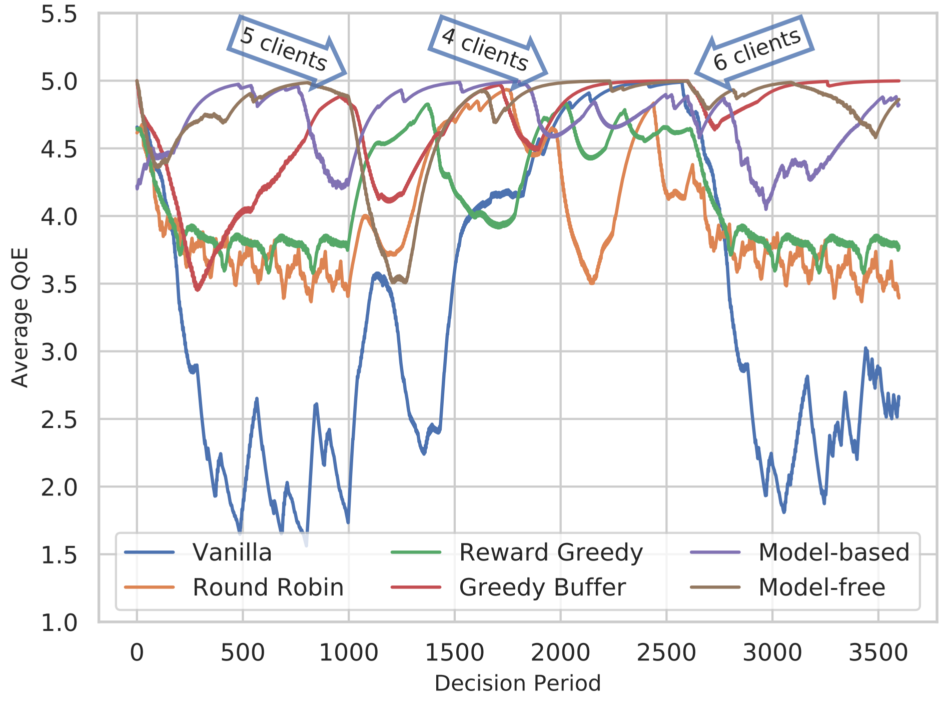

Figure 2.24: Evolution of QoE for dynamic clients

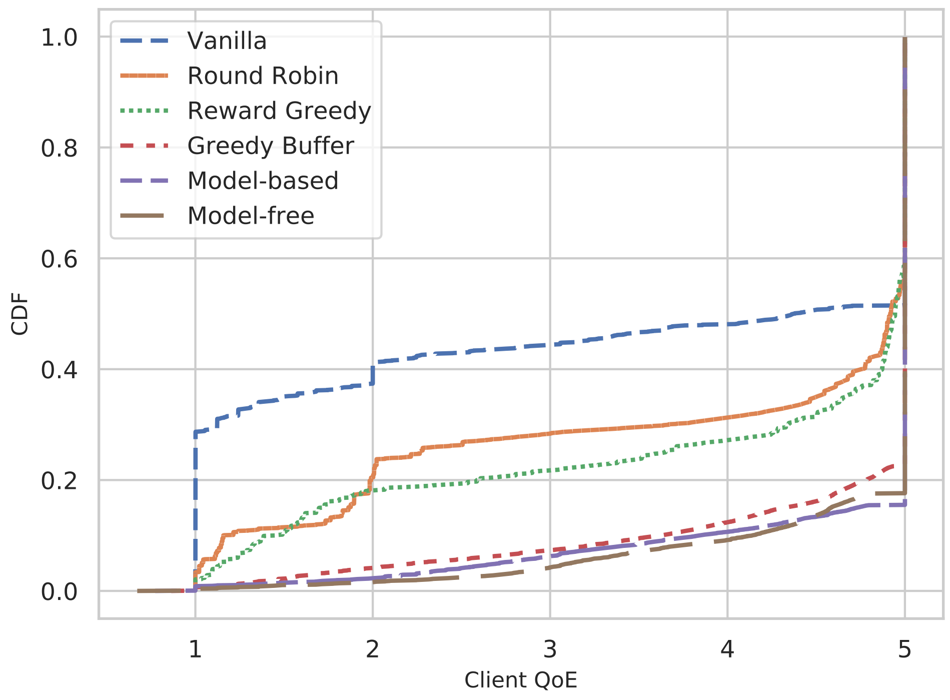

Figure 2.25: Comparison of client QoE CDF for dynamic clients

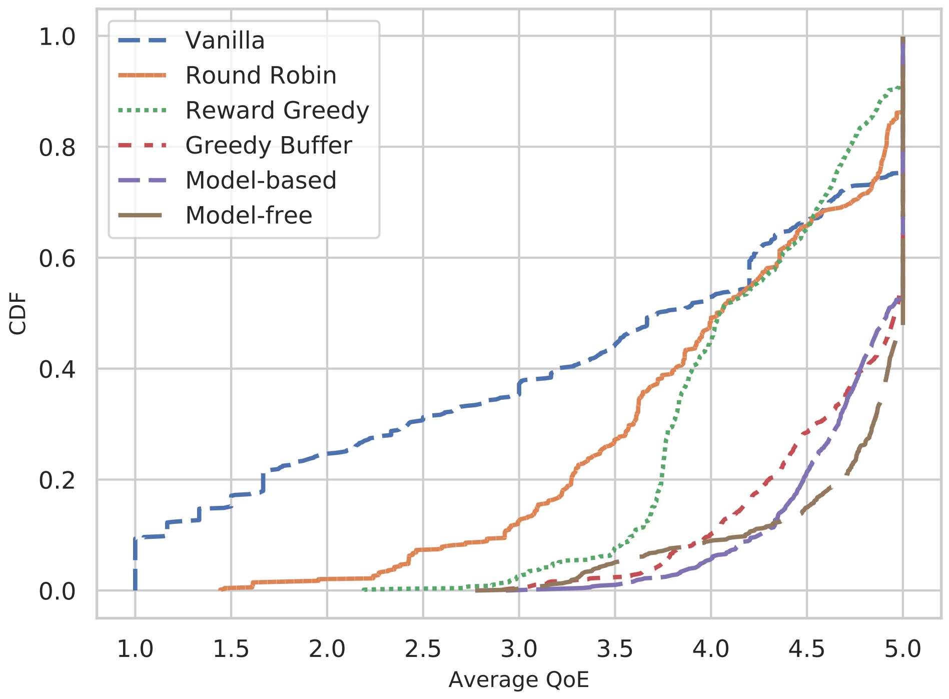

Figure 2.26: Comparison of average QoE CDF for dynamic clients

of the resource constraints and hence, other policies see an improvement in performance. This can be seen in Figures 2.25 and 2.26, where the CDF curves of the other policies are closer to those of the Model-based and the Model-free polciies. Even so, Model-based and Model-free policies exhibit the best performance, which reinforces their superiority over other policies in both static and dynamic client scenarios.

2.9 Conclusion

In this study, we considered the design, development and evaluation of QFlow, a platform for reinforcement learning based edge network configuration. Working with off-the-shelf hardware and open source operating systems and protocols, we showed how to couple queueing, learning and scheduling to develop a system that is able to reconfigure itself to best suit the needs of video streaming applications. The discussed RL algorithm was able to learn the optimal scheduling policy directly via interactions with the system. This success of this approach is based on the truthful declaration of state by all participating clients. We propose a market based system in the next chapter which promotes such truthful reporting of client state.

3. FLOWBAZAAR: A MARKET-MEDIATED SOFTWARE DEFINED COMMUNICATIONS ECOSYSTEM AT THE WIRELESS EDGE[*]

3.1 Introduction

A majority of Internet usage today occurs over wireless access networks, and this trend is only likely to accelerate with the growing penetration of connected televisions, VR headsets, and other smart home appliances. These access networks are growing ever more dense, and the difference between WiFi and cellular access is becoming less clear as 5G standards that require small, densely located cells, and next generation WiFi standards that utilize per-packet scheduling rather than random access become more popular.

A major fraction of the packets carried by these wireless access networks are related to media streaming, which have relatively stringent constraints on the required quality of service (QoS) provided by the network for ideal operation. These QoS metrics typically are measured as link statistics such as $[Throughput,\ RTT,\ Jitter,\ LossRate]$. The impact of such QoS on user satisfaction is identified in terms of Quality of Experience (QoE). QoE is measured as a number in the interval $[1, 5]$, and can be dependent on the application and its evolving state. For example, the application can be video streaming over the Web, with the state being the number and duration of stalls (re-buffering events) that have been experienced thus far. Supporting a large number of concurrent streams of this kind, while ensuring high QoE for all clients is a major challenge.

As a concrete example, consider Figure 3.1 that shows nine simultaneous YouTube clients that are supported over a wireless access network. This setup is used for our laboratory experiments, and can emulate a range of load and channel conditions by restricting the available QoS values at the access point. The traditional (vanilla) approach is to maintain a single queue, and to treat all packets identically regardless of the importance of the packets to the QoE of the clients. So a session that has already buffered up many seconds of video might get equal service as one that is

[*]This chapter has been submitted to WiOpt, 2019 for publication and is under review. It is also available at https://arxiv.org/abs/1801.00825.

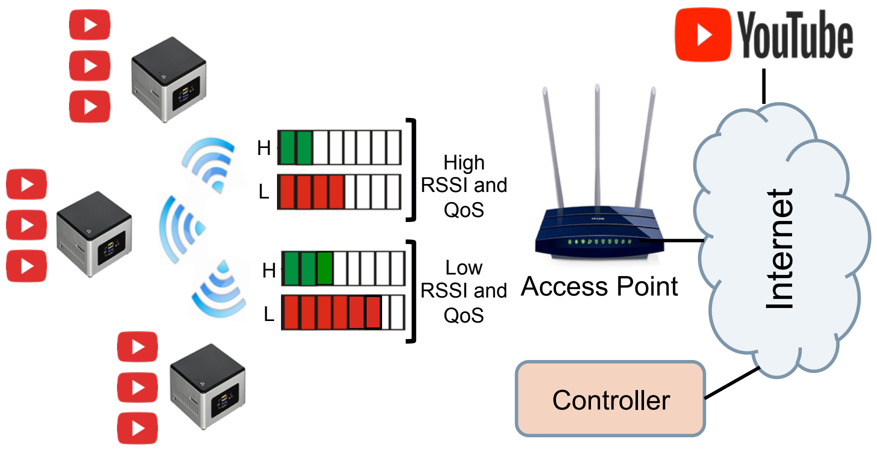

Figure 3.1: Ensuring high QoE video streaming via adaptive prioritization.

near stalling. While this approach might be acceptable when the number of streams is limited, the need to support multiple high quality streams motivates the desire to do better.

Given that queuing behavior is fundamental to all elements of the QoS statistics mentioned above, differentiated queuing at the access point immediately suggests itself. Token-bucket-based shaping can be used to create high-priority and low-priority queues, with the QoS statistics of the former being superior to that of the latter. Furthermore, we can create multiple "bins" of queues as shown in Figure 3.1, with each bin corresponding to similar client channel conditions (with worse channels implying lower achievable QoS), and allocate them similar time-spectrum resources. Then a basic question is that of periodically deciding client schedules: *Given the current QoE and video state at each client, how should the controller assign clients to queues for the next decision period?*

If we visualize the system state as being the QoE, stalls and buffered video of all clients, and actions as deciding assignments of clients to queues, then the problem of maximization of the sum QoE can be posed as a Markov Decision Process (MDP). However, the transition kernel of this MDP is unknown apriori. In the previous Chapter, we considered this problem from the perspective of reinforcement learning (RL), and utilized model-based and model-free RL approaches to finding the optimal policy over a platform entitled QFlow. Indeed, impressive improvements were seen on

QFlow—in a situation where the vanilla policy was only able to achieve a QoE of about 3, the RL approaches achieved a QoE of over 4.7.

However, there are two main drawbacks to the RL approach. First, is complexity. The model-free approach using Q-learning requires an enormous amount of training, which necessitates its being trained over a simulated data set. Furthermore, the model-based approach needs to obtain the complete transition kernel of the system, which does not scale well as the number of clients increases. The second is a question of incentives. The state of each client has to be supplied by the clients themselves, which implies that intelligent clients could obtain more than their fair share of resources through appropriate state declarations to the AP.

The goal of this paper is to design, develop and validate FlowBazaar, an auction-based system for ensuring high QoE video streaming using information volunteered by the clients themselves. FlowBazaar uses the same hardware platform as QFlow, but the focus is on eliciting true value functions from the clients, and using that to decide on prioritization policy.

3.1.1 Main Results and Organization

In this paper, we design FlowBazaar as a scheme for ensuring high QoE of video streaming. The system consists of a setup similar to Figure 3.1 under which all TCP flows corresponding to a particular client can be periodically assigned to any one of the queues. The system of setting up and assigning flows to queues follows 2.5, in which OpenFlow extensions are developed to enable an OpenFlow controller to instantiate policy decisions at the AP. The decision period is set as 10 seconds, so as to allow time for the TCP flows to attain equilibrium. Each client samples its video state, and maps this state to a standard video QoE model called Delivery Quality Score (DQS) [19]. The QoE forms the one-step reward seen by the client.

The core of FlowBazaar is an incentive compatible (truth-telling) auction that is conducted every 10 seconds. In our setup, we only permit two clients to be assigned to the high priority queue (in each bin), and hence select a third-price auction as our mechanism for queue assignment. Here, the clients are asked for bids, and the top two bidders are selected, with the price being charged equal to the third highest value. It is easy to show that such an auction will elicit the true value functions of the clients. Bids for the auction are placed via a smart middle-ware algorithm *(not by a human end-user, who may be unaware of the existence of the system).* The bids themselves are values in the interval $[1, 5]$, and can be interpreted as the number of cents that the bidding algorithm is willing to pay for high priority service for the next 10 seconds. We calculate that the eventual dollar price paid will be of the order to a few tens of dollars per month, consistent with cellular data access billing schemes of today.

Under this system, clients first learn the system model in an offline manner, i.e., the marginal transition kerenels that correspond to the change in state at a client, given the queue that it was assigned to. Once the kernel is known, the client solves an MDP to determine its bid, given a *belief distribution* of bids made by other clients. This belief is simply the empirical distribution of bids made thus far (using a weighted moving average to eliminate older information) and is collected and made available by the controller to all clients. Our earlier work shows analytically in a similar problem framework that an efficient equilibrium in the space of belief distributions exists under the so called mean field approximation [27].

FlowBazaar, solves the two main issues of the RL approach by leaving the MDP solution to the clients themselves under an auction-based scheduling scheme. Here, the clients only need to consider the *marginal transition kernel* (impact on their own states of the scheduling decision) while solving the MDP, and coordinate with one another via the auction conducted periodically at the access point. While one might think that the resulting allocation is suboptimal, it actually turns out to yield a superior QoE over reinforcement learning for all clients. This result suggest that a indexing of state is occurring in this problem, under which each client state is associated with a real-number index. The optimal policy simply picks the clients with largest indices to promote to the high priority queues. We empirically validate this hypothesis, and find that such an index policy performs as well or better than all others, lending credence to the indexing claim.

The paper is organized as follows. Section 3.2 reviews existing studies on QoS and queueing, OpenFlow extensions to wireless and auctions and pricing on the wireless edge. Section 3.3 describes our models for the system-wide model for overall QoE maximization, as well as the individual model for the auction. Section 3.4 describes the system architecture of FlowBazaar, emphasizing the algorithmic nature of the auctions with no human intervention. Section 3.5 determines both the system-wide and marginal kernels experimentally, and designs policies that apply to the different cases and belief structures. Section 3.6 presents our experimental results, that clearly illustrate the superior performance of FlowBazaar over tall other policies that we evaluated against. Finally, Section 3.7 concludes the paper.

3.2 Related Work

We review existing research work on Queueing, Auctions and OpenFlow extensions in this section.

Optimal Queueing: There has been significant work on QoS as a function of the scheduling policy, e.g., a sequence of work starting with [1], and follow on work in the wireless context that resulted in algorithms such as backpressure-based scheduling and routing in wireless networks [2] and more recently [3] that ensures that strict delay guarantees are met. Most of these works aim at maximizing throughput or loss rate, but they do not consider all the elements of QoS together. Also, they do not map received QoS to application QoE.

Auctions and Scheduling: There has also been work on using price or auction-based resource allocation in the wireless context. On the analytical side, [28] considered the problem of auction-based wireless resource allocation. Here, users participate in a second price auction and bid for a channel. It was shown that with finite number of users, a Nash Equilibrium exists and the solution is Pareto optimal. In [27], an auction framework is presented in which queues (representing apps on mobile devices) repeatedly bid for service in a second-price auction that determines which set of queues will be selected for service. They show that under a large system scaling (called the mean field game regime), the result of the auction would be the same as that of the longest-queue-first algorithm, and hence ensuring fair service for all. Our design of auction-based scheduling algorithms are motivated by these ideas. In the context of experiments, a recent trial of a price-based system is described in [29]. Here, day-ahead prices are announced in advance to users, who can choose to use their cellular data connection based the current price. Thus, the decision makers are the human end-users that essentially have an on/off control. Furthermore, the prices are not dynamic and have to be determined offline based on historical usage.

OpenFlow Extensions: There has been significant research into the development of OpenFlow extension to cross-layer wireless configuration selection. In this context, CrossFlow [7, 8] uses the SDN framework for configuring software defined radios. Similarly ÆtherFlow [9], extends OpenFlow for enabling remote configuration of WiFI access points. Finally, recent systems such as AeroFlux [10] and OpenSDWN [11] enable packet prioritization for flows that are identified by packet inspection as belonging to high priority applications, such as video streaming. However, these are all offline static policies in that they do not relate the prioritization policy with the state of the application.

3.3 System Model

We consider a resource constrained system in which clients are connected to a wireless Access Point (AP). We choose video streaming as the application of interest using the case study of YouTube, since video has stringent network requirements and occupies a majority of Internet packets today [5]. The AP has a high and low priority queue. Clients assigned to the high priority queue typically experience a better QoS (higher bandwidth, lower latency etc.) when compared to the clients assigned to the low priority queue. We assume that a fixed number of clients can be assigned to the high priority queue. The controller *optimally* assigns clients to each of these queues at every decision period (DP; 10 seconds in our implementation).

3.3.1 System-wide Model

We consider a discrete time system where time is indexed by $t \in \{0, 1, ...\}$. At each DP ($t = 0, 1, 2..$) the controller observes the *state* of the system and assigns clients to queues based on a *policy*. This problem can be modeled as a Markov Decision Process (MDP) consisting of an *Environment* that produces *states* and *rewards* and an *Agent* that takes *actions*.

Environment: The environment is composed of the AP and clients. Let $\mathcal{C}$ denote the set of all clients.

Client State: Each client keeps track of its state which consists of its current buffer (the number of seconds of video that it has buffered up), the number of stalls it has experienced (i.e., the number of times that it has experienced a break in playout and consequent re-buffering), and its current QoE (a number in $[1, 5]$ that represents user satisfaction, with 5 being the best). Let s_t^c denote the state of client c at time t.

System State: The state of the system is the union of the states of all clients. Let s_t denote the state of the system and $\mathbb{S}$ denote the set of all possible system states,

$$s_t^c = \textit{[Current Buffer State, Stall Information, Current QoE]} \ \forall c \in \mathcal{C}$$
$$s_t = [\cup_{\forall c \in \mathcal{C}} s_t^c]$$

Agent: The controller is the agent, that assigns flows to queue every decision period based on a policy. The queue assignment performed by the controller at time t is called action $a_t \in \mathcal{A}$. Let a_t^c denote the assignment of client c at time t,

$$a_t = [\cup_{\forall c \in \mathcal{C}} a_t^c]$$

Reward: The reward $R(s_t, a_t)$ obtained by taking action a_t at state s_t is the expected QoE of all clients in state s_{t+1}.

Policy: The goal of the agent is to maximize the overall QoE of the system. This goal can be formulated as determining the optimal policy π^*, that maximizes the Bellman Optimality Equation given by,

$$\pi^*(s_t) = \text{argmax}_{a_t \in \mathcal{A}}(R(s_t, a_t) + \gamma \sum_{s_{t+1} \in \mathbb{S}} \mathbb{P}(s_{t+1}|s_t, a_t)V^*(s_{t+1})) \tag{3.1}$$

where γ is the discount factor, $\mathbb{P}(s_{t+1}|s_t, a_t)$ is the *system transition kernel* and $V^*(s_{t+1})$ is the optimal value of state s_{t+1}.

3.3.2 Auction-based Model

We consider a market wherein clients bid for high priority service periodically. In each discrete time instant, a fixed number of clients N are assigned to the high priority queue. Clients participate in an $(N+1)^{th}$ auction to compete for admission to the high priority queue. The N winners who obtain high priority services will pay a price that is equal to the $(N+1)^{th}$ highest bid and the rest of the clients will be assigned to the low priority queue. We model the system in a Mean Field approach as described below.

Bid: The bid submitted by the client in each auction is denoted by $b \in \mathcal{B}$, where $\mathcal{B}$ is a set containing discrete bid values. One could interpret different bid values as the price each client c is willing to pay to obtain high priority service.

Bid Distribution: The clients must place their bid based on the beliefs of their competitors. We denote the assumed bid distribution in the market as ρ.

Payment: The amount of transaction after each auction is denoted by pay. Note that pay is a random variable that corresponds to the auction mechanism. In particular, the payment distribution in our system is exactly the distribution of the $(N+1)^{th}$ highest bid.

Client Reward: The reward $R(s_t^c, a_t^c)$ obtained by taking action a_t^c at state s_t^c is the expected QoE of client c in state s_{t+1}^c.

Client Transition Kernel Let $\mathbb{P}(s_{t+1}^c | s_t^c, a_t^c)$ denote the client transition kernel. The action $a_t^c = win$ and $a_t^c = lose$ corresponds to assignment of client c to the high and low priority queue respectively. Thus the probability of transitioning to state s_{t+1}^c is jointly defined by the probability of winning the auction when bidding b, $p_{win}(b)$ and $\mathbb{P}(s_{t+1}^c | s_t^c, a_t^c)$.

Policy: From a single client's perspective, all his opponents are placing bids according to the same bid distribution, which is a public belief. Thus we formulate the policy of the corresponding

MDP as follows,

$$b^*(s_t^c) = \text{argmax}_{b\in\mathcal{B}}\Bigg\{p_{win}(b)\Big[R(s_{t+1}^c, a_t^c = win) - pay + \sum_{s_{t+1}^c}\mathbb{P}(s_{t+1}^c|s_t^c, a_t^c = win)\gamma v(s_{t+1}^c)\Big] + (1 - p_{win}(b))\Big[R(s_{t+1}^c, a_t^c = lose) + \sum_{s_{t+1}^c}\mathbb{P}(s_{t+1}^c|s_t^c, a_t^c = lose)\gamma v(s_{t+1}^c)\Big]\Bigg\} \tag{3.2}$$

3.3.3 Measuring QoE for Video Streaming

Various models validated by extensive human experiments such as Delivery Quality Score (DQS) [19], generalized DQS [20], and Time-Varying QoE (TV-QoE) [21] identify the relation between video events and subjective user perception (QoE). These models are based on video stalling events if there is no rate adaptation. Since our goal is to support high video resolution, we fix the video resolution to prevent rate adaption. We choose the Delivery Quality Score (DQS) [19] as our QoE model.

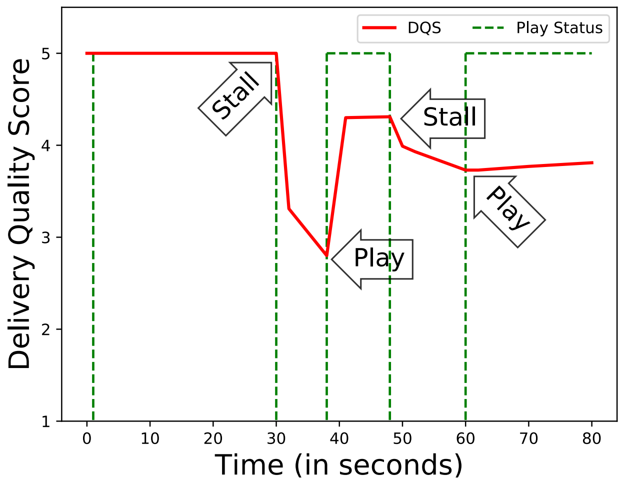

Figure 3.2: Evolution of DQS

The DQS model takes various factor into account (such as stall duration and number of stalls) to measure the impact of a stall on QoE. Figure 3.2 shows the evolution of QoE over the duration of a video. Observe that the impact of the first stall event on QoE is large when compared to subsequent stall events. The change in perceived QoE is captured by a function which is is a combination of raised cosine and ramp functions. Recovery of QoE from each stall event becomes progressively harder. Note that DQS has been validated using 183 videos and 53 human subjects [19], and we do not repeat the user validation experiments.

3.4 System Architecture

The architecture of FlowBazaar, which is illustrated in Figure 3.3, is an extension to QFlow described in Section 2.4.3. We briefly describe it in this section. The three main units in our system are an off-the-shelf WiFi Access Point running a custom operating system with extensions, multiple wireless stations installed with custom middleware and a centralized controller. The units contain different functional components which are shown in a color coded manner. These components have functionalities pertaining to packet mechanisms, QoS policy, application QoE, and end-user value. Tying together the units are a Controller Database in which we log all events, and a smaller Client Database at each station that obtains a subset of the data that it needs for decision making, both shown as yellow tiles. The components related to Network Interface and User Application are unaware of the rest of the system.

We briefly overview the different functional components below.

Queueing Mechanisms (Blue Tiles): We utilize reconfigurable queueing at the MAC layer of reconfigurable infrastructure stack. We create multiple MAC Layer queues, and apply different per-packet scheduling mechanisms over them. The impact of the application of such mechanisms on flows is seen on the resultant QoS statistics at the queue level, which in turn affects application performance. Flows in high priority queues experience much better performance when compared to those in low priority queues.

QoS Policy (Orange Tiles): The centralized controller makes policy decisions (assignment of flows to queues) which are communicated to the Access Point using the OpenFlow protocol. We reuse a custom message format described in Section 2.5 to send MAC Layer commands. The Access Point is installed with SoftStack, which interprets the received messages and implements

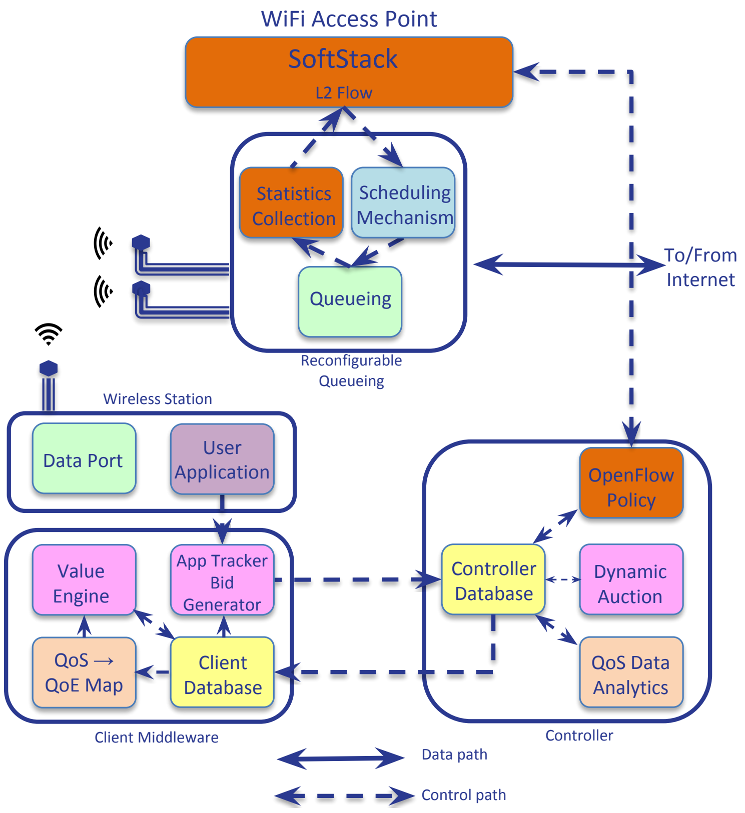

Figure 3.3: FlowBazaar architecture.

the policies selected by the controller. The Statistics Collection component at the Access Point periodically collects network connection statistics like throughput, drop rates and RTT and sends them back to the controller in a predefined message format using the OpenFlow protocol.

Application QoE (Beige Tiles): A smart middleware layer at clients is used to interface with our system. It retrieves the state of the foreground application and translates it into the impact on perceived QoE using the DQS model. This layer ensures that all the other functional components remain transparent to the end user.

End-user Value (Pink Tiles): Clients are offered high and low priority service under a market framework. The decision on which N flows to admit to a high-priority queue is taken via an $(N+1)^{th}$ price auction, which is conducted every 10 seconds. The resultant policy decisions in turn lead to a realization of network connection metrics, which in turn impact the end-user perceived QoE. The Value Engine retrieves the application state from the Client Database and the statistics of the current market conditions (bid distribution) from the Controller Database. On receiving both, it is responsible for running the Value Iteration to obtain the optimal Value Function, which determines what the value of winning and losing would be. The Value Engine then sends an appropriate bid over to the Controller Database. The Dynamic Auction module collects the bids of all participating clients and conducts the $(N + 1)^{th}$-price auction. The results of the auction translate into assignment of clients to queues, that remain in place for 10 seconds.

Interactions: The chronological order of the interactions among the three functional units is as follows. The Client Middleware, after determining the foreground application on the wireless station, calculates the corresponding QoE values. Based on these QoE values, the Middleware determines the value of winning and losing using the Value Engine and places a bid accordingly. Bids from all participating clients are sent to the Controller, which conducts the auction. The results of the auction, which are the policy decisions, are sent to the Access Point using OpenFlow Experimenter messages. SoftStack interprets and implements these policy decisions, which leads to the updation of network connection parameters and finally the application QoE. These steps are executed once every 10 seconds by the Client Middleware, the Controller and SoftStack. The (updated) list of client and queue statistics is sent back to the Controller every second.

3.5 Policy Design

In this section, we determine the policy obtained for the System-wide and Auction based models.

3.5.1 Transition Kernel of an Individual Client

To determine these policies, we first need to compute the transition kernel of an individual client, ie., we need to determine $\mathbb{P}(s_{t+1}^c | s_t^c, a_t^c)$. We do this as follows:

1. We generate state (s_t^c), action (a_t^c) and next state (s_{t+1}^c) tuples for clients by running the system described in Section 3.4 for a duration of 10 hours under vanilla, round robin and greedy buffer (discussed in Section 3.6) policy.

2. The state s_t^c is a 3 dimensional vector (Buffer, Stall, QoE) consisting of continuous values, thus we discretize the state space according to table 2.2. We encode the discretized state space to obtain a label for each state. Let NSB and NQB denote the number of stall and QoE bins respectively,

$$s_t^c = \text{Buffer} \times \text{NSB} \times \text{NQB } + \text{QoE} \times \text{NSB } + \text{Stall}$$

3. We fit an empirical distribution over the discretized data to obtain the transition kernel.

3.5.2 Optimal Policy for System-wide Model

To obtain the optimal policy for the system-wide model, we have to first determine the system transition kernel i.e., given the current state s_t of the system and the action taken a_t, we find the transition probabilities to the next states s_{t+1}. Given the transition kernel of the system, we can use policy or value iteration to solve for the optimal policy π^*. The system-wide approach is particularly interesting because of its special structure, since the state transitions of a client given its current state and action are independent of the states and actions of other clients in the system.

In other words,

$$\mathbb{P}(s_{t+1}|a_t, s_t) = \prod_{\forall c \in \mathcal{C}} \mathbb{P}(s^c_{t+1}|a^c_t, s^c_t)$$

It must also be noted that the state transitions of all clients in the system given their current states and actions are identical. Thus, we can determine the transition kernel of the system using the transition kernel of each individual client.

Directly determining the system transition kernel and solving it to obtain the optimal policy is intractable due to the large state space of the system. Consider a system with N clients (the state space is an N dimensional vector, with each dimension corresponding to the state of a client), where each client has a state space of the order of 10^3. The state space of the system ($\mathcal{S}$) is of the order of 10^{3N} which is very large. To combat the problem of state space explosion, we take advantage of the structure of the problem and identify the *popular states* of the system ($\mathcal{S}_p$), and approximate all the other states to the closest popular state under the L^2 norm.

To obtain the system transition kernel, we empirically fit a distribution over the transitions generated for each state in $\mathcal{S}_p$ under each action in $\mathcal{A}$ using the transition kernel of an individual client (Section 3.5.1). If the transitions generated are outside $\mathcal{S}_p$ we approximate it with the state closest in $\mathcal{S}_p$ under the L^2 norm. Once the system transition kernel is obtained, we run value iteration solve (3.1) to obtain the optimal policy. It must be noted that the reward obtained by taking action a_t in state s_t is the expected QoE of state s_{t+1}.

3.5.3 Optimal Policy for Auction-based Model

Using the transition kernel of a client (Section 3.5.1), we use Value Iteration to solve (3.2) and obtain the optimal value function for an individual client. This approach also provides a map between state of a client and its bid, which is subsequently used in the $(N+1)^{th}$ price auction. Since it is known that $(N+1)^{th}$-price auction promotes truth telling, a client participating in such an auction makes a bid which reflects its true value. The Auction Agent receives the bids from all clients, conducts the $(N+1)^{th}$-price auction and performs the assignment on the basis of the result.

3.5.4 Index Policy

The solution to (3.2) results in a *value* for each state s_t^c of the client. We can order states in increasing order of value, and associate each state with an *index,* which is its position in the order. Then these indices can be used to directly decide which clients to prioritise, and we call this as an *index policy.* Now, given the indices corresponding to a system with N clients, it would save computational effort if we could use the same indices for a system with $M < N$ clients, by simply setting indices of non-existent clients to 0. The question is whether the indices for a system with 6 clients are consistent with (for example) one that has 3 clients?

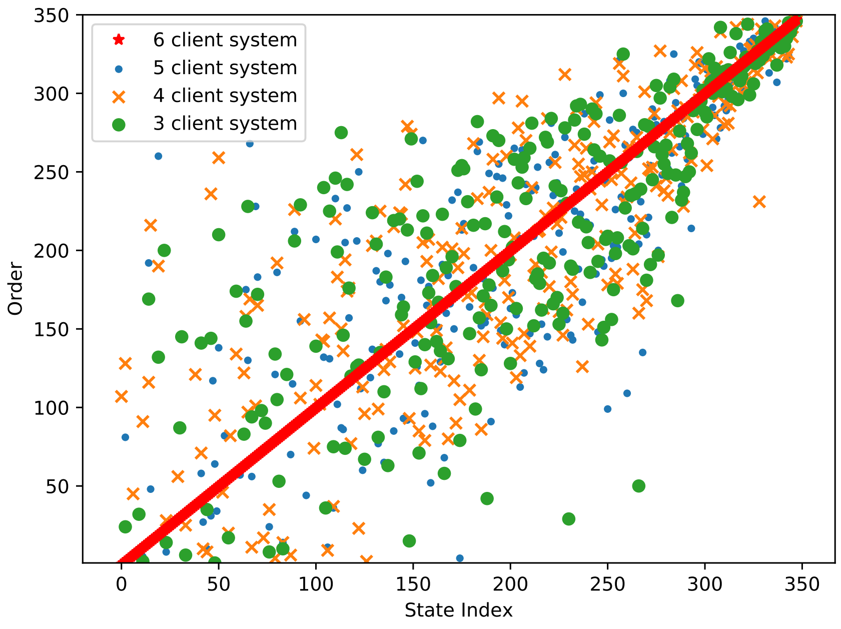

Figure 3.4: Ordering of states in different client configurations.

We experimentally determined the values for different numbers of clients, and calculated the state indices in each case. The comparison of the orderings for different client configurations (6, 5, 4, 3 clients) is shown in Figure 3.4, using the ordering for 6 clients as the base ordering. We observe that the relative ordering of most of the high value states is consistent across configurations, which indicates that an index policy for 6 clients is likely to perform well for one with fewer clients.

3.5.5 Dynamic Number of Clients and Varying Channel

In the previous subsections we assumed that the number of clients and channel state of the system were static. To deal with a dynamic environment, ie., varying number of clients and channel states (described in Section 3.6), we first obtain the optimal policy for all the different scenarios using the static approach described in section. We then construct a composite controller which chooses the appropriate policy based on the environment. This approach works well in practice since the time scale in which the environment changes is larger than the decision period.

3.6 Evaluation

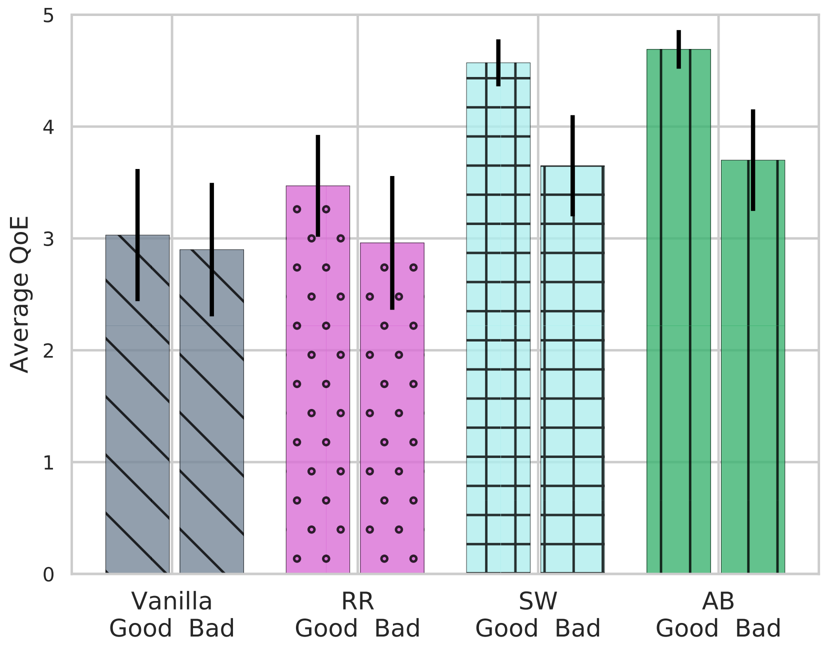

Figure 3.5: Comparison of average QoE

We ran a series of experiments to evaluate the performance of the described policies on a testbed hosting multiple YouTube sessions. A WiFi router installed with SoftStack is used as the

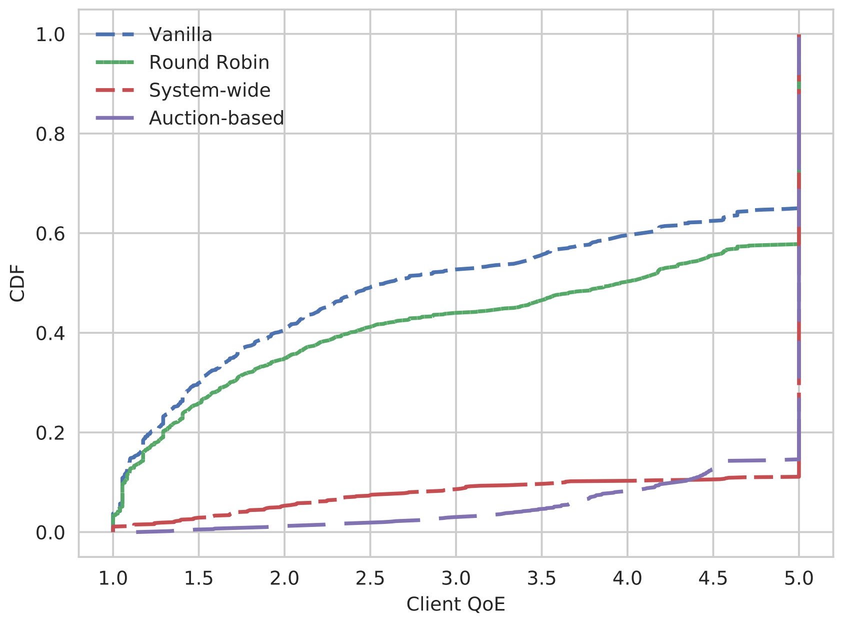

Figure 3.6: Client QoE comparison for good channel

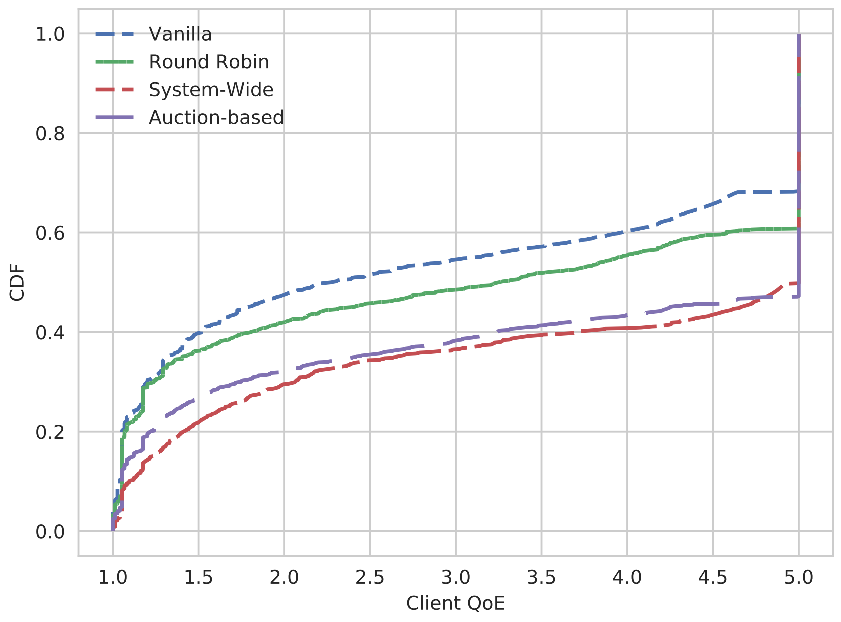

Figure 3.7: Client QoE comparison for bad channel

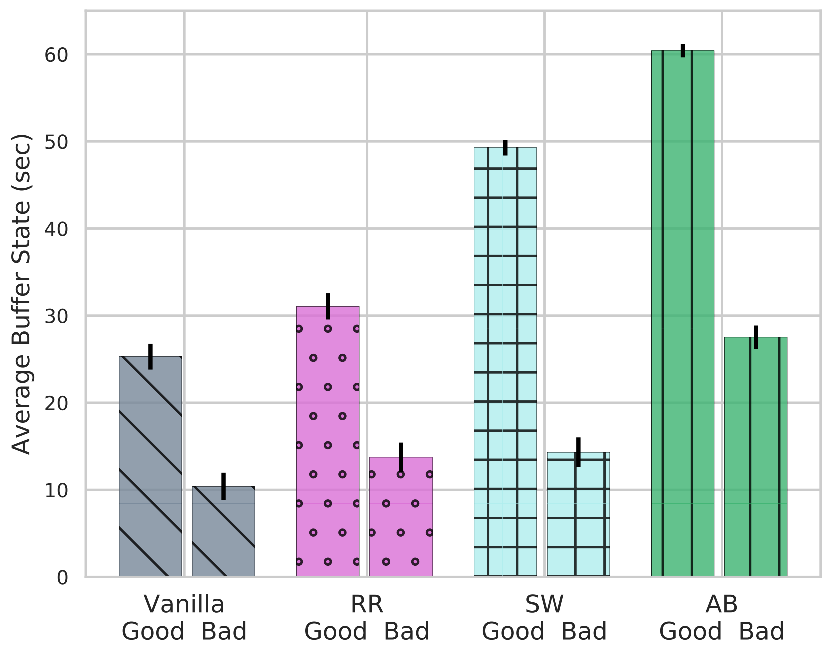

Figure 3.8: Comparison of average buffer

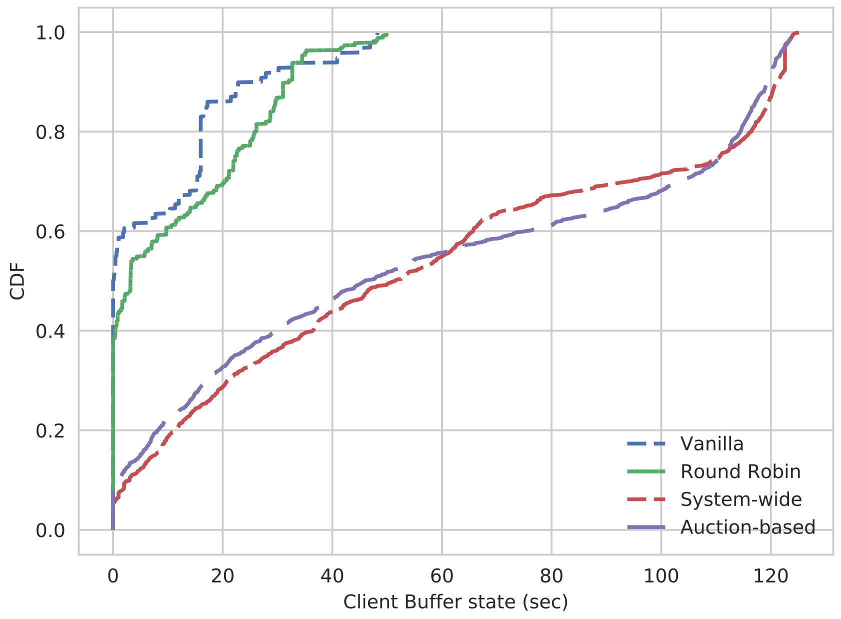

Figure 3.9: Client buffer comparison for good channel

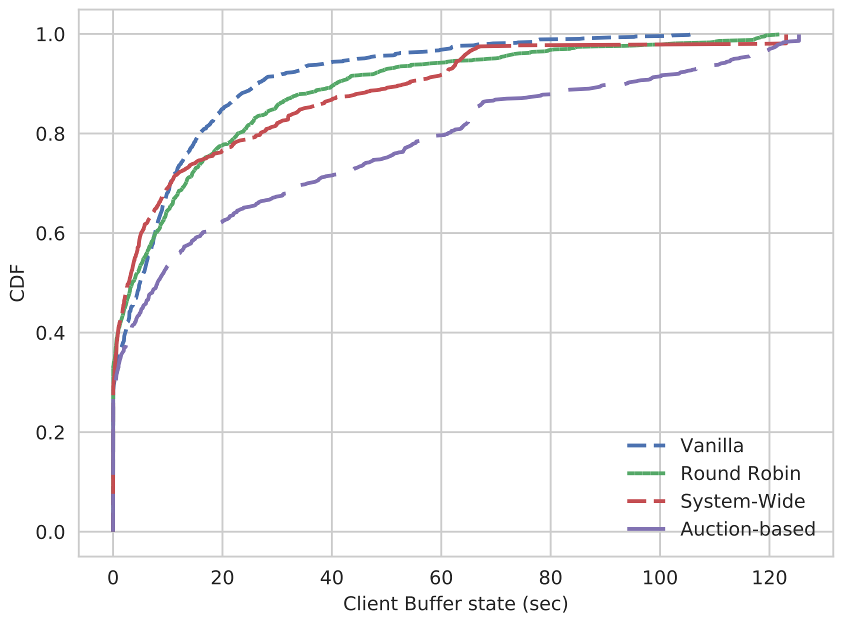

Figure 3.10: Client buffer comparison for bad channel

Access Point and three Intel NUCs are used to simulate up to 9 clients (YouTube sessions) for our experiments. Although each YouTube session can be associated with multiple TCP flows, all such flows are treated identically. Each of the NUCs is configured with an i7 processor and 8 GBs of memory, and is powerful enough to host multiple traffic intensive sessions simultaneously. Ubuntu Operating system is installed on the NUCs, which makes it easier to measure session specific information such as ports used by an application, play/load progress, bitrate and stall information. This information is collected every second and written to the centralized database for ease of sharing.

Since we are interested in studying the behavior of queues in routers, we create two bins of downlink queues, each containing a high priority and a low priority queue. The motivation behind the creation of two separate bins is to ensure that clients having similar signal strengths are eligible for the same bin and hence, do not adversely affect the performance of client with better signal strengths. Hence, we have a *Good* bin for clients with high signal strengths and a *Bad* bin for those who have low signal strengths. These bins are attached to the WiFi interface of our AP using *tc*. We allocate a higher bandwidth to each of the high priority queues using hierarchical token bucket queueing discipline. This is done so that clients assigned to these (high priority) queues experience better service than those in the other (low priority) queues. For the current scope of experiments, we set the admission limit of the high priority queues to two clients. We also create a default queue to accommodate any background traffic.

3.6.1 Emulation of bad network conditions

Since we have a fixed number of NUCs to host a total of upto 9 sessions, we decided to emulate bad channel by reducing the throughput and increasing the latency and loss rates of the queues in the Bad bin as compared to those in the $Good$ bin. We ran several hours of experiments with clients having low signal strengths to come up with these heuristics for emulating a bad channel. This enables us to mimic varying network conditions by dynamically assigning the sessions hosted on the NUCs to either the $Good$ or the Bad bin.

Given this setup, the control problem is to determine the assignment of sessions to queues under varying channels.

3.6.2 Policies

We compare the **System-wide** (SW) policy described in Section 3.3.1 and the **Auction-based** (AB) policy described in Section 3.3.2 with two other policies for determining the assignment of sessions to queues.

Vanilla: This baseline scenario consists of a single queue which treats all clients equally. The queue is allocated the total bandwidth of the queues used for the other policies.

Round Robin: The Round Robin (RR) policy assigns clients to the high priority queue in a cyclic manner. It is simple, easy to implement, work-conserving and starvation-free. But it might promote the wrong clients (like clients who have stalled multiple times) to the high priority queue instead of those who might benefit much more from the assignment.

3.6.3 Static Network Configuration

We run the first set of experiments in a static configuration consisting of 6 clients hosted on three NUCs under both channel conditions. For the $Good$ channel scenario, the total bandwidth allocation of the two downlink queues is set such that simultaneous playback of all the YouTube sessions cannot not be supported at HD (1080p), whereas for the Bad channel scenario, we further reduce the bandwidth and add latency and loss. Figure 3.5 shows the comparison of the average QoE achieved by the policies under the different channel conditions. It is evident from the figure that the System-wide and the Auction-based policies perform much better than the other policies. Interestingly, the Auction outperforms the System-wide policy. We believe that this is due to the coarse quantization of the state space. System-wide is worse affected by this due to the fact that 6 clients together are considered in the sate, whereas in Auction only 1 client is part of the (marginal) state. Hence, we believe that value identification is more accurate in the Auction case. The difference in achieved performance for the different policies becomes more clear in the comparison of the CDFs of the client QoE under $Good$ and Bad channel conditions in Figures 3.6 and 3.7. For example, we can observe from Figure 3.6 that the System-wide and the Auction-based policies are able to provide a QoE of 5 for almost 90 and 85% of the time for all clients, whereas it is only about 40% of the time for Round Robin. Similarly, we can deduce from Figure 3.7 that the Model-based and the Auction-based policies are able to achieve a QoE of 5 for all participating clients in the system about 50% of the time, whereas Round Robin is only able to achieve it 40% of the time. We can observe that this gap decreases in the Bad channel scenario, but the System-wide and the Auction-based policies still achieve higher QoE for the clients.

Since the QoE perceived by a client depends on the state of the buffer and the stalls experienced during video playback, we also study these individual components for the different policies under the different channel conditions. We can observe a similar trend when we compare the CDFs of the individual values for both these components in Figures 3.8 to 3.13. We can clearly infer from the figures that the System-wide and the Auction-based policies ensure higher number of buffered seconds and lower duration of stalls under both channel conditions when compared to the other

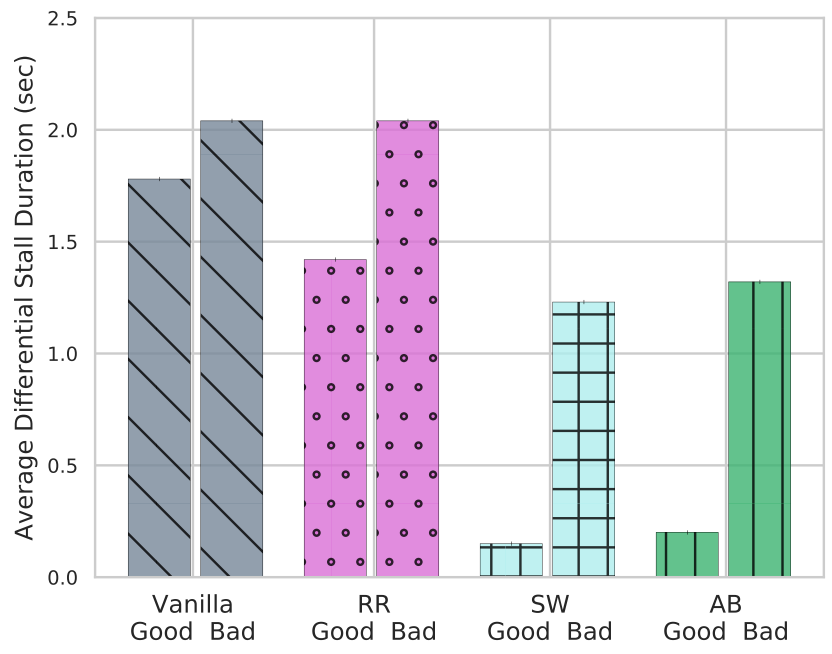

Figure 3.11: Comparison of average stall duration

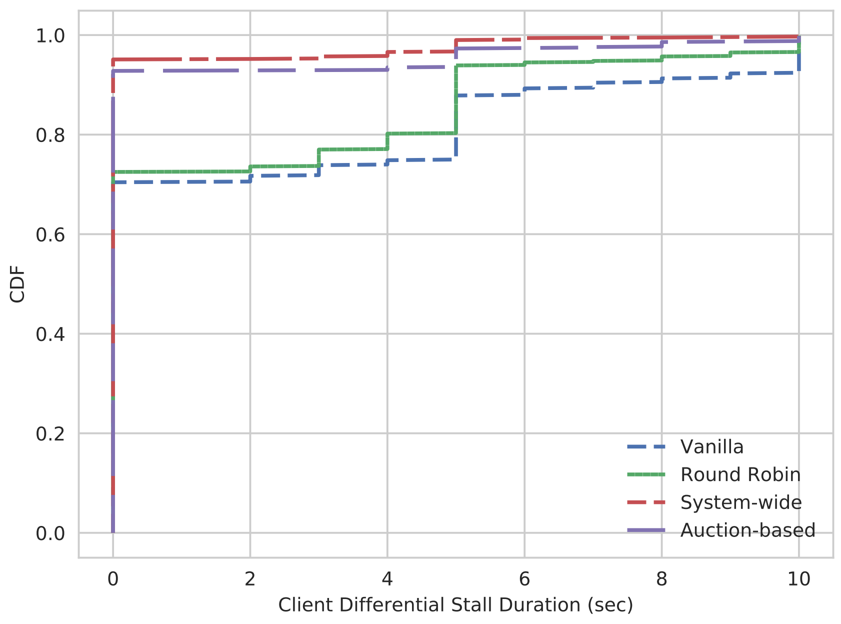

Figure 3.12: Stall duration comparison for good channel

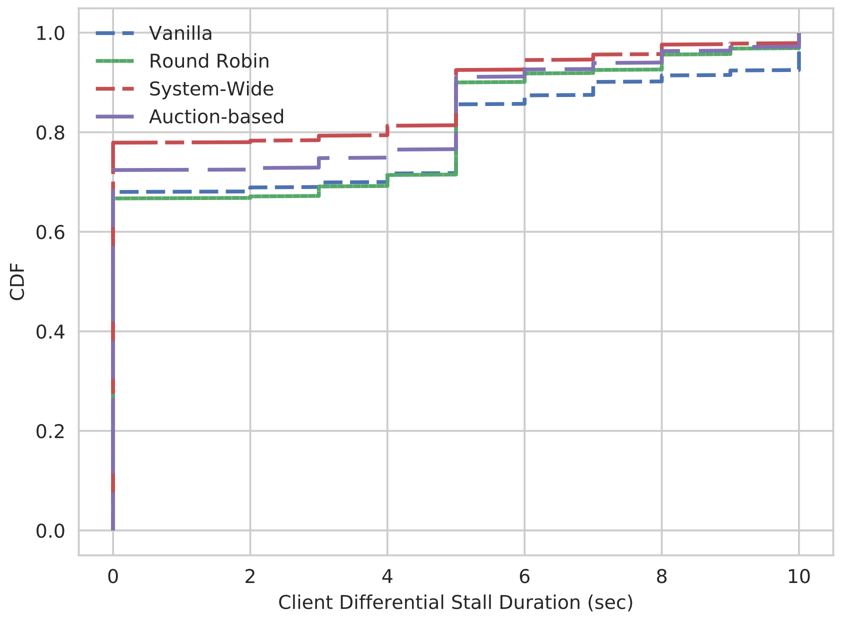

Figure 3.13: Stall duration comparison for bad channel

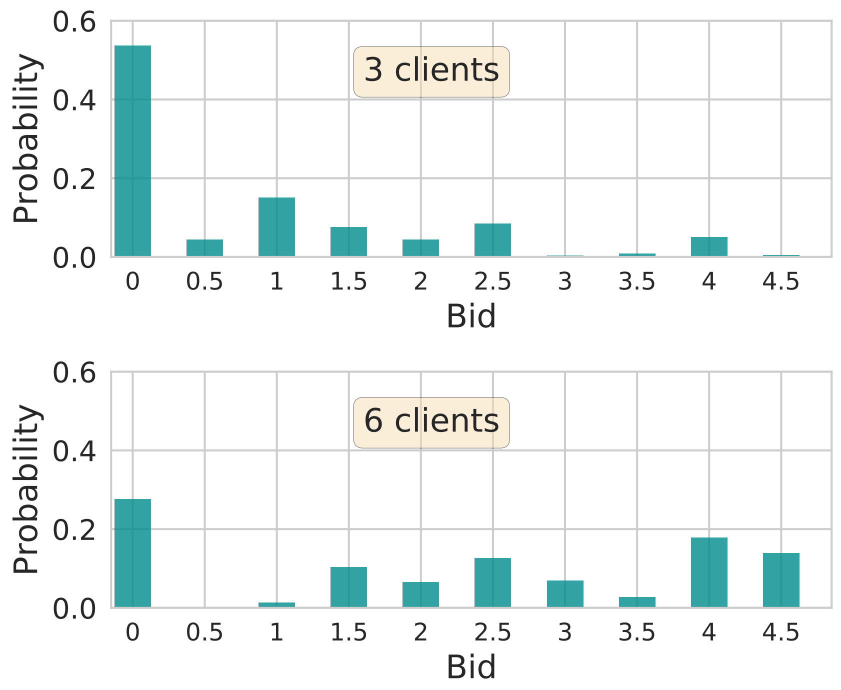

Figure 3.14: Bid distribution for 3 and 6 client configurations

policies, even though the gap in performance is much less in the Bad channel case.

We also compared the bid distributions of the clients in the Auction-based policy for two different client configurations. The first configuration had 3 clients whereas the second one had 6, with the total bandwidth allocation kept the same. The comparison of the two distributions is shown in Figure 3.14. When there are more clients participating, resources are scarce and valuable, and so, clients tend to bid higher in order to get into the high priority queue and experience better QoE. When the total number of clients is low, everyone experiences good QoE irrespective of the queue they are assigned to, and there is no incentive to bid higher.

3.6.4 Dynamic Number of Clients and Varying Channel

In a realistic setting, the number of participating clients as well as the channel conditions do not remain static. Hence, we would like to evaluate the performance of the policies with dynamic number of clients under varying channel conditions. We vary the number of active clients in the system between 3 and 6 for the next set of experiments, under the assumption that a particular configuration of clients remains unchanged (in the same channel condition) for a duration of 30 minutes. Since this duration is large as compared to the policy decision period of 10 seconds, the use of a composite controller as described earlier is only slightly sub-optimal.

We fix a sequence of client configurations (number of active clients) under each channel condition for the evaluation of all policies. The first configuration consists of 6 clients under $Good$ channel conditions and 3 under Bad channel conditions. We decrease the number of clients in the $Good$ scenario by 1 and increase that in the Bad scenario by 1 for the next three intervals. The evolution of the average QoE for each of the policies for the above sequence is shown in Figure 3.15. The System-wide and Auction-based policies exhibit a high average QoE in most of the configurations except for the last where it is not possible to achieve a high QoE for the 6 clients in the Bad channel. Even in such a scenario, the drop in QoE is more severe for the other policies.

Decrease in the number of active clients under a constant bandwidth allocation results in the relaxation of the constraints for each individual client and hence, Round Robin and even the Vanilla approach perform better in some configurations. This improvement in performance, when com-

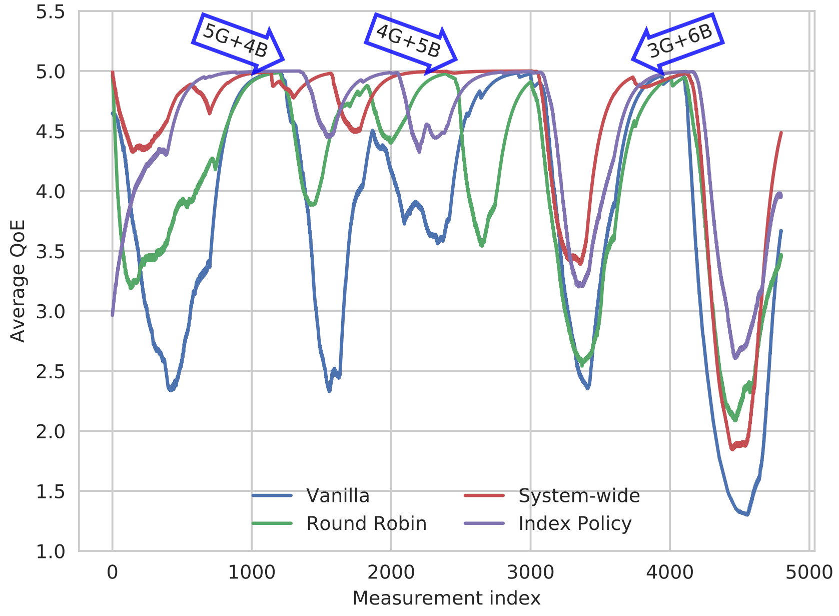

Figure 3.15: Evolution of QoE for dynamic clients in variable channel

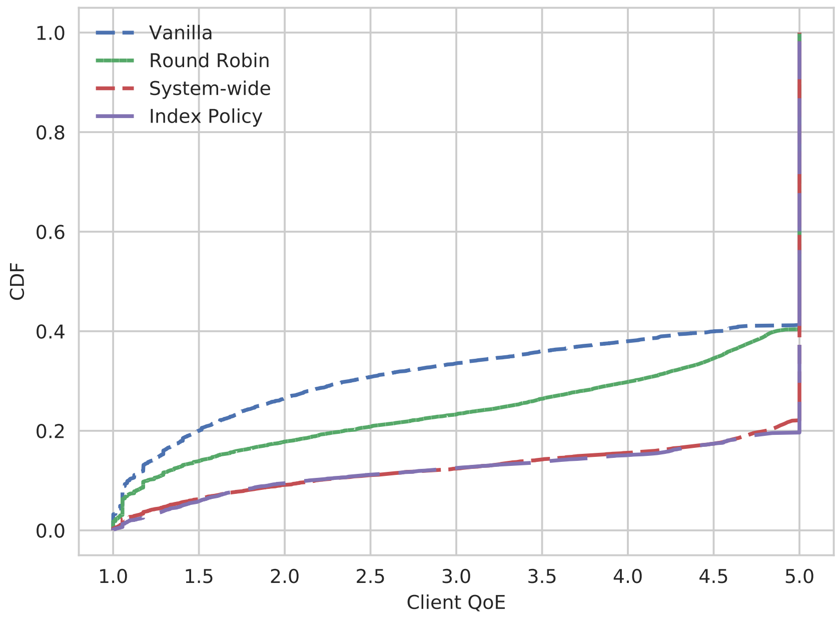

Figure 3.16: Comparison of client QoE CDF for dynamic clients in variable channel

pared with the constrained setting, can be seen in Figures 3.16, where the CDF curves of both policies are closer to those of the System-wide and the Auction-based policies. Nevertheless, it is evident that System-wide and Auction-based policies outperform the other policies, which strengthens our claim that they are superior to other policies in both static and dynamic scenarios.

3.7 Conclusion

In this paper, we described FlowBazaar, a platform for adaptive prioritization of flows in response to their self-declared values. We showed that using an auction framework is able to a elicit truthful proxy for state in terms of the bid made for prioritized service. Furthermore, the model needed at clients to make optimal bids is simply the marginal transition kernel of the system, which is learned quite easily. Using YouTube video streaming as an example application, we showed how FlowBazaar is able to make the correct choices on which clients to prioritize, and actually ensured higher overall QoE than system-wide optimization (using a coarsely quantized state space). We also discovered an index policy (ordering of state values) that can be applied directly as a simple and effective policy. Our future goal is to analytically characterize the nature of this index policy.

4. PULS: PROCESSOR-SUPPORTED ULTRA-LOW LATENCY SCHEDULING*

4.1 Introduction

Strict latency requirement is currently one of the most critical challenges for next-generation wireless networks. Emerging applications, such as virtual reality (VR) [30], factory IoT, and tactile Internet [31], require an end-to-end latency between 1 to 10 milliseconds to provide seamless user experience. However, the existing wireless networks cannot provide such stringent latency guarantees. For example, the round-trip time of LTE is estimated to be at least 20 milliseconds, including the transmission time, scheduling overhead, and processing delay [32]. In practice, the current LTE technology can only support voice or video streaming applications with round-trip time in the range of 20-60 milliseconds [33]. For Wi-Fi networks, due to the nature of random access, the round-trip time could vary from several milliseconds to hundreds of milliseconds depending on the traffic load [34]. Therefore, compared to the current technology, the latency budget is expected to be at least one order of magnitude smaller in next-generation wireless networks.

To provide strict per-packet latency guarantees, numerous theoretical solutions have been proposed to accommodate *per-packet deadline constraints* in wireless scheduling. For example, [35] proposes a theoretical framework to study wireless scheduling with per-packet deadline constraints. In this framework, packets not delivered on time are dropped. Later on, this framework has been applied to many other scenarios, such as utility maximization [36], scheduling for both latency-constrained and best-effort traffic [37], broadcast traffic [38], and multicast traffic [39]. The performances of these protocols are usually measured by *timely-throughput*, i.e. the time average of the amount of data delivered within their deadlines. While the above proposals are promising, there has been no implementation for these ultra-low-latency wireless protocols.

As such, we do not know what kinds of system throughputs and capacity regions are achievable with these systems. Without knowledge of the throughputs and capacity regions, these protocols

*Republished with permission of ACM, from PULS: Processor-Supported Ultra-Low Latency Scheduling, Simon Yau, Ping-Chun Hsieh, Rajarshi Bhattacharyya, K. R. Kartic Bhargav, Srinivas Shakkottai, I-Hong Hou, P. R. Kumar, MobiHoc, 2018; permission conveyed through Copyright Clearance Center, Inc.

may not function as expected.

Furthermore, future networks are expected to support many heterogeneous applications. Such diversity necessitates a system that is capable of switching between a set of MAC protocols for different applications. For example, one might choose to use backpressure/max-weight scheduling for throughput maximization [40, 41], deadline constrained scheduling for latency sensitive flows [35, 42] or perhaps a time division solution to support polling of IoT devices or in vehicular communication [43, 44]. Thus, a range of scheduling algorithms need to be instantiated, with the scheduling algorithm that is to be applied to a particular aggregate of packets being chosen based on its requirements. This can be difficult to implement on hardware due to the development time required, and the fact that instantiating each algorithm occupies hardware resources (on either FPGA or ASIC) that cannot easily be shared.

Another constraint appears in the form of the computational resources needed to support these algorithms. For example, the Max-Weight Independent Set problem that needs to be solved repeatedly for scheduling in multihop wireless networks is known to be NP-hard, and even polynomial approximations need significant computational resources [45]. More recently, scheduling algorithms that take a given schedule and "puncture" it with delay sensitive packets have been proposed to increase the efficiency of the MAC for 5G applications, and these algorithms also require significant computational resources [46]. As machine learning approaches make their way into MAC algorithms, the need for high speed computation at each scheduling decision will only become more pressing. Hence, the desire to support a diverse range of scheduling algorithms, coupled with the fact that some of these algorithms might require significant computational resources suggests the need for a platform that can support processor-assisted software scheduling. Thus, to prototype these ultra-low-latency wireless protocols and achieve realistic timely-throughput and system throughput, a powerful software-defined radio (SDR) platform with dedicated architectural design is required to compute complex algorithms quickly, provide enough latency budget for checking deadlines, as well as minimize the software and hardware processing and interfacing overhead.

Our main contribution is to bridge the gap between theory and implementation for heterogeneous wireless networks supporting flows with strict per-packet latency constraints (real-time flows), as well as flows that have no latency constraints (non-real-time flows), while maintaining high overall system throughput by proposing PULS, a processor-supported software-defined wireless platform that can support ultra-low-latency scheduling protocols. The PULS platform consists of a host machine that has significant computational power in the form of a general purpose multicore CPU, coupled with an SDR platform with FPGAs for low-level processing. PULS aims to leverage the higher clock speeds, and memory available at the host machine (which are at least an order of magnitude higher than what is available on SDRs) for performing complex scheduling algorithms, while leveraging the deterministic performance of the FPGA while performing simple repetitive tasks associated with PHY and low-level MAC layers. To achieve the required per-packet latency performance while performing scheduling on the host, PULS needs to address the following challenges:

1. **Enforce per-packet deadline on a software-defined wireless platform.** PULS aims to support per-packet latency as low as one millisecond. When packets arrive at software host, they are first queued and start waiting for transmission according to some scheduling policy. The deadline of each packet in the queue needs to be tracked and checked before transmission. A packet that misses its deadline should be dropped from the queue. Moreover, due to the nature of SDRs, packet transmission is carried out on hardware while packet scheduling is often done on software host. Therefore, it is not directly clear how deadlines should be maintained and checked on a software-defined platform if there is no synchronization between software and hardware. Moreover, it usually requires non-trivial efforts to provide an accurate reference timer shared by software host and hardware. In Section 4.4.3, we present a simple design that tracks packet deadlines accurately using only timestamps of software host, without any synchronization between software host and hardware.

2. **Low interfacing latency between software host and hardware.** There are three major factors that affect the end-to-end latency: (i) queuing delay on software host, (ii) interfacing

latency between software host and hardware, and (iii) hardware processing time. Queuing delay depends mainly on the scheduling policy, and hardware processing time can usually be made small due to the high clock rate supported by current technology. Therefore, with a proper choice of scheduling policy and hardware component, interfacing latency between software host and hardware needs to be minimized in order to achieve ultra-low latency. In Section 4.5, we present a simple experiment that demonstrates the interfacing latency of PULS is indeed small compared to packet deadlines.

3. **Achieve realistic per-flow timely-throughput.** Ultra-low latency needs to come with realistic timely-throughput. Given the same physical data rate, the overhead of enforcing per-packet deadlines could be quantified by the difference in total MAC-layer throughput between the networks with and without packet deadlines. In Section 4.6.1, through an experimental study on system capacity, we show that PULS achieves almost the same MAC-layer throughput as that with no deadlines.

4. **Support functions working on heterogeneous time scales.** MAC layer functions operate on very different time scales. For example, an ACK response needs to be done within tens of microseconds. The transmission time of a typical data packet is between 0.5 to 1 millisecond. The target per-packet deadline is between 1 to 10 milliseconds. The parameters of wireless protocols usually change over a period of at least several seconds to several minutes. In Section 4.3, we describe the separation principles of PULS which inherently incorporates the heterogeneity in time scale.

5. **Support various ultra-low-latency downlink applications.** For example, VR requires latency as low as 1 millisecond with moderate timely-throughput while factory automation needs ultra-low packet loss rate with latency of about 5-10 milliseconds. PULS is able to support applications with totally different performance requirements and provide a programmable environment for different wireless protocols.

To tackle the above challenges, PULS follows three major design principles. First, as a software-defined wireless testbed, PULS follows the Host-FPGA separation principle for both high flexibility and performance. Next, PULS addresses the heterogeneous time scales of MAC functions by applying Mechanism-Policy separation. Third, to support a broad class of scheduling policies, we borrow ideas from both WiFi as well as LTE standards, and build up a set of basic MAC functions required by most of the wireless protocols.

4.2 Related Work

Most of the existing experimental studies for wireless LANs focus primarily on maximizing system throughput or throughput-based network utility. For example, to achieve maximium throughput, the well-known backpressure algorithm has been tailored and implemented for various scenarios, such as multi-hop wireless networks [47], TDMA-based MAC protocol [48] and wireless networks with intermittent connectivity [49]. Besides, for wireless LAN with random access, [50] implements an enhanced version of 802.11 DCF and demonstrates that it achieves near-optimal throughput as well as fairness with the original DCF. However, all of the above studies provide no support for packets with latency constraints. To address latency requirement for industrial control applications, RT-WiFi, a WiFi-compatible TDMA-based protocol, has been proposed and implemented on commercial 802.11 interface cards [51]. However, it cannot achieve both ultra-low latency and satisfactory timely-throughput performance for each user at the same time due to the nature of TDMA.

On the cellular side, several preliminary studies about 5G provide candidate solutions to enhance the low-latency capability via either numerical and experimental evaluation. [52] studies the trade-off between latency budget and required bandwidth by applying the conventional OFDM framework to 5G networks through numerical analysis. However, these numerical results do not take the possible signaling and processing overhead into account. [53] provides experimental study for latency performance of 5G millimeter-wave networks with beamforming. However, this solution relies heavily on the beam-tracking technique and frame structure employed by cellular networks and cannot be directly applied to wireless LAN applications. Besides, [54] demonstrates a wireless testbed that is potentially capable of supporting millisecond-level end-to-end latency requirement. However, it supports only single link and does not take wireless scheduling issue into account.

4.3 Design of PULS

Our objective was to develop a platform for testing ultra-low latency protocols that require scheduling on a per packet basis, with a focus on downlink protocols. In this section, we will explain the basic design principles upon which PULS was built to achieve the goals described in the previous sections.

4.3.1 Basic MAC Functions for Wireless Scheduling

Our platform borrows ideas from both the WiFi and LTE standards. From the WiFi side, we used features such as Carrier Sense for loose synchronization between the nodes and some robustness to interference. We also use the same interframe space timing intervals as WiFi for transmission (and reception) of packets. Furthermore, ACKs are sent immediately following transmission of data packets after a SIFS period, which allows us to know in a short period of time whether the packet was transmitted successfully. On the other hand, we are using a completely centralized framework for scheduling the different queues, which is similar to what LTE does. By using ideas from WiFi and LTE, we are able to obtain a deployment that is both lightweight and can be incorporated more easily into our framework for ultra low latency scheduling.

4.3.2 Mechanism-Policy Separation

In our design of PULS, we utilize a mechanism-policy separation used in [55]. Mechanisms are functions or hardware blocks used to handle the low-level operations of packet transmissions over the network, whereas policy refers to the high-level specification of the scheduling protocol itself. This mechanism-policy separation builds on the framework of Wireless MAC Processors, introduced by Tinnirello et al in [56].

Each mechanism has a set of inputs, outputs, events, conditions to check and possible actions that can be performed. Inputs and outputs of a mechanism take the form of register values (e.g. channel state and average energy), or an array of bytes (e.g. my address and packet).

Mechanisms provide the set of actions, events and also act as condition checkers, whereas the policy specifies the set of enabling functions, the parameters for the conditions, the set of update functions and the transition relations for the state machine of the scheduling protocol. Having the distinction between the different mechanisms and its associated events, actions, and conditions allows us to design new mechanisms more cleanly and reuse previously developed mechanisms. In addition to the mechanisms used in [55] and in [56], we implemented mechanisms to allow for deadline checking, packet dropping, controlling the packet arrival process, as well as features to increment and decrement some notion of deficit according to user-speficied conditions. Deficits can be used in various ways to achieve some performance guarantees in delay-sensitive traffic, or to control the ratio of service times between inelastic and elastic flows. We will focus more on these mechanisms since they allow us to achieve certain guarantees on delay-sensitive traffic. These mechanisms can also be changed at runtime, allowing us to switch between different protocols on-the-fly. The implementation details of these mechanisms are laid out in section 4.4. Note, in the design of our testbed, mechanisms are implemented both on the FPGA as well as on the host machine. Furthermore, the inputs and outputs for each mechanism can be modified accordingly to allow for cross-layer designs. For example, the update deficit mechanism can use the packet's MCS as an input for the function to increment and decrement the deficits.

4.3.3 Flexible MAC through Host-FPGA Separation

For flexible MAC scheduling decisions, we employed a Host-FPGA separation, where high-level MAC functions such as packet scheduling, and packet dropping, are done on the host, and low-level functions such as packet encoding/decoding, carrier sensing, ACK processing, and CRC checking are done on the FPGA. This is done to allow for easy changes in packet scheduling decisions, while still being able to achieve our latency requirements for the platform.

Our testbed was designed with the following goals in mind:

- Ultra low-latency scheduling of packet transmissions on a packet by packet basis.
- Decoupling of specification of protocol from the underlying features of the system.

- Specification of scheduling protocols in the software domain.
- Allows for cross-layer design of protocols.
- Supports protocol changes at runtime.

4.4 Implementation of PULS

We will discuss the implementation details of PULS in this section, namely, the flow of events, and the design of mechanisms for the nodes. Since our platform focuses on downlink scheduling, the mechanisms shown here are geared towards that, although most can be re-used for uplink scheduling as well. Starting with the National Instruments 802.11 Application Framework, we implemented additional mechanisms on the Host machine and on a USRP-2953R for flexible scheduling protocols. To support packets with strict deadlines, PULS presents a data transmission procedure which enables per-packet scheduling on the host while keeping the FPGA design simple.

4.4.1 Packet Generation

For packet generation, we implemented a packet generation mechanism that allows three main parameters that can be tweaked to replicate the different kinds of traffic that a user might want to experiment with: 1) interarrival times, 2) probability that packets get generated, 3) number of packets that are generated. Packets can be generated with interarrival times up to 1ms in resolution, with a certain number of packets being generated at each time. In addition to that, packets can be generated deterministically or stochastically. At each interrival time, users can specify the probability that a certain number of packets get generated. Furthermore, the number of packets can be set to be a fixed number, or can take on a random number based on a distribution. For our purposes, in the random case, we use a simple uniform distribution for the number of packets that are generated. The maximum number of packets that can be generated are specified by the user. These arrival patterns and rates are by no means exhaustive, but it is sufficient for us to justify the performance of the platform and of particular protocols that we've implemented. (The details of the results are in Section 4.6.)

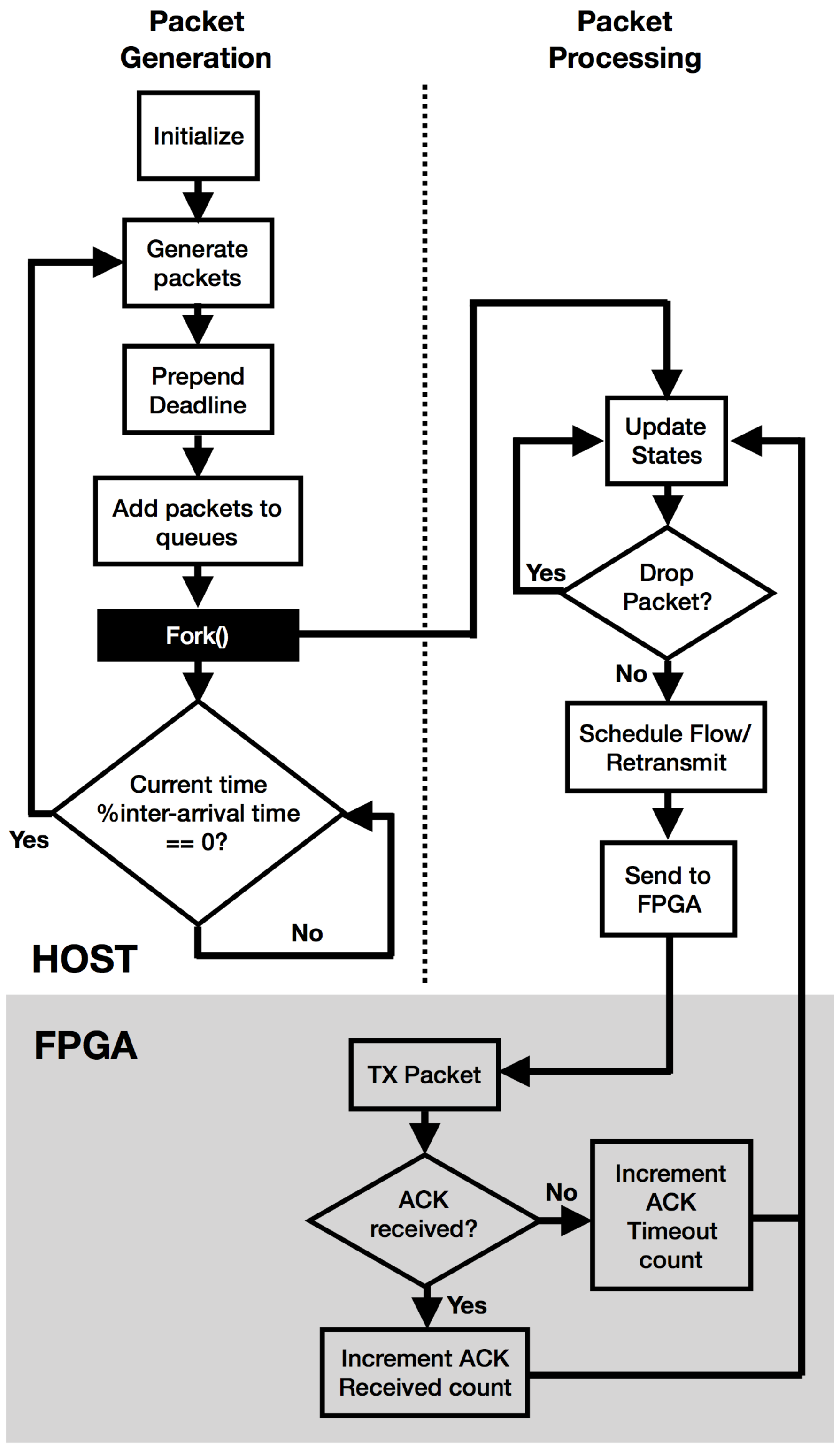

Figure 4.1: Architecture of PULS.

4.4.2 Queueing, Deadlines and Types of Flows

Each data flow, either elastic or inelastic, is associated with a queue on the Host. To add deadlines to the packets, we implemented a simple mechanism to prepend deadlines to each packet arriving at the queue. The input of this block is the current tick count of the system, the deadline, the incoming packet, and all the references to the queues associated with each flow. This block takes the current tick count of the system, adds the correct tick count corresponding to the deadline of the packet and prepends it to the packet before adding it to the correct queue. For inelastic flows, when a packet is generated (according to the Packet Generation mechanism), the corresponding *deadline identifier*, in the format of Host reference timer, is prepended onto the packet and then the packet is queued for scheduling. Since the absolute packet deadlines of each flow are assumed to form a non-deceasing sequence, every queue on the Host is always inherently arranged in an earliest-deadline-first manner. (Note, this may not be true if packet deadlines are changed frequently in a non-increasing manner. However, applications typically just require a baseline performance guarantee so this is valid for most cases. In the event that deadlines for packets are not arranged in an earliest-deadline-first manner, we can sort the packets while packets are being transmitted.) On the other hand, for elastic flows, the packet deadlines are set to be negative one for simplicity and better modularity in design.

4.4.3 Scheduling and Transmission Procedures

Once packets are generated and the packets are in their respective queues, scheduling is performed and repeated on a per-packet basis. In PULS, scheduling is aided by the use of several mechanisms to achieve our desired latency goals. First, we have packet dropping mechanism which scans the head-of-line packet of each queue, and drops the packets that have expired. This block only requires the references of each queue, and the current tick count of the system as inputs. It also has an output to inform other mechanisms whether or not a packet has been dropped. Next, we have a mechanism to update the state of the flows. Its job is to update the deficits associated with each flow based on whether or not an ACK was received and if a packet has been dropped from the queue. It uses the references of the queues, ACK received count, ACK timeout count, and the output of the packet dropping mechanism as its inputs. Lastly, we have a scheduling decision block that decides which flow to schedule based on the current states of the flows. This mechanism uses the deficits associated with each flow, the ACK counters for each of the flow, and the output of the packet dropping mechanism as inputs. All scheduling-related mechanism blocks are executed on the Host.

Scheduling and transmission executions are triggered when at least one queue is non-empty, and the scheduling state is UNLOCKED. The scheduling state becomes LOCKED as soon as a packet has been scheduled and is being processed for transmission. The state becomes UNLOCKED again when either of the two following FPGA events happen: an ACK reception or an ACK timeout. A successful ACK reception happens when an ACK packet is decoded successfully. This in turn increments the number of ACKs that have been received. The Host machine polls the register that stores the number of ACKs that have been received, and when the numbers differ from one iteration of the while loop to the next, the Host registers an ACK reception. For ACK timeouts, we implemented a mechanism that starts a counter after a packet has been transmitted on the FPGA. If an ACK is not received within a specified time period (set to be 75 microseconds), the count for timeout is increased on the FPGA. As in the case of ACK reception, if the count for ACK timeouts that the Host polls from the FPGA differ from one while loop iteration to the next, an ACK timeout event is registered. The timing of loop executions will be described further in Section 4.5. In every loop cycle, exactly one packet is scheduled for transmission. Before making a scheduling decision, the Host first "cleans up" the queues and updates the deadline-related state information by dropping expired packets in each queue using the mechanisms detailed above. Given the computing power of the Host, this cleanup can be finished almost instantaneously compared to the transmission time of a packet. (We should note that there will be more overhead if a batch of packets have expired, since we are only dropping one packet per loop execution, but this time is also negligible relative to the packet transmission time.) Given the queues in a clean state, the flow scheduler sets priorities of the flows according to the scheduling policy and schedules the flow with a non-empty queue and the highest priority. The scheduled queue then sends the head-of-line packet to the Prepare ICP Packet block, which removes the deadline identifier and appends the ICP header to the packet. The ICP header carries the information required by the FPGA, such as packet length, modulation and coding scheme, source MAC address, destination MAC address, and flow identifier, etc. Upon receiving the scheduled packet from the Host, FPGA simply triggers the required channel access procedure of the MAC layer as well as the physical transmission procedure in the PHY layer. By placing all the scheduling complexity on the Host, the design of PULS can be easily reproduced on an existing wireless interface card. The Access Point (AP) packet transmission procedure is summarized in Algorithm 1 and the function blocks associated with the transmission procedure are depicted in Figure 4.2.

4.4.4 Retransmission

As reliable transmission is often required by mission-critical low-latency applications, PULS also supports retransmission for both elastic and inelastic flows to recover packet losses. Note that while MAC-layer retransmission is usually implemented in the hardware for conventional SDRs to minimize latency, the retransmission block of PULS is located in the Host for two reasons: (i) Packet deadlines need to be checked before any retransmission. Since deadlines are tracked in the Host, it is straightforward to handle retransmission in the Host. (ii) The Hostto-FPGA interfacing latency is low enough for supporting retransmission in the Host.

In PULS, retransmission is built on top of the scheduling and data transmission procedure described in Section 4.4.3. An additional retransmission mechanism with retransmission queue is created on the Host for temporarily storing the duplicate of the scheduled packet in the current loop cycle. The inputs to this block are the references to all the queues, the ACK counts for the flows, and the maximum retransmission attempts which can be easily configured in the Host according to the user specified scheduling policy. If an ACK timeout is received and the maximum retransmission count is not met, then the scheduler will attempt to retransmit the same packet in the next cycle; otherwise, the duplicate packet is removed from the retransmission queue.

The ICP packet is sent to the SDR's FPGA fabric via a Direct Memory Access (DMA) Chan-

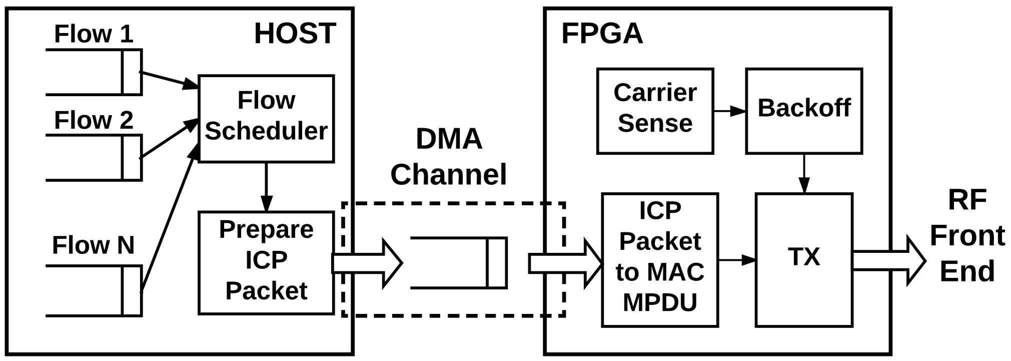

Figure 4.2: Packet transmission on PULS.

nel. All PHY layer processing required to transmit the packet is performed on the FPGA using the Intellectual Properties (IP) provided by the 802.11 AF. While the platform supports mechanisms for random backoff, we set backoffs to zero since our scheduling algorithms are completely centralized.

After the packet is transmitted, the Received Packet mechanism will determine whether an ACK was received or not. If an ACK was received, then ACK count will be incremented. Otherwise, the ACK timeout count will be incremented. Received packets are then sent back to the Host for processing, again via a DMA Channel. Concurrently, the Host polls the FPGA registers for transmitted packet count, ACK count, and ACK timeout count, and updates the deficits accordingly based on these counts.

4.5 Measuring Host-to-FPGA Interfacing Latency and Round-Trip Latency

The interfacing latency between hardware and software significantly affects the achievable link throughput of a software-defined wireless testbed. As discussed in Section 4.4, flow scheduling is repeated on a per-packet basis in a loop where scheduling-related function blocks are executed. The time between two consecutive loop executions depends on the round-trip transmission time of each packet plus the Host-to-FPGA interfacing latency. The interfacing latency between Host and FPGA needs to be low enough to support a packet deadline as low as 1ms. Meanwhile, *round-*

Algorithm 1 AP transmission procedure.

```
Initialize the state variables and the Host queues
While (station is ON)
        update Host queues and state variables
        schedule the flow with the highest priority
        send the scheduled packet to Prepare ICP Packet
        state of flow scheduler $\leftarrow$ LOCKED
        While (state of flow schedule is LOCKED)
            If (receive ACK response)
                state of flow scheduler $\leftarrow$ UNLOCKED
            end
        end
end
```

trip latency, which is defined as the elapsed time between a packet arrival at the Host and the ACK reception of the packet, inidcates the minimum achievable per-packet deadline of a wireless platform. To measure interfacing latency and round-trip latency, we devise a simple experiment by using the FPGA counter for the timestamps of the packet events.

The experiment can be summarized as follows:

1. Test packets of fixed payload size arrive at the Host periodically. The period is set to be large enough such that at each time there is only 1 test packet waiting for transmission in the Host queue. This completely eliminates the effect of queueing delay in the Host.

2. When a test packet arrives at the Host, it is given a timestamp denoted by t_h (read by the Host from an FPGA register) and then forwarded to the FPGA through a DMA Channel immediately.

3. When FPGA detects the new test packet, FPGA starts processing the ICP header and retrieves t_h from the header. Along with the current FPGA counter denoted by t_f, the Host-to-FPGA interfacing latency can be derived as $t_f - t_h$.

4. The packet is then transmitted. When the corresponding ACK is received, FPGA reads the current timestamp t_r and calculates the round-trip latency as $t_r - t_h$.

In the experiments, we measure both latency metrics for 50000 packets with a fixed interarrival time of 10ms. Figure 4.3 shows the empirical cumulative distribution function (CDF) of the Host-to-FPGA interfacing latency for packets with 1500 bytes payload at a data rate of 54Mbps. The mean interfacing latency is around 192 μs, and the 90, 95, and 99 percentiles are 233.4, 257.6, and 316.1 μs, respectively. Table 4.1 further summarizes the statistics of the interfacing latency for different data rates and payload sizes. We see that both the mean and the percentiles of the Host-to-FPGA latency are almost invariant, regardless of data rate and payload size. Therefore, PULS indeed exhibits low and predictable interfacing latency.

Next, Figure 4.4 shows the empirical CDF of round-trip latency, and Table 4.2 summarizes the statistics of round-trip latency for different data rates and payload sizes. Since round-trip latency consists of both transmission time and Host-to-FPGA interfacing latency, it varies with the physical data rate and the payload size. For the six test cases listed in Table 4.2, the maximum 99 percentile round-trip latency is 933.7 μs. Therefore, PULS is indeed able to guarantee a round-trip latency of less than 1ms with high probability even for large packet sizes and moderate physical data rates.

Table 4.1: Host-to-FPGA latency results

Data rate	Payload size	Host-to-FPGA latency (μs)			
(Mbps)	(bytes)	Mean	90%	95%	99%
54	500	185.2	227.6	251.5	301.8
54	1000	189.7	235	259	315
54	1500	192.2	233.4	257.6	316.1
24	500	187.8	228.4	252.7	305.7
24	1000	188.3	230.6	254.3	304
24	1500	189.2	231.5	255	304.6

As further verification of the system, we also plotted the receive throughput of clients against packet deadlines. This is to ensure that packets are not transmitted when they have expired. For this test, 20 packets arrive deterministically on the AP every 11ms. (Note, packets are only expired when the current time exceeds the deadlines. Hence even 0ms deadlines give some throughput.)

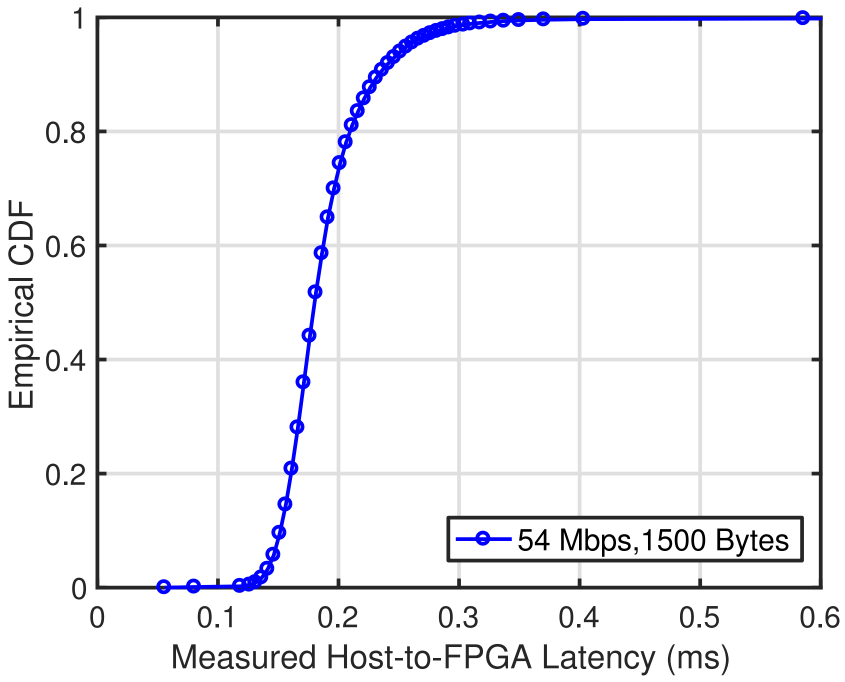

Figure 4.3: Empirical CDF of Host-to-FPGA latency for payload size = 1500 bytes and data rate = 54Mbps.

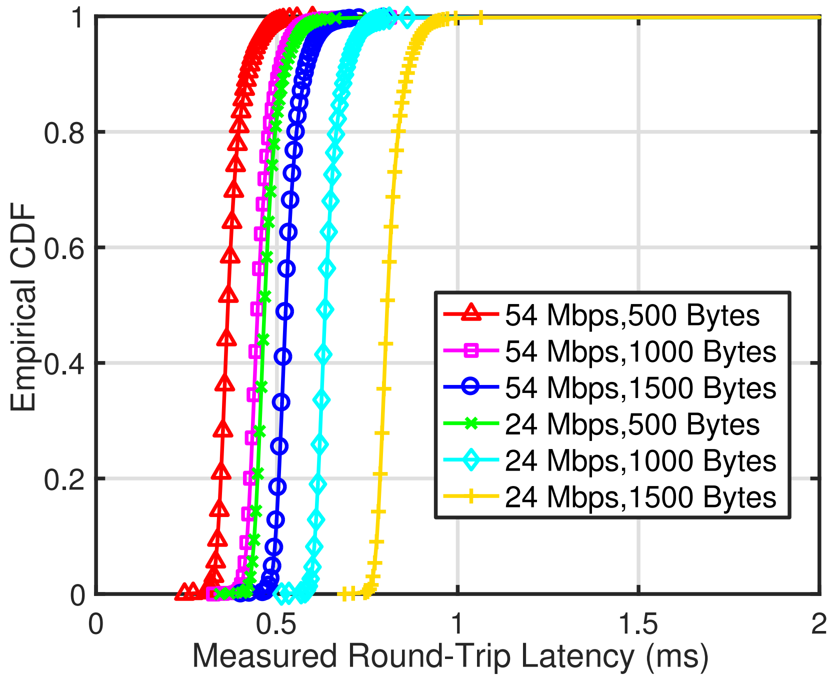

Figure 4.4: Empirical CDFs of round-trip latency for various data rates and payload sizes.

Table 4.2: Round-trip latency results

Data rate (Mbps)	Payload size (bytes)	Round-trip latency (μs)			
		Mean	90%	95%	99%
54	500	377.0	419.2	443.4	494.4
54	1000	459.9	505.1	529.2	585.0
54	1500	536.9	578.5	602.3	660.8
24	500	479.5	520.2	544.4	598.1
24	1000	646.6	688.9	712.7	762.8
24	1500	817.9	860.2	883.9	933.7

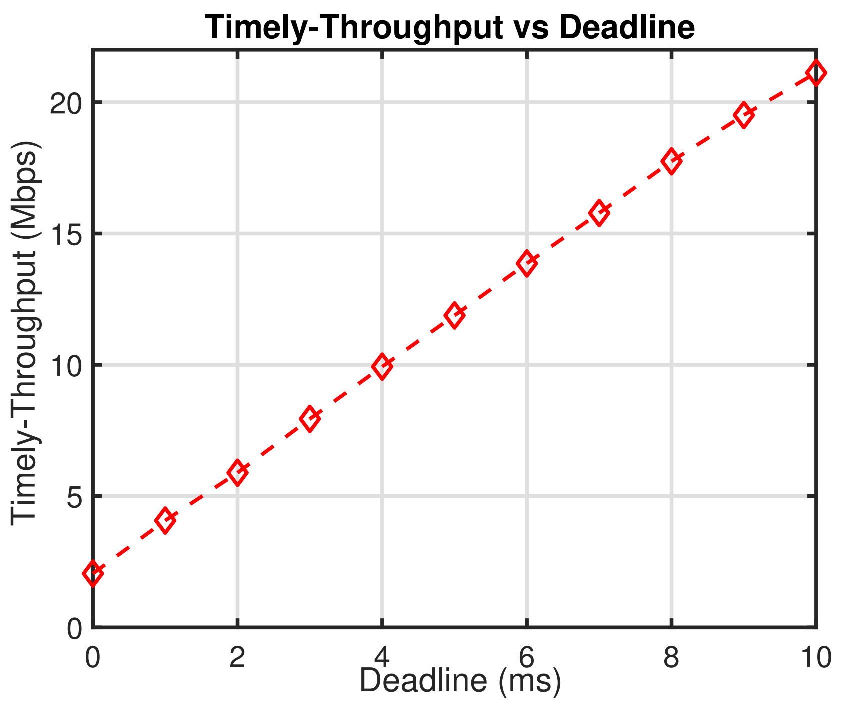

Figure 4.5: Throughput vs. Deadline for 20 packets generated every 11ms.

As we can see from Figure 4.5, the throughput increases linearly as deadline increases, which shows expired packets are indeed getting dropped by the AP.

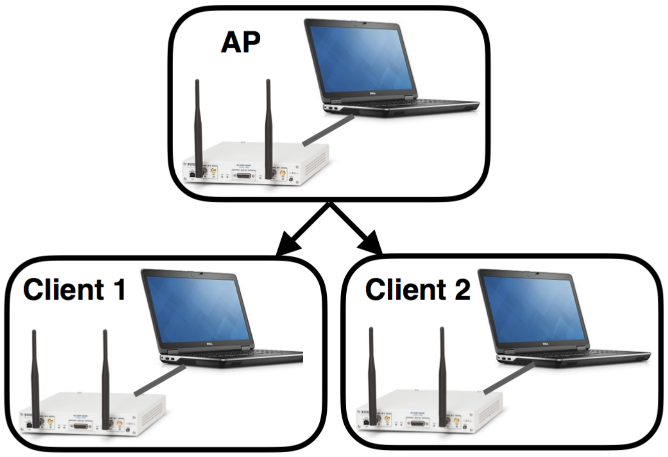

Figure 4.6: Experiment setup with host machines and USRP-2153Rs being used as wireless AP and clients.

4.6 Experimental Results

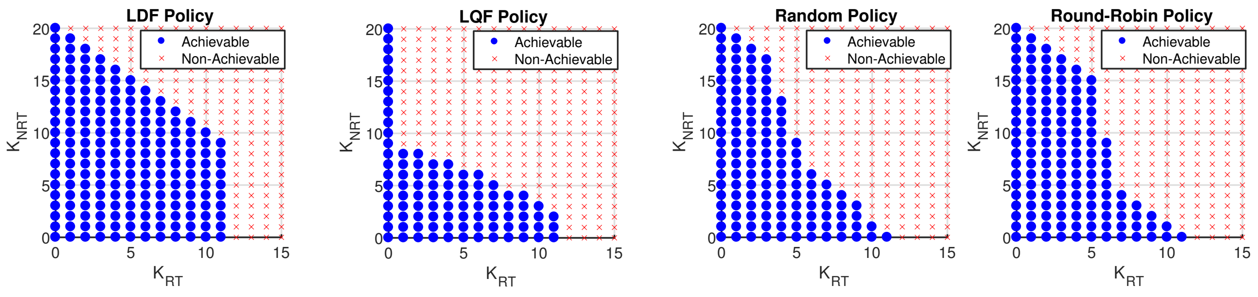

Figure 4.7: Capacity regions for the different policies with 5ms deadline and 0.99 delivery ratio.

We provide experimental results for a network with one AP and two downlink clients (as shown in Figure 4.6) in various scenarios. Each client is associated with one real-time flow with per-packet deadlines as well as one non-real-time flow without deadline constraints. Each of the nodes

(AP and clients) is a USRP-2153R that is connected to a Windows laptop acting as the Host machine. A specific scenario will have a certain arrival process, as well as predefined requirements for deadlines and delivery ratios for the real-time flows. These experiments were run using 1500B packets and IEEE 802.11a MCS 7 (54Mbps theoretical link data rate). We consider four scheduling policies: Largest Deficit First (LDF) [35], Longest Queue First (LQF), Round Robin (RR), and Random. Based on the definition of deficit introduced in [35], LDF schedules the real-time flows with the largest deficit and selects non-real-time flows with the largest queue length if the real-time flows are empty. Ties are broken randomly. LQF, as the name suggests, schedules the flow with the longest queue, with ties broken randomly. Random policy randomly picks a flow to schedule among non-empty queues. RR schedules flows in the following order: real-time flow for client 1, real-time flow for client 2, non-real-time flow for client 1, and then non-real-time flow for client 2. If any of the queues are empty, it will schedule the next queue. A point to note is that dropping expired packets is not done in most implementations today. To ensure a fair comparison, we decided to enable packet dropping for all policies, which improves the performance for all policies.

The results are presented in this section as follows. In section 4.6.1, we show the capacity regions of a single client under the different policies for one scenario to show the achievable regions of our system. Then, in section 4.6.2, we present the throughputs of the policies under different scenarios to compare the system performance. In these sections, packets are generated every 5ms and the number of packets generated are uniformly distributed from 0 to K_{RT} for real-time flows, and K_{NRT} for the non-real-time flows. We used a different arrival process in section 4.6.3 to have packets to arriving more frequently to show that ultra low latencies with 1ms deadlines are indeed achievable.

4.6.1 Capacity Regions

For our capacity region experiments, we defined achievable regions as regions where the system deficit (accumulated when packets are dropped) and queue length of the clients does not grow to infinity. Per-packet deadline is set to 5ms and the delivery ratio is set to 0.99, which means that the deficit increases by 0.99 every time a packet is dropped, and decreases by 0.01 when the packet is successfully delivered. In other words, if we require 99% of real-time packets to arrive in 5ms, we say the policy can achieve that for any incoming packet rate where the deficits do not grow to infinity. As we can see from Figure 4.7, the LDF policy has a bigger achievable region than LQF, Random and RR. Random and RR has similar capacity regions, but Random performs slightly better than RR when K_{RT} is higher, and worse when K_{RT} is smaller. This is because the amount of time waiting for service is bounded for RR, so for small number of real-time packets, they will always be served before the packets expired. On the other hand, when K_{RT} is high, RR will not be able to serve the packets in time, but random has a chance of serving the packets before they expire. LQF works better for higher value of K_{RT} than Random and RR, since it would schedule real-time flows more frequently, but suffers significantly when the value of K_{NRT} is higher than K_{RT}, since non-real-time flows will be schedule first. LDF always schedules real-time flows first, so the drop off in the capacity region is linear until K_{RT} = 11, after which the deficits start increasing to infinity for this delivery ratio and deadline. In this scenario, all policies can support up to a K_{RT} = 11, and serve up to 20 packets every 5ms.

4.6.2 Throughput Performance

Using the capacity region plots, we can know what kind of arrival rates our platform is capable of supporting. However, this did not tell us much about system performance is affected, in terms of the throughputs for real-time and non-real-time flows when operating in different scenarios. For brevity, in this section, throughputs for real-time flows will be referred to as timely-throughput, and throughputs for non-real-time flows will just be referred to as throughputs. So, the next thing we did was to run experiments for multiple clients, each with a real-time and non-real-time flow, under various scenarios to see how each protocol performed. We ran two symmetric scenarios (clients have the same traffic and requirements), and two asymmetric scenarios. In the first scenario, we had K_{RT} = 4 and K_{NRT} = 4 for both clients, with deadlines set to 3ms and delivery ratio set to 0.98. As we can see in Figure 4.8, while most protocols perform relatively well, LDF has a higher timely throughput and overall throughput for both clients. Next, we increased non-real-time traffic (K_{NRT} = 6) and decreased real-time traffic (K_{RT} = 3), and changed the requirements of the real-

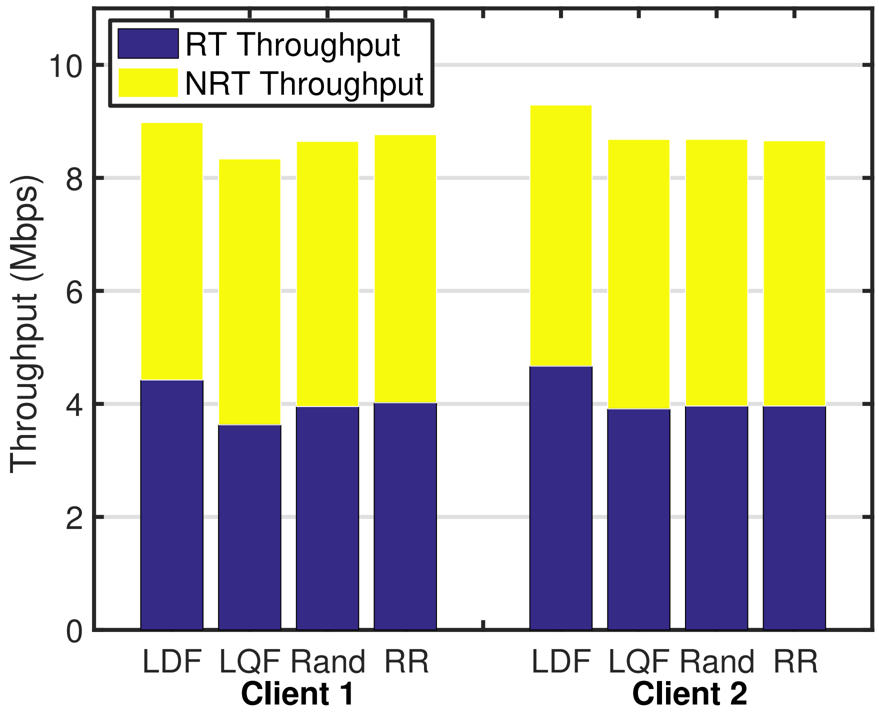

Figure 4.8: Throughputs for K_{RT}=4 and K_{NRT}=4.

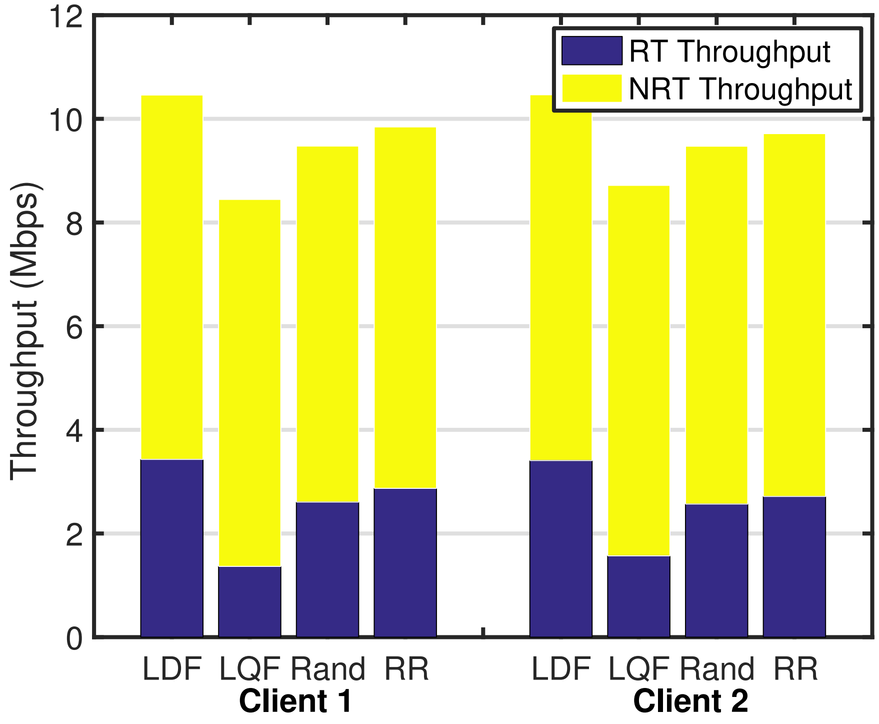

Figure 4.9: Throughputs for K_{RT}=3 and K_{NRT}=6.

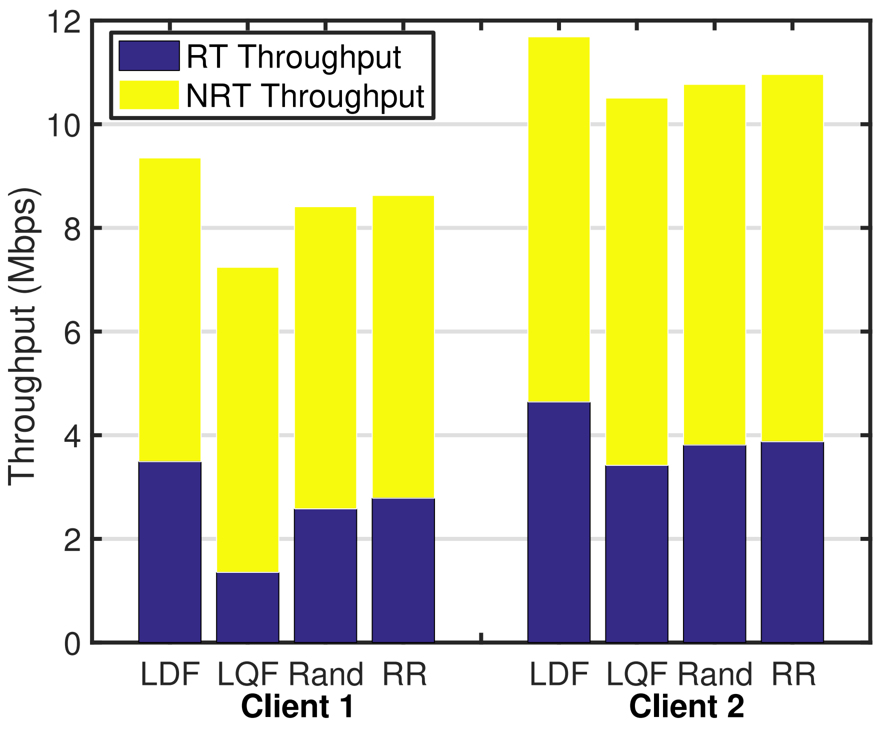

Figure 4.10: Throughputs for K_{RT}=3, K_{NRT}=5 for client 1, K_{RT1}=4, K_{NRT}=6 for client 2.

time flows to 2ms deadlines and 0.95 delivery ratios to see how the system performs with lower latencies. This could be an example of two clients downloading a large file while streaming videos. As we can see in Figure 4.9, LQF has the worst performance since it will tend to serve non-real-time traffic first, followed by Random and RR. LDF has the best performance in both of these symmetrical scenarios.

For the first asymmetrical scenario, we set K_{RT} = 3 and K_{NRT} = 5 for client 1, K_{RT} = 4 and K_{NRT} = 6 for client 2. For client 1, deadline of real-time packets was set to 2ms, with 0.97 delivery ratio, whereas client 2 has a deadline of 3ms with 0.98 delivery ratio i.e. client 1 has a real-time flow requirement where 97% of packets have to arrive in 2ms and client 2 has a real-time flow requirement where 98% of packets arrive in 3ms. As we can see in figure 4.10, LDF outperforms the other policies in terms of timely throughput and overall throughput. This difference is even more apparent when we 'increase' the asymmetry between the requirements of both clients. In our second asymmetrical scenario, we set K_{RT} = 7 and K_{NRT} = 4 for client 1, K_{RT} = 3 and K_{NRT} = 5

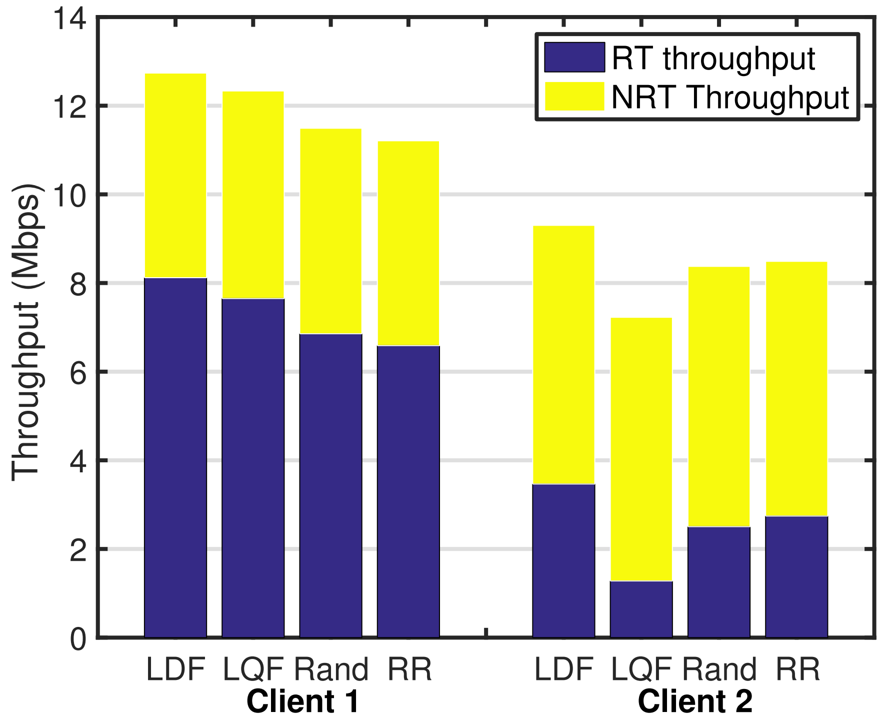

Figure 4.11: Throughputs for K_{RT}=7, K_{NRT}=4 for client 1, K_{RT1}=3, K_{NRT}=5 for client 2.

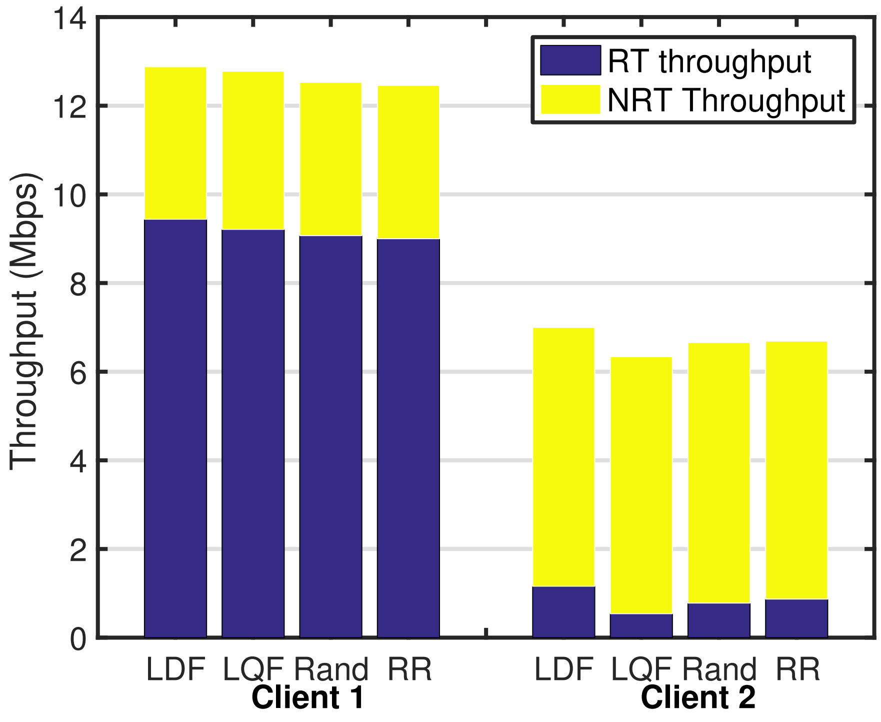

Figure 4.12: Throughputs for the second arrival process with P(K_{RT1}) = 0.8, P(K_{NRT1}) = 0.3, P(K_{RT2}) = 0.1, P(K_{NRT2}) = 0.5.

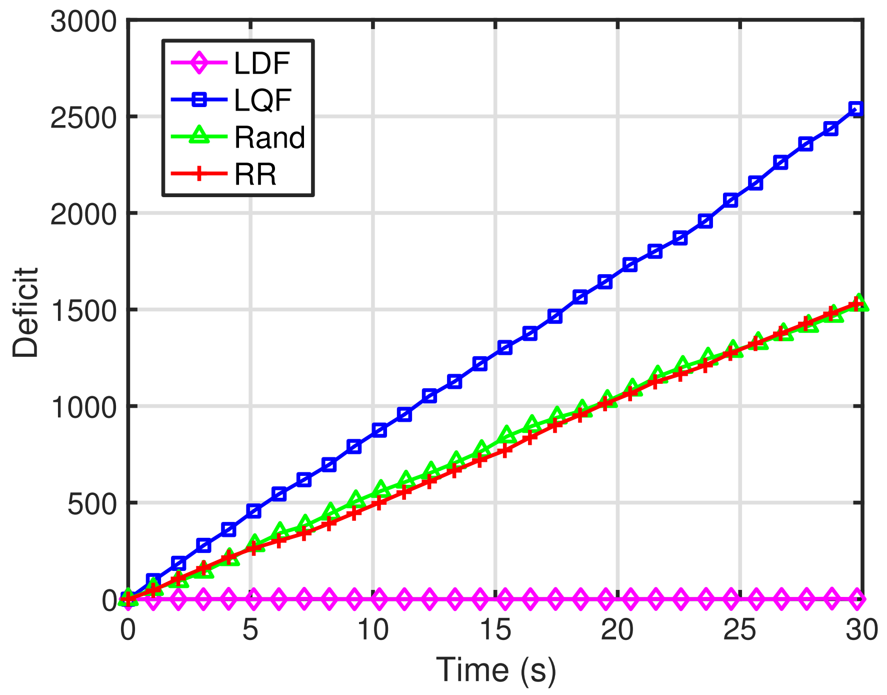

Figure 4.13: Deficit growth for the K_{RT}=4 and K_{NRT}=4 for both clients.

for client 2. Client 1's deadline was set to 5ms with 0.98 delivery ratio, and client 2's deadline was 2ms with 0.99 delivery ratio. This could be a case where client 1 is streaming a video where 5ms latency is tolerable but it generates a lot of traffic, and client 2 was running a control application that does not produce as many packets but has stricter latency and delivery ratio requirements. We see that Random and RR does not have a good timely throughput for client 1, while LQF does not perform well for client 2. LDF outperforms all policies in both clients since it smartly schedules client 1 and client 2's real-time flows so that both requirements are met, without sacrificing overall system throughput. Note, since we are operating on the boundaries of LDF, the deficits of LQF, Random and RR are growing to infinity as well. The deficit evolution from the first symmetrical scenario is shown in Figure 4.13.

Even though the throughput does not differ by much, the picture becomes very different when we examine the number of real-time packets that were dropped. Using the deficits of the clients, we calculated the loss ratios of the real-time flows for all the scenarios above. (Since the deficits of LDF stays at 0, the most we can conclude is the loss ratio is below 1 - delivery ratio). In Table

Table 4.3: Loss ratios of real-time flows

Arrival Rate	Deadline (ms)		Delivery Ratio		Policy	Loss Ratio (%)	
	C1	C2	C1	C2		C1	C2
K_{RT1} = 4, K_{NRT1} = 4, K_{RT2} = 4, K_{NRT2} = 4	3	3	0.98	0.98	LDF	<2	<2
					LQF	23.51	17.12
					Rand	15.17	15.62
					RR	15.07	16.03
K_{RT1} = 3, K_{NRT1} = 6, K_{RT2} = 3, K_{NRT2} = 6	2	2	0.95	0.95	LDF	<5	<5
					LQF	60.24	54.97
					Rand	26.48	27.22
					RR	19.96	21.27
K_{RT1} = 3, K_{NRT1} = 5, K_{RT2} = 4, K_{NRT2} = 6	2	3	0.97	0.98	LDF	<3	<2
					LQF	61.45	27.35
					Rand	27.2	18.07
					RR	20.09	17.13
K_{RT1} = 7, K_{NRT1} = 4, K_{RT2} = 3, K_{NRT2} = 5	5	2	0.98	0.99	LDF	<2	<1
					LQF	7.18	63.51
					Rand	17.01	29.46
					RR	19.14	21.42

4.3, we see that the loss ratios of LQF, Random and RR are much higher than LDF. The biggest difference is for the fourth scenario, when LDF can maintain less than 1% loss rate for Client 2, but LDF, Rand and RR experience loss rates of 63.5%, 29.5% and 21.4% respectively.

4.6.3 Changing the Arrival Process

In our final experiment, we modified the arrival process to further evaluate the policies under more stringent deadline requirements. Instead of packets being generated every 5ms, packets can now be generated every 1ms. However, instead of generating it uniformly from 0 to K_{RT} (or K_{NRT}), 1 packet is generated with some probability, which is specified for each flow by the client. Let P(K_{RTi}) and P(K_{NRTi}) be the probability of generating a packet for client i's real-time flow and non-real-time flow respectively. As in the previous experiment, the contrasts are most stark in the asymmetrical case, which is presented in the following scenario. Client 1 has P(K_{RT1}) = 0.8, P(K_{NRT2}) = 0.3 with the real-time flow requirement of 2ms and 0.99 delivery ratio. This could be a user streaming a video while downloading some files in the background. Client 2 on the other hand has P(K_{RT2}) = 0.1, P(K_{NRT2}) = 0.5 with a deadline of 0ms and 0.99 delivery ratio.

Note that packets are only expired when the current time exceeds the deadlines, and hence even 0ms deadlines give some throughput. This could be a mission-critical application which does not generate much traffic but requires control packets to be delivered within 1ms. As we can see in Figure 4.12, client 1's timely throughput is lower for Random and RR and LQF causes client 2's timely throughput to suffer. However, LDF is able to support both clients and give good timely throughput and throughputs as well. Furthermore, this also shows that the system is capable of delivering packets under 1ms.

These results show that our system is indeed capable of supporting experimentation of policies for ultra-low latency applications with various packet arrival patterns and deadline requirements.

4.7 Conclusion

Applications today have increasingly stringent requirements, especially in terms of latency and throughput. This presents next generation networks with one of its critical challenges: providing some measure of guarantee for applications with strict latency and throughput requirements. There exist theoretical frameworks to develop protocols that are able to do this, but there is still a gap between theory and implementation of these protocols. We aim to bridge this gap by developing PULS, which we have shown to be capable of supporting per-packet scheduling for downlink with latencies on the order of 1ms, with realistic system throughputs. PULS was developed for with reprogrammability in mind, so new scheduling policies can be more easily implemented and experimented on it.

This chapter, along with the previous two, dealt with the design of scheduling policies in a centralized manner. In the next chapter, we propose a distributed approach for enhancing application performance by intelligent sharing among end user devices interested in a common objective.

5. REALTIME STREAMING WITH GUARANTEED QoS OVER WIRELESS D2D NETWORKS*

5.1 Introduction

There has recently been a steep increase in the use of smart, handheld devices for content consumption. These devices, such as smart phones and tablets, are equipped with multiple orthogonal wireless communication interfaces. Interfaces include expensive (both dollar-cost and energy) long-range base-station-to-device (B2D) interfaces (*e.g.* 3G/4G), as well as low-cost short-range interfaces like WiFi. More recently, the use of short-range interfaces such as WiFi-Direct and FlashLinQ for device-to-device (D2D) communication is starting to make an appearance. Simultaneously, there has been an explosion in available content, and it is expected that streaming of live events will play a big part in future demand [57].

In this chapter, we focus on live-streaming of content to multiple co-located devices, as shown in Figure 5.1. Here, data is generated in realtime by a server, and must be delivered to all the devices quickly in order to maintain the "live" aspect of the event. Each device is both capable of unicast communication via a B2D interface, and broadcast D2D communication with peer devices. Devices desire to minimize the usage of their B2D interfaces to reduce cost, while maintaining synchronous reception and playout of content. While it might be possible for a cellular base station to broadcast live events to multiple handsets, such content would be restricted to a few selected channels, and only available to subscribers of a single provider. Utilizing both interfaces enables users to pick any event of interest, and "stitch together" their B2D capacities regardless of provider support. Apart from content consumption in a social setting, such a scheme would also be valuable in an emergency response or battlefield setting.

In our model, content is generated in realtime in the form of *blocks*, with one block corresponding to a playout time called a *frame*. In other words, each device must receive one block of

*Republished with permission of ACM, from Realtime Steaming with Guaranteed QoS over Wireless D2D Networks, Naved Abedini, Swetha Sampath, Rajarshi Bhattacharyya, Suman Paul, Srinivas Shakkottai, MobiHoc, 2013; permission conveyed through Copyright Clearance Center, Inc.

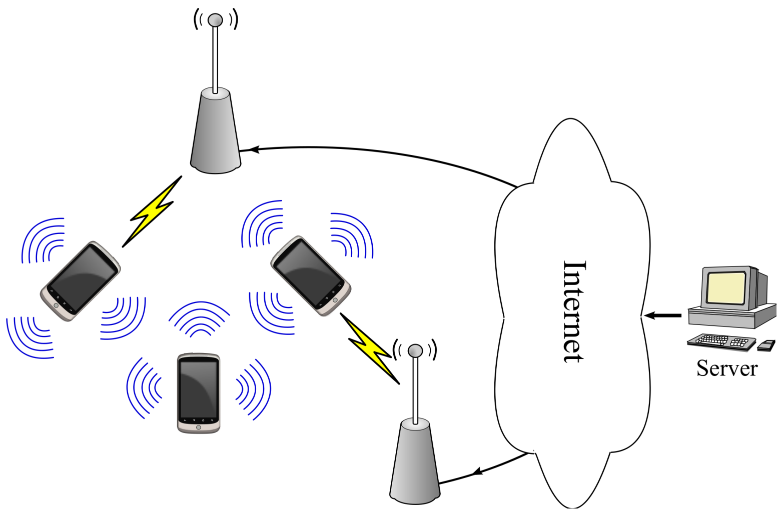

Figure 5.1: Hybrid wireless network.

data within one frame of time for smooth playout. The blocks are divided into *chunks* for sharing via different interfaces. Coordination of chunk identities across all devices and all base-stations is near impossible, so a coding solution is needed for transmitting chunks. Hence, the server performs random linear coding [58, 59] across each block, and unicasts the resulting coded chunks to the devices via the Internet and through the B2D connections. Thus, each coded chunk can be thought of as an element in a vector space with the scalars in a finite field. The information at each device can then be represented by a matrix that contains all the vectors that it has received thus far, and the block of information can be decoded when this matrix is of full rank. Both the Internet and B2D links are lossy, but since the chunks are coded, there is no need for feedback and the server simply generates a new chunk for each transmission. The devices receive coded chunks from the server, and in turn transmit these chunks to each other over a potentially lossy D2D broadcast channel, again without feedback. If each device receives enough chunks to decode the block, it can play it, else it has to skip that block.

We illustrate the timing diagram for block transmission in Figure 5.2. Here, we divide time into *slots*, and each frame consists of T slots. Each block is divided into N chunks, and coding is performed over these chunks. We require all devices to synchronously play out the k^{th} block during the k^{th} frame. Since our application is that of *live* streaming, the k^{th} block would be available at the server to disseminate to devices only within some short interval before the k^{th} frame. Hence, we assume that the k^{th} block is available at the server not before frame $k-2$. Thus, devices receive chunks of block k using their individual B2D interfaces during frame $k-2$. These chunks may be disseminated over the broadcast D2D network during frame $k-1$, and must be played out in frame k. Device i is able to play out block k, only if it has received enough degrees of freedom to decode the N chunks of the k^{th} block by the beginning of frame k. Otherwise, i will be idle during this frame.

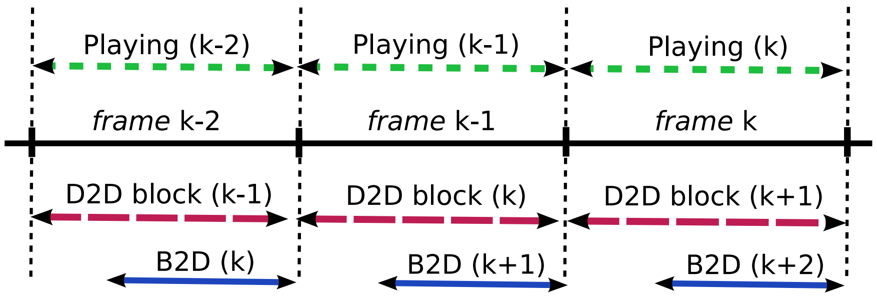

Figure 5.2: Sequence of transmissions over B2D and D2D networks. Each block must be delivered within two frames of its generation or lost.

We use a recent model of service quality for realtime wireless applications in which each station's QoS is parametrized by a *delivery-ratio*, which is the average ratio of blocks desired to the blocks generated [35]. For example, a delivery ratio of 90% would mean that 10% of the blocks can be skipped without noticeable impairment (which depends on the video coding used). We desire an algorithm that minimizes the usage of the expensive B2D connections, while maintaining an acceptable quality of service for each device. We note that the scenario is different from traditional peer-to-peer sharing in a wired network, wherein streaming with full chunk coordination using tree construction is the norm. In our situation, each device is independently connected to a base-station (perhaps using different service providers), and there is no coordination across base stations.

The broadcast D2D medium is shared across the devices, and interference across the devices is a limiting factor.

The objectives of this work are three-fold. First, we would like to systematically design provably-cost-optimal chunk exchange algorithms that would minimize the usage of expensive B2D transmissions, while ensuring that quality constraints for live streaming can be met. Second, we would like to conduct performance analysis of our schemes using simulations to understand how different parameters affect the algorithms. Third, we desire to implement the developed algorithms on Android smart phones in order to observe real-world behavior. In what follows, we will describe these objectives in greater detail, and present our results in attaining each.

5.2 Related Work

The problem of exchanging random linear coded information while using the minimum number of transmissions over a broadcast D2D network (starting from some initial condition) was studied in [60, 61, 62]. There is also recent work on quick file transfer to a set of peer devices using a hybrid network, while minimizing the amount of B2D usage [63]. All consider a reliable model for the D2D links, and their common objective is to be able to decode a single block of information at the end. Unlike these papers, our objective and QoS metric—minimum cost timely synchronization of a live stream—is quite different.

There is a rich body of literature in the area of P2P multimedia streaming applications. Lava [64] and UUSee [65] are among the systems built based on the idea of network coding. While all the above models only consider unicast (wired) transmissions among peers, our D2D framework is one of the few models in this context that allows having broadcast transmissions over the D2D network.

Closest to our multimedia streaming model is [66] that investigates the problem of managing multiple interfaces for cooperatively sharing content over a D2D network. Unlike our block-by-block delay sensitive model for realtime streaming, they try to maximize a utility function of the average information flow rate achieved by the peers, which is relevant to elastic traffic such as file transfer or streaming of stored content.

5.3 Organization and Main Results

In Section 5.4, we present the live streaming model and the QoS metric. In our model, we assume that time is slotted, and in each time slot there can be one unicast B2D transmission to each device, and one broadcast D2D transmission over all devices. The probabilities of success of each are different, and these are assumed to be known. There is no feedback in the system, consistent with the idea of using UDP-based communications. Our frame timing structure allows us to study the B2D and the D2D communications as two sub-problems, namely, (i) B2D stopping time: how long should each device use its B2D channel for each block? and (ii) D2D broadcast scheduling: which device should broadcast at each time?

We consider the problem of D2D scheduling in Section 5.5. Since the B2D communications result in a random initialization of coded chunks with each peer device at the beginning of each frame, our first objective is to find a set of necessary and sufficient conditions for achievability of a given QoS. We then use ideas from queueing theory and the Foster-Lyapunov stability criterion to find an optimal D2D scheme for sufficiently large field sizes over which we perform coding. We next study the special case of fully reliable D2D broadcast in Section 5.6. The optimal D2D scheduling algorithm in this case has a simple and intuitive form, which allows for easy implementation, as well as development of good heuristics.

Our next problem is to answer the question of how to determine the stopping time for B2D, which we tackle in Section 5.7. This problem is hard to solve explicitly, given the form of the necessary conditions for QoS achivability. However, it has to be solved only once for any particular set of users, and we provide a general framework to find the optimal B2D usage times for any given cost structure.

We assumed in the above sections that coding is performed using fields of sufficiently large size as to ignore the performance degradation effects of receiving linearly dependent (and hence useless) chunks. In Section 5.8, we show that the degradation decreases inversely proportional to the field size, and is independent of other parameters like the number of devices or the number of chunks in a block, which is a useful result as the system scales.

We validate the proposed algorithms through simulations in Section 5.9, comparing with some intuitive heuristic algorithms. Finally, we discuss implementation ideas for music streaming in Section 5.10. Our system is more like an emulation for B2D, as we did not yet have 3G service on the phones, and hence initialized each device with coded chunks periodically. However, the D2D part was achieved accurately through WiFi broadcast. Finally, we conclude in Section 5.11.

5.4 System Model

We have M co-located peer devices in our system, denoted by $i \in \{1, \ldots, M\}$, all interested in synchronously receiving the same stream of data. The data source generates the stream in the form of a sequence of blocks. Each block is further divided into N chunks for transmission. We use random linear network coding over the chunks of each block, which implies that each coded chunk (a degree of freedom) is now a random linear combination (with coefficients in finite field F_q of size q) of the N original chunks in the corresponding block, and can be represented by a vector in F_q^N.

Time is divided into slots, and at each time slot τ, each device can receive up to one chunk using its B2D interface, and one using its D2D interface. We assume that the two interfaces are orthogonal. Also, both B2D unicasts and D2D broadcasts are made in a connectionless fashion with no feedback. The probabilities of success when using each interface are different, and we can use these probabilities to reflect the relative throughputs of the two interfaces.

Thus, each device has an expensive but unreliable B2D unicast channel to a base-station, whose success probability depends on the loss probability over the Internet as well as the condition of the B2D channel. For each device i, we model the chunk reception event for the B2D interface by a Bernoulli random variable with parameter β_i, independent of the other devices. Each device also has a low-cost D2D broadcast interface, and only one device (denoted by $u[\tau] \in \{1, \ldots, M\}$) can broadcast over the D2D network at each time τ. Losses over the D2D channel could happen due to collisions that occur as an overhead of distributed scheduling. Hence, we assume that a D2D broadcast is either received by all devices with a probability equal to δ, or lost. It is straightforward to extend our results and framework to a more general D2D model in which only a subset of devices receive each broadcast chunk, but at the cost of greater algorithmic complexity. Wireless devices could apply channel estimation techniques to determine their success probabilities β_i and δ, and we will develop algorithms assuming that these values are known. We will also study a system (similar to the ones investigated in [60, 61, 62]) in which the D2D broadcast is assumed to be completely reliable.

Since each D2D broadcast is either received by all devices or none, there is no need to rebroadcast any information received via D2D. Following this observation, it is straightforward to verify that the order of D2D transmissions has no impact on the performance. Thus, we only need to keep track of the number of chunks transmitted and received over the interfaces during a frame in order to determine the state of the system. We denote the total number of coded chunks of block k delivered to device i via the B2D network during frame $k-2$ using $b_i^{(k)}$. Also, $t_i^{(k)}$ and $r_i^{(k)}$ are used to denote, respectively, the total number of transmitted and received chunks of block k by device i via D2D during frame $k-1$ (*i.e.,* before frame k, which is the block k's play out time). Note that according to the model

$$\textstyle\sum_i t_i^{(k)} \leq T, \tag{5.1}$$

i.e., only one device can broadcast over the D2D network at each slot, and hence at most T transmissions can occur during frame $k-1$.

Let $\hat{n}_i^{(k)}$ denote the total number of coded chunks of block k possessed by device i at the beginning of frame k,

$$\hat{n}_i^{(k)} = b_i^{(k)} + r_i^{(k)}.$$

Hence, device i has a set of $\hat{n}_i^{(k)}$ vectors of coefficients corresponding to the coded chunks that it has received. We denote by $n_i^{(k)}$ (called the k^{th} *rank* of device i) the dimension of the subspace spanned by these vectors. In order to decode the original N chunks of block k at the beginning of frame k, device i must have $n_i^{(k)} = N$. Otherwise, i skips the block and will be idle during this frame. Let $R_i[k] \in \{0,1\}$ denote the success or failure of the k^{th} block w.r.t i, *i.e.,* $R_i[k] = 0$ means device i is idle during frame k, and $R_i[k] = 1$ denotes the event of successful decoding of block k. Clearly, $R_i[k] = 0$, if $n_i^{(k)} < N$.

Each device i has a delivery ratio $\eta_i \in (0, 1]$, which is the minimum acceptable long-run average number of frames device i must playout. Hence, we require

$$\eta_i \leq \lim_{K\to\infty} \frac{1}{K} \sum_{k=1}^{K} \mathbb{E}[R_i[k]]. \tag{5.2}$$

The objective of this chapter is to find a scheme to coordinate both interfaces that would satisfy the delivery ratio requirement of all devices at the lowest cost of using B2D transmissions. Since the B2D and D2D transmissions corresponding to a particular block k occur during the disjoint frames $k-2$ and $k-1$, respectively, we will explore schemes that consist of two parts, namely:

1. **B2D Stopping Time:** We use a fixed number of transmissions on the B2D channel for each block and for each device. Hence, for each block k, device i receives some random number $\mathrm{b}_i^{(k)}$ of coded chunks via B2D at the beginning of frame $k-1$. By assumption, the values of $\mathrm{b}_i^{(k)}$ are independent over devices. Also, since we fix the B2D usage in each frame, $\mathrm{b}_i^{(k)}$ are independent and identically distributed over blocks k for each device i. We need to determine the minimum stopping time such that delivery requirements can be met.

2. **D2D Scheduling Algorithm:** Given the B2D stopping time, each device receives chunks at the beginning of each frame in an *i.i.d* fashion according to an arrival process denoted by b_i. We must determine a scheduling algorithm that can achieve the delivery requirements if at all it is possible to do so, given the D2D broadcast channel reliability.

In what follows, we will first solve the problem of D2D scheduling, and will then show how to select the B2D stopping time. We will do so under different assumptions on channel reliability and coding schemes.

5.5 D2D Scheduling Algorithm

Under the assumption of *i.i.d* B2D arrivals, we first need to determine whether the desired QoS metric $(\eta_1, ..., \eta_M)$ is achievable for the given arrival process $(\mathbf{b}_1, ..., \mathbf{b}_M)$. For large field sizes, distinct randomly coded chunks are linearly independent with high probability. In what follows, we assume that the field size q is large enough that we can ignore the effect of its finiteness on the performance of the linear coding. We will consider the impact of finite field size on performance in Section 5.8.

Definition 1. *We say the QoS $(\eta_1, ..., \eta_M)$ is achievable, if there exists a feasible policy to coordinate the D2D transmissions such that, on average each device i successfully receives η_i fraction of the blocks before their deadlines.*

Since the B2D arrivals are assumed to be *i.i.d.* over frames, the achievability of the QoS metric can be evaluated based on the existence of a randomized stationary D2D policy as follows:[1]

Lemma 1. *The QoS $(\eta_1, ..., \eta_M)$ is (strictly) achievable if and only if there exists a D2D policy $\mathbf{P}^*$ such that,*

for each block k, given the B2D arrivals $b^{(k)} = (b_1^{(k)}, ..., b_M^{(k)})$, $\mathbf{P}^*$ chooses a feasible D2D schedule $t^{(k)} = (t_1^{(k)}, ..., t_M^{(k)})$ (satisfying (5.1)) with probability $\mathbb{P}\left(t^{(k)} | b^{(k)}\right)$, such that for each device i,

$$\mathbb{E}_{\mathbf{b}}\left[\sum_{t^{(k)}} \mathbb{E}\left[R_i[k] \mid t^{(k)}, b^{(k)}\right] \mathbb{P}\left(t^{(k)} | b^{(k)}\right)\right] > \eta_i \tag{5.3}$$

where $\mathbb{E}_{\mathbf{b}}[.]$ is expectation with respect to the B2D arrival processes.

5.5.1 Optimal D2D Scheme

In order to keep track of quality of service, each device maintains a so-called deficit queue. The length of this queue $d_i[k]$, for device i at frame k, follows the dynamic below

$$d_i[k] = d_i[k-1] + \eta_i - R_i[k]. \tag{5.4}$$

[1]It is easy to verify this characterization of achievability, which is used in much recent work. For instance, a detailed proof can be found in [67].

Recall that $R_i[k] = 0$ means device i is not successful in receiving the block k before its deadline, and hence is idle in frame k. In this case, the deficit value of this device increases by an amount equal to its delivery ratio η_i. Otherwise, $R_i[k] = 1$ and the deficit value decreases by $1 - \eta_i$. Therefore, the deficit queue can be thought of as a counter that captures the accumulated "unhappiness" of a device experienced thus far,

$$d_i[k] = k\eta_i - \sum_{l=1}^{k} R_i[l].$$

The evolution of these deficit queues can be understood by using a Markov chain $\mathcal{D}$, whose state at each step k is $([d_1[k]]^+, ..., [d_M[k]]^+)$, where $[a]^+ = \max\{a, 0\}$. If our D2D algorithm is such that $\mathcal{D}$ is stable (positive recurrent), then $\mathbb{E}[[d_i[k]]^+] < \infty$ for all k, and we will have

$$\lim_{K\to\infty} \frac{1}{K}\mathbb{E}[d_i[k]] \leq \lim_{K\to\infty} \frac{1}{K}\mathbb{E}[[d_i[k]]^+] = 0.$$

Hence, $\eta_i \leq \lim_{K\to\infty} \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}[R_i[k]]$, which implies that the QoS requirement of device i is satisfied. We next define a D2D scheme in Algorithm 2, whose optimality is shown in Theorem 2.

Algorithm 2 Optimal D2D scheme (unreliable broadcast)

At the beginning of each frame $k-1$: Given the B2D arrivals $(b_1^{(k)}, ..., b_M^{(k)})$ and the deficit values $([d_1[k-1]]^+, ..., [d_M[k-1]]^+) = (d_1, ..., d_M)$, solve the following maximization problem to find the optimal number of transmissions $(t^{(k)})^* = ((t^{(k)})_1^*, ..., (t^{(k)})_M^*)$:

$$\begin{array}{c} \max \sum_i d_i \mathbb{P}(\sum_{j\neq i} \min(\hat{t}_j^{(k)}, b_j^{(k)}) \geq N - b_i^{(k)}) \\ \text{s.t.} \\ \hat{t}_j^{(k)} = Bin(\delta, t_j^{(k)}), \quad \forall j \\ \sum_j t_j^{(k)} \leq T \end{array} \tag{5.5}$$

Devices take turns (in any arbitrary order) to broadcast over the D2D network: Device i first broadcasts $\min(b_i^{(k)}, (t^{(k)})_i^*)$ of its initial coded chunks received from the B2D network. For the remaining $\max((t^{(k)})_i^* - b_i^{(k)}, 0)$ transmissions, device i randomly combines the initial $b_i^{(k)}$ B2D chunks.

Theorem 2. *The D2D scheme in Algorithm 2 is throughput optimal,* i.e., *it can satisfy* all *achievable QoS metrics* $(\eta_1, ..., \eta_M)$.

Proof. In this proof, we will use the Lyapunov criterion [68] to show the stability of Markov chain $\mathcal{D}$. Consider the candidate Lyapunov function $V[k] = \frac{1}{2}\sum_i([d_i[k]]^+)^2$. We will show that for any achievable QoS, the proposed D2D algorithm results in an expected drift $\Delta V[k] =$

$$\mathbb{E}\left[V[k] - V[k-1] \mid \text{state of the system at frame } k-1\right],$$

which is negative except in a finite subset of the state space. Hence, the Lyapunov Theorem implies that the Markov chain $\mathcal{D}$ is stable. Now, we present the details of the proof.

$$\begin{aligned}
&\Delta V[k] = \mathbb{E}\left[V[k] - V[k-1]\middle|[d_i[k-1]]^+ = d_i : \forall i\right] \\
&= \tfrac{1}{2}\mathbb{E}\left[\sum_i \left([d_i[k-1] + \eta_i - R_i[k]]^+\right)^2 \middle| [d_i[k-1]]^+\right] \\
&-\tfrac{1}{2}\sum_i (d_i)^2 \overset{(a)}{\leq} \tfrac{1}{2}\mathbb{E}\left[\sum_i (d_i + \eta_i - R_i[k])^2 - (d_i)^2\right] \\
&= \mathbb{E}\left[\sum_i d_i\left(\eta_i - R_i[k]\right)\right] + \tfrac{1}{2}\mathbb{E}\left[\sum_i (\eta_i - R_i[k])^2\right] \\
&\overset{(b)}{\leq} M/2 + \sum_i d_i\eta_i - \mathbb{E}\left[\sum_i d_i R_i[k]\right],
\end{aligned}$$

where (a) follows from $([X+Y]^+)^2 \leq ([X]^+ + Y)^2$, and (b) holds since $(\eta_i - R_i[k])^2 \leq \max((\eta_i)^2, (1-\eta_i)^2) \leq 1$.

In order to get a negative drift (except in a finite subset), we minimize the above upperbound. Hence, at the beginning of each frame $k-1$, for given deficit values $([d_1[k-1]]^+, ..., [d_M[k-1]]^+) = (d_1, ..., d_M)$ and any realization of the B2D arrivals $(b_1^{(k)}, ..., b_M^{(k)})$, we solve the following

$$\max_{t^{(k)}} \sum_i d_i \mathbb{E}\left[R_i[k] \mid t^{(k)}, b^{(k)}\right], \tag{5.6}$$

to find the optimal schedule $(t^{(k)})^*$. Note that policy $\mathbf{P}^*$ (in Lemma 1) randomly chooses a feasible

$t^{(k)}$ according to some distribution $\mathbb{P}\left(t^{(k)}|b^{(k)}\right)$, hence one can easily verify that

$$\begin{aligned}\sum_i d_i \mathbb{E}\left[R_i[k] \mid (t^{(k)})^*, b^{(k)}\right] \geq \\ \sum_{t^{(k)}} \sum_i d_i \mathbb{E}\left[R_i[k] \mid t^{(k)}, b^{(k)}\right] \mathbb{P}\left(t^{(k)}|b^{(k)}\right).\end{aligned} \tag{5.7}$$

Taking expectation on both sides of the above inequality with respect to the arrival processes results in

$$\begin{aligned}&\sum_i d_i \mathbb{E}\left[R_i[k] \mid (t^{(k)})^*\right] \\ &\geq \sum_i d_i \mathbb{E}\left[\sum_{t^{(k)}} \left[R_i[k] \mid t^{(k)}, b^{(k)}\right] \mathbb{P}\left(t^{(k)}|b^{(k)}\right)\right] \\ &\overset{(c)}{>} \sum_i d_i \eta_i \geq \sum_i d_i(\eta_i + \epsilon)\end{aligned} \tag{5.8}$$

for some small enough $\epsilon > 0$, where (c) follows from (5.3). By considering (5.8) in inequality (b), we conclude that using schedule $(t^{(k)})^*$ will result in

$$\Delta V[k] \leq \frac{M}{2} - \epsilon \sum_i d_i,$$

which is negative for large enough d_i values, and hence can stabilize the deficit queues.

We have just shown that we can satisfy any achievable QoS metric by using the schedule obtained by solving (5.6) for each frame. In what follows, we will show (5.6) is equivalent to the optimization problem (5.5) that appears in Algorithm 2.

Recall that $R_i[k] = 1_{\{n_i^{(k)}=N\}}$, where the indicator variable $1_{\{A\}}$ is equal to 1 if A is true, and 0 otherwise. Hence, $\mathbb{E}\left[R_i[k]\right] = \mathbb{P}\left(n_i^{(k)} = N\right)$. Suppose device j broadcasts $t_j^{(k)}$ chunks generated from random linear combinations of its initial $b_j^{(k)}$ chunks. Since each transmission is successful with probability δ, the number of successful transmissions by device j is distributed as a Binomial random variable $\hat{t}_j^{(k)} = Bin(\delta, t_j^{(k)})$ with parameters δ and $t_j^{(k)}$.

Now, we have assumed that the random linear coding is performed over a field of sufficiently large size q such that N distinct randomly coded chunks are linearly independent. Consequently, each successful transmission increases the rank of the receiving devices by one. Thus, the transmissions by device j introduces $\min(\hat{t}_j^{(k)}, b_j^{(k)})$ new degrees of freedom (DoF) at the other devices. Also, device i is full-rank (*i.e.,* $n_i^{(k)} = N$) if and only if it has received at least $N - b_i^{(k)}$ new DoFs from other devices, that is $b_i^{(k)} + \sum_{j \neq i} \min(\hat{t}_j^{(k)}, b_j^{(k)}) \geq N$. Therefore, we have

$$\begin{aligned} &\mathbb{E}\left[R_i[k]\right] = \mathbb{P}\left(n_i^{(k)} = N\right) \\ &= \mathbb{P}\left(b_i^{(k)} + \textstyle\sum_{j \neq i} \min(\hat{t}_j^{(k)}, b_j^{(k)}) \geq N\right). \end{aligned} \tag{5.9}$$

Substituting (5.9) in (5.6) results in the optimization (5.5). □

We have just shown the throughput optimality of Algorithm 2 for the unreliable D2D case. However, we note that in practice we need a central entity with significant computational capabilities to implement this optimal algorithm, since it takes a combinatorial form. In the following section, we will consider the special case when D2D broadcast is completely reliable. We will show that the optimal algorithm under this case possesses an intuitively appealing form that lends itself to easy implementation. We also will develop a heuristic modification of this algorithm when the D2D broadcast is not reliable, which does not have the complexity of the optimal algorithm.

5.6 D2D Algorithm Under Reliable Broadcast

In this section, we assume that D2D broadcasts are always successfully received by all intended recipients. Of course, the constraint that only one device can broadcast at a time still applies. As before, we first model the B2D arrivals as *i.i.d.* random variables $\mathbf{b}_i$ for each device i, and focus on scheduling the D2D transmissions. Since broadcasts are deterministically successful, $\sum_{j \neq i} t_j^{(k)} = r_i^{(k)}$. In other words, the total number of D2D chunks received by device i is equal to the number of transmissions performed by all other devices. Also, each device i can transmit at most $b_i^{(k)}$ times during frame $k-1$, since any further transmissions by i will not add to other devices' information, *i.e.,*

$$t_i^{(k)} \leq b_i^{(k)}. \tag{5.10}$$

Consequently, it is easy to see that $n_i^{(k)} = \min\{N, \hat{n}_i^{(k)}\}$ holds for large field size q, *i.e.,* device i will obtain full-rank if it receives at least a total of N coded chunks from B2D and D2D interfaces.

Further, since the chunks received from the B2D interfaces are initially randomly coded, combining them further with random coefficients cannot improve performance. Hence, at time τ, the device chosen to transmit, $u[\tau]$, can simply broadcast any of the chunks received via its B2D interface, which it has not transmitted yet. We are primarily interested in the case where $N > T$, because otherwise there are enough number of time slots in each frame for devices to broadcast all N degrees of freedom and hence the optimal D2D scheme becomes trivial.

5.6.1 Achievability of QoS Metric

In Section 5.5, Lemma 1 implicitly characterizes the achievability conditions of a given QoS requirement $(\eta_1, ..., \eta_M)$, and can be applied for the reliable D2D case as well. However, we will see below that the QoS achievability condition can be determined explicitly based on a set of necessary and sufficient conditions for this case.

Devices have T slots in each frame to exchange the received B2D chunks. Hence, each device i can potentially recover block k, only if (i) it has received enough B2D chunks initially (i.e, $b_i^{(k)} \geq N - T$) and (ii) the whole system is full-rank at the beginning of the frame (i.e, $\sum_j b_j^{(k)} \geq N$). Therefore, for each device i we have,

$$R_i[k] \leq 1_{\{b_i^{(k)} \geq N-T,\ \sum_j b_j^{(k)} \geq N\}}. \tag{5.11}$$

Since $(b_1^{(k)}, ..., b_M^{(k)})$ is assumed to be identically and independently distributed across frames according to $(\mathbf{b}_1, ..., \mathbf{b}_M)$, from (5.2) we get the following necessary condition on the achievability of η_i :

$$\eta_i \leq \mathbb{P}\left(\mathbf{b}_i \geq N - T,\ \sum_j \mathbf{b}_j \geq N\right). \tag{5.12}$$

Let $N_s(b_1^{(k)}, ..., b_M^{(k)}) = \sum_i R_i[k]$ be the number of devices that successfully receive the whole block k, given the B2D arrivals $(b_1^{(k)}, ..., b_M^{(k)})$. The following lemma provides an upper bound on $N_s(b_1^{(k)}, ..., b_M^{(k)})$.

Lemma 3. *Given the B2D arrivals $(b_1^{(k)}, ..., b_M^{(k)})$, we have*

$$\begin{aligned} & N_s(b_1^{(k)}, ..., b_M^{(k)}) \\ & \leq N_s^*(b_1^{(k)}, ..., b_M^{(k)}) \\ & \equiv \frac{\min\left(|\mathcal{S}|(N-T), \left[\sum_i b_i^{(k)} - T\right]^+\right)}{N-T}, \end{aligned} \tag{5.13}$$

where $\mathcal{S} = \{i \in \{1, ..., M\} : N - b_i^{(k)} \leq T,\ b_i^{(k)} + \sum_{j \neq i} b_j^{(k)} \geq N\}$.

Proof. Since $n_i^{(k)} \leq \hat{n}_i^{(k)}$, we have

$$R_i[k] = 1_{\{n_i^{(k)} \geq N\}} \leq 1_{\{\hat{n}_i^{(k)} \geq N\}} = 1_{\{b_i^{(k)} - t_i^{(k)} + \sum_j t_j^{(k)} \geq N\}}.$$

Therefore, we can solve the following maximization problem in order to find an upper bound on $N_s(b_1^{(k)}, ..., b_M^{(k)})$,

$$\begin{aligned} \max\ & \textstyle\sum_i 1_{\{b_i^{(k)} - t_i^{(k)} + \sum_j t_j^{(k)} \geq N\}} \\ \text{s.t.}\quad & t_i^{(k)} \leq b_i^{(k)} \qquad \forall i \\ & \textstyle\sum_j t_j^{(k)} \leq T \end{aligned} \tag{5.14}$$

The first constraint implies $\sum_j t_j^{(k)} \leq \sum_j b_j^{(k)}$. Hence, to achieve the maximum objective we let $\sum_j t_j^{(k)} = \min(\sum_j b_j^{(k)},\ T)$.

We partition the set of devices $\{1, ..., M\}$ into sets $\mathcal{S}$ and $\mathcal{S}^c = \{1, ..., M\} \backslash \mathcal{S}$. Here, $\mathcal{S}^c$ is the set of devices which, either individually or collectively, have not received enough number of B2D arrivals to possibly recover the block, and $R_i[k] = 0$ for $i \in \mathcal{S}^c$.

Suppose $\sum_j b_j^{(k)} < N$. Then we have $N_s(b_1^{(k)}, ..., b_M^{(k)}) = |\mathcal{S}| = 0$. Otherwise $\sum_j b_j^{(k)} \geq N$, and with our assumption that $T < N$, the optimization in (5.14) can be rewritten as

$$\begin{aligned} &\max\ \textstyle\sum_{i\in\mathcal{S}} 1_{\{b_i^{(k)}-t_i^{(k)}\geq N-T\}} \\ \text{s.t.}\quad & t_i^{(k)} \leq b_i^{(k)} \qquad\qquad \forall i \\ & \textstyle\sum_j t_j^{(k)} = \min(\sum_j b_j^{(k)},\ T) = T. \end{aligned} \tag{5.15}$$

The maximum value above can be shown to be

$$\left\lfloor \frac{\min\left(|\mathcal{S}|(N-T), \left[\sum_i b_i^{(k)} - T\right]^+\right)}{N-T} \right\rfloor.$$

Consequently, we obtain

$$N_s(b_1^{(k)}, ..., b_M^{(k)}) \leq \frac{\min\left(|\mathcal{S}|(N-T), \left[\sum_i b_i^{(k)} - T\right]^+\right)}{N-T}.$$

□

Now, from Lemma 3 and (5.2), since $(b_1^{(k)}, ..., b_M^{(k)})$ is *i.i.d* over frames, the following necessary condition on $\sum_i \eta_i$ can be obtained:

$$\sum_i \eta_i \leq \mathbb{E}\left[\lfloor N_s^*(\mathbf{b}_1, ..., \mathbf{b}_M)\rfloor\right]. \tag{5.16}$$

The following theorem summarizes our results.

Theorem 4. *The QoS metric $(\eta_1, ..., \eta_M)$ is achievable with respect to* i.i.d *B2D arrivals $(\mathbf{b}_1, ..., \mathbf{b}_M)$ if and only if the following conditions are satisfied*

$$\begin{aligned} &(C1)\ \forall i: \quad \eta_i \leq \mathbb{P}\left(\mathbf{b}_i \geq N-T,\ \textstyle\sum_j \mathbf{b}_j \geq N\right) \\ &(C2) \qquad\ \ \textstyle\sum_i \eta_i \leq \mathbb{E}\left[\lfloor N_s^*(\mathbf{b}_1, ..., \mathbf{b}_M)\rfloor\right]. \end{aligned} \tag{5.17}$$

Further, we can show that for the symmetric case where $\eta_i = \eta$, and $\mathbf{b}_i$ are identically distributed *for all devices i, the necessary and sufficient condition reduces to*

$$(C2') \quad \eta \leq \tfrac{1}{M}\mathbb{E}\left[\lfloor N_s^*(\mathbf{b}_1, ..., \mathbf{b}_M)\rfloor\right]. \tag{5.18}$$

Proof. The necessity part was shown in (5.12) and (5.16). To prove the sufficiency of these conditions, we will propose an algorithm in the next subsection which can fulfill any QoS constraints satisfying $(C1)$ and $(C2)$. The proof of (C2') is not hard, and is omitted here due to space constraints. □

5.6.2 Optimal Reliable D2D Scheme

In this subsection, we propose a simple algorithm that can achieve any QoS metric satisfying the conditions in (5.17). As a result, $(C1)$ and $(C2)$ are sufficient conditions on achievability of a QoS metric (Theorem 4). Also, the proposed algorithm is throughput optimal.

We follow the same approach as in Section 5.5 to study the D2D system using deficit queues $d_i[k]$. Algorithm 3 describes an optimal D2D scheme that can stabilize these deficit queues for any achievable QoS metrics.

Algorithm 3 Optimal D2D scheme (reliable broadcast)

```
At the beginning of each frame $k-1$, given the arrivals $(b_1^{(k)}, ..., b_M^{(k)})$:
Partition the devices into sets $\mathcal{S} = \{i \in \{1, ..., M\} : N - b_i^{(k)} \leq T, b_i^{(k)} + \sum_{j \neq i} b_j^{(k)} \geq N\}$ and $\mathcal{S}^c$.
If $\mathcal{S} = \emptyset$, nobody can get full-rank. Otherwise,
Phase 1) Let all the devices in $\mathcal{S}^c$ transmit all they have initially received for the next $T_1 = \min\{\sum_{i \in \mathcal{S}^c} b_i^{(k)}, T\}$ slots.
If there exist time and a need for more transmissions,
Phase 2) Let each device $i \in \mathcal{S}$ transmit up to $b_i^{(k)} + T - N$ of its initial chunks.
Phase 3) While there exist time and a need for more transmissions, let devices in $\mathcal{S}$ transmit their remaining chunks in an increasing order of their deficit values.
```

Theorem 5. *The D2D scheme in Algorithm 3 is throughput optimal when the D2D transmissions are reliable.*

Proof. As in the proof of Theorem 2, we use the Lyapunov criterion. Equivalent to the optimization in (5.6), we need to solve the following

$$\begin{aligned} \max\ & \textstyle\sum_i d_i R_i[k] = \sum_i d_i 1_{\{b_i^{(k)} + \sum_{j\neq i} t_j^{(k)} \geq N\}} \\ \text{s.t.}\quad & t_i^{(k)} \leq b_i^{(k)} \qquad \forall i \\ & \textstyle\sum_i t_i^{(k)} \leq T. \end{aligned} \tag{5.19}$$

The optimization in (5.19) is similar to the one in (5.14). Hence, we can apply a similar argument and verify that the following optimization problem is equivalent to (5.19),

$$\begin{aligned} \max\ & \textstyle\sum_i d_i z_i \\ \text{s.t.}\quad & z_i \leq 1_{\{b_i^{(k)} \geq N-T,\ \sum_j b_j^{(k)} \geq N\}} \quad \forall i \\ & \textstyle\sum_i z_i \leq N_s^*(b_1^{(k)}, ..., b_M^{(k)}), \end{aligned} \tag{5.20}$$

where $N_s^*(b_1^{(k)}, ..., b_M^{(k)})$ is defined in (5.13).

Using the solution to the above maximization in the drift formula $\Delta V[k]$ will result in

$$\begin{aligned} & \Delta V[k] \leq M/2 + \textstyle\sum_i d_i \eta_i - \mathbb{E}\left[\max \sum_i d_i z_i\right] \\ & \leq M/2 + \textstyle\sum_i d_i \eta_i - \max \sum_i d_i \mathbb{E}\left[z_i\right]. \end{aligned} \tag{5.21}$$

From (5.20), we notice that $\mathbb{E}\left[z_i\right]$ must satisfy

$$\begin{aligned} \mathbb{E}[z_i] \leq \mathbb{P}\left(b_i^{(k)} \geq N-T,\ \textstyle\sum_j b_j^{(k)} \geq N\right) \\ \textstyle\sum_i \mathbb{E}[z_i] \leq \mathbb{E}\left[N_s^*(b_1^{(k)}, ..., b_M^{(k)})\right]. \end{aligned} \tag{5.22}$$

In Subsection 5.6.1, we have shown that an achievable QoS metric $(\eta_1, ..., \eta_M)$ needs to satisfy the above conditions. This suggests that for a strictly achievable QoS metric $(\eta_1, ..., \eta_M)$, for which these conditions hold with strict inequalities, there exists some $\epsilon > 0$ such that

$$\max \sum_i d_i \mathbb{E}\left[z_i\right] \geq \sum_i d_i \eta_i (1+\epsilon). \tag{5.23}$$

Consequently, the drift in (5.21) reduces to

$$\Delta V[k] \leq M/2 - \epsilon \textstyle\sum_i d_i \eta_i. \tag{5.24}$$

Thus, for large enough deficit values d_i, the drift is negative. This means the D2D scheme implied by solving (5.19) at each frame can satisfy any achievable QoS metric, and hence is optimal. Furthermore, it proves the sufficiency of the conditions in Theorem 4 on the achievability of a QoS metric.

The argument showing that the D2D scheme in Algorithm 3 actually solves the optimization problem in (5.19) is straightforward, and follows from dividing devices into those that cannot complete (and so might as well transmit all that they have), then considering those that can complete (and hence can transmit a limited number of chunks), and finally considering those with the largest deficit (and so should stop transmitting after phase two). □

5.6.3 Simple Suboptimal Scheme for Unreliable D2D Network

The optimal D2D scheme, presented in Section 5.5 for the unreliable D2D system is hard to implement in practice due to high complexity. Further, this algorithm requires estimates on the channel quality. Here, we propose a modified version of Algorithm 3, which inherits its simple 3-phase form and tries to account for the unreliability of the broadcast network by letting devices retransmit a number of times.

We incorporate the following modifications in Algorithm 3:

1. In the second phase, we choose devices in an increasing order of their deficit values to transmit upto the same threshold that Algorithm 3 suggests.

2. Each device chosen to transmit by Algorithm 3 holds onto the channel for ρ time slots. For

each transmission, it generates a new chunk as a random combination of its initial B2D chunks.

The parameter ρ can be chosen in accordance with the quality of broadcast channel. The heuristic algorithm attempts to improve the performance of Algorithm 3 when loss rates are not too high by accounting for deficits during Phase 2, as well as improving reliability by retransmissions. We will investigate the performance of this algorithm in Section 5.9.

5.7 Selection of B2D Stopping Time

In this section, we would like to determine the number of time slots that the B2D interface should be used by each device over each frame. We consider this as an offline stopping time problem, and will show how to calculate this value. Thus, we want to find $0 \leq T_B(i) \leq T$, which is the number of times in each frame that device i attempts to receive a coded chunk from the server using the B2D interface.

The B2D usage times should be such that the QoS target can be met with the lowest total cost $C(T_B(1), ..., T_B(M))$, where $C(.)$ is a general cost function. Note that the function could be linear ($\sum_i T_B(i)$), if usage-based costs are considered. Furthermore, it makes intuitive sense that devices that desire a higher QoS η_i, should contribute more to the system so as to maintain a level of fairness. This requirement can also be captured in the cost function, by simply choosing weights for each device that are dependent on its desired QoS value.

Recall that the B2D interface of each device i is assumed to be Bernoulli with success probability β_i. This means that the B2D arrivals $b_i^{(k)}$ to each device i are independently and identically distributed over frames k as a Binomial random variable $Bin(\beta_i, T_B(i))$ with parameters β_i and $T_B(i)$:

$$\mathbb{P}(b_i^{(k)} = a) = 1_{\{0 \leq a \leq T_B(i)\}} \binom{T_B(i)}{a} \beta_i^a (1 - \beta_i)^{T_B(i) - a}.$$

For the fully reliable D2D scenario, we can determine the achievability of a QoS requirement based on conditions $(C1)$ and $(C2)$ in Theorem 4. In order to achieve a given QoS metric $(\eta_1, ..., \eta_M)$, $T_B(i)$ values must be large enough such that these conditions are satisfied. Hence, the optimal values of $T_B^*(i)$ can be obtained by solving the following problem:

$$\begin{aligned} &\min C(T_B(1), ..., T_B(M)) \\ &\text{s.t.} \\ &0 \le T_B(i) \le T \qquad \text{for all } i \\ &\mathbf{b}_i = Bin(\beta_i, T_B(i)) \quad \text{for all } i \\ &(C1) \text{ and } (C2) \text{ in (5.17) are satisfied.} \end{aligned} \tag{5.25}$$

Note that the minimization problem in (5.25) does not have a simple form that can be solved efficiently. However, since the region for feasible $T_B(i)$ values is finite (*i.e.,* $\{0, 1, ..., T\}^M$), and the values need to be determined only once for the given set of system parameters, we could conduct an exhaustive search to find the optimal $T_B^*(i)$ values. We can also improve the search algorithm by applying more efficient search methods like *branch-and-bound*.

In the case of the unreliable D2D channel, there is only an implicit achievability condition as given by Lemma 1. We therefore cannot analytically solve for the optimal B2D usage times. Since $T_B(i)$ values are to be calculated in an offline manner, we can run stochastic simulations of the optimal D2D scheme presented in Algorithm 1 in order to find the optimal values numerically. We present results of this nature in Table 5.1.

5.8 Finite Field Case

So far we assumed that the field size q is large enough that all randomly coded chunks are linearly independent almost surely. Under this assumption, it was sufficient for each device to receive N distinct coded chunks in order to recover the original block. In this section, we turn our attention to the case where field size $q < \infty$, and there is a non-zero probability that randomly coded chunks are linearly dependent. More specifically, we are interested in evaluating the performance of our proposed B2D and D2D schemes in the case of finite field sizes. We require the following useful Lemma, whose proof is omitted for brevity.

Lemma 6. *A matrix of dimension $R \times N$, whose elements are drawn uniformly at random from a* *finite field F_q, is full-rank with probability at least $1 - \frac{1}{q^{|N-R|}(q-1)}$, where q is the field size.*

We now have the following result.

Theorem 7. *Suppose that the coefficients for coding the chunks are drawn uniformly at random from a field of size $q \geq 2$. If we apply Algorithm 2 (or Algorithm 3 for the reliable D2D case) to coordinate the D2D broadcasts and choose the B2D usage times as discussed in Section 5.7, then for each device i we have*

$$\lim_{K\to\infty} \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}[R_i[k]] \geq \eta_i - \frac{1}{q-1}. \tag{5.26}$$

Proof. We define $\hat{R}_i[k] = 1_{\{b_i^{(k)} + \sum_{j\neq i} \min(\hat{t}_j^{(k)}, b_j^{(k)}) \geq N\}}$. For the case of infinite field size, we had $n_i^{(k)} = \min\left(N, b_i^{(k)} + \sum_{j\neq i} \min(\hat{t}_j^{(k)}, b_j^{(k)})\right)$ and $\hat{R}_i[k] = R_i[k]$.

In the proof of throughput optimality of Algorithms 2 and 3, we showed that

$$\lim_{K\to\infty} \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}[\hat{R}_i[k]] \geq \eta_i \tag{5.27}$$

holds true, when we employ the proposed D2D schemes. For $R_i[k]$, we have

$$\begin{aligned}
&\mathbb{E}[R_i[k]] = \mathbb{E}[R_i[k]|\hat{R}_i[k] = 1]\mathbb{P}(\hat{R}_i[k] = 1) \\
&+\mathbb{E}[R_i[k]|\hat{R}_i[k] = 0]\mathbb{P}(\hat{R}_i[k] = 0) \\
&\overset{(a)}{=} \mathbb{E}[R_i[k]|\hat{R}_i[k] = 1]\mathbb{P}(\hat{R}_i[k] = 1) \\
&= \mathbb{E}[R_i[k]|\hat{R}_i[k] = 1](1 - 1 + \mathbb{E}[\hat{R}_i[k]]) \\
&\overset{(b)}{\geq} \mathbb{E}[R_i[k]|\hat{R}_i[k] = 1] + \mathbb{E}[\hat{R}_i[k]] - 1 \overset{(c)}{\geq} \mathbb{E}[\hat{R}_i[k]] - \tfrac{1}{q-1},
\end{aligned} \tag{5.28}$$

where (a) holds because if $\hat{R}_i[k] = 0$, then i did not receive enough distinct coded chunks and cannot get full-rank (*i.e.,* $R_i[k] = 0$). (b) follows from $-1 + \mathbb{E}[\hat{R}_i[k]] \leq 0$ and $\mathbb{E}[R_i[k]|\hat{R}_i[k] = 1] \leq 1$, and (c) holds by Lemma 6, since $\mathbb{E}[R_i[k]|\hat{R}_i[k] = 1]$ is equal to the probability that i who has received at least N distinct randomly coded chunks is full-rank. We then sum both sides of (5.28) from $k = 1$ to K, divide the result by K, and let $K \to \infty$. Then the desired bound in (5.26) follows from (5.27). □

Note that the effect of finiteness of the field size is limited by the value $\frac{1}{q-1}$, for example for field size $q = 32$, there is only around 3% reduction in the quality of service. In the simulation results presented in Figure 5.3, we see the actual reduction in the QoS is even less than this value. Note that the achieved delivery ratio is sometimes higher than desired since T_B is chosen as the smallest integer value that can support η.

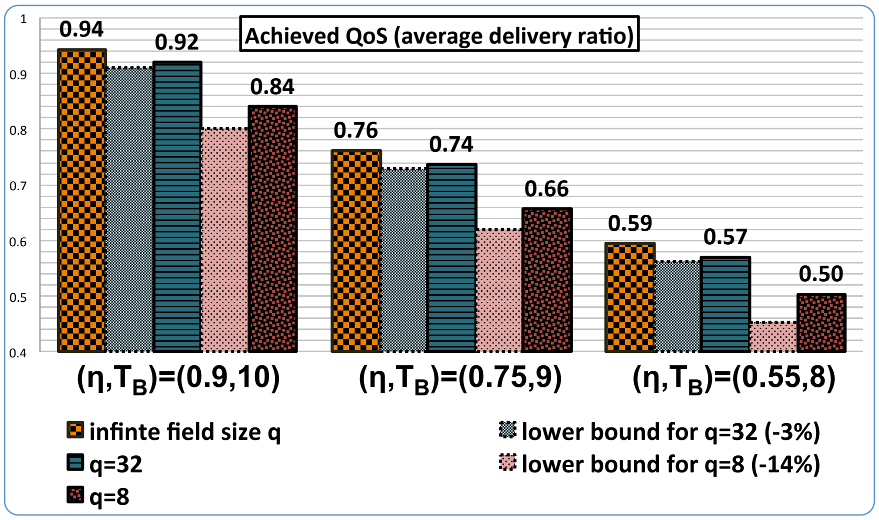

Figure 5.3: Achievable delivery ratios with finite field sizes ($N = 20$, $T = 15$, $M = 4$, $\beta_i = 0.9$ in a reliable D2D network).

5.9 Simulation Results

We now study the performance of our different algorithms through Matlab simulations. We chose the chunks per block $N = 20$, frame length $T = 15$, B2D success probability $\beta_i = 0.9$ for all i, and varied the other parameters. The results are listed below.

B2D Stopping Time: In Section 5.7, we saw that the B2D stopping time problem needs a numerical solution. We present the optimal stopping time T_B^* for a system with M devices with QoS requirement η, probability of success of the D2D channel δ, and a linear cost criterion in

Table 5.1. We use values from this table as the optimal B2D stopping times T_B^* in the other simulations presented in this section.

Table 5.1: Optimal B2D stopping time T_B^*

	η \ δ	1	0.95	0.9	0.85	0.8
$M = 4$	0.8	9	11	12	13	14
	0.9	10	12	13	14	15
$M = 5$	0.8	8	10	11	12	13
	0.9	9	11	12	13	14

Heuristic algorithm for unreliable D2D: In Section 5.6.3, we proposed a simple heuristic scheme based on the optimal algorithm derived for the reliable D2D case. Table 5.2 depicts the achieved QoS by this algorithm for number of per-chunk transmissions $\rho = 2$, in a network of $M = 4$ devices and a target QoS of $\eta = 0.8$. We observe that if we set $T_B = T_B^*$, the degradation

Table 5.2: QoS achieved by the heuristic algorithm for $T_B \geq T_B^*$

δ \ T_B	11	12	13	14	15
0.8				73.88%	74.78%
0.85			72.54%	74.45%	74.93%
0.9		68.92%	73.54%	74.77%	74.96%
0.95	61.79%	70.78%	74.11%	74.85%	74.97%

in the QoS when using the heuristic algorithm is around $7\% - 22\%$ for different values of δ. We can improve the achieved QoS by increasing the B2D usage time, *e.g.,* the degradation reduces to $6\% - 11\%$ if we let $T_B = T_B^* + 1$. The results suggest that the simple heuristic algorithm can be used successfully if the D2D channel quality is reasonably good, or if the frame length T is long enough to permit retransmissions.

Other intuitive D2D schemes: In Figure 5.4, we compare the optimal scheme, Algorithm 3, with other algorithms in the case of reliable D2D: (i) Round Robin: each device broadcasts in turn, (ii) Min-Deficit-First: the device with smallest deficit broadcasts, and (iii) Max-Rank-First: the device with highest current rank broadcasts. We see that the optimal algorithm outperforms the others by combining the best of each.

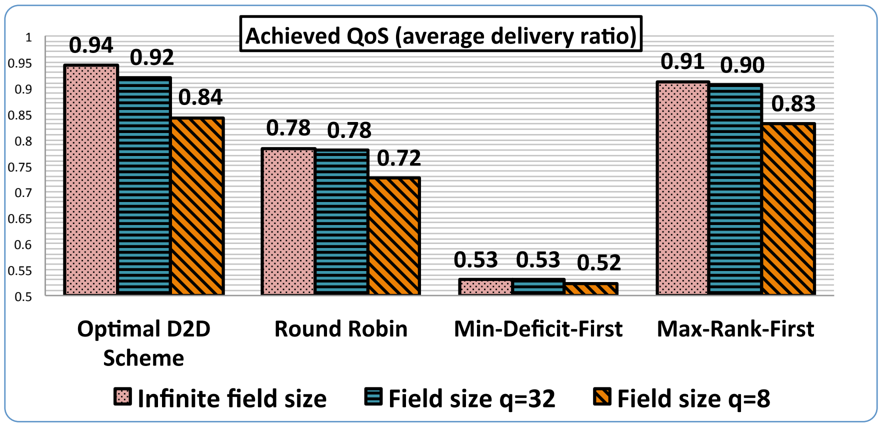

Figure 5.4: Comparison of Algorithm 3 with others for $M = 4$ and $\eta = 0.9$.

Playout smoothness: Since our QoS metric is in terms of the average delivery ratio, we do not know if a device loses a few chunks with a small periodicity, or a large number of chunks with a large periodicity, or indeed if it is periodic at all. The first case is preferred, since by using an outer code over the blocks, the device will be able to recover the dropped blocks to some extent. We define a *smooth play out interval* as the time between two consecutive block drops. Figure 5.5 demonstrates the distribution of these smooth play out intervals. We observe that the actual QoS implied by our algorithm is acceptable, since the smooth play out interval has a small variance (*i.e.,* almost periodic reception) and a small average (no loss of big portions).

5.10 Android Experiments

We now describe experiments on an Android testbed consisting of HTC Evo smart phones. A full scale system with peer discovery, multimedia coding and so on is beyond the scope of this

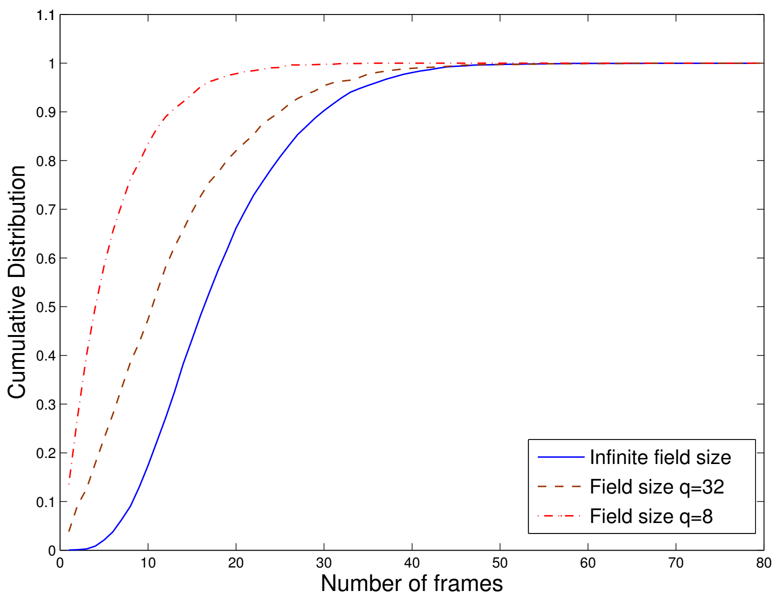

Figure 5.5: Distribution of smooth play out times ($\eta = 0.9$).

chapter, and our objective is primarily to observe real-world performance of the D2D algorithms. As discussed earlier, we emulate B2D transmissions by initializing each device periodically with chunks representing their B2D arrivals to test the D2D algorithms. Since Android does not support ad-hoc UDP broadcast (D2D) over 802.11 (WiFi), we rooted the phones to enable this service. To allow experiments with more (unrooted) phones, we also used the approximation of UDP broadcast via a WiFi access point (AP). Here, the phone unicasts to the AP, which broadcasts to the rest.

As in the analytical model, we divide the file into blocks, with the blocks being further divided into chunks. We chose the chunk size as 1450 Bytes to roughly correspond to the usual packet size in 802.11. Chunks are generated using an open source random linear coding library [69] with field size 256, and the degrees of freedom per block is 10, *i.e.*, a block is decodable with high probability if 10 chunks are received successfully across both interfaces. Each chunk has a header that contains the block number that it corresponds to, as well as the device's current deficit.

While the analytical model assumes slotted time, we do not have fine grain control of when exactly transmissions will take place in 802.11. Instead, we transmit a pilot chunk (whose header simply contains frame number) periodically to indicate the beginning of a new frame, but do not have distinct time slots within the frame. Recall that our D2D scheduling in Algorithm 3 proceeds in three phases. We approximate these phases by backing off by different amounts of time, based on the estimate of which phase that device is in before sending the chunk to the UDP socket. Note that after doing this, 802.11 itself follows a back off procedure to avoid collisions.

Now, the three backoff times are chosen as follows. Devices that cannot complete (*i.e.,* Phase 1 devices) should be the most aggressive in D2D channel access and transmit all their chunks. We therefore set them to randomly backoff between 1 and 5 ms before transmission. Devices that can afford to transmit some number of chunks should be less aggressive, and transmit chunks upto the limit as given by Phase 2 of Algorithm 3 by backing off between 1 and 15 ms. Finally, once each device completes Phase 2, it enters Phase 3, and modulates its aggressiveness based on deficits in the system. Each device normalizes its deficit based on the values of deficits that it sees from all other transmissions, and backs off proportional to this deficit within the interval of 10 to 20 ms. To allow for all the backoffs and the transmission of 10 or more D2D chunks, we set the frame duration as 450 ms.

Devices ensure that they are all synchronized to each other by observing the frame ID present in the header of each chunk. If a device is not able to recover a block by the time a new frame pilot chunk or a chunk bearing a new frame ID is received, it reports that block as lost and updates its deficit.

We conducted experiments involving 3 to 5 phones, using a standard MP3 audio file as the source, since playout is easily accomplished using the built in player on Android. An example trajectory of the (smoothed) deficit queue for a run with 3 phones, with each phone being initialized with 4 chunks in each frame, and desired delivery ratio 0.95 is shown in Figure 5.6. The deficit queue exhibits periodic decrease, and clearly does not increase to infinity, showing stabilization.

We also conducted experiments to determine the stable delivery ratio achieved using D2D for different B2D initializations per frame. We present these results for 5 phones (labeled Ph $1-5$), with delivery ratio indicated out of 100% in Table 5.3. It is clear we can have significant savings in

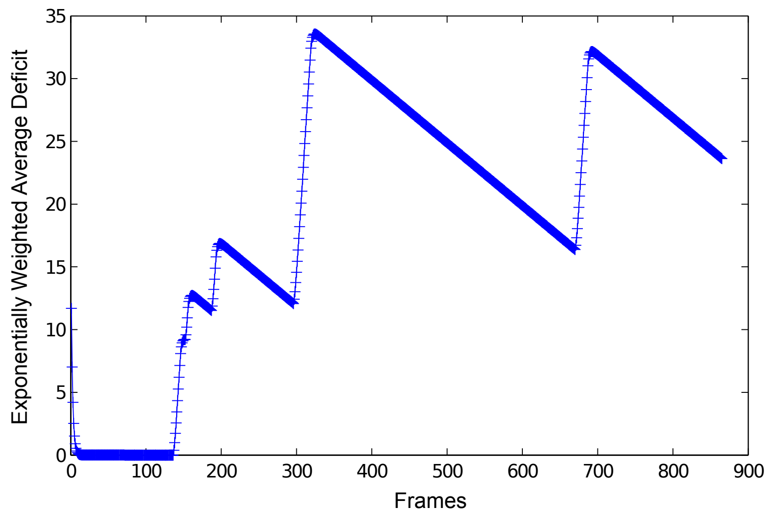

Figure 5.6: Sample trajectory of deficit.

B2D usage. For instance, with a target delivery ratio of 94%, we save about 60% of the B2D costs for each phone.

Table 5.3: Delivery ratios achieved with different initializations.

B2D	Ph 1	Ph 2	Ph 3	Ph 4	Ph 5	Avg
2	56.38	38.44	56.37	41.7	43.1	47.198
3	86.9	70.85	81.8	75.3	83.6	79.69
4	85.75	94.6	94.3	98.31	98.45	94.282
5	98.34	98.03	97.3	99.48	100	98.63

5.11 Conclusion

We studied the problem of realtime streaming applications for wireless devices that can receive coded data chunks via both expensive long-range unicast and an inexpensive short-range broadcast wireless interfaces. QoS was defined in terms of the fraction of blocks recovered successfully on average.

We utilized a Lyapunov stability argument to propose a minimum cost D2D transmission algorithm that can attain the required QoS, in the case of unreliable and reliable D2D transmissions. We also showed how to calculate the minimum cost B2D usage per device. We showed how the infinite field size assumption under which the algorithms are derived does not have a significant impact on scaling and performance, and then showed how performance changes with parameters using simulations. Finally, we described our design decisions for an Android implementation. Future work includes a full scale implementation, as well as multihop D2D transmission.

6. CONCLUSION

The goal of this dissertation was to exploit the increasingly prevalent paradigm of software reconfigurable infrastructure with the objective of optimizing application performance under the resource constraints of the wireless edge. We studied several problems in the space of cross layer optimization, and demonstrated that a holistic approach, which accounts for the interactions between a configuration and the effect it has on the application performance, is able to achieve both Quality of Service (QoS) and Quality of Experience (QoE) guarantees that a narrow focus on individual components of the system is incapable of achieving.

We first studied reconfigurability in the context of (priority) queueing with High Definition video streaming application as our use case. We developed Reinforcement Learning control policies which are able to prioritize the desired clients in a bandwidth restrained scenario, and hence exceed the performance exhibited by the state of the art policies. We next designed an open-market based approach which promotes truthful declaration of value by the clients. Such an approach not only enables applications to obtain the necessary resources for optimal performance, but also is vital for the correct determination of control policies using the learning approach. We also explored a custom hardware platform to achieve finer grained reconfiguration capabilities such as per-packet scheduling. Our processor supported implementation enables flexible scheduling policies on software and high hardware function re-usability, while meeting ultra-low latency latency requirements of modern applications like Virtual Reality. Finally, we also demostrated a distributed approach for satisfying application performance requirements by leveraging end user devices interested in a shared objective. Such an approach empowered us to achieve the performance guarantees with minimal use of centralized infrastructure.

We believe that the application of such software-defined, adaptive resource allocation will be in upcoming small cell wireless architectures such as 5G, and our goal will be to extend our ideas to such settings.

REFERENCES

[1] L. Tassiulas and A.Ephermides, “Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks,” *IEEE Trans. Automat. Contr.*, vol. 37, no. 12, pp. 1936–1948, 1992.

[2] A. Eryilmaz, R. Srikant, and J. Perkins, “Stable scheduling policies for fading wireless channels,” *IEEE/ACM Trans. Network.*, vol. 13, pp. 411–424, April 2005.

[3] I. Hou, V. Borkar, and P. Kumar, “A theory of QoS for wireless,” in *IEEE INFOCOM 2009*, (Rio de Janeiro, Brazil), April 2009.

[4] S. Yau, P.-C. Hsieh, R. Bhattacharyya, K. Bhargav, S. Shakkottai, I. Hou, and P. Kumar, “Puls: Processor-supported ultra-low latency scheduling,” in *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pp. 261–270, ACM, 2018.

[5] Ericsson, “Ericsson Mobility Report: On the Pulse of the Networked Society.” `https://www.ericsson.com/assets/local/mobility-report/documents/2015/ericsson-mobility-report-june-2015.pdf`, 2015.

[6] R. Singh and P. Kumar, “Optimizing quality of experience of dynamic video streaming over fading wireless networks,” in *Decision and Control (CDC), 2015 IEEE 54th Annual Conference on*, pp. 7195–7200, IEEE, 2015.

[7] P. Shome, M. Yan, S. M. Najafabad, N. Mastronarde, and A. Sprintson, “Crossflow: A cross-layer architecture for SDR using SDN principles,” in *Proceedings of IEEE NFV-SDN*, 2015.

[8] P. Shome, J. Modares, N. Mastronarde, and A. Sprintson, “Enabling dynamic reconfigurability of SDRs using SDN principles,” in *Proceedings of Ad Hoc Networks*, 2017.

[9] M. Yan, J. Casey, P. Shome, A. Sprintson, and A. Sutton, “Ætherflow: Principled wireless support in SDN,” in *Proceedings of IEEE ICNP*, 2015.

[10] J. Schulz-Zander, N. Sarrar, and S. Schmid, "AeroFlux: A near-sighted controller architecture for software-defined wireless networks," in *Proceedings of USENIX ONS*, 2014.

[11] J. Schulz-Zander, C. Mayer, B. Ciobotaru, S. Schmid, and A. Feldmann, "OpenSDWN: Programmatic control over home and enterprise WiFi," in *Proceedings of ACM SOSR*, 2015.

[12] R. Mok, W. Li, and R. Chang, "IRate: Initial video bitrate selection system for HTTP streaming," *IEEE Journal on Selected Areas in Communications*, vol. 34, pp. 1914–1928, June 2016.

[13] T. Spetebroot, S. Afra, N. Aguilera, D. Saucez, and C. Barakat, "From network-level measurements to expected quality of experience: The Skype use case," in *Proceedings of IEEE M&N*, 2015.

[14] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, "SDN-based application-aware networking on the example of YouTube video streaming," in *Proceedings of EWSDN*, 2013.

[15] H. Nam, K.-H. Kim, J. Y. Kim, and H. Schulzrinne, "Towards QoE-aware video streaming using SDN," in *Proceedings of IEEE GLOBECOM*, 2014.

[16] S. Ramakrishnan, X. Zhu, F. Chan, and K. Kambhatla, "SDN based QoE optimization for HTTP-based adaptive video streaming," in *Proceedings of IEEE ISM*, 2015.

[17] P. Georgopoulos, Y. Elkhatib, M. Broadbent, M. Mu, and N. Race, "Towards network-wide QoE fairness using Openflow-assisted adaptive video streaming," in *Proceedings of ACM FhMN*, 2013.

[18] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 197–210, ACM, 2017.

[19] H. Yeganeh, R. Kordasiewicz, M. Gallant, D. Ghadiyaram, and A. C. Bovik, "Delivery quality score model for Internet video," in *Proceedings of IEEE ICIP*, 2014.

[20] N. Eswara, K. Manasa, A. Kommineni, S. Chakraborty, H. P. Sethuram, K. Kuchi, A. Kumar, and S. S. Channappayya, "A continuous QoE evaluation framework for video streaming over HTTP," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. In press, 2017.

[21] D. Ghadiyaram, J. Pan, and A. C. Bovik, "Learning a continuous-time streaming video QoE model," *IEEE Transactions on Image Processing*, vol. 27, pp. 2257–2271, May 2018.

[22] CPqD, "OpenFlow Software Switch." `http://cpqd.github.io/ofsoftswitch13/`, 2015.

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

[24] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.," in *AAAI*, vol. 2, p. 5, Phoenix, AZ, 2016.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[26] M. Schaarschmidt, A. Kuhnle, and K. Fricke, "Tensorforce: A tensorflow library for applied reinforcement learning," *Web page* *`https://github.com/reinforceio/tensorforce`*, 2017.

[27] M. Manjrekar, V. Ramaswamy, and S. Shakkottai, "A mean field game approach to scheduling in cellular systems," in *Proceedings of IEEE INFOCOM*, pp. 1554–1562, 2014.

[28] J. Sun, E. Modiano, and L. Zheng, "Wireless channel allocation using an auction algorithm," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 5, pp. 1085–1096, 2006.

[29] S. Ha, S. Sen, C. Joe-Wong, Y. Im, and M. Chiang, "TUBE: Time-dependent pricing for mobile data," in *Proceedings of ACM SIGCOMM*, pp. 247–258, 2012.

[30] "NSF Workshop on Ultra-Low Latency Wireless Networks," November 2016.

[31] ITU-T, "The Tactile Internet," August 2014.

[32] H. Holma and A. Toskala, *LTE for UMTS: Evolution to LTE-advanced.* John Wiley & Sons, 2011.

[33] M. Laner, P. Svoboda, P. Romirer-Maierhofer, N. Nikaein, F. Ricciato, and M. Rupp, "A comparison between one-way delays in operating HSPA and LTE networks," in *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), 2012 10th International Symposium on*, pp. 286–292, IEEE, 2012.

[34] K. Sui, M. Zhou, D. Liu, M. Ma, D. Pei, Y. Zhao, Z. Li, and T. Moscibroda, "Characterizing and improving WiFi latency in large-scale operational networks," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 347–360, ACM, 2016.

[35] I. H. Hou, V. Borkar, and P. R. Kumar, "A Theory of QoS for Wireless," in *IEEE INFOCOM 2009*, pp. 486–494, April 2009.

[36] I.-H. Hou and P. Kumar, "Utility maximization for delay constrained qos in wireless," in *INFOCOM, 2010 Proceedings IEEE*, pp. 1–9, IEEE, 2010.

[37] J. J. Jaramillo and R. Srikant, "Optimal Scheduling for Fair Resource Allocation in Ad Hoc Networks With Elastic and Inelastic Traffic," *IEEE/ACM Transactions on Networking*, vol. 4, no. 19, pp. 1125–1136, 2011.

[38] I.-H. Hou, "Broadcasting delay-constrained traffic over unreliable wireless links with network coding," *IEEE/ACM Transactions on Networking*, vol. 23, no. 3, pp. 728–740, 2015.

[39] K. S. Kim, C.-p. Li, and E. Modiano, "Scheduling multicast traffic with deadlines in wireless networks," in *INFOCOM, 2014 Proceedings IEEE*, pp. 2193–2201, IEEE, 2014.

[40] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE Transactions on Automatic Control*, vol. 37, pp. 1936–1948, Dec. 1992.

[41] A. Eryilmaz and R. Srikant, “Joint Congestion Control, Routing and MAC for Stability and Fairness in Wireless Networks,” *IEEE Journal on Selected Areas in Communications*, vol. 24, pp. 1514–1524, August 2006.

[42] R. Li, A. Eryilmaz, and B. Li, “Throughput-optimal wireless scheduling with regulated inter-service times,” in *INFOCOM, 2013 Proceedings IEEE*, pp. 2616–2624, 2013.

[43] R. K. Lam and P. Kumar, “Dynamic channel reservation to enhance channel access by exploiting structure of vehicular networks,” in *Vehicular Technology Conference (VTC 2010-Spring), 2010 IEEE 71st*, 2010.

[44] R. K. Lam and P. Kumar, “Dynamic channel partition and reservation for structured channel access in vehicular networks,” in *Proceedings of the seventh ACM international workshop on VehiculAr InterNETworking*, pp. 83–84, ACM, 2010.

[45] S. Sanghavi, D. Shah, and A. S. Willsky, “Message passing for max-weight independent set,” in *Advances in Neural Information Processing Systems*, pp. 1281–1288, 2008.

[46] A. Anand, G. de Veciana, and S. Shakkottai, “Joint scheduling of urllc and embb traffic in 5g wireless networks,” in *INFOCOM (to appear), 2018 Proceedings IEEE*, 2018.

[47] A. Warrier, S. Janakiraman, S. Ha, and I. Rhee, “DiffQ: Practical differential backlog congestion control for wireless networks,” in *INFOCOM 2009, IEEE*, pp. 262–270, IEEE, 2009.

[48] R. Laufer, T. Salonidis, H. Lundgren, and P. Le Guyadec, “XPRESS: A cross-layer back-pressure architecture for wireless multi-hop networks,” in *Proceedings of the 17th annual international conference on Mobile computing and networking*, pp. 49–60, ACM, 2011.

[49] J. Ryu, V. Bhargava, N. Paine, and S. Shakkottai, “Back-pressure routing and rate control for ICNs,” in *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, pp. 365–376, ACM, 2010.

[50] J. Lee, H. Lee, Y. Yi, S. Chong, E. W. Knightly, and M. Chiang, “Making 802.11 DCF near-optimal: Design, implementation, and evaluation,” *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1745–1758, 2016.

[51] Y.-H. Wei, Q. Leng, S. Han, A. K. Mok, W. Zhang, and M. Tomizuka, "RT-WiFi: Real-time high-speed communication protocol for wireless cyber-physical control applications," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pp. 140–149, IEEE, 2013.

[52] O. N. Yilmaz, Y.-P. E. Wang, N. A. Johansson, N. Brahmi, S. A. Ashraf, and J. Sachs, "Analysis of ultra-reliable and low-latency 5G communication for a factory automation use case," in *Communication Workshop (ICCW), 2015 IEEE International Conference on*, pp. 1190–1195, IEEE, 2015.

[53] S. Yoshioka, Y. Inoue, S. Suyama, Y. Kishiyama, Y. Okumura, J. Kepler, and M. Cudak, "Field experimental evaluation of beamtracking and latency performance for 5G mmWave radio access in outdoor mobile environment," in *Personal, Indoor, and Mobile Radio Communications (PIMRC), 2016 IEEE 27th Annual International Symposium on*, pp. 1–6, IEEE, 2016.

[54] J. Pilz, M. Mehlhose, T. Wirth, D. Wieruch, B. Holfeld, and T. Haustein, "A Tactile Internet demonstration: 1ms ultra low delay for wireless communications towards 5G," in *Proc. of INFOCOM WKSHPS*, pp. 862–863, IEEE, 2016.

[55] S. Yau, L. Ge, P.-C. Hsieh, I. Hou, S. Cui, P. Kumar, A. Ekbal, N. Kundargi, *et al.*, "Wimac: Rapid implementation platform for user definable mac protocols through separation," in *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 109–110, ACM, 2015.

[56] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli, "Wireless mac processors: Programming mac protocols on commodity hardware," in *INFOCOM, 2012 Proceedings IEEE*, pp. 1269–1277, IEEE, 2012.

[57] Cisco, *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2010-2015*. Cisco, February 2011.

[58] S. Deb, M. Médard, and C. Choute, "Algebraic gossip: A network coding approach to optimal multiple rumor mongering," *IEEE Trans. on Information Theory*, vol. 52, no. 6, pp. 2486–2507, 2006.

[59] A. ParandehGheibi, M. Medard, A. Ozdaglar, and S. Shakkottai, "Avoiding interruptions - a QoE reliability function for streaming media applications," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 5, pp. 1064–1074, 2011.

[60] A. Sprintson, P. Sadeghi, G. Booker, and S. El Rouayheb, "A randomized algorithm and performance bounds for coded cooperative data exchange," in *Proc. of IEEE ISIT*, pp. 1888–1892, 2010.

[61] N. Milosavljevic, S. Pawar, S. El Rouayheb, M. Gastpar, and K. Ramchandran, "An optimal divide-and-conquer solution to the linear data exchange problem," in *Proc. of IEEE ISIT*, 2011.

[62] T. Courtade and R. Wesel, "Coded cooperative data exchange in multihop networks," *Arxiv preprint arXiv:1203.3445v1*, 2012.

[63] N. Abedini, M. Manjrekar, S. Shakkottai, and L. Jiang, "Meeting deadlines: Server-assisted wireless p2p networks," in *Proc. of ITA 2012*, February 2012.

[64] M. Wang and B. Li, "Lava: A reality check of network coding in peer-to-peer live streaming," in *INFOCOM 2007*, pp. 1082–1090, 2007.

[65] Z. Liu, C. Wu, B. Li, and S. Zhao, "UUSee: Large-scale operational on-demand streaming with random network coding," in *INFOCOM, 2010 Proceedings IEEE*, pp. 1–9, 2010.

[66] H. Seferoglu, L. Keller, B. Cici, A. Le, and A. Markopoulou, "Cooperative video streaming on smartphones," in *Allerton*, pp. 220–227, IEEE, 2011.

[67] M. Neely, "Energy optimal control for time varying wireless networks," *IEEE Transactions on Information Theory*, vol. 52, pp. 2915–2934, July 2006.

[68] F. Foster, "On the stochastic matrices associated with certain queuing processes," *The Annals of Mathematical Statistics*, vol. 24, no. 3, pp. 355–360, 1953.

[69] "Network coding utilities." Library available at `http://arni.epfl.ch/software`.