Extending REACT to Support Quality of Service: Algorithms and Implementation, With Screening and Performance Experiments on a Wireless Testbed

by

Daniel J. Kulenkamp

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2021 by the
Graduate Supervisory Committee:

Violet R. Syrotiuk, Chair
Charles J. Colbourn
Ilenia Tinnirello

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

REACT is a distributed resource allocation protocol that can be used to negotiate airtime among nodes in a wireless network. In this thesis, REACT is extended to support quality of service (QoS) airtime in an updated version called $\text{REACT}_{\text{QoS}}$. Nodes can request the higher airtime class to receive priority in the network. This differentiated service is provided by using the access categories (ACs) provided by 802.11, where one AC represents the best effort (BE) class of airtime and another represents the QoS class. Airtime allocations computed by $\text{REACT}_{\text{QoS}}$ are realized using an updated tuning algorithm and $\text{REACT}_{\text{QoS}}$ is updated to allow for QoS airtime along multi-hop paths. Experimentation on the w-iLab.t wireless testbed in an ad-hoc setting shows that these extensions are effective. In a single-hop setting, nodes requesting the higher class of airtime are guaranteed their allocation, with the leftover airtime being divided fairly among the remaining nodes. In the multi-hop scenario, $\text{REACT}_{\text{QoS}}$ is shown to perform better in each of airtime allocation and delay, jitter, and throughput, when compared to 802.11. Finally, the most influential factors and 2-way interactions are identified through the use of a locating array based screening experiment for delay, jitter, and throughput responses. The screening experiment includes a factor on how the channel is partitioned into data and control traffic, and its effect on the responses is determined.

DEDICATION

This thesis is dedicated to my best friend Becki. I'm lucky to have you in my life, and I'm not sure where I'd be without you.

ACKNOWLEDGMENTS

First, I have to acknowledge my fantastic committee chair Dr. Violet Syrotiuk. Thank you for always being more confident in me than I am! Without you I don't think I would have gone after a graduate degree, let alone a thesis. I'm lucky I took your operating systems class.

Next, Dr. Charles Colbourn, for constantly challenging me, bending my brain in your classes (always a good time!), and always having a good perspective. Thanks for always making me think!

I also thank Dr. Ilenia Tinnirello for being so warm and welcoming when I visited a few summers ago, and for coming up with the idea for chapter five of this thesis. I think the results worked out pretty nicely and they wouldn't have happened without you!

Lastly, thank you to everyone who has supported me over the years and who made it possible for me to get to this point. I'm grateful for all your support.

This thesis was supported in part by the U.S. National Science Foundation (NSF) NeTS Award 1813451.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

In the modern world, wireless networks are everywhere and are used by billions of people. According to the Cisco Annual Internet Report for 2018 to 2023, the 5.1 billion mobile subscribers in 2018 are predicted to grow to approximately 5.7 billion by 2023 [1]. Connected devices can range from laptops or desktops connected through a home *wireless local area network* (WLAN), to a cell phone connected to a cell tower using 5G, to a low power Internet of Things sensor. With the emergence of 5G, new applications that require *quality of service* (QoS) support are possible including ultra reliable, low latency applications, such as autonomous vehicles and high capacity applications such as virtual reality. However, 5G has some limitations, depending on the technology used, such as the need for line of sight with the cell tower. These hold it back from being the complete solution for every situation. WLANs have become the access method of choice for the home, the workplace, and public spaces such as libraries or coffee shops. They are envisioned as an offloading solution for 5G networks, where the 5G infrastructure is unavailable or overloaded [2]. However, this means that WLANs also need to support QoS at a level comparable to 5G.

One mechanism for achieving QoS support in WLANs is airtime allocation. *Airtime* is the amount of time the channel is sensed busy due to frame transmissions [3]. This means that the amount of time a device spends transmitting is its airtime allocation. In order to provide QoS support in a wireless setting, we need to ensure that applications that require a higher airtime allocation than others get it, if possible. To provide guarantees on metrics such as delay, airtime allocations need to be consistent and unaffected by other transmitting devices.

The most common channel access method for WLANs today is the Wi-Fi standard, or IEEE 802.11 [4]. Its main mechanism is the *distributed coordination function* (DCF), a carrier sense multiple access protocol with collision avoidance (CSMA/CA). The IEEE 802.11e amendment introduced traffic prioritization in the enhanced distributed channel access (EDCA) protocol using *access categories (ACs)* for different traffic classes [5]. Each class has its own contention window values, allowing for higher priority traffic to be transmitted in the network before lower priority traffic. The QoS support provided by IEEE 801.11e is still limited, as there are no guarantees on delay, jitter, throughput, or other metrics.

IEEE 802.11 does not allocate airtime to individual nodes; rather, each node attempts to gain access to the channel as much as possible. A distributed resource allocation protocol, REACT negotiates an airtime allocation among neighboring nodes, where nodes concurrently *demand* and *offer* airtime [6]. If the sum of all of the nodes' demands is more than is available, the remaining airtime is divided equally among the nodes for fairness. Once REACT has converged on an allocation for a node, the allocation must be realized. The original work on REACT proposed a scheduled MAC protocol where the allocation corresponded to slot assignments, implemented in simulation [6]. Later work used a contention-based method, tuning the contention window using an algorithm based on renewal theory (RENEW [3]), and with a control theoretic approach (SALT [7]). Through experimentation on a testbed with these two tuning approaches, REACT had lower delay and jitter statistics than IEEE 802.11, with only a relatively small reduction in throughput [7].

We propose an extension to REACT, $\text{REACT}_{\text{QoS}}$, which extends the original algorithm to support two different classes of airtime, QoS airtime and *best effort* (BE) airtime. Using the same concept of demanding and offering airtime, $\text{REACT}_{\text{QoS}}$ allows nodes to request and receive QoS airtime, which is prioritized by the algorithm.

Any remaining airtime is distributed among the remaining BE nodes as in the original algorithm. We study $\text{REACT}_{\text{QoS}}$ in an ad-hoc wireless network setting because REACT was originally designed, implemented, and evaluated in an ad-hoc scenario as well.

The contribution of this work is summarized here. First, the REACT algorithm and the SALT tuning algorithm are extended to support differentiated traffic. Secondly, $\text{REACT}_{\text{QoS}}$ is implemented and also updated to allow for dynamically changing demands, where prior work focused on static demands. Next, to improve the performance of the tuning, a traffic shaping mechanism using Linux *traffic control* (`tc`) is introduced to prevent nodes from obtaining higher airtime than they are allocated. Multi-hop $\text{REACT}_{\text{QoS}}$ is enabled through a reservation server, allowing flows to reserve airtime along a multi-hop path, and enabling QoS support in more complex networks. We demonstrate that the new algorithm can successfully allocate differentiated airtime for nodes requesting a higher priority of service, while equally sharing remaining airtime among BE nodes. Finally, a set of screening experiments is conducted to uncover the most influential factors and 2-way interactions in our ad-hoc scenario with REACT. This experiment is conducted as part of an investigation into REACT control parameters.

The rest of this thesis is organized as follows. We begin by providing some background and more in depth explanations of the 802.11 Wi-Fi standard and past work on REACT, as well as the evaluation environment and an overview on screening experiments in Chapter 2. Then we present the QoS extensions to REACT in Chapter 3 and describe the implementation in detail. In Chapter 4 we discuss the design of the evaluation experiments for $\text{REACT}_{\text{QoS}}$ under single-hop settings and in Chapter 5 we discuss the experiments in a multi-hop setting. The screening experiments are

discussed and analyzed in Chapter 6. Finally, we describe our conclusions as well as next steps for REACT in Chapter 7.

Chapter 2

RELATED WORK

Supporting QoS in wireless networks is a difficult problem. Many attempts have been made, and no one solution is the best for all situations. In this chapter we give some background on the current 802.11 Wi-Fi standard and the past work on REACT. We also describe the testing environment used for experimentation, and give an overview of screening experiments using locating arrays which is a technique we use to screen for significant factors in our experimentation.

2.1 802.11 Wi-Fi Standard

The 802.11 Wi-Fi standard is the dominant protocol used today in WLANs [4]. As mentioned in Chapter 1, the main mechanism of the protocol is the distributed coordination function, or DCF. As mentioned in the introduction, DCF is a CSMA/CA. CSMA protocols with collision avoidance differ from those with *collision detection* (CSMA/CD) due to the underlying limitations of the technology. Technologies that uses CSMA/CD do not try to avoid collisions because it is possible for them to detect a collision in the middle of sending data. One such example is the Ethernet protocol (802.3) [8]. WLANs using half-duplex technologies, however, cannot simultaneously transmit and receive; this means that they cannot detect if a collision is occurring during a transmission. In fact, once a transmission is started, 802.11 carries it out until it is complete. This could result in extreme under-utilization of the channel, hence the need for collision avoidance.

The DCF protocol operates by competing for channel resources. It maintains a *contention window* (CW), from which a backoff counter is randomly selected. The

backoff counter must count down to zero before attempting to transmit, and once it does, DCF checks to see if the channel is busy. If the channel is quiet, it can then attempt its transmission after waiting a period of time called the *distributed interframe space* (DIFS). When a station receives a transmission, it sends an acknowledgement after waiting a period called the *short interframe space* (SIFS). The SIFS, plus the propagation delay, is shorter than the DIFS, so no stations are able to detect the station idle and attempt to transmit during the SIFS. Acknowlegements are required because half-duplex nodes cannot both listen and transmit at the same time.

Collisions are detected if a node does not receive an acknowledgement within a timeout period. If a collision is detected, the size of the contention window is doubled until it reaches its maximum value. This increases the probability that the next backoff counter value chosen from the CW is larger than before. If a transmission is successful, the contention window is reset to its minimum value, giving that node a higher chance of being able to transmit again sooner. The minimum and maximum values are hardware dependent, but typical values include $CW = [15, 1023]$. The use of the contention window (and randomly selecting values from it), reduces the chance of a collision between nodes. This operation of the protocol is called the two-way handshake and is the basic access mechanism of the protocol. In addition, 802.11 also has a four way handshake that allows nodes to send a *request to send* (RTS) message to "reserve" the channel for a transmission. On reception of an RTS message, a node responds with its own *clear to send* (CTS) message letting the original node know it can start its transmission. Then the node can transmit, and receives an acknowledgement once the data has been received. The RTS/CTS messages contain information about the transmission, such as the length of the packet being transmitted. This allows nearby nodes to stay quiet during the length of the transmission, ensuring that longer packets can be sent without collisions. 802.11 has been shown to

perform fairly well, and the RTS/CTS mechanism tends to perform better than the basic access mechanism [9]. However, it can also be a source of unfairness [10]. This is due to the fact that upon a successful transmission the CW is reset to its minimum value, but when a collision is detected the window is doubled. This can lead to nodes having successful transmission having a higher chance to transmit sooner, where unsuccessful nodes have a lower chance.

The 802.11e amendment, introduced in 2005, proposed the enhanced distributed coordination access (EDCA) protocol, which uses different *access categories* (ACs) to provide traffic differentiation in WLANs [5]. The amendment outlines four ACs, for different types of traffic, each with an increasing level of priority: BK for background traffic, BE for best effort traffic, VI for video traffic, and VO for voice traffic. EDCA still uses DCF to operate, but instead of just one CW, it now uses one for each AC. Now, instead of just competing with other nodes attempting to transmit, there is contention among the different ACs. Each AC has a different value for its maximum and minimum CW sizes, with the higher classes having much smaller maximum CWs than the lower classes. Additionally, EDCA introduces the *arbitration interframe space* (AIFS), which is a period of time the AC has to wait after the backoff counter has expired and before attempting to transmit. Again, the higher classes have smaller AIFS values than the lower ones, giving them higher priority since they can attempt to access the channel sooner. If an intra-node collision is detected between two ACs, the AC with the higher priority gets to transmit and the lower AC behaves as if it detected an inter-node collision.

Although EDCA provides some level of QoS support, there are still no guarantees on delay, jitter, or other statistics. It essentially amounts to prioritizing certain traffic over others with no guarantees. An alternative protocol enhanced improved delay and jitter characteristics is described next.

2.2 REACT

REACT is a distributed resource allocation protocol that uses the metaphor of an auction to decide an airtime allocation for the network. Initially described in [6], REACT operates in an ad-hoc setting with each node running both an *auctioneer* and *bidder* program. The auctioneer auctions off a certain *resource* which has a given *capacity*, while the bidder maintains a *demand* on that resource. The auctioneer attempts to satisfy all of the demands for both itself and its neighbors. If the total demands exceed the available capacity, it splits the remaining airtime equally among the nodes. In this way, the airtime allocation is fair among the nodes. Dividing the remaining airtime equally is a choice; other divisions are possible as well.

Deciding on an airtime allocation for nodes in the network is only the first step; after receiving an allocation, nodes need to be able to *realize* their allocation. Several techniques for realizing an allocation have been researched. In the first version of REACT, the allocations decided by REACT are realized by using a random scheduled MAC protocol called ATLAS [6]. After allocations are determined by REACT, ATLAS randomly schedules each node a percentage of slots in which to transmit, based on the allocation computed by REACT. ATLAS was shown in simulation to have better delay than 802.11.

Later work focuses on implementing REACT in a testbed, and due to the difficulties implementing a scheduled MAC protocol in hardware, they are focused on contention-based approaches. These approaches focus on tuning the contention window in order to achieve the airtime allocation desired. The idea of tuning works because the channel access probability depends on the average size of the contention window [11], and has been successfully done in non-REACT settings such as in [12]. There, the authors use a tuning mechanism to achieve the theoretical maximum

throughput, ignoring delay and jitter performance. In contrast, REACT wants to improve the delay and jitter performance without taking throughput into account. For that purpose, one such tuning approach is RENEW, which is based on renewal theory [3]. The goal of this approach is to tune the contention window to achieve the airtime allocation computed by REACT. RENEW relies on the total airtime achieved, the amount of time the channel is frozen, the number of channel accesses, and the previous contention window size to predict the size of the next contention window. In this way, the allocation can be realized in a contention-based manner. In experimentation, this technique was successful in achieving the airtime allocations from REACT [3].

Another tuning algorithm, called SALT, is based on a control theoretic approach [13]. The SALT tuner first observes the current airtime over a given period, and uses exponentially weighted smoothing technique to predict the size of the contention window for the next period. The equation used to calculate the next CW value, S_t, is given in Equation 2.1 (for values of $t > 0$; if $t = 0$, the CW is set to its default values of $[15, 1023]$). The value of β is constrained to the interval $[0, 1]$ and is used to control how much past measurements influence S_t. The terminology and equations are taken from [13].

$$S_t = \begin{cases} a_1 & \text{if } t = 1 \\ \beta a_t + (1 - \beta) S_{t-1} & \text{if } t > 1 \end{cases} \tag{2.1}$$

Once S_t is calculated, the smoothed airtime is scaled by a factor k to transform it from a percentage to an actual CW value, C_t. SALT then sets $CW_{min} = CW_{max} = C_t$, to ensure the CW value is enforced (i.e., CW values are now chosen from the range $[C_t, C_t]$, so only one value is available). Using this approach, REACT can realize an

allocation using the SALT tuner. SALT has been shown to perform well, yielding superior delay and jitter characteristics when compared to the 802.11, with only a small reduction in throughput [7].

2.3 w.iLab-t testbed

All experiments in this thesis are conducted using the w.iLab-t wireless testbed in Zwijnaarde, Belgium. The testbed was developed by the CREW project and is available for use in research experimentation [14]. There are around 40 ZOTAC type nodes, each of which are equipped with two 802.11abgn antennas which use the Atheros `ath9k` driver. In this work, the `ath9k` driver is extended to both expose the contention window values to the user, and to allow them to be set to values other than powers of two. In order to run experiments on complex topologies, an experiment was first conducted to determine which nodes are in transmission range, allowing a graph representing connectivity to be established. The transmission power of the nodes is set to its minimum value, and pings were sent between each of the nodes in the testbed to determine which have solid connections. The experiment is the same as was conducted in [13]. All experiments were run on channels that had little traffic to ensure minimal interference from other transmitting nodes. In addition, all the ZOTAC nodes were reserved during experiments, although other node types with Wi-Fi cards are available. As with all wireless experimentation, not all interference can be avoided.

2.4 Screening Experiments

In addition to the work done on achieving differentiated airtimes, this thesis also focuses in part on screening experiments in wireless networks. The screening experiments are intended to aid in both research and experimentation. As such, some

background on experimental design, screening experiments, and locating arrays is presented here. The screening experiments specific to this work are presented in Chapter 6, as are the motivations for conducting them in this scenario.

2.4.1 Experimental Designs

In this section, we define terminology used to describe experimental designs. These definitions were adapted from [15].

Suppose that a system under study has k parameters, $P_1, \ldots, P_k$, and that each parameter P_j has a set $V_j = \{v_{j,1}, \ldots, v_{j,\ell_j}\}$, of ℓ_j possible values. A *test* is an assignment of a value from V_j to P_j, for each parameter $j = 1, \ldots, k$. An *experimental design* (or, design) is a collection of tests.

When a design has N tests (i.e., of size N), it is represented by an $N \times k$ array $A = (a_{ij})$ in which each column j corresponds to a parameter and each row i corresponds to a test. When run on the system, a test yields the measurement of one or more performance metrics. An *experiment* consists of running each test in the design.

A *full-factorial design* has tests that include all possible combinations of *levels* (a value of a parameter) for every parameter used in the experiment. The size of such a design is equal to the product of $|V_j|$ for each parameter $j = 1, \ldots, k$. All significant parameters and t-way interactions for $t = 1, \ldots, k$ can be identified using *analysis of variance* (ANOVA).

Another experimental design, called a *supersaturated design (SSD)*, is much smaller than a full-factorial design; the number of parameters is $k = N - 1$, where N is the number of tests [16]. The much smaller size of supersaturated designs makes them much more feasible in experimentation than a full-factorial design. However, they are only used for identifying factors, not interactions [16].

In order to discover interactions with many fewer tests than full-factorial designs, covering arrays can be used. Again, we borrow the definition from [15]. An assignment of values to any subset $t \leq k$ of the parameters is a *t-way interaction*. A *covering array* of strength t is an $N \times k$ array in which for every $N \times t$ subarray, each t-way interaction occurs in at least one test.

In order to demonstrate covering arrays (and later, locating arrays), we use an example of an experiment containing four factors: A and B have two values each $\{0, 1\}$, and factors C and D have three values each $\{0, 1, 2\}$. An example covering array of strength 2 is shown in Table 2.1. There are 37 two-way interactions possible, and these nine tests cover all of them. An example of a two-way interaction is where A is set to 0 and C is set to 2 (or, A_0C_2, for short) occurs in at least one test (test 5 in this case).

Table 2.1: Covering Array (Left) and Locating Array (Right)

Test	A	B	C	D
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	2
5	0	1	2	1
6	1	0	2	2
7	1	1	0	2
8	1	1	1	1
9	1	1	2	0

Test	A	B	C	D
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	1
5	0	0	2	2
6	0	1	0	2
7	0	1	1	2
8	0	1	2	0
9	0	1	2	1
10	1	0	0	2
11	1	0	1	2
12	1	0	2	0
13	1	0	2	1
14	1	1	0	0
15	1	1	0	1
16	1	1	1	0
17	1	1	1	1
18	1	1	2	2

Covering arrays are useful to ensure that every interaction of strength t is covered, however, they make no guarantees that it is possible to distinguish the effects of different interactions. For example, if the response for test 5 of the covering array is an anomaly, it is not be possible to distinguish which interaction is responsible—A_0B_1, A_0C_2, or C_2D_1—because all three occur only in test 5. In order to overcome this limitation, locating arrays can be used instead.

A (d, t)-*locating array* (LA) is a covering array of strength t, with an additional *locating property*: any set of d interactions each involving t parameters can be distinguished from any other such set by appearing in a distinct set of tests [15]. An example of a $(1, 2)$-locating array for the same factors A-D is shown in Table 2.1 (taken from [15]).

The locating array in Table 2.1 distinguishes the three interactions in test 5 of the CA that the covering array cannot. Each of the three interactions occur in at least one unique row of the locating array: A_0B_1 occurs in rows $\{6, 7, 8, 9\}$, A_0C_2 in rows $\{5, 8, 9\}$, and C_2D_1 in rows $\{9, 13\}$.

The size of locating arrays is larger than for covering arrays, although it is much smaller than the full-factorial design, since they grow logarithmically. For larger examples, such as in the experiment run in §6.3, the benefits of the smaller size of locating arrays is more apparent. This is discussed further in Chapter 6.

2.4.2 Experimental Analysis

One important consequence of using locating arrays instead of standard experimental designs is that analysis is more difficult. This is because standard experimental designs are typically *balanced*, which means each interaction occurs the same number of times. Locating arrays are often unbalanced, and so a specialized tool developed for this purpose is used [17]. The analysis tool uses a level-wise screening method

to identify the significant factors and 2-way interactions using two algorithms [17]. The first is a breadth-first search algorithm that uses orthogonal matching pursuit to identify a number of models that are 'best' explanations of a response. Each factor and interaction, also called effects, has a score based on its contribution to the R^2 of the model. The second algorithm aggregates the effects of the generated models and reports them in non-increasing order by their scores. The ranking output by the screening algorithm is what is used to determine which are the most influential. It is important to note that the R^2 score can sum to more than one. This is because the score is based on the aggregate contribution of that factor to the R^2 across all of the individual models generated. This analysis program provides the tools needed to conduct screening experiments using locating arrays and to discover the influential factors and 2-way interactions in a system.

2.5 Summary

In this chapter we provided background on the various protocols, techniques, and technology that supports the work done in this thesis. We discussed the 802.11 Wi-Fi protocol, which is the dominant MAC protocol for WLANs, as well as REACT which is an alternative protocol that provides improved fairness over 802.11. We provided a high-level overview of the testbed and testing environment and concluded with a section on screening experiments using locating arrays. Now we move on and describe our updated REACT algorithm, which we do in the next chapter.

Chapter 3

$\text{REACT}_{\text{QOS}}$

In this chapter we describe the extensions made to the REACT algorithm, the new version that we call $\text{REACT}_{\text{QoS}}$. In §3.1 we present Algorithms 1 and 2 which together compose the $\text{REACT}_{\text{QoS}}$ algorithm. In §3.3 we describe how $\text{REACT}_{\text{QoS}}$ is implemented as well as the extensions for allowing demands to be changed dynamically. We describe the new traffic shaping mechanism to assist the tuning algorithm in §3.4, and finally give the updated reservation algorithm used to support multi-hop flows in §3.6. $\text{REACT}_{\text{QoS}}$ was first presented in [18], along with Experiments 1-4.

3.1 QoS Extensions to REACT

As mentioned in §2.2, REACT negotiates the airtime allocations for nodes in a wireless network. The idea behind QoS support in $\text{REACT}_{\text{QoS}}$ is to provide two classes of airtime that nodes can request. This allows some nodes to receive a higher airtime allocation for their applications. In an ad-hoc setting, each node runs both an *auctioneer* and a *bidder* algorithm. It maintains the list of *offers* and *claims*, respectively, for itself and its adjacent auctions (neighbors).

We define two traffic classes: QoS and BE. In the QoS class, a node is guaranteed its allocation if available, while the BE class offers no guarantees on how much airtime a node receives. In the single-hop experiments described in Chapter 4, all traffic from a node is treated as a single flow. Therefore, each node is considered either a QoS node or a BE node, and must select whether it is requesting QoS airtime or BE airtime. In contrast, in Chapter 5 nodes have to forward traffic from both a BE and a

Algorithm 1 REACT Bidder for node i.

1: **upon** initialization **do**
2: $\alpha_i \leftarrow \emptyset$ ▷ set of neighboring auctions of bidder i
3: $B_i^O \leftarrow \emptyset$, $Q_i^O \leftarrow \emptyset$ ▷ set of BE, QoS offers
4: $q_i \leftarrow 0$, $b_i \leftarrow 0$ ▷ QoS, BE demand for bidder i
5: **upon** receiving a new demand magnitude (q_i, b_i) **do**
6: UpdateClaim()
7: **upon** receiving offer (x_j, y_j) from auctioneer j **do**
8: $Q_i^O[j] \leftarrow x_j$
9: $B_i^O[j] \leftarrow y_j$
10: UpdateClaim()
11: **upon** bidder i joining auction j **do**
12: $\alpha_i \leftarrow \alpha_i \cup j$
13: UpdateClaim()
14: **upon** bidder i leaving auction j **do**
15: $\alpha_i \leftarrow \alpha_i \setminus j$
16: **procedure** UpdateClaim()
17: **for** offer $\in Q_i^O$ **do**
18: **if** offer $< q_i$ **then**
19: $q_i \leftarrow 0$
20: *QoS claim* $\leftarrow q_i$
21: *BE claim* $\leftarrow \min(\{offers[j] : j \in B_i^O\}, b_i)$
22: **send** (*QoS claim*, *BE claim*) to all auctions in α_i

QoS flow, so two flow classes are available. We now present the bidder and auctioneer algorithms that form $\text{REACT}_{\text{QoS}}$.

Algorithm 1 presents the $\text{REACT}_{\text{QoS}}$ bidder. Each bidder i maintains three sets: B_i^O is the set of BE offers, Q_i^O is the set of QoS offers, and α_i is the set of nearby auctions for bidder i. Additionally, the bidder keeps track of two variables, q_i and b_i which are its QoS and BE *demands* or *claims*, respectively. Initially, all sets are empty and demands are zero.

Initially, for the single-hop experiments, nodes are considered as either QoS or BE nodes, so only one of the two demand types is allowed to be positive at one time. However, for the multi-hop experiments, it is necessary for each node along the path

to have both QoS and BE demands, therefore both are allowed to be positive. When the bidder receives a new offer from auctioneer j, it updates its sets accordingly. The main function for the bidder is UpdateClaim (lines 16-22). The node must check whether it has received QoS offers from each of its neighboring nodes, and that the offer is at least as large as the demand. If any of the offers from neighboring nodes is not as large as the demand, it sets q_i to zero. After it checks all neighboring offers, it sets *QoS claim* to q_i and *BE claim* to the minimum of all offers in B_i^O and b_i. Finally, it sends the tuple (*QoS claim*, *BE claim*) to all auctions in α_i.

Algorithm 2 presents the $\text{REACT}_{\text{QoS}}$ auctioneer. Similar to the bidder, the auctioneer maintains three sets: B_j^C is the set of BE claims, Q_j^C is the set of QoS claims, and β_j is the set of bidders at auction j. It maintains one variable c_j, the capacity of its resource, which is the airtime to auction at that node.

The main part of this algorithm is the function UpdateOffer (lines 9-36). In this function, two sets are maintained: R is the set of all satisfied QoS bidders, and C is the set of all satisfied BE bidders. As the algorithm works through claims, it adds bidders to these sets accordingly. The variable A_j is the remaining airtime that has not been allocated and is initially set to the capacity of auction j, c_j. The boolean flag *done* keeps track of when the algorithm has terminated.

Lines 13-15 check if all bidders have been satisfied or constrained by this auction; a bidder is *constrained* by an auction i if it cannot increase its claim based on a higher offer from a different auction j. Then, *BE offer* is set to the remaining airtime plus the maximum claim in B_j^C, to ensure that a bidder can increase its claim if it is no longer constrained by an adjacent auction. Lines 16-26 are where QoS claims are satisfied and whether BE claims need to be constrained.

$\text{REACT}_{\text{QoS}}$ first attempts to satisfy each of the QoS bidders, subtracting out their claims from the pool of available airtime (lines 18-23). If the claim is less than the

Algorithm 2 REACT Auction for node j.

1: **upon** initialization **do**
2: $\beta_j \leftarrow \emptyset$, $c_j \leftarrow 0$ ▷ set of bidders, capacity at auction j
3: $B_j^C \leftarrow \emptyset$, $Q_j^C \leftarrow \emptyset$ ▷ set of BE, QoS claims at j
4: **upon** bidder i joining auction j **do**
5: $\beta_j \leftarrow \beta_j \cup i$
6: **upon** bidder i leaving auction j **do**
7: $\beta_j \leftarrow \beta_j \setminus i$
8: UpdateOffer()
9: **procedure** UpdateOffer()
10: $C \leftarrow \emptyset$, $R \leftarrow \emptyset$ ▷ set of *satisfied* BE, QoS bidders
11: $A_j \leftarrow c_j$, *done* $\leftarrow$ `False`
12: **while** (*done* = `False`) **do** ▷ all bidders are satisfied
13: **if** $(R \cup C = \beta_j)$ **then**
14: *done* $\leftarrow$ `True`
15: *BE offer* $\leftarrow A_j + \max(\{claims[i] : i \in B_j^C\})$
16: **else**
17: *done* $\leftarrow$ `True`
18: **for** $q \in \{Q_j^C \setminus R\}$ **do**
19: *QoS offers*$[q] \leftarrow A_j$
20: **if** *claims*$[q]$ <= *QoS offers*$[q]$ **then**
21: $R \leftarrow R \cup q$
22: $A_j \leftarrow A_j -$ *claims*$[q]$
23: *done* $\leftarrow$ `False`
24: **else** ▷ cannot satisfy QoS demand
25: *QoS offers*$[q] \leftarrow 0$
26: *done* $\leftarrow$ `False`
27: **if** $|B_j^C \setminus C| > 0$ **then**
28: *BE offer* $= A_j / |B_j^C \setminus C|$
29: **else**
30: *BE offer* $= A_j$
31: **for** $b \in \{B_j^C \setminus C\}$ **do**
32: **if** (*claims*$[b] <$ *BE offer* **then**
33: $C \leftarrow C \cup b$
34: $A_j \leftarrow A_j -$ *claims*$[b]$
35: *done* $\leftarrow$ `False`
36: **send** (*QoS offers*, *BE offer*) to all bidders in $Q_j^C \cup B_j^C$

remaining airtime, we can satisfy this request and move the bidder to the set R (lines 20-23). We set `done` to `False`, because we may need to update the BE offer with the

new available airtime. If we cannot satisfy the claim (line 25), we set the offer for that bidder to zero, and set *done* to `False`. This notifies the node requesting QoS airtime that its demand cannot be satisfied, leaving the node to decide if it wants to make a smaller demand instead. However, this does not affect the nodes BE request; that is taken care of in lines 27-30.

In line 27, we check if there are any unsatisfied bidders, and if so, we divide the remaining airtime up among them; otherwise, we set it to the available airtime as all claims are satisfied. Finally, for each remaining unsatisfied bidder, if its claim is less than the BE offer, we can satisfy that bidder, subtract its claim from the remaining available airtime, and set `done` to `False` to ensure we iterate again to update the offers. Once all bidders are satisfied or constrained, we send the set of QoS offers and the BE offer to all bidders in $Q_j^C \cup B_j^C$.

Figure 3.1 gives an example of the operation of $\text{REACT}_{\text{QoS}}$. In this case, node 4 is a QoS node and is demanding 50% QoS airtime. The other nodes are demanding 100% BE airtime. In the figure, b_t gives node t's initial BE demand, q_t is node t's initial QoS demand, and s_t gives node t's ultimate airtime allocation. White backgrounds represent BE nodes, gray background represent QoS nodes, and double edged circles represent nodes whose airtime is constrained. The dotted lines represent bidirectional links between the nodes.

Figure 3.1: Example of $\text{REACT}_{\text{QoS}}$ on a Line Topology.

We see that node 4 receives its QoS request, but the rest end up claiming less than their BE request. Nodes 2 and 3 are constrained due to node 4's request, because

the auction at node 3 only has 50% of its airtime left over to offer as BE airtime. It splits it evenly between the remaining nodes at its auction (nodes 2 and 3), leaving each to claim half. Node 1 claims more airtime because node 2 offers 50%. Node 2 can offer 50% because after nodes 2 and 3 claim 25% each, it has 50% left over. No auction here is aware of all four nodes due to the line topology and so the algorithm relies on indirect information from neighboring auctions to determine allocations.

3.2 Updated SALT Tuner

In past work, the allocations computed by REACT were realized in several ways, as discussed in §2.2. For this work, we use the SALT tuning algorithm, which uses a control theoretic approach to achieve the airtime allocations [7]. However, because we now have two classes of airtime, the tuning algorithm is extended in order to tune two contention windows instead of one.

EDCA uses four different ACs to provide differentiated traffic and we utilize two of these ACs for our two classes of airtime: the QoS class uses the VI AC, and the BE class uses the BE AC. As in past work, control messages are enqueued to the VO AC, to ensure they are sent with the highest priority. We set the CW values for each AC according to the standard, which we present in Table 3.1. For $\text{REACT}_{\text{QoS}}$, these are the starting values, but they change as they are tunend by SALT. For the original version of REACT and for DCF, we set the values for all ACs to be: AIFS, 2; CW min, 15; CW max, 1023.

Table 3.1: Values for EDCA Access Categories

Queue	AIFS	CW min	CW max
VO	2	3	7
VI	2	7	15
BE	3	15	1023
BK	7	15	1023

In order to measure the airtime of each of these ACs, two statistics are collected from the nodes: the first is the amount of time the station has spent transmitting divided by the amount of time the station has been active (airtime of the node); the second is number of bytes sent for each AC, divided by the total number of bytes sent by all ACs. We multiply these two values together to achieve an approximation for the airtime per AC. We then use these approximations for the airtime to perform the tuning. Our updated SALT tuner uses essentially the same mechanism as before, except that now it maintains two smoothed airtime values, S_t^1 and S_t^2, instead of one. These two airtimes correspond to the two ACs being used, and are updated separately. At each interval, the CW values for both queues are updated simultaneously.

3.3 New Implementation

Several improvements and changes to the existing implementation of REACT were made to simplify development and ensure maintainability of $\text{REACT}_{\text{QoS}}$. For example, the original code base was implemented in Python 2, but as Python 2 has reached end of life as of 2020 [1] , the code base was upgraded to Python 3. This brought some challenges, as the newer version of Python handles multithreading differently. In order to ensure performance as well as enable new, dynamic demands, REACT was rewritten to use a producer-consumer model, where messages are passed between threads using queues.

Figure 3.2 shows the implementation of and interactions between threads. There are four threads: the manager thread, used for running the algorithms and coordinating the message passing between the threads as shown in the figure; a sniffer thread, used to listen for $\text{REACT}_{\text{QoS}}$ control messages; a sender thread, used to send out $\text{REACT}_{\text{QoS}}$ control messages; and a CW updater which sets the CW according to the

[1] https://www.python.org/dev/peps/pep-0373/

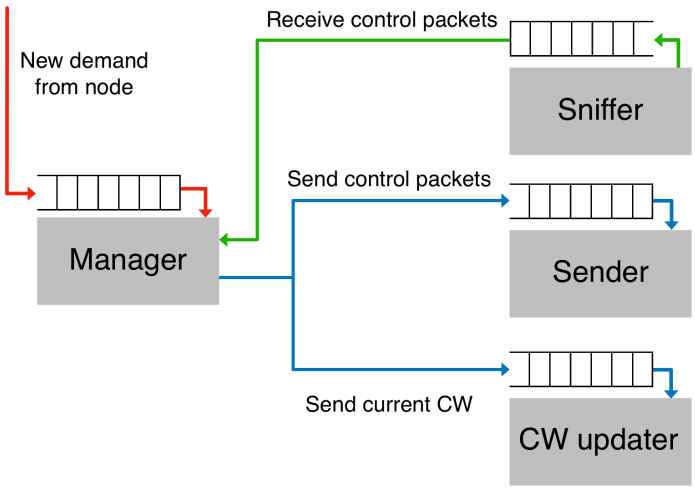


Figure 3.2: Interaction of Threads in REACT$_{\text{QoS}}$ Implementation.

tuning algorithm. Messages are passed between the four threads using the queues. The size of the CW's queue is set to one, to ensure that the updater has the most recent CW value (i.e., if there is a value already there, it is removed and replaced by the newer value). The fourth queue is used to pass new demands from outside the REACT$_{\text{QoS}}$ object, allowing a script to dynamically adjust demands. An example of this in action is provided in Figures 4.7 and 4.8 in Chapter 4.

Figure 3.3 shows the high level overview of the REACT$_{\text{QoS}}$ architecture, and how it interacts with the various components. REACT$_{\text{QoS}}$ communicates directly with the driver functions, the SALT contention window tuner, and the traffic shaper (see §3.4). REACT$_{\text{QoS}}$ has two separate queues, one for data (TX-Q-DATA) and one for control messages (TX-Q-CTL). By separating the data from the control messages, we can ensure that control messages are sent with a higher priority, therefore ensuring the REACT algorithm can execute even when the network is under high load. In addition, as done in earlier experimentation, we only allow each node to offer 80% of the airtime for data, leaving 20% for the control messages being sent from the separate queue [3][7].

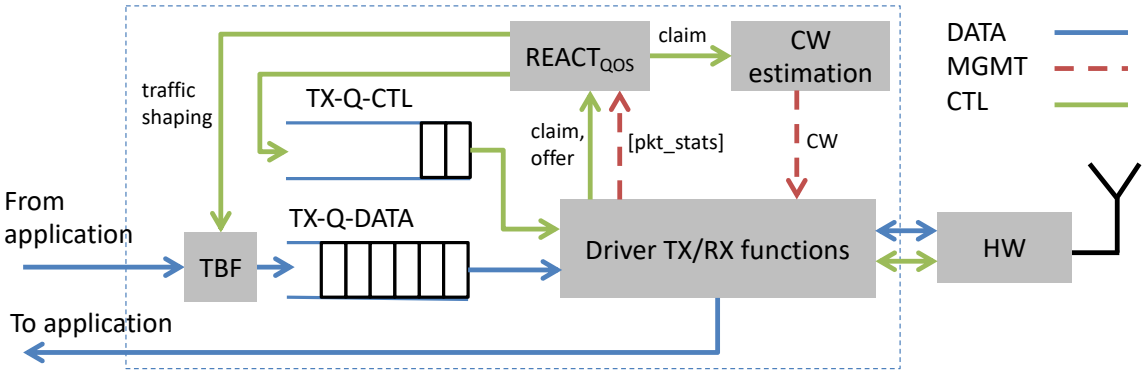


Figure 3.3: Architecture of REACT$_{QoS}$ Implementation.

Additionally, REACT$_{QoS}$ interacts directly with the driver to receive control messages and packet statistics, which it uses to conduct the auction as well as to tune the contention window. It passes the claim for its node to the CW estimation module, which sets the CW parameters in the driver.

3.4 Tuning with Linux `tc`

During development of REACT$_{QoS}$, it was discovered that the current SALT tuning algorithm has trouble ensuring varied allocations in the network. Specifically, if one node is receiving a higher allocation than the remaining nodes (for example if one node is requesting 50% QoS airtime and the others split the remaining airtime equally), then there will be drops that occur at that node where its airtime drops to zero for a period of time before returning to its allocation. An example of the issue can be seen in the airtime chart shown in Figure 3.4. In this scenario, an experiment was conducted where node zotacB3 received an airtime allocation of 44%, which it achieved for more than half of the experiment. However, around second 25 and second 90, the airtime dropped to zero. After about 10 seconds for the first drop and 25 seconds for the second drop, the airtime spiked as SALT attempted to correct the issue, before settling on the correct allocation. In this experiment, all nodes were

transmitting at the channel rate, which in this case was 6 Mbps. In past work, all nodes requested 100% of the channel and transmitted at the full channel rate and this issue did not occur; however, in this case nodes request less than the channel rate, yet still transmit at the channel rate. We speculate that the issue is caused by buffer overload at the QoS node along with the remaining nodes "stealing" more airtime than they have been allocated due to transmitting more than they request.

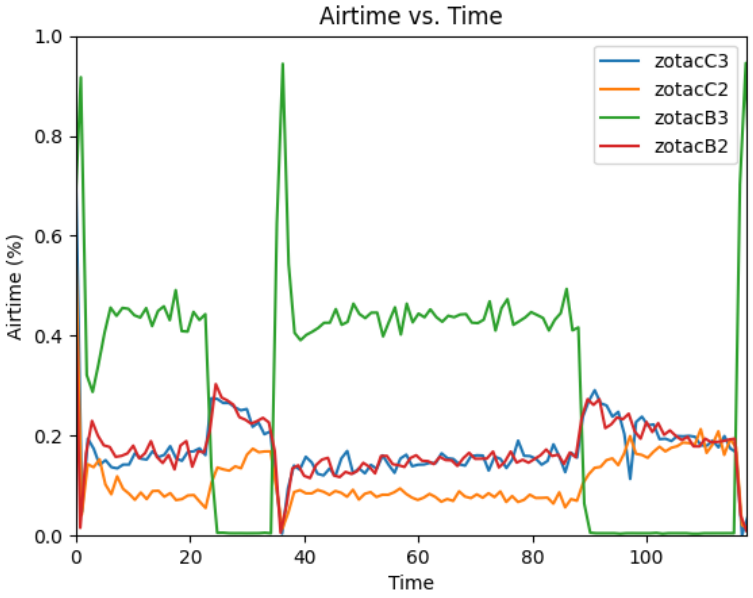


Figure 3.4: Example of Airtime Drops for REACT$_{\text{QoS}}$ Without Traffic Shaping.

In order to assist SALT in tuning the airtime at the nodes, a traffic shaper was introduced to ensure that nodes are not able to send more traffic than their allocation allows. In other words, if the theoretical limit of the channel is 10 Mbps, and a node was allocated 50% airtime, its approximate traffic rate should be 5 Mbps. REACT$_{\text{QoS}}$ uses this approximation to limit nodes from transmitting more than their allocation, to ensure that the node experiencing a drop can recover quickly. The traffic shaper can be seen in Figure 3.3 and operates on data traffic before it enters the queue (TX-

Q-DATA). This ensures that $\text{REACT}_{\text{QoS}}$ can use the shaper to control the rate at which the node sends traffic.

The alternative to introducing a traffic shaper is to require nodes to limit their flow rates to approximately what their allocations are. This is done in §4.4. For experimentation, this is slightly more complicated for static experiments, because nodes now need to adjust their flow rates according to $\text{REACT}_{\text{QoS}}$. However, for dynamic demands, discussed in §3.5, this is much more complicated because demands change throughout the duration of the experiment. In this case, the traffic shaper simplifies things considerably. Nonetheless, we know there are tradeoffs when using the traffic shaper and that future work needs to improve on this subject.

To implement the shaping, we selected a *token bucket filter* (TBF), specifically the `tbf` module from Linux traffic control [2] (`tc`). $\text{REACT}_{\text{QoS}}$ sets the rate of the TBF whenever the offers for the node change. In the case where there are both QoS and BE demands at a given node, the traffic shaper is set to the sum of the QoS and BE load approximation. It is important to note that the traffic shaping mechanism is not precise; its main purpose is to assist the tuning algorithm to ensure QoS nodes receive their allocation consistently. Additionally, we have noticed that the TBF can have a negative impact on delay and jitter characteristics. However, we leave tuner improvements to eliminate the need of a traffic shaper to future work.

3.5 Dynamic Demands

Another benefit of the updated implementation of REACT discussed in §3.3 is the ability to have dynamically changing demands in experimentation. The Python implementation of $\text{REACT}_{\text{QoS}}$ is written as an importable module, allowing the experimenter to change the demands programatically. This allows for more interesting

[2] https://www.man7.org/linux/man-pages/man8/tc-tbf.8.html

scenarios where nodes change their demands throughout the experiment to see more complex behavior from $\text{REACT}_{\text{QoS}}$. In addition, to simplify experimentation further, the experimenter can pass a JSON file to $\text{REACT}_{\text{QoS}}$ with a list of events (including the demand and the timestamp of the event) and $\text{REACT}_{\text{QoS}}$ adjusts the demands accordingly. An experiment including dynamic demands is given in §4.3.

3.6 Reservation Server Updates

In order to support multi-hop flows, each auction along the path must be aware of all flows passing through it. In past work, this was solved through the use of a reservation server which runs on all nodes in the network [7]. Nodes register their demand with servers along the path and the servers deduct the demand from the available capacity. REACT then uses the reservations to preallocate the airtime for those flows. In other words, REACT sets aside a certain amount of airtime for the reservations and only offers the remaining available capacity in its auction. We use the same technique to ensure multi-hop flows are considered in the $\text{REACT}_{\text{QoS}}$ auction. However, the algorithm must be extended to support the two traffic classes. The updated pseudocode is presented in Algorithm 3.

As before, the $\textsc{Reserve}(s, d, q, b)$ function is a recursive function that first attempts to reserve airtime at its node, then its neighbors, and finally forwards that reservation along the path. In lines 1 and 2, it checks if it is possible to satisfy both the QoS and BE amount and, if not, the reservation fails. If so, it reduces capacity by that amount (line 3). If the source matches the destination, this means that we are just reserving airtime, and not doing any forwarding (lines 4 and 5). Otherwise, we attempt to reserve the airtime at all neighbors (lines 8 and 9) and if we can, then we pass along the reservation to the next hop in the path (lines 10-14). Finally, if we were not able to reserve at all neighbors, we tear down all of the completed reserva-

Algorithm 3 RESERVE(S,D,Q,B)

```
1: if (capacity_s − q − b) ≤ 0 then
2:     return False
3: capacity_s ← capacity_s − q − b
4: if (s == d) then
5:     return True
6: else
7:     status ← True
8:     for each n ∈ NEIGHBORS(s) do
9:         status ← status and RESERVE(n, n, q, b)
10:    if (status == True) then
11:        if (NEXT_HOP(s) ≠ d) then
12:            return RESERVE(NEXT_HOP(s), d, q, b)
13:        else
14:            return True
15:    else
16:        tear down completed reservations made by neighbors
17:        of s and s itself for this multi-hop flow
```

tions made by this node and its neighbors, and the reservation fails. One additional improvement to this version of the reservation code, is that reservations can be made in either direction along a path.

One drawback to this implementation is that both the QoS and BE are considered together. There may be available capacity to reserve only one of the two, perhaps requiring the prioritization of the QoS reservation, but in our experimentation this is currently unnecessary. We make reservations that are feasible, and adding this functionality creates unneeded complexity for our experiments. Hence, we stay with the simpler algorithm.

3.7 Experiment Setup and Measuring Statistics

In order to evaluate the performance of $\text{REACT}_{\text{QoS}}$, it is necessary to conduct experiments and collect measurements to accurately and fairly compare the two pro-

tocols. In this subsection we describe the general overview of how experiments are conducted as well as how measurements are collected.

As discussed in §2.3, all experiments are conducted on the w.iLab-t testbed. Ubuntu 16.04 was installed on the nodes and the code was written using Python 3. Experiments were orchestrated using the Python `fabric` library [3] , which allows scripts to send commands to remote hosts. All of the code used in experimentation for this thesis can be found online [4] .

In order to perform the tuning necessary for SALT, two kernel extensions [5] need to be installed: one is used to provide an interface for exposing and manually setting the contention window values, while the other is used to allow contention window values to be set to values other than powers of two. These are necessary because SALT sets these values, and because the standard only allows setting CW values to powers of two; without the extension we limit how precisely SALT can tune.

Flows are sent between nodes using `iperf` which provides the necessary parameters for adjusting flow sizes, transport protocol, and TOS fields needed for experimentation. It is also used to collect throughput statistics for the various flows. To collect delay and jitter statistics, the `ping` utility is used. Pings are sent between the same nodes as the flows and the outputs are saved to disc. Jitter is calculated based on the delay measurements. Airtime is also collected for all experiments using the $\text{REACT}_{\text{QoS}}$ module, though for experiments using 802.11 the module is disabled and is only used to collect airtime statistics.

$\text{REACT}_{\text{QoS}}$ uses separate ACs to differentiate between traffic, so both `ping` and `iperf` packets must map to the correct AC. The `ath9k` driver maps the type of service

[3] https://www.fabfile.org

[4] https://github.com/danielkulenkamp/react80211

[5] https://github.com/mmellott/backports-cw-tuning

(TOS) field from an IP packet into the four ACs. REACT$_{\text{QoS}}$ uses three of the four available queues and so we make sure to set the field appropriately. For REACT$_{\text{QoS}}$ control messages, we use an association request header and append our messages to the header–association requests automatically map to the VO AC, so control messages are handled correctly. For QoS flows and pings, we set the TOS field to `0xa0` and for BE flows and pings we set it to `0x00`. These values are taken from the ranges described by the documentation for the `ath9k` driver [6] .

3.8 Summary

In this chapter we introduced REACT$_{\text{QoS}}$, the updated tuning mechanism, the traffic shaper, and the updated reservation server. We described the new implementation of REACT$_{\text{QoS}}$and the overall architecture, as well as giving an overview of the experimentation environment and how measurements are collected. In the next chapter, we describe our single-hop experiments and present the results.

[6]https://wireless.wiki.kernel.org/en/developers/documentation/mac80211/queues

Chapter 4

SINGLE-HOP EXPERIMENTS

In this chapter, we present the experiments that are designed to evaluate the performance of REACT$_{\text{QoS}}$ in a setting with single-hop flows (a setting with multi-hop flows is evaluated in Chapter 5). We begin the chapter by describing the topologies selected for use in experimentation and the logic behind choosing them in §4.1. Next, in §4.2, we provide an overview of the experiments conducted and we present the results from the experiments in §4.3. In §4.4 we detail an additional experiment demonstrating the performance of REACT$_{\text{QoS}}$ without the traffic shaper enabled and finally, we summarize the chapter in §4.5.

4.1 Topologies

In evaluating REACT$_{\text{QoS}}$ we conduct experiments on two different topologies: a complete topology of four nodes (i.e., fully connected) and a line topology of four

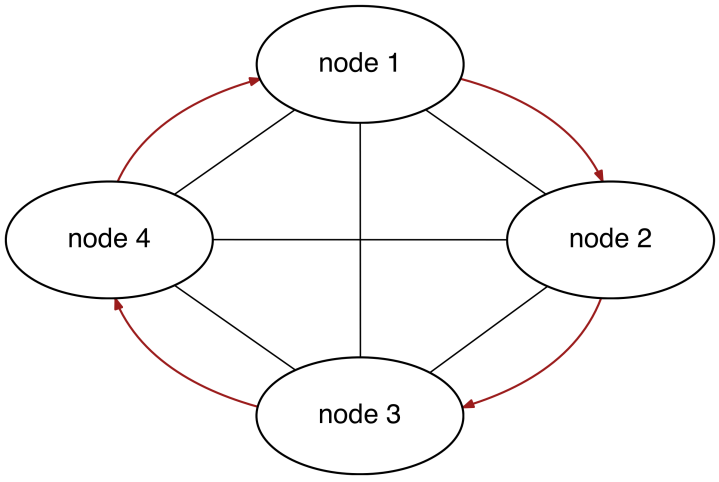


Figure 4.1: Complete Topology of Four Nodes With Flow Paths.

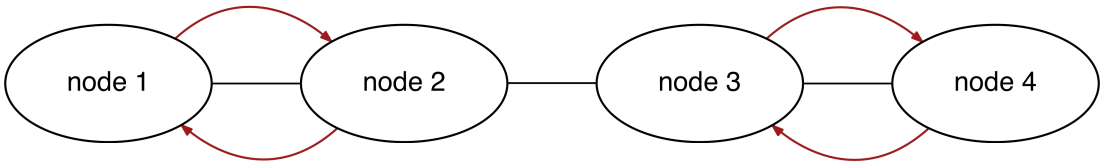


Figure 4.2: Line Topology of Four Nodes With Flow Paths.

nodes (i.e., each node can only hear one or two of its neighbors). These were selected in order to show the performance of REACT$_{\text{QoS}}$ in a scenario where each node has full information of all nodes' demands and offers, and one where nodes only have indirect information based on the messages from its neighbors. Additionally, the exposed terminal problem is present in the line topology which can cause degraded performance [19]. The complete topology is shown in Figure 4.1 and the line topology in Figure 4.2. In the figures, black lines represent connections between nodes and the red lines represent the direction of the flow paths in the experiments. For the complete topology, flows were sent in a circular fashion, while in the line topology flows were sent from the end nodes to their nearest neighbors, and from their nearest neighbors to the end nodes.

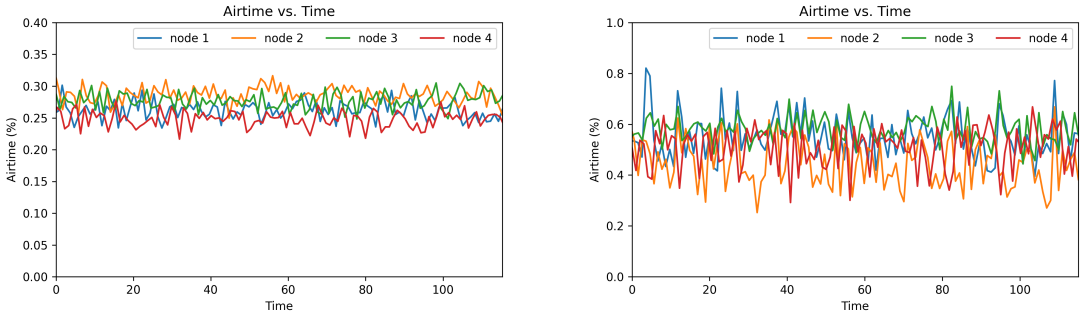
4.2 Description of experiments

Four experiments were designed to demonstrate the differences between the 802.11 standard, the original REACT algorithm, and the updated REACT$_{\text{QoS}}$ algorithm. Each experiment was run twice, once on each topology. In experiment 1, 802.11 DCF is the active protocol and the REACT$_{\text{QoS}}$ module is only running to capture airtime statistics. Experiment 2 was designed to demonstrate the performance of the original REACT protocol, so the QoS extensions are disabled. Next, experiment 3 consists of REACT$_{\text{QoS}}$ running and one of the four nodes requesting 50% of the available airtime. Finally, experiment 4 combines REACT$_{\text{QoS}}$ with dynamic demands enabled by the

new implementation. The events and their occurrence times are shown in Table 4.1. For the single-hop experiments, flows are set to a rate of 6 Mbps, and traffic shaping is active to ensure that nodes do not overwhelm the tuning mechanism. Experiments 1-3 were all run for a duration of two minutes, while Experiment 4 was run for five minutes.

4.3 Results of experiments

Beginning with experiment 1, we see that in the complete topology IEEE 802.11 performs quite well (Figure 4.3a). Each node is allocated about 25% of the airtime for the entire experiment run. For the line topology in Figure 4.3b, however, IEEE 802.11 has much more variable performance.

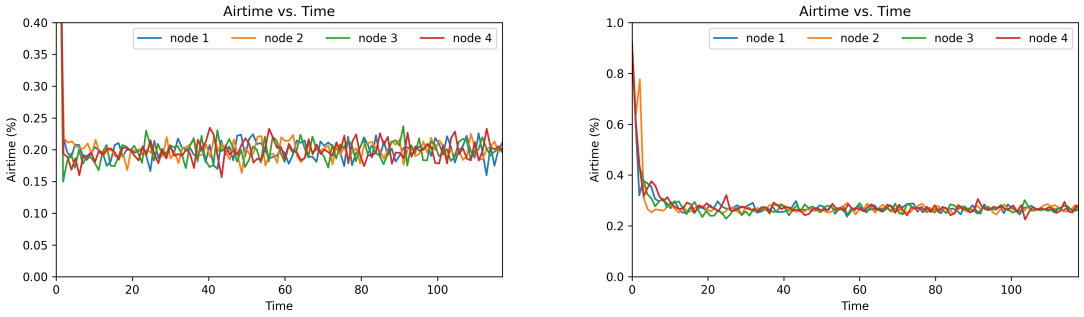


(a) 802.11 complete topology

(b) 802.11 line topology

Figure 4.3: Experiment 1: 802.11 Airtime for the Complete and Line Topology.

Experiment 2 provides an insight into how REACT performs. Here, each node is requesting 100% of the offered airtime (because 80% is offered, this results in about a 20% allocation per node). From Figures 4.4a and 4.4b we see that REACT converges on a much tighter airtime allocation, i.e., the means of the airtime of the four nodes are much closer. The airtime each node receives in REACT is less than under IEEE 802.11 (partially due to 80% of the airtime being allocated), but this much more consistent allocation is an improvement.



(a) REACT complete topology (b) REACT line topology

Figure 4.4: Experiment 2: REACT Airtime for the Complete and Line Topology.

To compare between the two approaches, we analyze the variance; however, to account for the period of time that REACT takes to converge, we only compute the variance for the time after REACT has converged (at second 3 for the complete topology; at second 10 for the line). For the complete topology, the variances for the nodes range from 0.017 to 0.471 for 802.11, while for REACT they range from 0.016 to 0.024. For the line topology, variances for 802.11 range from 0.3 to 0.877, while for REACT they range from 0.012 to 0.02. There is some improvement in the complete topology, but with the line topology the improvement is much more apparent, as can be seen when comparing between Figure 4.3b and Figure 4.4b.

In Experiment 3, node 4 requests 50% of the offered airtime as QoS airtime, which translates to 40% of the actual airtime, while the remaining nodes request 100% of the BE airtime. With the complete topology (Figure 4.5), node 4 receives about 40% airtime, while the remaining three nodes receive about 13% each. The airtimes for the line topology are more complicated to understand (Figure 4.6), however, this is the same discussion for the scenario presented in Figure 3.1 on page 19. As before, node 4 requests half of the offered airtime, which it correctly receives. Node 1 also received half of the offered airtime, because it is only constrained by the auction at node 2. Nodes 2 and 3 are constrained by multiple auctions, which results in them receiving

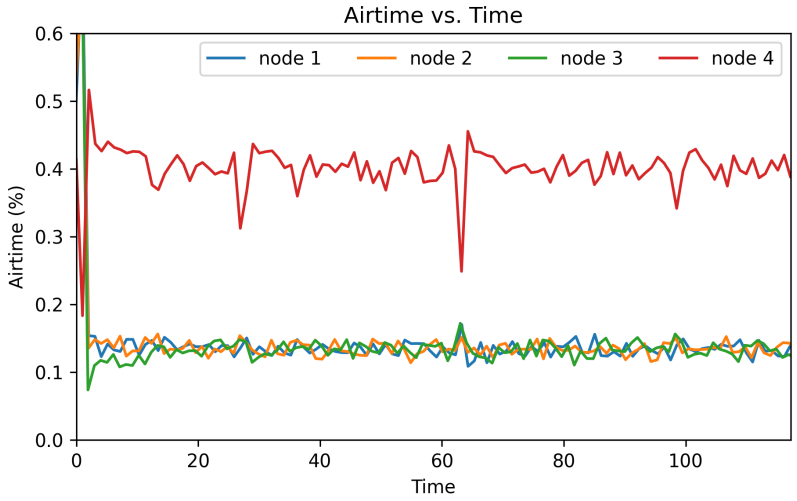


Figure 4.5: Experiment 3: $\text{REACT}_{\text{QoS}}$ in the Complete Topology.

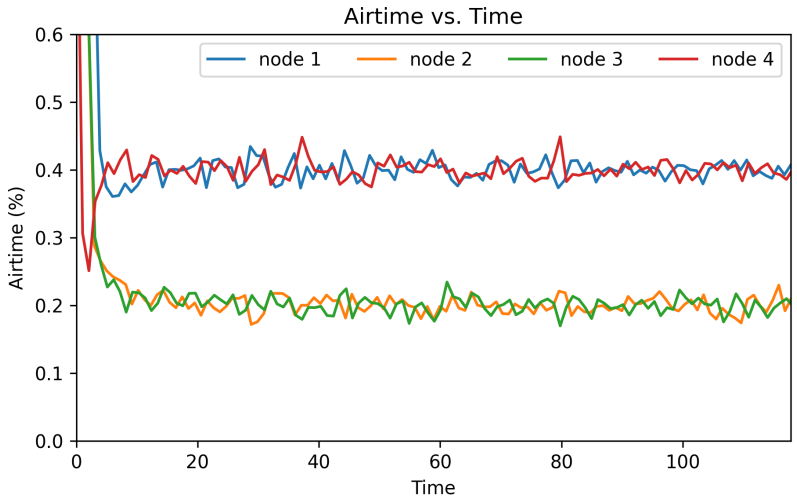


Figure 4.6: Experiment 3: $\text{REACT}_{\text{QoS}}$ in the Line Topology.

less airtime than either node 1 or node 4. This can be seen in Figure 4.6, where nodes 1 and 4 receive half the offered airtime (40%) and nodes 2 and 3 receive a quarter (20%). Additionally, $\text{REACT}_{\text{QoS}}$ takes four seconds to converge in the complete topology, and 8 seconds for the line topology. The line topology takes longer to converge because information has to travel two hops in the worst case, where a complete topology is by definition fully connected and information has to travel at most one hop.

Finally, for Experiment 4, Figures 4.7 and 4.8 show node 1 requesting 50% QoS airtime, with remaining nodes adjusting their demands dynamically according to the events in Table 4.1. In this experiment, node 1 has a consistent allocation throughout the experiment and is not affected by the demand changes at other nodes. In the complete topology, even when nodes 2 and 3 adjust their demands at seconds 120 and 180, the sum of the three BE demands is higher than 50% of the airtime split three ways.

Table 4.1: Events for Experiments 4

Time (s)	Node	Demand	QoS
0	node 1	50%	Yes
0	node 2	10%	No
0	node 3	10%	No
0	node 4	100%	No
60	node 2	20%	No
60	node 3	20%	No
120	node 2	50%	No
180	node 2	80%	No
180	node 3	50%	No

Therefore the airtime remains equally split between the three nodes, so no change is reflected in the figure. In contrast, in the line topology, we see that when node 2 increases its demand to 50% of the airtime at second 120, it is allocated more of the airtime, because there are fewer nodes competing for the airtime at node 2. Later at second 180, nodes 2 and 3 increase their demands further — they are both constrained

the neighboring auctions and have to split the airtime. Though $\text{REACT}_{\text{QoS}}$ took twice as long to converge initially in the line topology (20 s vs. 9 s in the complete topology), re-convergence times for both topologies averaged only four seconds.

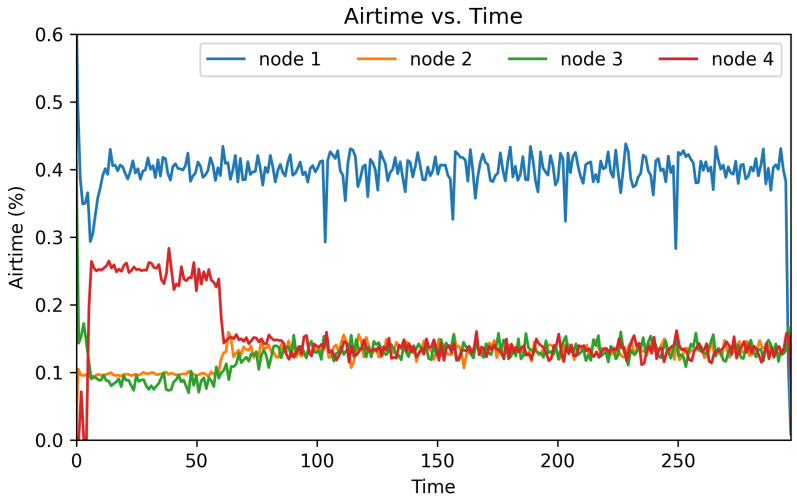


Figure 4.7: Experiment 4: $\text{REACT}_{\text{QoS}}$ in the Complete Topology with Dynamic and Varied Demands According to Table 4.1.

4.4 $\text{REACT}_{\text{QoS}}$ without Traffic Shaping

Traffic shaping has a negative impact on the delay and jitter characteristics of $\text{REACT}_{\text{QoS}}$, because the traffic shaper introduces artificial delay onto the packets. Additionally, the jitter is impacted based on how the traffic shaper dequeues the packets. The shaper's purpose is to ensure nodes do not transmit more than their channel allocation allows (therefore "stealing" airtime from the QoS node), so we can instead insist that nodes only sent flows of approximately the bandwith of their portion of the channel. Instead of having the shaper induce this requirement, we can require the nodes themselves to limit what they send.

Another experiment (Experiment 5) was run to demonstrate this behavior. Traffic shaping is disabled in $\text{REACT}_{\text{QoS}}$, but nodes use flow sizes that approximate their

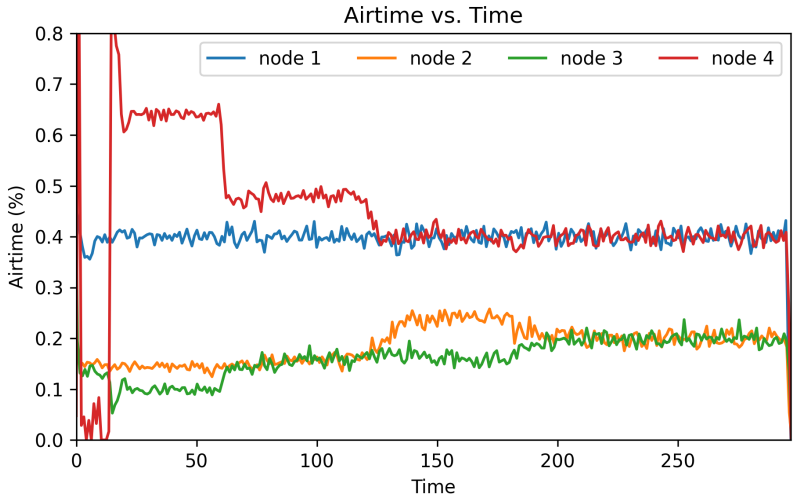
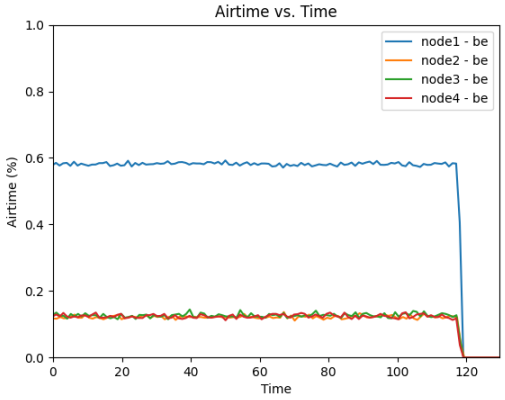


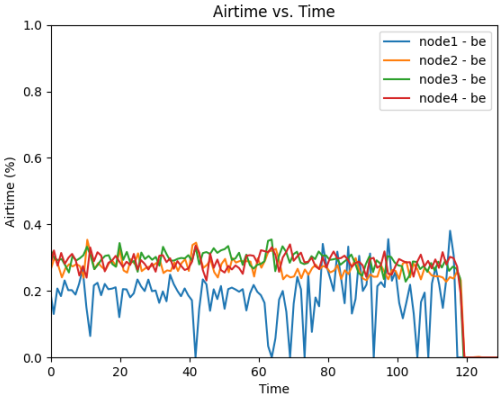
Figure 4.8: Experiment 4: REACT$_{\text{QoS}}$ in the Line Topology, Dynamic and Varied Demands According to Table 4.1.

eventual allocation. This allows us to view some of the same characteristics that REACT originally demonstrated; namely, that delay and jitter improve with a small reduction in throughput. This experiment is essentially the same as Experiment 2, but with flow sizes set according to the allocation and traffic shaping disabled. This experiment was run on the complete topology and delay and jitter measurements were collected. We compare REACT$_{\text{QoS}}$ to DCF and we do not consider it fair to only compare a scenario where the nodes reduce their transmisison rate for REACT$_{\text{QoS}}$, but not for DCF. Therefore, we run the experiment for REACT$_{\text{QoS}}$ and DCF with the QoS node transmitting at a rate of 3 Mbps, and the BE nodes transmitting at 1 Mbps. Additionally, we run DCF again with a rate of 6 Mbps for all nodes. This way we can see how REACT$_{\text{QoS}}$ works compared to both scenarios under DCF.

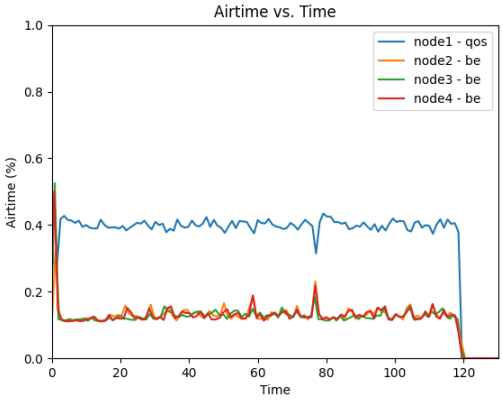
The airtime charts for both DCF runs are shown in Figures 4.9a and 4.9b, and the airtime chart for REACT$_{\text{QoS}}$ is shown in 4.9c. A comparison of the delay, jitter, and throughput measurements is given in Table 4.2.



(a) DCF Airtime Performance (small)



(b) DCF Airtime Performance (large)



(c) REACT$_{\text{QoS}}$ Airtime Performance

Figure 4.9: Experiment 5: DCF vs REACT$_{\text{QoS}}$ Airtime Performance for a Complete Topology.

Right away we can see that when the flows are sent at the full channel capacity (Figure 4.9b), DCF is much more variable in its performance than when the flow rates are reduced (Figure 4.9a). Looking at Figure 4.9a, the behavior of DCF under these loads looks similar to REACT$_{\text{QoS}}$. Additionally, the way that REACT$_{\text{QoS}}$ performs in this experiment appears to be slightly worse than DCF (see Figure 4.9c). Node 4 achieves 60% airtime in DCF with lower loads, while with REACT$_{\text{QoS}}$ node 1 (the QoS node) only achieves 40%. The difference here is 20%, and accounts for the airtime REACT$_{\text{QoS}}$ reserves for control traffic. Interestingly, the node sending the higher

Table 4.2: Delay, Jitter, and Throughput for Experiment 5

	Node 1	Node 2	Node 3	Node 4
DCF small				
Delay	536.19 ms	8.74 ms	8.67 ms	66.18 ms
Jitter	131.66 ms	7.06 ms	6.47 ms	24.29 ms
Throughput	426.78 Kbps	629.15 Kbps	629.15 Kbps	629.16 Kbps
DCF large				
Delay	1871.79 ms	649.27 ms	648.72 ms	777.81 ms
Jitter	496.75 ms	170.03 ms	156.85 ms	236.98 ms
Throughput	938.10 Kbps	1.29 Mbps	1.29 Mbps	1.21 Mbps
$\text{REACT}_{\text{QoS}}$				
Delay	536.59 ms	4.37 ms	4.13 ms	3.33 ms
Jitter	137.39 ms	4.27 ms	3.92 ms	2.5 ms
Throughput	2.59 Mbps	629 Kbps	628 Kbps	629 Kbps

traffic in both scenarios is node 1, illustrating how DCF has no way to differentiate classes of traffic.

We present statistics for the three scenarios in Table 4.2; the DCF scenario with reduced loads is called "DCF small" in the table, while the DCF scenario with all loads at 6 Mbps is called "DCF large". Looking at these statistics, we can see that $\text{REACT}_{\text{QoS}}$ achieves better delay and jitter for nodes 2,3, and 4, than DCF in either scenario, while it only improves the delay and jitter for DCF under small loads. $\text{REACT}_{\text{QoS}}$ achieves essentially the same delay and jitter performance for node 1 (the QoS node) as DCF does under small loads. It is interesting that DCF is able to perform this well in comparison to $\text{REACT}_{\text{QoS}}$. The total aggregate throughput for DCF under large scenarios is 4.778 Mbps and for $\text{REACT}_{\text{QoS}}$ it is 4.476 Mbps. By reducing the load for $\text{REACT}_{\text{QoS}}$ and allowing the SALT tuner to tune the contention windows, we were able to achieve essentially the same throughput, but with much improved jitter and delay performance with $\text{REACT}_{\text{QoS}}$.

4.5 Summary

In summary, with this implementation we initially sacrificed the delay and jitter performance of $\text{REACT}_{\text{QoS}}$ to ensure that the varied airtime demands of the QoS nodes (and the dynamic behavior enabled by the new implementation) were achievable. However, if instead the flow rates of the nodes are adjusted according to the allocations decided by $\text{REACT}_{\text{QoS}}$, we can retain the delay and jitter characteristics (and even throughput), while achieving superior delay and jitter performance. We move on to the more challenging multi-hop scenario in the next chapter.

Chapter 5

MULTI-HOP EXPERIMENTS

In contrast to the last chapter, where flows only traveled at most one hop, in this chapter we study multi-hop flows. Multi-hop flows present their own problems in a network, as nodes must forward traffic from other nodes in addition to sending their own traffic. In §5.1 we provide an overview and explain the experiment scenario. In §5.2 we describe our experiment and in §5.3 we report experiment results.

5.1 Overview

Past work on REACT has focused on using a reservation server running on each node in the network to forward information about multi-hop flows along the path of the flow [7]. As discussed in §3.6, we use the same technique with new extensions to support $\text{REACT}_{\text{QoS}}$. We consider a scenario where four nodes are arranged in a line topology, as for the single-hop experiments in Chapter 4. Now, there are two flows, each running the entire path of the topology; in other words, they must travel three hops to travel from the source node of the line topology to the destination. One of the flows is a QoS flow (running from node 1 to node 4) and the other is a BE flow (running from node 4 to node 1). There are applications for this in a real-world setting, as it is common to have full-duplex traffic between two senders along a path: perhaps there is real-time data flowing in one direction, with the other flow consisting of background traffic. This type of scenario would necessitate a QoS flow in one direction. The topology and flows are shown in Figure 5.1, where the black lines represent connections between nodes, the red arrows represent the QoS flow, and the blue arrows represent the BE flow.

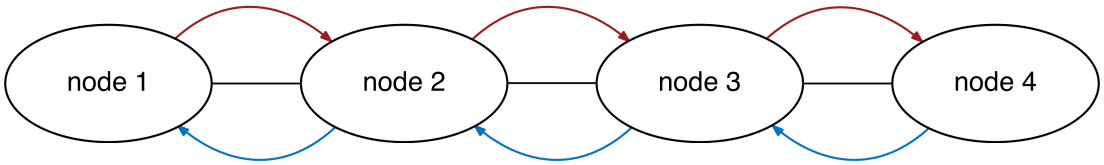


Figure 5.1: Multi-Hop Line Topology with Flow Paths.

Before, we considered QoS nodes requesting a certain portion of the airtime. However, now that we are considering a multi-hop scenario, we consider *flows* requesting the airtime because a flow requests airtime at multiple nodes. Deciding how much airtime the QoS flow should request is tricky, since now nodes have forwarding requirements that must also share the airtime. In other words, each node must have enough airtime preallocated for all flows going out of the node, and all flows transmitting at neighboring nodes.

First let us focus on a flow in one direction. Considering Figure 5.1, if we ignore the blue arrows representing BE flows, node 1 must preallocate enough airtime for one outgoing flow as well as for the traffic occurring at node 2. Meanwhile, nodes 2 and 3 must preallocate enough for an outgoing flow as well as the traffic at their two neighboring nodes. Node 4 does not need to reserve airtime for an outgoing flow, but still needs to preallocate airtime for its neighbor.

In summary, the first node in the path needs to reserve twice its request, nodes in the middle of the path need to reserve three times the request, the second to last node (node 3 here) needs to reserve twice the request (since the destination node is not forwarding) and the destination node needs to reserve enough for the request. This limits how much a flow can reserve among the path. For example, let us assume that the QoS flow wants 50% of the airtime. This can be reserved at nodes 1 and 3 ($2 * 50\% = 100\%$) and node 4 ($1 * 50\% = 50\%$), but it cannot be reserved at node 2

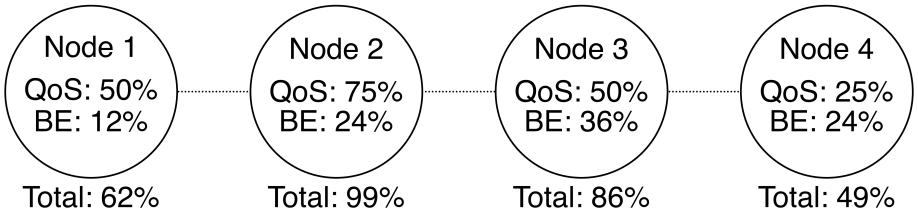


Figure 5.2: Reservation Amounts at Nodes in a 3-Hop Line Topology.

$(3 * 50\% = 150\% > 100\%)$. This does not even consider having a flow in the opposite direction, so we are limited in how much airtime can be reserved for a single flow.

Instead, we can have the QoS flow reserve 25% of the available airtime. This means that node 1 will have 50% preallocated, node 2 will have 75% preallocated, node 3 will have 50% preallocated, and node 4 will have 25% allocated. This reservation is feasible and it leaves space for a flow in the opposite direction. For the BE flow in the opposite direction, we are most limited by the available capacity at node 2 (25%) and node 3 (50%). This is because we need to reserve three times the request at node 3 and two times the request at node 2. This means the maximum we can reserve for the BE flow is $\lfloor \texttt{max}(\frac{25\%}{2}, \frac{50\%}{3}) \rfloor = 12\%$. This means that the final reservation will be a total of 62% at node 1, 99% at node 2, 86% at node 3, and 49% at node 4. Figure 5.2 illustrates the reservations at each node, along with the total amount reserved per node. This leaves us with an experiment of two flows moving in opposite directions, with enough airtime reserved to support them in the multi-hop path. The remaining capacity is free to be claimed by the nodes at the various auctions.

5.2 Description of Experiment

For the experiment, we run the flows for a total of two minutes. Since $\text{REACT}_{\text{QoS}}$ only offers 80% to nodes as available airtime, our reservation values from above need to be scaled accordingly. Node 1 sets a reservation of $0.8 * 25\% = 20\%$ and node 4 reserves $0.8 * 12\% = 9.6\%$. Once the reservations have been made, $\text{REACT}_{\text{QoS}}$

pulls the values from the reservation server and preallocates this amount at the nodes (i.e., the capacity offered to bidders at that auction is reduced by the preallocation amount). Flow rates are also set depending on the amount of airtime requested, so the QoS flow has a rate of $6\,\text{Mbps} * 0.2 = 1.2\,\text{Mbps}$ and the BE flow has a rate of $6\,\text{Mbps} * 0.096 \approx 0.6\,\text{Mbps}$. These flow rates are used for both $\text{REACT}_{\text{QoS}}$ as well as for EDCA in our experiments.

5.3 Results of Experiments

We present the results of our experiment in Figures 5.3 and 5.4 as well as Table 5.1. Figure 5.3 presents the airtime per access category for EDCA, and Figure 5.4 presents the airtime per access category for $\text{REACT}_{\text{QoS}}$. The difference in the two figures is fairly striking, as the airtimes for EDCA vary wildly. Some of the ACs on the nodes achieve quite high airtime values, such as node 1 receiving almost 100% for a majority

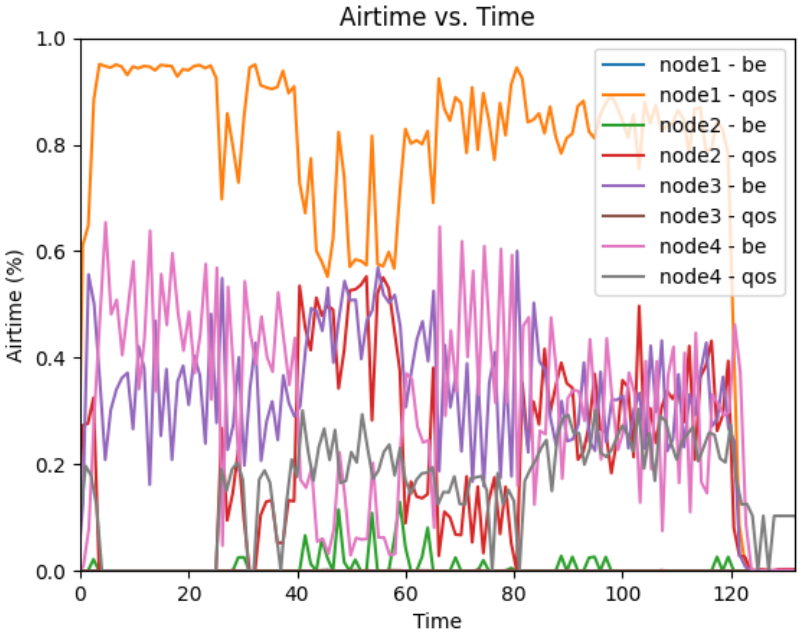


Figure 5.3: Multi-Hop Experiment: EDCA Airtime per AC per Node.

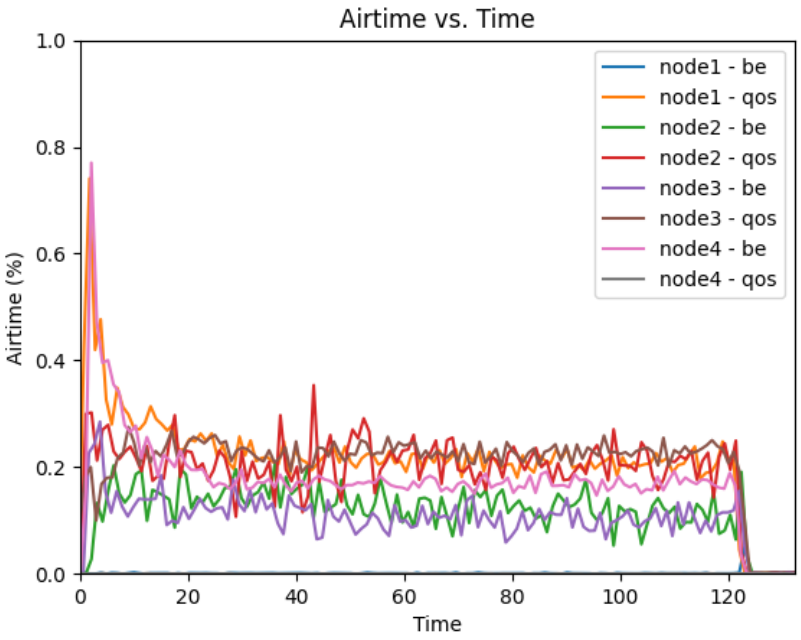


Figure 5.4: Multi-Hop Experiment: $\text{REACT}_{\text{QoS}}$ Airtime per AC per Node

of the experiment. However, some of the nodes, such as the BE access category at node 2, receive much less airtime. This means that the BE flow is going to suffer and have lower throughput than $\text{REACT}_{\text{QoS}}$. In contrast, the airtimes for the various ACs are much more consistent for $\text{REACT}_{\text{QoS}}$, with none of them achieving airtime above 40% except during initial convergence. This is the expected behavior by $\text{REACT}_{\text{QoS}}$ and allows it to achieve more stable performance. The three ACs that transmit the QoS flow (at nodes 1, 2, and 3) all achieve slightly above 20%, which is what they were allocated. For the BE ACs, nodes 2 and 4 achieve slightly above 10%, while node 3 achieves almost 20%. The BE AC at node 3 should not be achieving quite this much, although it is probably due to the extra capacity available at node 3 that goes unclaimed. The SALT tuner attempted to slow its transmissions by setting the CW value to its maximum of 1023, but was not quite enough to slow it down. However, the QoS AC at this node receives slightly more than its allocation, so this

	Delay		Jitter		Throughput	
	EDCA	$\text{REACT}_{\text{QoS}}$	EDCA	$\text{REACT}_{\text{QoS}}$	EDCA	$\text{REACT}_{\text{QoS}}$
BE	2629 ms	597 ms	909 ms	106 ms	34 Kbps	110 Kbps
QoS	839 ms	312 ms	157 ms	72 ms	126 Kbps	156 Kbps

Table 5.1: Delay, Jitter, Throughput Results for the Multi-Hop Experiment.

does not affect overall performance drastically. Quantitatively, the average variance for airtimes for EDCA is 0.011914, while for $\text{REACT}_{\text{QoS}}$ it is 0.000571; the variance for EDCA is over 20 times larger than for $\text{REACT}_{\text{QoS}}$!

Table 5.1 presents the delay, jitter, and throughput measurements for the QoS and BE flows. These numbers are the average for the measurements over the duration of the experiment. We can see that $\text{REACT}_{\text{QoS}}$ beats EDCA on all measurements collected. Interestingly, EDCA does provide traffic differentiation between the QoS and BE access categories, with the QoS flow achieving much better performance than the BE flow, by a factor of three for delay, over a factor of five for jitter, and over a factor of three for throughput. This makes sense, especially when considering Figure 5.3, since the QoS access categories had much higher airtime that the BE ones. Additionally, it follows that both the delay and jitter for EDCA is much worse than $\text{REACT}_{\text{QoS}}$, due to the wild variance of the airtimes for EDCA in comparison with the airtimes for $\text{REACT}_{\text{QoS}}$. $\text{REACT}_{\text{QoS}}$ also provides good traffic differentiation since the QoS flow achieves better metrics for all three statistics. Additionally, the difference between the BE and QoS flows is much smaller than with EDCA. For example, the delay for the BE flow in EDCA is 1790 ms longer than the QoS flow, while for $\text{REACT}_{\text{QoS}}$ it is only 285 ms longer. That is a sizeable improvement over EDCA. Similarly for jitter, EDCA has a difference of 752 ms while $\text{REACT}_{\text{QoS}}$ has a difference of 34 ms. Finally, the same is true for throughput, with $\text{REACT}_{\text{QoS}}$ gets a combined 266 Kbps while EDCA gets 160 Kbps. Comparing numerically, delay with

REACT$_{\text{QoS}}$ improved by 440% for the BE flow, and by 260% for the QoS flow, while jitter improved by 850% for the BE flow and by 210% for the QoS flow. Similarly, throughput improved by 320% for the BE flow, and by 120% for the QoS flow. What this tells us is that not only did REACT$_{\text{QoS}}$ provide better performance for the QoS flow, but it improved the BE flow performance by an even larger margin. Overall, this mostly confirms to us that EDCA provides good traffic differentiation at the local level, but struggles with the multi-hop scenario. In contrast, REACT$_{\text{QoS}}$ outperforms EDCA in the multi-hop scenario by sizeable margins on all metrics, for both the QoS and BE flows.

5.4 Summary

In summary, in a multi-hop scenario with two opposing flows, REACT$_{\text{QoS}}$ performs better than EDCA in all metrics considered and converges on a much more stable airtime allocation when considering the variance. The reservation server allows for successful forwarding of flow information and the new, multiple-AC SALT tuner achieves per-AC airtimes that are very close to the desired allocations. In the next chapter, we discuss the screening experiments, analyze the results, and discover the influential factors and interactions for our REACT$_{\text{QoS}}$ scenario.

Chapter 6

SCREENING EXPERIMENTS

This chapter describes an overview of the screening experiments conducted in this thesis. §6.2 provides a description of the experimental design, and §6.3 presents the results, including the most influential factors and 2-way interactions.

6.1 Overview

Typically, screening experiments are conducted first, ahead of any "real" experimentation on the system being investigated. One reason for this is to discover the most important factors on a response in a system, as well as those that are less important. In a system with many factors (such as most experiments in networked settings), this can allow the experimenter to cut down on the number of parameters used in experimentation. Intuitively, this makes sense because if a factor does not affect the outcome of an experiment, it can be safely excluded without fear of affecting the results.

In our case, in addition to screening factors to help reduce their number in future experimentation, we hope to discover more information about the effect of one of REACT's parameters on the responses. $\text{REACT}_{\text{QoS}}$, along with earlier versions, reserves a portion of the airtime for control messages. These control messages are essential to the algorithm, and need to be exchanged constantly to ensure that auctions in the network are up to date. These messages implicitly convey topology information along with the demands and requests, and so can become outdated quickly if there is a delay in transmission or reception of them. For this reason, all versions of REACT to date only offer 80% of the airtime to data traffic from nodes, and reserve 20%

of the airtime solely for REACT control messages. This is quite a large amount of overhead with REACT, and is one of its major downsides as the bandwidth of the channel is reduced by a fifth before any actual data is sent.

The 80% value was determined when REACT was first developed in a contention-based scenario [3], and so far has seemed to work very well. However, no further investigation of whether 80% is the best choice has been conducted since. If the amount needed for control traffic could be reduced, one of REACT's largest downsides could be mitigated. Therefore, part of our intent with the screening experiments conducted here is to provide an investigation into the amount of airtime reserved for control traffic. We use three different levels for the reserved amount, $80\%, 85\%$ and 90%, and hope to discover whether varying this amount has any affect on the selected responses. This information could inform as to whether it is worthwhile to conduct a more thorough investigation into the amount of airtime to reserve for control traffic.

Additionally, it is important to note that in our experimental design we use many more factors and levels than just the airtime reserved for control traffic. The purpose for this is to see if the amount of airtime reserved interacts in any way with the other factors. This is also critical in further experimentation, as perhaps the amount of airtime reserved does not directly affect the responses, but does when interacting with another parameter. By screening for 2-way interactions in addition to single factors, we can provide a more clear picture into what most strongly affects the responses we collect and inform how we might change REACT in the future.

6.2 Description of Screening Experiments

For the screening experiments, factors were selected for use in the single-hop scenario discussed in Chapter 4. Two sets of screening experiments are designed to

Table 6.1: Factors and Levels for the Screening Experiments.

Factor	Levels
traffic shaping	yes, no
CW min	$0, 15$
CW max	$511, 1023$
QoS flow type	BE, QoS, both
QoS node*	node1, node2, node3, node4
QoS request	$0\%, 10\%, 25\%, 50\%$
MAC protocol	DCF, EDCA, REACT80, REACT 85, REACT90, REACTQoS80, REACTQoS85, REACTQoS90
Transport protocol	UDP(500K), UDP(1M), UDP(5M), UDP(10M), TCP(8K), TCP(64K), TCP(128K), TCP(256K), TCP(8K,noDelay), TCP(64K,noDelay), TCP(128K,noDelay), TCP(256K,noDelay)

screen for the most significant factors in the ad-hoc REACT setting. The factors selected for experimentation are shown in Table 6.1.

There are eight factors for the line topology and seven for the complete topology. The factor "QoS node" is marked with a star because it is used in the line topology but not in the complete topology. This is because changing the QoS node does not change the allocation in a complete topology, but it does in a line topology. In other words, because all nodes are aware of all others in a complete topology, switching the QoS node to another position does not affect the information that all of the nodes have. In a line topology, nodes only know information about their direct neighbors and not nodes further down the line. The selected factors yield a full-factorial design of $36,864$ tests for the line topology and $9,216$ tests for the complete topology. In contrast, the locating arrays created only have 104 tests and 98 tests, respectively. This drastically reduces the number of experimental runs needed to determine significant factors and 2-way interactions.

Each factor affects either the flows or the behavior of $\text{REACT}_{\text{QoS}}$. Starting with the first factor in Table 6.1, the traffic shaping factor has two values and determines

whether traffic shaping is used by the SALT tuner. In the case that $\text{REACT}_{\text{QoS}}$ is disabled, the traffic shaper still operates according to this factor. Next, we have minimum and maximum values for the CW. SALT tunes the CW, but it does not tune outside the minimum and maximum values. This factor limits the space that SALT has to tune. Again, if $\text{REACT}_{\text{QoS}}$ is disabled, these factors still take effect as the minimum and maximum values for the CW set at the start of the experiment. The QoS flow type determines whether the QoS node is transmitting either a BE or QoS flow, or both at the same time. For the line topology only, QoS node determines the node that has QoS requirements. Next we have the size of the QoS request for the QoS node, followed by the MAC protocol used in the network. DCF is present even though it does not have traffic differentiation; in this case, the QoS node still has its QoS requirements, even though it does not have a mechanism in the network for achieving differentiated traffic. The same applies for REACT without QoS support. The levels for REACT have numbers attached; they represent the amount of airtime offered. Due to limits on the analysis techniques, factors cannot depend on other factors. This causes us to collapse the offered airtime factor onto the MAC protocol factor, so each is treated as a "different" protocol for analysis. Finally, we have the transport protocol, which varies between UDP and TCP, as well as protocol specific parameters such as bandwidth (UDP), window size (TCP), and enabling/disabling Nagle's algorithm (for TCP, "noDelay" in table). Again, because there are factors dependent on the protocol type (UDP does not have a window size), they are collapsed into a single factor for screening.

All experiments are run as in Chapter 4, with flows running as shown in Figures 4.1 and 4.2. The only difference is in the case of the QoS flow type, when the QoS node has two flows running. In this case, the QoS flow is sent as before, and in the complete topology the BE flow is sent to node 3. In the line topology, if the node has two

neighbors, the BE flow is sent to the other neighbor, otherwise it is sent along with the QoS flow to the only neighbor. All flows run at most one hop. The responses collected include delay, jitter, and throughput. These are the same responses that were collected for single-hop experiments in Chapter 4. To reduce the effect of run order between experiments, the order is randomized between each set of runs. The screening experiments are run multiple times, for a total of three replicates, and results are averaged from the three replicates to achieve the final responses. The replicates were run over a period of a few days on the w.iLab-t testbed, with all of the ZOTAC nodes reserved to ensure minimal interference. After all responses are collected, the results are analyzed using the technique from [17] and results are presented in §6.3.

The analysis tool from [17] generates models with a configurable number of terms. The tool outputs the score of each factor and interaction, and in this case the score is aggregate contribution to R^2 across all models generated (scores can be greater than one, recall §2.4.2). We generate many different models, starting with 2 terms to 25 terms, and we attempt to find a model with the best fit. The 'best' model is chosen when the change R^2 for the new model is less than 0.01 higher than the old model. This is not an exact science, but we felt this provided a decent way to select the models and is similar to what was done in [17]. For each topology, we present the two to four most important factors or interactions, depending on how many significant effects were reported. This information gives us insight into how important the factor or interaction is to the model and can inform us on the factors or interactions that are most important to our system.

6.3 Results of Screening Experiments

In this section we present the results of running analysis on our screening experiments. The analysis for the complete topology is presented in §6.3.1 and the line topology analysis in §6.3.2.

6.3.1 Complete Topology Analysis

Beginning with the delay response, the selected model has seven terms and an R^2 of 0.964. The most significant factor and the two most significant interactions are reported in Table 6.2.

Table 6.2: Top Effects for the Delay Response on the Complete Topology

Effect	R^2 score
`transport_protocol`	702.773
`transport_protocol & mac`	46.3876
`transport_protocol & traffic_shaping`	43.7748

In this case, the most important factor is the transport protocol used. The contribution to R^2 is much higher than for the two interactions. The interactions discovered include an interaction between the transport protocol and the MAC protocol, as well as one between the transport protocol and the traffic shaper. The remaining effects and interactions have even lower contributions to the R^2, and can probably be eliminated as non-significant parameters in this case. In fact, because the transport protocol has such a high contribution compared to the others (almost sixteen times higher), it may be correct to say we could also safely ignore these two interactions in this case as well. The most dominant effect according to this analysis is the transport protocol.

Moving on to the jitter response for the complete topology, we present the two most significant effects in Table 6.3. The selected model has an R^2 of 0.968 and

has 5 terms. For this response, the third effect in the ranking has an R^2 score of 2.52, much less than the top two at around 148, so we only include the top two in the ranking. Here, again as with the delay response, the top two effects are the transport protocol and the interaction between the transport protocol and the MAC protocol. In this case, the R^2 scores are very close. This implies that these effects are equally important, whereas in the delay response the transport protocol was much more active. Based on this analysis, we can safely exclude the remaining factors, and just test using the transport protocol and the MAC protocol.

Table 6.3: Top Effects for the Jitter Response on the Complete Topology

Effect	R^2 score
`transport_protocol`	147.605
`transport_protocol & mac`	148.535

Our last response is throughput, and the selected model has 16 terms and an R^2 of 0.972. This model has many more terms than for the other two responses, and as the terms were added the R^2 increased by a much more consistent amount. This implies that as terms were added they were consistently contributing to the R^2. However, the scores for the effects drops again after the third highest effect. We present the three effects in Table 6.4.

Table 6.4: Top Effects for the Throughput Response on the Complete Topology

Effect	R^2 score
`transport_protocol`	396787
`transport_protocol & mac`	144023
`transport_protocol & QoS_request`	144247

Again, the transport protocol is the most influential factor, followed by the interaction between the transport and MAC layers. The third effect here is an interaction between the transport protocol and the QoS request factor. Recall that this factor corresponds to how much airtime the QoS node requests, from zero to fifty percent.

Based on this analysis, the factors we should retain in experimentation are the transport protocol, the MAC protocol, and the QoS request factor.

6.3.2 Line Topology Analysis

Now we move on to the more complicated line topology. Again, we discuss analysis for each of the responses in turn.

For the delay response, the selected model has 11 terms with an R^2 of 0.945. We report the top four effects in Table 6.5. Once again, the transport protocol is overwhelmingly the most dominant effect. In this case, the interaction between the transport protocol and the MAC protocol is ranked fourth, while the MAC protocol individually ranks third. In second is the interaction between the MAC protocol and the QoS node factor. Recall that the QoS node factor determines the node along the line that has QoS requirements. This was not included in the complete topology because all nodes have complete information and changing the QoS node does not affect the allocation computed by $\text{REACT}_{\text{QoS}}$. In the line topology, changing the QoS node does affect the allocation because nodes do not have complete network information. For this response, it makes sense to include the transport and MAC protocols, along with the QoS node factor in future experimentation, and exclude the remaining factors.

Table 6.5: Top Effects for the Delay Response on the Line Topology

Effect	R^2 score
`transport_protocol`	13583.7
`mac & QoS_node`	1400.19
`mac`	744.716
`transport_protocol & mac`	555.404

For jitter, the selected model contains 16 terms and has an R^2 of 0.965. We present the top three effects in Table 6.6. Similarly to the other responses, the transport

protocol ranks first, with the MAC protocol second and the interaction between the two, third. Based on this, we can include those two factors and exclude the remaining factors.

Table 6.6: Top Effects for the Jitter Response on the Line Topology

Effect	R^2 score
`transport_protocol`	6945.52
`mac`	5284.72
`transport_protocol & mac`	4543.56

Lastly, we look at throughput for the line topology. This model has seven terms and the R^2 is 0.970. There are only two significant effects, in this case both are interactions. We present them in Table 6.7. The first effect is the transport/MAC protocol interaction, and the second is an interaction between the transport protocol and the QoS flow type. Recall that this factor determines what flows the QoS node sends (one BE flow, one QoS flow, or one of each). This implies that we should include the transport protocol, the MAC protocol, and the QoS flow type in future experimentation and can safely exclude the remaining factors.

Table 6.7: Top Effects for the Throughput Response on the Line Topology

Effect	R^2 score
`transport_protocol & mac`	544.951
`transport_protocol & QoS_flow_type`	317.697

6.3.3 Airtime Reserved for Control Traffic

Part of our motivation for conducting the screening experiments was to look for indications that the amount of airtime reserved for control traffic in $\text{REACT}_{\text{QoS}}$ had an effect on any of the responses. Just looking at the ranking of the terms does not give an indication for this, because we included the control airtime amount in the MAC factor, so instead we look to the actual models generated. The terms in the

models include a factor set to a specific level, so we observe these values and check if they include the MAC factor set to one of the REACT protocols set to specific airtime values (i.e., `REACTQoS85`).

Among the various responses, one of the `mac` levels including REACT appears in the delay and throughput responses for both topologies. It does not appear in the jitter response for either topology. For the complete topology, two terms in the delay model include a REACT value (airtime reserved set to 90 and 85) while for throughput it contains five terms including REACT with airtime values set to 80 and 90. Meanwhile, for the line topology, one term in the delay model includes REACT with the reserved value set to 80, while for throughput it appeared twice with the value set to 85.

From this we can conclude that the airtime value set does have some sort of an impact on the delay and throughput responses. Otherwise, we expect to either not see the terms present, or to see that value stay the same in the models. This provides an indication that the amount reserved for control traffic does have an impact, and further study could provide more insight into whether 80% for control traffic is indeed the best value for $\text{REACT}_{\text{QoS}}$.

6.4 Summary

In summary, the most significant parameters in all of the typically include both the MAC protocol choice and the transport protocol choice. For both the delay and throughput responses for the both topologies, we have found indications that the amount of airtime reserved for control traffic has an impact on those responses in our system. The fact that it does not appear in the jitter models could signify that it does not affect jitter to a significant degree. This gives us insight into whether the amount reserved for control traffic could be reduced, and gives us a starting point for

a more detailed investigation into the best amount of airtime to reserve for control traffic under REACT. Finally, we have determined that the most significant factors in this system are the transport and MAC protocols (with some other factors specific to certain responses) and that the majority of the remaining factors could be excluded in experimentation. In the next chapter we briefly outline future directions for this work and describe our conclusions.

Chapter 7

CONCLUSIONS AND FUTURE WORK

In this chapter, we begin by outlining future directions for this work in §7.1 and conclude in §7.2 by providing an overview of the work presented in this thesis and give our conclusions.

7.1 Future Work

$\text{REACT}_{\text{QoS}}$ is a first step to a more complete solution to realize airtime allocations and provide QoS support in a wireless network. However, there is much that could be done to further the work presented here.

First, this implementation of $\text{REACT}_{\text{QoS}}$ can impact the delay and jitter performance. Prior work shows that REACT sacrifices some amount of throughput in order to achieve superior delay and jitter [7]. Due to the traffic shaper used (the `tbf` module from Linux `tc` [1]), this implementation has worse delay and jitter. As we showed in Experiment 5 in § 4.3, if the nodes adjust their traffic to match the allocation they received from the $\text{REACT}_{\text{QoS}}$ auction, this can be mitigated. However, it may be possible to improve the tuning algorithm, perhaps by tuning the *arbitration interframe space* (AIFS) in addition to the contention window size. However, as our focus in this thesis is to achieve varied airtime allocations, we leave it to future work to improve the tuning algorithm to not sacrifice delay or jitter.

An alternative solution to eliminating the need for a traffic shaper is to build a cross-layer, application-level module that interacts with $\text{REACT}_{\text{QoS}}$ and automatically adjusts the flow rate based on the allocation decided by $\text{REACT}_{\text{QoS}}$. This would

[1] https://www.man7.org/linux/man-pages/man8/tc-tbf.8.html

allow the SALT tuner to be able to tune without drops without the need for a traffic shaper, while maintaining the delay and jitter characteristics of $\text{REACT}_{\text{QoS}}$. Even if SALT could be improved to not require the traffic shaper (or this application level module), it may be desirable to enforce that nodes only send according to their allocation. Intuitively this makes sense, as nodes should not send more than they have been allocated.

Another improvement that could be made is to implement an explicit admission control scheme for the QoS requests. Currently, the algorithm gives QoS nodes an allocation of zero if the request cannot be satisfied. An explicit mechanism to request and deny/accept the requests at the $\text{REACT}_{\text{QoS}}$ auction may be a better solution and would allow for easier interaction with the $\text{REACT}_{\text{QoS}}$ module. This mechanism could integrate directly with an application-level module.

As a result of our screening experiments, several cross-layer interactions were discovered (particularly between the transport protocol and the MAC protocol, which appeared for every response for both topologies). This begs the question of how these cross-layer interactions impact the performance of $\text{REACT}_{\text{QoS}}$. Further experimentation to determine the effect of these interactions is needed and can help inform what changes to make to $\text{REACT}_{\text{QoS}}$ to improve performance. Furthermore, we determined that the MAC protocol is an influential factor and that the control traffic reservation value for REACT appeared in several of the models during the analysis. This observation is important for improving the performance of REACT and informs us that the airtime dedicated to control messages has an impact on the responses. Through further study, we could potentially reduce this value without losing the improved delay and jitter characteristics of $\text{REACT}_{\text{QoS}}$. Finally, the results of our screening experiments allow us to eliminate many many factors that are unimportant to the

responses collected. This aids in all future research because fewer factors are needed in experimentation.

Next, $\text{REACT}_{\text{QoS}}$ was implemented and evaluated in an ad-hoc wireless network scenario. We believe that the algorithm could help improve the performance of infrastructure WLANs in a managed access point (AP) scenario. In this case, one REACT auction could be run per-AP, with clients only running the bidder portion of REACT. The AP then has full knowledge of its network and could make decisions in a centralized manner. Furthermore, if APs were in the same collision domain, a REACT auction could be run between entire AP-subnets, over a wired connection, to negotiate airtime among APs.

The use of $\text{REACT}_{\text{QoS}}$ in an AP scenario suggests the use of Software-Defined Networking (SDN) in a wireless network. There has been much previous work on SDN and network slicing in the wireless domain, such as the EmPOWER system [20],[21]. But much of this work (including EmPOWER) operates higher in the network stack and still depends on IEEE 802.11 for the lower level MAC protocol. EmPOWER, for example, relies on manipulating the dequeuing frames at the AP using an adaptive deficit weighted round robin (ADWRR) scheme to provide the slicing functionality. Instead, it may be possible to use the more precise airtime realization mechanism of $\text{REACT}_{\text{QoS}}$ to give nodes a precise allocation determined by a centralized controller. This could provide more precise slicing either by replacing the ADWRR algorithm altogether, or complementing it by replacing 802.11 at the lower MAC layer. Additionally, conducting slicing in the uplink direction is not a solved problem, which $\text{REACT}_{\text{QoS}}$ has the potential to improve because SALT could tune the airtime for the client nodes. Some work has been done using traffic shapers interacting the with the controller [22], but again this takes place higher in the network stack and still relies on 802.11 at the MAC layer.

7.2 Conclusion

In this work we proposed REACT$_{\text{QoS}}$, a distributed protocol to allocate airtime with QoS support. This allows nodes to request a higher class of airtime which, if available at adjacent auctions, guarantees the node a higher airtime allocation. These airtime allocations are realized through the use of an updated SALT tuner that tunes two separate ACs from the EDCA protocol used by the 802.11 standard. Through experimentation on the w-iLab.t testbed we have shown that this mechanism is successful at achieving varied allocations in an ad-hoc wireless network in a single-hop scenario. Additionally, through the use of a reservation server running at all REACT$_{\text{QoS}}$ nodes, multi-hop flows can reserve airtime along their path to make nodes aware of flows running through them. Our experimentation shows that REACT$_{\text{QoS}}$ achieves better performance in all metrics when compared to EDCA, including throughput. Finally, by conducting screening experiments we discovered that we can eliminate many unimportant factors from further experimentation. In addition to identifying the influential factors, several influential interactions were discovered and indications were found that the amount of airtime reserved for control traffic has an impact on the responses. This essential information will aid in future research and will contribute to improvements to REACT.

This extension, combined with traffic shaping and dynamic demands, allows us to tune the airtime nodes receive and opens the door to improved QoS support in wireless networks in both single-hop scenarios and the more difficult multi-hop scenarios. This work has contributed to the REACT protocol and improved quality of service in networks, as well as improving future REACT research needs by discovering the influential factors through screening experiments.

REFERENCES

[1] Cisco, "Cisco annual internet report (2018-2023)," March 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[2] I. Elgendi, K. S. Munasinghe, and A. Jamalipour, "Traffic offloading for 5g: LTE or Wi-Fi," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017, pp. 748–753.

[3] D. Garlisi, F. Giuliano, A. L. Valvo, J. Lutz, V. R. Syrotiuk, and I. Tinnirello, "Making WiFi Work in Multi-hop Topologies: Automatic Negotiation and Allocation of Airtime," in *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*. Columbus, OH, USA: IEEE, Jun. 2015, pp. 48–55. [Online]. Available: http://ieeexplore.ieee.org/document/7165083/

[4] "IEEE Std 802.11™-2016, IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements—Part 11: Wireless LAN Medium Access Control," p. 3534, 2016.

[5] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification. Amendment 7: Medium Access Control (MAC) Quality of Service (QoS), ANSI/IEEE Std 802.11e, LAN/MAN Standards Committee of the IEEE Computer Society Std., 2005.

[6] J. Lutz, C. J. Colbourn, and V. R. Syrotiuk, "ATLAS: Adaptive Topology- and Load-Aware Scheduling," *IEEE Transactions on Mobile Computing*, vol. 13, no. 10, pp. 2255–2268, Oct. 2014.

[7] M. J. Mellott, D. Garlisi, C. J. Colbourn, V. R. Syrotiuk, and I. Tinnirello, "Realizing airtime allocations in multi-hop Wi-Fi networks: A stability and convergence study with testbed evaluation," *Computer Communications*, vol. 145, pp. 273–283, 2019.

[8] *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015) - Redline: IEEE Standard for Ethernet - Redline*. IEEE, 2018.

[9] G. Bianchi, "Performance analysis of the ieee 802.11 distributed coordination function," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 3, pp. 535–547, 2000.

[10] S. Xu and T. Saadawi, "Revealing the problems with 802.11 medium access control protocol in multi-hop wireless ad hoc networks," *Computer networks (Amsterdam, Netherlands : 1999)*, vol. 38, no. 4, pp. 531–548, 2002.

[11] I. Tinnirello, G. Bianchi, and Yang Xiao, "Refinements on IEEE 802.11 Distributed Coordination Function Modeling Approaches," *IEEE Transactions on Vehicular Technology*, vol. 59, no. 3, pp. 1055–1067, Mar. 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5191039/

[12] F. Cali, M. Conti, and E. Gregori, "Dynamic tuning of the ieee 802.11 protocol to achieve a theoretical throughput limit," *IEEE/ACM Transactions on Networking*, vol. 8, no. 6, pp. 785–799, 2000.

[13] M. Mellott, "Smoothed airtime linear tuning and optimized REACT with multi-hop extensions," Master's thesis, 2018. [Online]. Available: http://search.proquest.com/docview/2050224947/

[14] S. Bouckaert, P. Van Wesemael, J. Vanhie-Van Gerwen, B. Jooris, L. Hollevoet, S. Pollin, I. Moerman, and P. Demeester, "Distributed spectrum sensing in a cognitive networking testbed," in *Towards a Service-Based Internet: 4th European Conference, ServiceWave 2011, Poznan, Poland, October 26-28, 2011. Proceedings*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6994, pp. 325–326.

[15] S. A. Seidel, M. T. Mehari, C. J. Colbourn, E. De Poorter, I. Moerman, and V. R. Syrotiuk, "Analysis of large-scale experimental data from wireless networks," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Honolulu, HI: IEEE, Apr. 2018, pp. 535–540. [Online]. Available: https://ieeexplore.ieee.org/document/8407023/

[16] D. C. Montgomery, *Design and analysis of experiments*, eighth edition.. ed. Hoboken, NJ: John Wiley Sons, Inc., 2013.

[17] Y. Akhtar, F. Zhang, C. J. Colbourn, J. Stufken, and V. R. Syrotiuk, "Scalable level-wise screening using locating arrays," *Unpublished manuscript*, October 2020.

[18] D. J. Kulenkamp and V. R. Syrotiuk, "Supporting differentiated airtime in wireless networks," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2021.

[19] J. Mvulla, Y. Kim, and E.-C. Park, "Probe/PreAck: A Joint Solution for Mitigating Hidden and Exposed Node Problems and Enhancing Spatial Reuse in Dense WLANs," *IEEE Access*, vol. 6, pp. 55 171–55 185, 2018.

[20] E. Coronado, R. Riggio, J. Villalon, and A. Garrido, "Lasagna: Programming Abstractions for End-to-End Slicing in Software-Defined WLANs," in *2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. Chania, Greece: IEEE, Jun. 2018, pp. 14–15.

[21] E. Coronado, S. N. Khan, and R. Riggio, "*5G-EmPOWER* : A Software-Defined Networking Platform for 5G Radio Access Networks," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 715–728, Jun. 2019.

[22] P. H. Isolani, D. J. Kulenkamp, J. M. Marquez-Barja, L. Z. Granville, S. Latré, and V. R. Syrotiuk, "Support for 5G mission-critical applications in software-defined IEEE 802.11 networks," *Sensors (Basel, Switzerland)*, vol. 21, no. 3, pp. 693–, 2021.