Smoothed Airtime Linear Tuning and Optimized REACT with Multi-hop Extensions

by

Matthew J. Mellott

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2018 by the
Graduate Supervisory Committee:

Violet Syrotiuk, Chair
Charles Colbourn
Ilenia Tinnirello

ARIZONA STATE UNIVERSITY

May 2018

ABSTRACT

Medium access control (MAC) is a fundamental problem in wireless networks. In ad-hoc wireless networks especially, many of the performance and scaling issues these networks face can be attributed to their use of the core IEEE 802.11 MAC protocol: distributed coordination function (DCF). Smoothed Airtime Linear Tuning (SALT) is a new contention window tuning algorithm proposed to address some of the deficiencies of DCF in 802.11 ad-hoc networks. SALT works alongside a new user level and optimized implementation of REACT, a distributed resource allocation protocol, to ensure that each node secures the amount of airtime allocated to it by REACT. The algorithm accomplishes that by tuning the contention window size parameter that is part of the 802.11 backoff process. SALT converges more tightly on airtime allocations than a contention window tuning algorithm from previous work and this increases fairness in transmission opportunities and reduces jitter more than either 802.11 DCF or the other tuning algorithm. REACT and SALT were also extended to the multi-hop flow scenario with the introduction of a new airtime reservation algorithm. With a reservation in place multi-hop TCP throughput actually increased when running SALT and REACT as compared to 802.11 DCF, and the combination of protocols still managed to maintain its fairness and jitter advantages. All experiments were performed on a wireless testbed, not in simulation.

DEDICATION

This thesis is dedicated to my parents, Dr. Ramona Mellott and Dr. Micheal Mellott. Without your constant support I would have never been able to complete this work. And without your encouragement I may never have tried.

ACKNOWLEDGMENTS

I would like to acknowledge my wonderful committee chair, Dr. Violet Syrotiuk. There were ups. There were downs. And it took longer than we thought, and then longer than that. Thank you for sticking with me through it all and especially for having faith in me when I was ready to throw in the towel. Special thanks also goes to my committee members Dr. Charles Colbourn and Dr. Ilenia Tinnirello for supporting this work and for waiting so long in suspense.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Most wireless networks we encounter regularly rely on fixed network infrastructure. At a coffee shop your laptop accesses the network through an access point that is backed by the wired network provided by that shop's ISP. Cellular coverage depends on a vast network of towers and each tower covers an fixed area or "cell". Conversely, ad-hoc wireless network are formed by nodes communicating directly. Such a network can emerge spontaneously because it does not require the presence of infrastructure like an access point or cell tower. In the presence of seemingly ubiquitous Wi-Fi and cell coverage ad-hoc networks might seem irrelevant, but there are still many situations where network infrastructure is not present or unusable or perhaps even maliciously inoperative. For example after disasters, like the Boston marathon bombing [1], phone networks (landlines and cellular) are frequently overwhelmed. If first responders could form an ad-hoc network they could communicate across the disaster area and relay data to allow radios to communicate beyond their physical range limitations. In the 2016 Gambian presidential election both the Internet and cell phone networks were blocked [2]. Hong Kong street protesters faced a similar problem in 2014 and turned to an app called FireChat [3]. This app uses the capabilities already present in mobile phones, through support in the 802.11 standard (i.e., Wi-Fi), to let users communicate on an ad-hoc basis. Thus despite the government's crackdown, protesters were still able to communicate with each other. But while successful in this case, 802.11 based ad-hoc networks still face many challenges.

Performance problems are one of the main challenges facing ad-hoc networks. Data

flows can exhibit severe instability problems [4], [5], and when there are many hops *transmission control protocol* (TCP) throughput degrades quickly [4]. Under load these networks suffer from a high degree of contention and reach saturation at data rates much lower than in comparable wireless local area networks (WLANs) [6]. Unfairness in regards to transmission opportunities (where some nodes get a disproportionately large share of the channel while others are starved) is also a problem and drives or exacerbates the other problems [4], [5]. These performance problems can primarily be attributed to the *medium access control* (MAC) protocol used in wireless ad-hoc networks [4]–[7].

The *distributed coordination function* (DCF) is the default 802.11 MAC protocol. To transmit, a node must first wait the duration of a *DCF interface space* (DIFS). If the channel is not sensed busy during the DIFS then the node is allowed to transmit. However if the channel was sensed busy the node must wait for the channel to be idle for a DIFS and then start decrementing its backoff counter. The backoff counter is picked randomly from the interval $[0, C)$, called the contention window, where C is the contention window size. C starts at C_{min} and is doubled every transmission failure up to C_{max}. On success C is reset to C_{min}. The resizing of the contention window is one of the causes of unfairness. It causes nodes that have already acquired the channel to be more likely to acquire it again than nodes that have sensed it busy. If the channel is sensed busy at any point while counting down then backoff becomes frozen; the counter is no longer decremented until the channel is idle for a DIFS. Once transmission does occur the receiving node transmits an ACK after waiting the duration of a Short Interframe Space (SIFS) after the packet is received. Positive acknowledgment is used instead of requiring the transmitting node to detect a collision itself while transmitting. Since the SIFS plus its propagation delay is shorter than a

DIFS no node is able to detect the channel idle for a DIFS until the end of the ACK. This allows that ACK to follow right after a received packet without contending and with a lower probability of being interfered with. If a transmitting node does not get an ACK after a certain amount of time then it determines its transmission failed.

When all nodes are within transmission range of each other the ACK scheme can work well. However when there are nodes in the network outside the transmission range of each other we can encounter problems like the hidden node problem and the exposed node problem (shown in Figure 1).

(a) Hidden Node (b) Exposed Node

Figure 1: The hidden and exposed node problems

The hidden node problem is where two nodes A and C are interfering with each other at node B but neither A nor C can detect that the other is transmitting. This is especially problematic because when B tries to ACK a transmission from either A or C the other node might interfere with the ACK because it never sensed the transmission B is acknowledging.

The exposed node problem involves two transmitting nodes within range of each other that are contending with each other for the channel even though at their separate destinations their transmissions would not interfere. Both of these problems can show up in WLANs but in ad-hoc networks there is no central access point that has full knowledge of the network and that could coordinate transmissions. An example of such coordination are RTS and CTS packets. When a station receives at CTS packet it knows to not transmit, even if the medium is not physically sensed busy, for an

interval specified in the packet. In a wireless LAN the access point will be sending out the CTS packet in response to a RTS from some node connected to it that wants to transmit. The access point is by definition within range of all nodes in the network so the CTS it sends out will prevent all nodes from interfering with the node that is ready to transmit. Ad-hoc networks can also utilize the RTS/CTS (i.e., virtual carrier sense) feature of 802.11. However they lack a central coordinator like an access point and so nodes that are outside of CTS reception range but within interference range can still cause contention. Virtual carrier sense also does nothing to address the problem of exposed nodes in ad-hoc networks.

Many solutions to the problems of unfairness and high contention in ad-hoc networks have been proposed. In [8] the authors propose a protocol, SSCH, that has nodes switching between channels so that communicating nodes share the same channel while nodes with disjoint communications do not. SSCH reduces contention when communicating nodes can be efficiently partitioned onto separate channels. However there can still be a high degree of contention on the single shared control channel, and the scheme is limited by the number of separate frequencies available. [9] takes a similar approach, and has similar drawbacks, but instead implements channel assignment at the MAC level. In [10] the authors make significant improvements to fairness using rate limiting. Each node has a fixed maximum data rate, and because there is a limit no node can saturate the channel, which would lead to unfairness. Thus rate limiting solves the problem of unfairness by not letting it occur. This approach, however, is dependent on precise knowledge of the channel's capacity (to ensure the chosen rates will prevent saturation). The authors of [11] address the problem of contention directly by using contention window tuning. Their CW tuning scheme is formulated such that the network achieves its theoretical maximum aggregate

throughput, but the question of fairness is completely neglected. The REACT protocol was first proposed in [12] as part of a scheduled MAC protocol and takes a different approach to address both unfairness and high contention using a distributed auction for channel resources. In [13] REACT was transformed into a contention based scheme using a contention window tuning algorithm to realize the airtime allocated to nodes by REACT. In this thesis we will build upon REACT and the work already done on contention window tuning algorithms.

This thesis will demonstrate that REACT and contention window tuning are even more effective techniques than shown in previous work and can be extended to multi-hop scenarios. Chapter 2 discusses REACT and contention window tuning in more depth and introduces the testbed and other background knowledge that might be helpful when reading subsequent chapters. In Chapter 3 we introduce a new user-space implementation of REACT and discuss related optimizations for this implementation. Perhaps the most significant contribution of this thesis comes in Chapter 4 where we introduce a new contention window tuning algorithm and a experimental evaluation of this new algorithm on the testbed. Finally in Chapter 5 we extend REACT and the new tuning algorithm to the multi-hop scenario by introducing a new multi-hop reservation algorithm.

Chapter 2

BACKGROUND

In this chapter we discuss some of the information that is required to understand the contributions of this thesis. We introduce the testbed that our experiments were performed on in Section 2.1. This introduction includes explaining how topologies were found on the testbed and describing which topologies we used in our experiments in Section 2.1.1 and Section 2.1.2. A metric for quantitatively comparing the convergence times of our experiments is described in Section 2.2. Finally we describe REACT and the tuning algorithm from [13] in Section 2.3 and Section 2.4 respectively.

2.1 w-iLab.t Testbed

All experiments were performed on the CREW project w-iLab.t testbed [14]. This testbed has two separate locations: the "office" location in Ghent, Belgium and the "Zwijnaarde" location in Zwijnaarde, Belgium. We used the latter and it is commonly referred to as the Wilab2 testbed. Wilab2 is composed of 60 wireless nodes, called the "zotac" nodes, with two IEEE 802.11a/b/g/n antennas along with several other interfaces we did not use (e.g., IEEE 802.15.4, IEEE 802.15.1, Software Defined Radio, spectrum scanners). Some of these nodes are also mobile and are mounted on Roomba vacuum robots. We used one of these mobile nodes but it remained stationary. While the testbed is large and provides many nodes, finding specific wireless topologies is a non-trivial task.

2.1.1 Finding Topologies on the w-iLab.t Testbed

For REACT especially it is very important that topologies are "high-quality". This means that nodes cannot successfully receive packets from other nodes that are not supposed to be within transmission range. If a single broadcasted control packet makes it to a node "outside" of transmission range this can ruin an experiment by completely changing the airtime offered to all nodes at that auction. The high-quality topologies we found were created in two steps: first we performed a ping experiment and then we used the results of these experiments to create graphs from which topologies could be derived.

This ping experiment was fairly straightforward. It involved reserving every node on the testbed and then having each node ping every other node 100 times. Each node starts off by pinging itself as a sanity check. Then each node pings the next node after itself in order, wrapping around if need be. For example node 5 pings node 6 after pinging itself and node 60 pings node 1. This was done so that no two nodes would be pinging the same node at the same time. The number of successful pings out of 100 is the transmission probability between those two nodes. While there certainly can exist uni-directional links in a wireless network pings require a response. Since we are not interested in uni-directional links this is fine and checking that the transmission probabilities from a node to another and then vice versa are the same provides another sanity check.

With the transmission probability data we created two graphs of the network. The first graph, G_{100}, only has an edge between two nodes if their transmission probability is 100% and can be seen in Figure 2. The second graph, G_{all}, has an edge between every two nodes that have a non-zero transmission probability and can be seen in

Figure 4. Both graphs are important because we want topologies with high quality links, which we can find with G_{100}, but we also want to ensure there are no low quality links that would violate our topologies by checking G_{all}. For example to find a line topology with a certain number of hops the logic is as follows. For each pair of nodes, check if the length of the shortest paths (there can be multiple shortest paths) between these nodes in G_{100} is if of the target length. If it is, check that there there does not exist a shorter path between the nodes in G_{all}. If there is not a shorter path then the shortest paths in G_{100} are "high-quality" paths. This is because each link in the path has a 100% transmission probability and there are no lower probability links that would form a shorter path between the pair of nodes. Several different topologies were found on the testbed using logic similar to this.

2.1.2 Topologies Found

Three topologies were used in all of our experiments. There is the *complete topology* with four fully-connected nodes. There is the *star topology* with four leaf nodes connected to one center node. There is the *line topology* with four nodes arranged in a line with three hops. A map of the testbed can be seen in Figure 6 and Figure 7 shows each topology superimposed on top of that map. The red lines represent links in the network. If there are no lines connecting nodes then they cannot directly communicate. The nodes used in each topology are colored purple (instead of the normal blue).

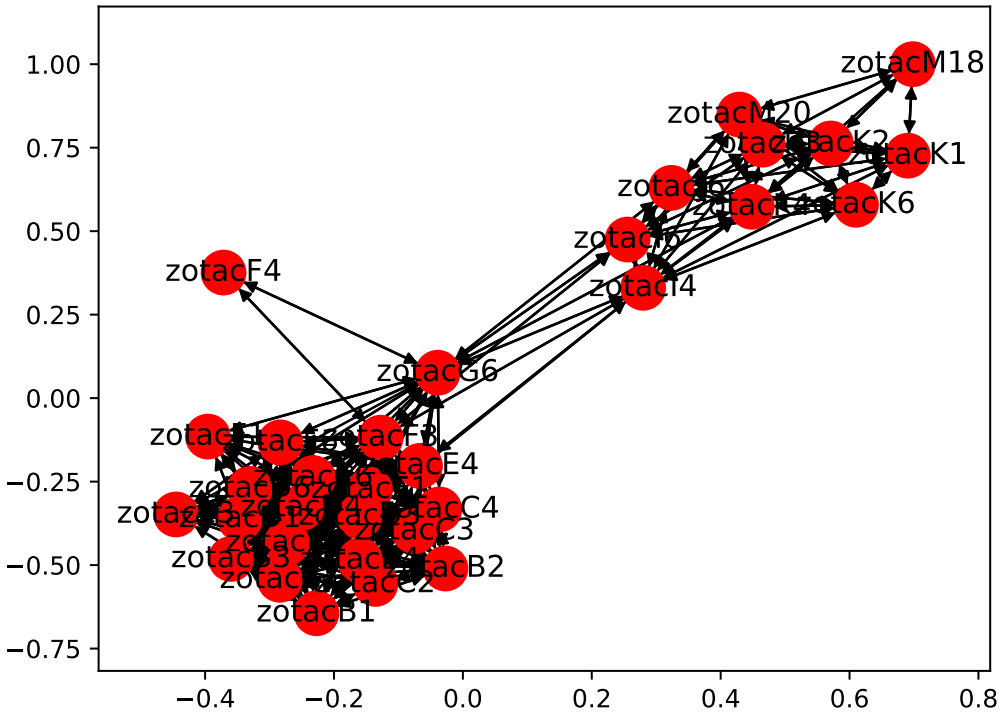


Figure 2: Map of 100% transmission probability links on the testbed G_{100}

2.2 Quantitative Convergence Comparisons

In order to compare our experimental results we had to come up with a way to measure convergence time. Convergence is an easy thing to qualitatively gauge when looking at a graph. In some of our airtime vs. time graphs especially you can see the measurements converge quickly to values that they remain around for the rest of the experiment. One thing that is harder to gauge from looking at a graph however is how *tightly* things converge. You can see that the values did in fact converge but trying to eyeball whether values remain within a 20% or 15% or 5% range of the

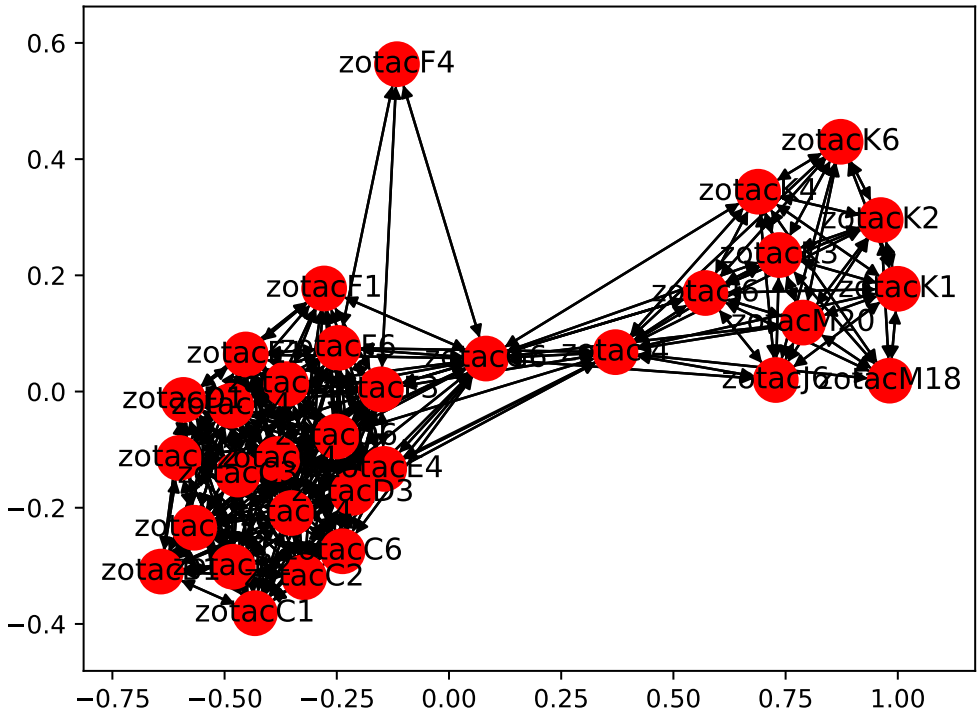


Figure 4: Map of non-zero transmission probability links on the testbed G_{all}

value they converged to is a futile exercise. This convergence threshold, as we call it, is an important factor when you are comparing two experiments on the basis of convergence time. If in one experiment values converge more slowly but at a lower convergence threshold that is still an important and perhaps interesting distinction. In order to quantitatively compare convergence times for various thresholds we use a statistic called coefficient of variation.

The coefficient of variation (CV) is the ratio of standard deviation to the mean. It is a statistical measurement of the dispersion of our measurements expressed as a percentage. In our experiments there are multiple nodes each with a separate series of

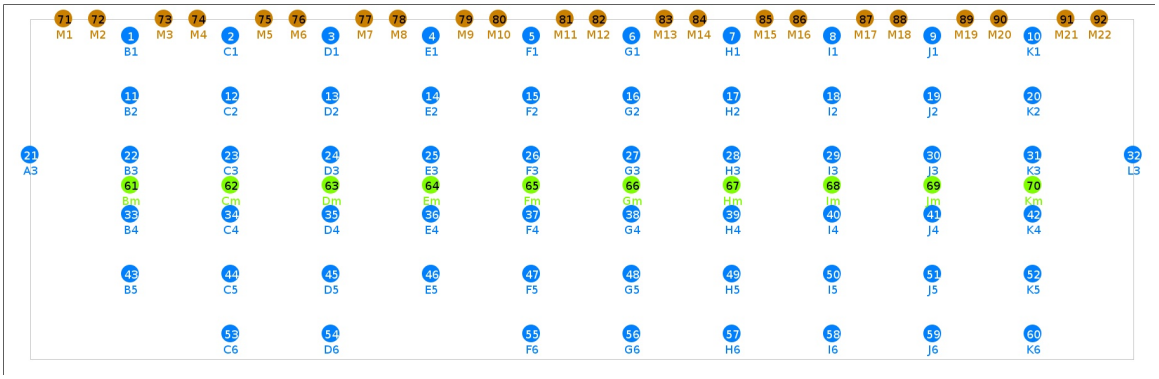
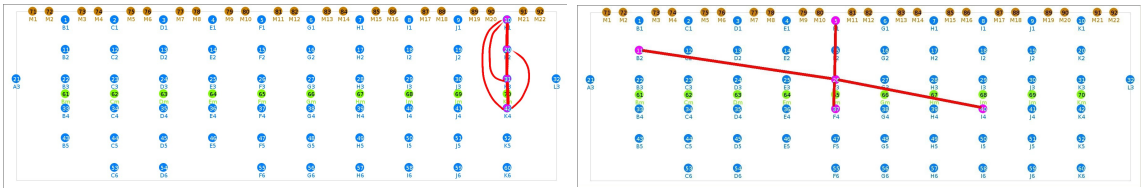
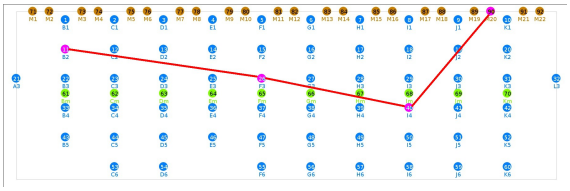


Figure 6: Testbed map showing the position of each node



(a) Complete Topology (b) Star Topology



(c) 3-hop Line Topology

Figure 7: Topologies used in upcoming experiments shown on testbed map

measurements over time that may be converging to different values. CV is useful in this scenario because it does not matter what value our measurements are converging to. We can compare every CV to a single threshold to determine if measurements have converged to the value they will be converging to and if that convergence is within the threshold we have set. In order to determine if values for a particular trial have converged we must first set T, the convergence threshold (e.g., $T = 0.1$). If the CV for all of a node's values after a certain time t are below T then that node has converged. Once all nodes converge then the trial has converged. The time of the last node to converge is the convergence time for the trial. Like our intuitive notion of

convergence this measure quantitatively captures the same idea: after a certain point in time no node's values vary by more than a certain amount around its mean. All the convergence times we mention from here are on out were found using this method and are accompanied by the convergence threshold used to compute them.

2.3 REACT

REACT is a distributed resource allocation protocol that uses the metaphor of an auction. When used in the context of wireless networks the resource being allocated (or being put up for "auction") is *airtime*, the percentage of time a node controls the medium over a given period. Each node runs both an auctioneer (Algorithm 2) and a bidder (Algorithm 1), and auctioneers offer capacity while bidders claim capacity at adjacent auctions to satisfy their own airtime demand. Auctioneers update their offers to satisfy all nodes bidding at their auction while also ensuring that all nodes receive a fair allocation of the resource. Bidders update their claims to ensure that they are not consuming any more airtime than can be offered at any adjacent auction. In [12] it was shown that the nodes participating in REACT converge to a lexicographic max-min airtime allocation. Auctioneers in our experiments are programmed to allocate 80% of the channel while bidders all start with a demand for 100% of the channel. These values were also used in previous work [13].

2.4 Garlisi et al. Tuning

In [13] the authors transformed REACT, which had previously been used in a scheduled MAC protocol [12], into the basis for a novel contention based MAC protocol.

Algorithm 1 REACT Bidder for Demand i.

upon initialization
    $R_i \leftarrow \emptyset$
    $w_i \leftarrow 0$
    UpdateClaim()
end upon
upon receiving a new demand magnitude w_i
    UpdateClaim()
end upon
upon receiving *offer* from auctioneer j
    offers$[j] \leftarrow$ *offer* // *Remember the offer of auctioneer* j.
    UpdateClaim()
end upon
upon bidder i joining auction j
    $R_i \leftarrow R_i \cup j$ // *Resource* j *is now required by demand* i.
    UpdateClaim()
end upon
upon bidder i leaving auction j
    $R_i \leftarrow R_i \setminus j$ // *Resource* j *is no longer required by demand* i.
    UpdateClaim()
end upon
procedure UpdateClaim()
    // *Select the claim to be no larger than the smallest offer or* w_i.
    claim $\leftarrow \min(\{\textit{offers}[j] : j \in R_i\}, w_i)$
    send *claim* to all auctions in R_i
end procedure

This new system takes a node's airtime allocation from REACT and tries to realize it at runtime by tuning the contention window size parameter that is part of the 802.11 backoff process. REACT combined with the contention window tuning algorithm, which we will call the "Garlisi et al. [13]" algorithm from now on, was very successful. The authors show that the system provided for fairer opportunities for transmission than 802.11 DCF while also reducing contention and with only a small reduction in throughput due to the auction's overhead.

Garlisi et al. [13] considers, over a given observation period, the amount of time the channel is frozen, the actual airtime achieved, the number of channel accesses, and the previous contention window size in order to determine the contention window size for the next interval. The equation below shows exactly how each of these parameters

Algorithm 2 REACT Auctioneer for Resource j.

```
1: upon initialization
2:     D_j ← ∅
3:     c_j ← 0
4:     UpdateOffer()
5: end upon
6: upon receiving a new capacity of c_j
7:     UpdateOffer()
8: end upon
9: upon receiving claim from bidder i
10:    claims[i] ← claim // Remember the claim of bidder i.
11:    UpdateOffer()
12: end upon
13: upon bidder i joining auction j
14:    D_j ← D_j ∪ i // Demand i now requires resource j.
15:    UpdateOffer()
16: end upon
17: upon bidder i leaving auction j
18:    D_j ← D_j \ i // Demand i no longer requires resource j.
19:    UpdateOffer()
20: end upon
21: procedure UpdateOffer()
22:    D*_j ← ∅
23:    A_j ← c_j
24:    done ← False
25:    while ( done = False ) do
26:        // If D*_j contains all bidders in D_j, then auction j does not
27:        // constrain any of the bidders in D_j.
28:        if ( D*_j = D_j ) then
29:            done ← True
30:            offer ← A_j + max({claims[i] : i ∈ D_j})
31:        // Otherwise, auction j constrains at least one bidder in D_j.
32:        else
33:            done ← True
34:            // What remains available is offered in equal portions to the
35:            // bidders constrained by auction j.
36:            offer ← A_j/|D_j \ D*_j|
37:            // Construct D*_j and compute A_j for the new offer.
38:            for all b ∈ {D_j \ D*_j} do
39:                if ( claims[b] < offer ) then
40:                    D*_j ← D*_j ∪ b
41:                    A_j ← A_j − claims[b]
42:                    done ← False
43:    send offer to all bidders in D_j
44: end procedure
```

are accounted for. W_i^* is the next contention window size. s_i^* is the airtime allocation. σ is the length of a backoff slot as defined by the 802.11 specification. C is the length of the observation period (note C as used here is different from how we typically use it to represent the contention window size). F is amount of time backoff was frozen during C. A is airtime during C. n is number of channel accesses during C.

$$W_i^* = \frac{2}{\sigma} \cdot \frac{A(1 - s_i^*)}{n \cdot s_i^*} \cdot \left(1 - \frac{F}{C - A}\right). \tag{2.1}$$

2.5 Summary

In the following chapters any experiment we present can be assumed to have been performed on the Wilab2 testbed discussed in this chapter. Similarly all references to the complete, star, or line topologies refer to the topologies shown in this chapter and use the exact testbed nodes highlighted on the testbed maps. Next we will build upon previous work on REACT by considering its efficiency and possible optimizations.

Chapter 3

REACT PROTOCOL OPTIMIZATIONS

When considering REACT protocol optimizations there are two areas of concern: convergence time and protocol overhead. There are two different aspects of the protocol that could be targeted for optimization, namely the update algorithms or the messaging scheme, in order to improve performance in either of these areas. Both aspects of the protocol affect convergence time but in different ways.

If fewer rounds of sending out messages are required for the protocol to converge then convergence time is reduced by the number of rounds saved multiplied by the amount of time each of those rounds took. Optimizing the update algorithms reduces the time each round takes; if updated offers and claims can be computed more quickly then a new message can be sent out sooner. When it comes to protocol overhead, however, only the messaging scheme is really relevant. Reducing the time it takes to compute updated offers and claims does not affect the number of messages that are required for the protocol to converge.

In Section 3.1 shows that convergence time is tightly bounded. Section 3.2 focuses on reducing messaging overhead.

3.1 Convergence Time

There are two aspects of the REACT protocol that are relevant to convergence times. The time it takes to update claims and offers affects convergence time and the number of rounds of messages that must be sent before the auction converges does as

well. Below we show updates can happens quickly and that at most three rounds of messaging are required for convergence.

3.1.1 Update Time

Claims and offers can be updated quickly. To show this we ran a REACT node with varying numbers of simulated neighbors and measured how long it took the node to update its claim and offer. This "simulation" (in which REACT was running on a testbed node) was done by generating and sending to the node a number claim and offer update messages equivalent to what it would get if it had the number of neighbors in that trial. Table 1 shows the results for a number of neighbors ranging from 10 to 100000. In the first column are the number of neighbors that were used in that trial. The second column shows how long a full update took, which includes updating the node's neighbor list, claim, and offer for each simulated message. Even for the completely unrealistic value of 100,000 neighbors all the updates happen in very little time, 237.78 ms.

Table 1: REACT update time vs. number of neighbors

Neighbors	Processing Time (ms)
10	0.03814
100	0.63896
1000	6.1771
10000	21.211
100000	237.78

3.1.2 Convergence Proof

After working with the REACT protocol on many topologies we noticed that it only ever took 3 rounds at most for it to converge. We now prove that this is the case.

First we start with a few simplifying assumptions. There are n nodes. All nodes start running REACT at the same time and send out a message after some set interval. The length of this interval is considered one round. At startup all nodes send out a message containing offers and claims to the other nodes. Each node can process every message received and update its own offer and claim before the end of the round. There are no one-way connections; if a node gets a message from another node then that node also got its message. All these assumptions are generally true in our experiments.

The first offer and claim is received by every node from its neighbors. To compute its new offer a node tries to satisfy all claims it has received. If it can satisfy all claims it has received then its new offer is whatever airtime was left after satisfying all claims plus the maximum claim. When it sends this offer out no neighbor increases its claim because every node's claim was satisfied and the new offer is at least as big as the largest claim. If the node cannot satisfy all claims then it sends out an offer that is equal to its capacity divided by the number of neighbors. Now begins round two. Every node gets new offers from their neighbors. Their claim, at this point, can only go down. Either their claim was satisfiable or they got an offer that reduces their claim. The node's claim now becomes the smallest offer or remains unchanged (equal to its original demand). Now begins round three. The new claim is sent out to the first node. All claims are now at least as small as the offer it sent out previously and so it able to satisfy every offer. REACT has now converged in three rounds.

3.2 Messaging Overhead

Protocol overhead is one of the primary drawbacks to using REACT when compared to 802.11 DCF. Every byte in every control message REACT sends, or every byte that is added to an existing header, is a byte of information that would not be sent or need to be sent when using 802.11 DCF. In previous work REACT improved fairness in throughput between nodes but at the cost of lower aggregate throughput for the whole network [13]; less relevant data (i.e., data nodes care about and not control messages) were sent when using REACT. By reducing REACT protocol overhead we mitigate this problem and make it less costly to use REACT from an aggregate throughput perspective. To do this we must first look at the different options for implementing protocol messaging. The version of REACT that was implemented at the kernel level from [13] did this by adding fields to existing 802.11 headers. The new user-space REACT introduced in this thesis instead sends out dedicated control messages. Since user-space REACT sends out control messages we must also determine when or if to send them out, and this seemingly simple but nuanced decision can affect if and how REACT converges as well as protocol overhead.

3.2.1 Comparison of Options for Reducing Messaging Overhead

In [13] REACT was implemented within the Linux kernel. The implementation added a specific header encapsulating the full state of the protocol for that node to each 802.11 packet. The fields in this header and their sizes can be seen in Figure 8. A total of 15 bytes was added to every single 802.11 packet. This is an additional 15 bytes of overhead for each packet sent, and the overhead for 802.11 packets is already

high. On a cumulative basis this overhead increases with every packet sent, and as the data rate increases the associated overhead also increases (unless the application is sending out more data in each packet, as opposed to more packets per unit of time).

```
1  struct state_header {
2      u8 mac[6];
3      u8 closed;
4      u8 finished;
5      u8 offer;
6      u8 claim;
7      u8 w;
8      u8 status_bidd;
9      u8 status_auct;
10     u16 reserved;
11 } state_header;
```

Figure 8: REACT kernel level implementation headers appended to 802.11 headers

There are also two disadvantages to this scheme that are not completely relevant to messaging overhead but still worth mentioning. Since the state of the REACT protocol is communicated via headers, nodes won't be updated on their neighbors offers and claims unless their neighbors are transmitting. Thus if even a single node can't transmit the protocol won't converge until all nodes have gotten sufficient chances to transmit. Since the protocol influences which node gets to transmit this could have the double effect of keeping the auction from converging and excluding a node that hasn't had sufficient chances to transmit because it is not part of the auction and will have trouble getting sufficient chances because it is not part of the auction. In previous work this problem did not manifest to the point where it could be deemed significant. However it is ironic that the overhead associated with modified headers happens only and exactly when we don't want it to happen—when nodes

are transmitting data—and not when the network is idle and could accept control messages at basically no cost to aggregate data throughput.

The second disadvantage with this scheme is that the update algorithms must run every time a packet is sent out. Similarly, updated state must be extracted from each incoming packet. Before a packet is sent out the offer and claim must be recomputed based on these values extracted from incoming packets, and the fresh offer and claim are put into the state header that is inserted in to the standard 802.11 packet. As we have seen, the update algorithms are not a big concern time-wise because they do not add much latency to each packet. However this is processing time that is spent for each packet in addition to the transmit time overhead added to each packet due to the 15-byte header.

In this thesis we introduce a new implementation of REACT that runs in user-space. It does not modify 802.11 packet headers. Instead, user-space REACT sends out dedicated control messages. Each control message updates other nodes on the sender's state, and in the implementation this message is 63 bytes in size. The naïve implementation of user-space REACT follows the exact description of the REACT protocol in [12] sends out control messages as fast as it processes messages from other nodes. This means a large number of control messages get generated and in fact so many that almost no data can be sent. The entire network capacity would be taken up by control messages if user-space REACT was implemented exactly how the protocol is specified in pseudo-code.

One way this deluge of messages could be reduced might be to only send out a message if a received offer or claim from a particular node changed from last time an offer and claim was received from that node. This turns out to be a bad idea because if for any reason an offer or claim is lost in transmission the protocol can grind to a

halt without converging. The two solutions for user-space REACT that were actually implemented and tested were a dual-queue queuing discipline and simply sending control messages out on a timer.

In the Linux kernel a *queueing discipline* (qdisc) determines how packets get buffered when waiting to get transmitted and in what order they are finally taken out of a buffer to be transmitted or dropped. For user-space REACT we implemented a dual-queue qdisc. Qdiscs are part of the kernel, and so the REACT qdisc was implemented as an external kernel module that can be dynamically loaded into a running kernel. Once loaded the qdisc is then assigned to a particular network interface, and its queuing/dequeueing interface is then used transparently by the network stack.

The first queue in our qdisc is for non-control packets; it operated with standard FIFO mechanisms and is bounded by the system's default `tx_queue_len` for that device. The second queue is only for control packets. Its length is bounded to one, and every time a new control packet enters the queue if there is an older packet in the queue the older packet is dropped and the new packet takes its place. This means that whenever a packet is taken out of the control queue to be sent it contains the most up-to-date state for that node. The control queue also take precedence over the non-control queue; if there is a packet in the control queue it will be sent first. A control packet is still sent out whenever there is a control packet available to send, but the qdisc reduces the number of packets sent because stale control packets are dropped.

The other option for user-space REACT, after the qdisc, is to send control messages out after a set time interval. This interval must be longer than the amount of time it takes to update bids and offers, but that is the only restriction. It is possible that using the approach makes it take longer for REACT to converge if the interval time is

much longer than the time it takes to compute offers and bids, but the interval should also greatly reduce the overhead of control data. With this method the overhead for control data does not increase when the data rate increases, unlike with the modified headers method.

3.2.2 Experimental Evaluation for Reducing Messaging Overhead

To evaluate the different options for reducing messaging overhead an experiment was performed. In this experiment there were two nodes. There was one greedy UDP flow between the nodes. A comparison was made between four options: 802.11 DCF (as a control), the naïve method, the qdisc method, and the time interval method. For each method throughput of the greedy flow and convergence time was measured. The results are given in Table 2. We see that with no control messaging, just 802.11 DCF, the throughput is very high. This throughput drops off very quickly with the naïve method. Naïve user-space REACT is sending so many control messages that throughput suffers considerably. In terms of throughput the qdisc method is even worse. It turns out that even with the evicting of stale control messages REACT can generate them so quickly that if they are prioritized almost no data can be sent. The qdisc method converges the fastest, but this means almost nothing when barely a few kilobits of data is being sent. In fact, the qdisc method does not even converge that much faster than the naïve method. Finally we see that the time interval method's throughput is almost as good as 802.11. Unlike any of the other methods the time interval method's throughput is actually comparable to 802.11, but its convergence time is more than double the other methods. The interval time was set to 100 ms according to results from the update time experiment. This interval is long enough

that even updates for 10,000 neighbors would in less than a quarter of the interval time. Even with the slowest convergence time the interval method's convergence time is not a problem and insignificant compared to the tuning convergence times seen in Chapter 4.

Table 2: Throughput and convergence time by messaging implementation strategy

Variant	Throughput (kbps)	Convergence Time (ms)
802.11	4970.2	N/A
Naïve	571.70	95.749
qdisc	2.6834	67.720
Interval	4969.0	237.42

3.3 Summary

In this chapter we have dived into the REACT protocol and considered how it can be best implemented. Our implementation works primarily in user-space and has been carefully designed to reduce REACT protocol messaging overhead. This sets the stage for the introduction of our new tuning algorithm in the next chapter that works alongside REACT to achieve airtime allocations.

Chapter 4

CONTENTION WINDOW TUNING

The REACT protocol allocates resources between nodes participating in an auction, and contention window tuning is the tool we use to realize that allocation. Tuning of this sort is a potent tool and its power comes from the fact that it operates at the MAC layer. As discussed in the introduction, this layer is the primary source of performance issues and unfairness in ad-hoc networks. Moreover every packet that is sent out onto the wireless medium must pass through the MAC layer. Thus the total airtime consumed by a node, without distinction by flow, is affected by contention window size choices. In this chapter we describe our novel algorithm SALT (Smoothed Airtime Linear Tuning) and its implementation in sections 4.1 and 4.2. Next we describe how SALT was evaluated on the testbed in Section 4.3. To evaluate SALT it is first important to determine what values should be used for the algorithm's parameters, β and k. In Section 4.3.1 we make this determination. With β and k worked out we can compare SALT to the original contention window tuning algorithm as presented in [13]. This comparison is done in Section 4.3.2.

4.1 SALT: Smoothed Airtime Linear Tuning

We propose SALT as a new contention window tuning technique. Contention windows are used by the 802.11 backoff algorithm to determine how long a node should wait before attempting to transmit again after there has been a collision. SALT changes the backoff process by programming a value C_t such that $C_{min} = C_{max} = C_t$. The

backoff counter is still picked from the range $[0, C_t)$, but C_t is already at its maximum value and cannot be doubled after transmission failures. We bound the size of the contention window in this way because in [15] it was shown that a node's transmission probability depends on the average size of its contention window. Since SALT directly controls this size by programming C_t it can change the node's transmission probability and tune C_t to achieve the airtime allocated to the node by REACT.

SALT measures a node's airtime over a set interval and then uses that measurement, a_t, to set C_t, the contention window size for the next interval. However a_t is not passed directly to the tuning component of SALT. Using an exponentially weighted smoothing technique the smoothed airtime, S_t, is computed from a_t and then used for tuning. β is the smoothing parameter and is limited to the range $(0, 1]$. It controls how heavily past airtime measurements are weighed when computing S_t. The closer β is to zero the less C_t fluctuates in response to changes in the measured airtime. Smoothing is done to reduce the effect of random background noise on the contention window tuning process.

$$S_t = \begin{cases} a_1 & \text{if } t = 1 \\ \beta a_t + (1.0 - \beta) S_{t-1} & \text{if } t > 1 \end{cases}$$

You will notice that the t subscript for S_t and a_t start at 1. This is because a_t represents the airtime measured over interval t so there is no a_0; our first airtime measurement is available after the end of the first interval. There is however a C_0 in SALT's tuning component, shown below, because before the beginning of the first

airtime observation interval $C_0 = 0$ is programmed.

$$C_t = \begin{cases} 0 & \text{if } t = 0 \\ \lfloor S_t - \alpha \rfloor k + C_{t-1} & \text{if } t > 0 \text{ and } 0 \leq \lfloor S_t - \alpha \rfloor k + C_{t-1} < 1024 \\ 1023 & \text{if } t > 0 \text{ and } \lfloor S_t - \alpha \rfloor k + C_{t-1} \geq 1024 \\ 0 & \text{if } t > 0 \text{ and } \lfloor S_t - \alpha \rfloor k + C_{t-1} < 0 \end{cases}$$

A starting value of 0 was chosen to minimize the amount of time nodes spend in backoff. If the nodes do not achieve their airtime because there was too much contention the successive C_t will be adjusted upwards but realizing allocations with the lowest possible C_t will maximize throughput. In the equation above α is the node's airtime allocation from REACT and k is a constant scaling factor. C_t is inversely related to a node's transmission probability [15] and so C_t increases when $S_t > \alpha$ or decreases when the opposite is true. The difference between S_t and α is scaled by k to convert the difference of unit-less airtime ratios into a contention window size value.

4.2 SALT Implementation

SALT is implemented as a user-space Python program running on each Linux testbed node in an experiment. It is part of the same program that is running REACT, and the airtime allocation is passed from the thread running REACT to the thread running SALT. The airtime measurement interval used is one second. SALT is invoked after each of these one second intervals and uses data collected by the networking subsystem to determine the airtime during the last interval. The kernel interface for accessing the data used to compute airtime is a preexisting interface, but the interface that SALT uses to set C_{min} and C_{max} required modifying the kernel.

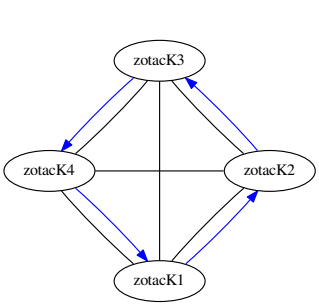
Linux was modified to expose C_{min} and C_{max} to user-space programs. The interface

allows a user-space program to set these parameters and then the Linux kernel's networking subsystem honors the values. In the future patching the Linux kernel in this way might not even be necessary. The 802.11e standard includes a feature called Enhanced Distributed Channel Access (EDCA) and allows for the contention window size to be set for each EDCA access category. The wireless subsystem also had to be patched to accept contention window sizes that were not powers of 2. This patch would still be necessary with full EDCA support.

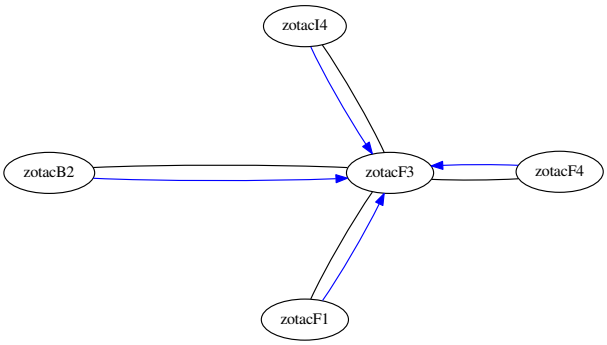
4.3 SALT Evaluation

SALT was evaluated on the three different Wilab2 testbed topologies described in Section 2.1.1. All the topologies are shown in Figure 9 with links between nodes shown as black lines (all connections are bi-directional) and the flows shown as blue arrows (denoting directionality). In the complete topology there are four single-hop flows, one originating at each node to the next one. The star topology also has four flows but five nodes. Each flow originates from one of the four outer nodes and terminates at the center node. The line topology has four flows: two from the outside nodes in and two from the inside nodes to each other.

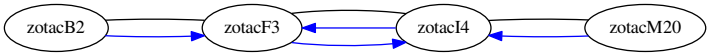
In each topology in each experiment in this chapter we used greedy UDP flows. These flows were setup with a target 1 Gbps UDP bandwidth far beyond the capabilities of the wireless link (i.e., the channel was saturated), and we measured the throughput of these flows as seen by the destination node. Jitter, drop rate, aggregate throughput, and airtime measurements were also collected.



(a) Complete Topology



(b) Star Topology

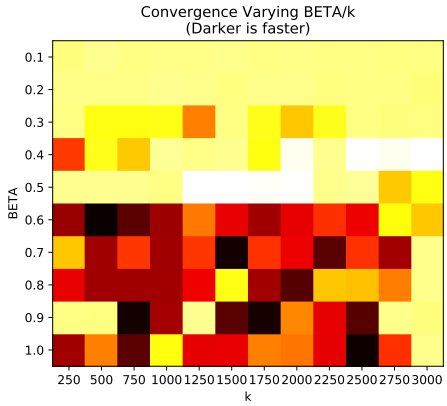


(c) Line Topology

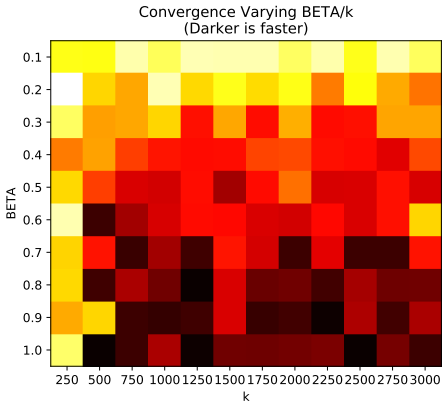
Figure 9: All testbed topologies with single-hop flows

4.3.1 β,k Parameters Experiment

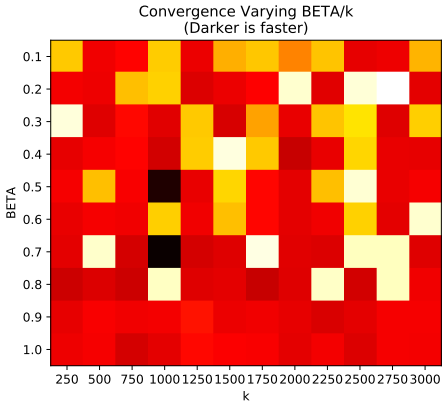
In order to answer the question of what values we should use for β and k we performed an experiment where these parameters were both varied over ranges of their possible values. We varied β from 0.1 to 1.0 in steps of 0.1 and k from 250 to 5000 in steps of 250, making for 120 trials on each of the three topologies. Each trial lasted for 15 seconds and we measured airtime data for each node in the trial over that time. The amount of time it took for airtime to converge in each trial was then computed using the CV measurement. In this case we used a CV threshold of 0.15. The heat maps in Figure 10 shows the convergence results for each topology and each combination of β and k. The darker the square in the heat map the faster that particular trial converged. All graphs share a small clustering of dark squares towards the bottom left quadrant.



(a) Complete Topology



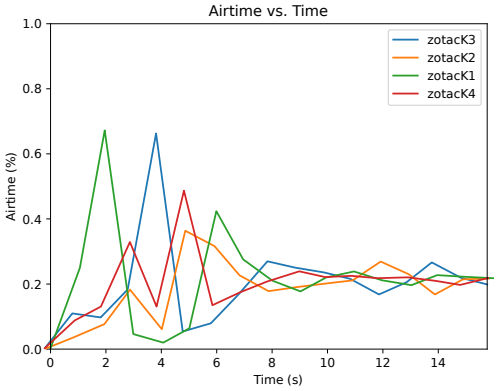
(b) Star Topology



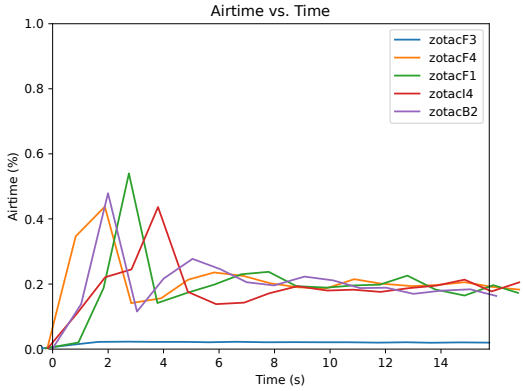
(c) Line Topology

Figure 10: Convergence time heat maps for each β,k on each topology

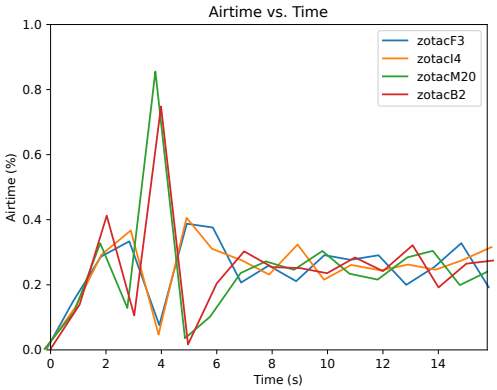
Now with this data the question remains regarding which β and k pair to select. To do this we averaged the convergence time for each trial on each topology that used the same β, k and then picked the lowest average. This turned out to be a β of 0.6 and k of 500. This gives the best average convergence time of 7.44 seconds. The airtime versus time graphs using this β, k are shown in Figure 11 for each topology. We can see in these graphs that the airtime consumed by each node did in fact converge very quickly in each trial on each topology.



(a) Complete Topology



(b) Star Topology



(c) Line Topology

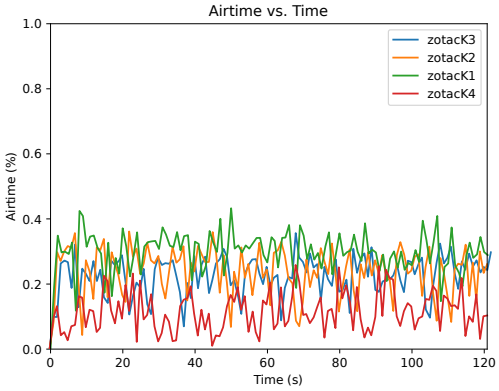
Figure 11: Airtime vs. time graphs with $\beta = 0.7, k = 500$ for all topologies

4.3.2 Comparison Experiment

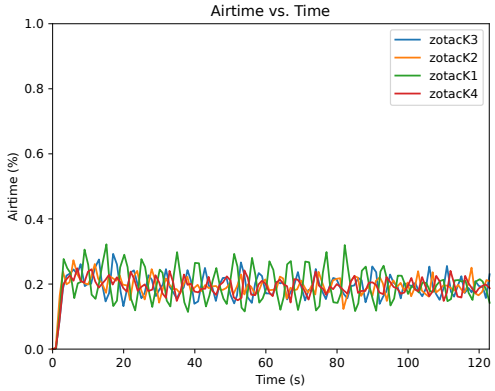
In this experiment we compared SALT to the Garlisi et al. [13]. 802.11 DCF was also tested as a control, and both contention window tuning algorithms were being run alongside REACT. The setup for this experiment was the mostly the same as the last experiment. However the β and k parameters for SALT were set according to our experimental results ($\beta = 0.6$, $k = 500$), and one trial was performed per 802.11 DCF, SALT, and Garlisi et al. [13]. The flows in this experiment were still greedy UDP

flows setup as shown in Figure 9. A wider variety of measurements were taken during these trials including airtime, throughput, drop rate, and jitter.

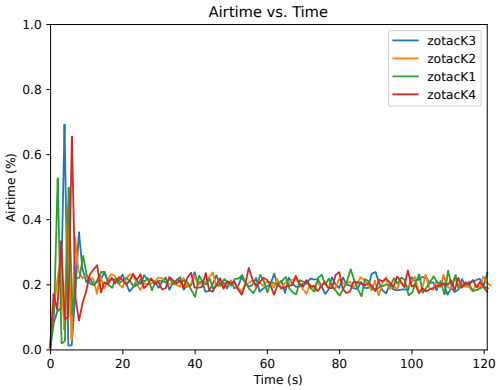
The airtime results for the complete topology can be seen in Figure 12. On this topology like all the others the 802.11 DCF airtime measurements are all over the place. Garlisi et al. [13] converges much more quickly than SALT but not as tightly. In fact, with a lower CV threshold of 0.10 it does not converge until after more than 100 seconds have passed. Figure 15 shows throughput, jitter, drop rate, and aggregate throughput graphs for the complete topology. The tighter convergence with SALT leads to the higher throughput and generally lower jitter that can be seen in these graphs. Both Garlisi et al. [13] and SALT have a near zero drop rate while 802.11's drop rate is very high, above 0.6 for one node. However even with a high drop rate 802.11 still comes out ahead in throughput with SALT coming in second above Garlisi et al. [13]. These results are broadly similar to the ones for the star and line topologies seen in Figures 13 and 16 for the star topology and Figures 14 and 17 for the line topology. One aberration of note is the throughput for Garlisi et al. [13] in the line experiment. The airtime graph shows that all the nodes converged very quickly but the throughput graph shows almost zero throughput for any node but zotacB2. The explanation for this can be seen in the drop rate graph in that the drop rate was very high for all the nodes except zotacB2. The throughput that we measured was that as seen by the receiver so while these nodes were getting airtime and transmitting, drop rates were high and actual data was not getting received. In the line topology this a particular problem because it combines both hidden and exposed node flows.



(a) 802.11



(b) Garlisi et al. Tuning Algorithm

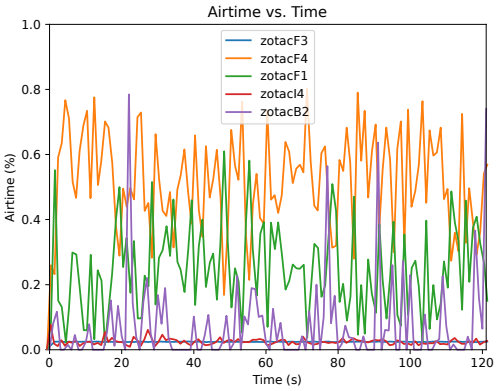


(c) SALT

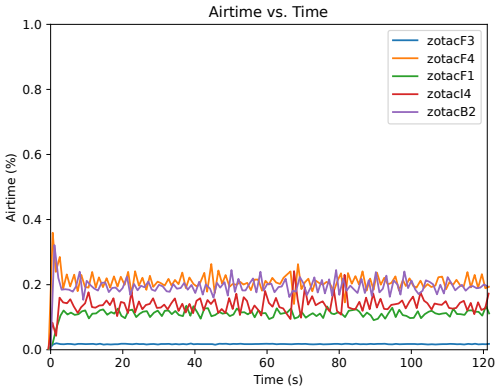
Figure 12: Comparison experiment airtime vs. time graphs for complete topology

4.4 Summary

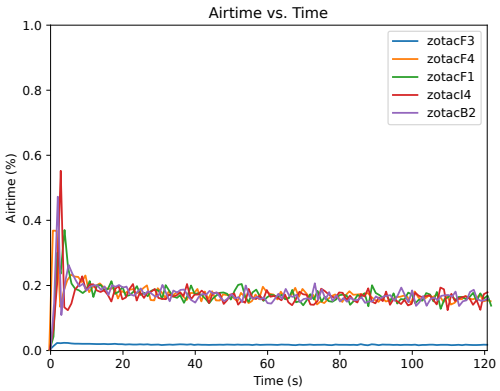
In this chapter we introduced SALT and showed it is an effective contention window tuning algorithm. However all the experiments in this chapter used single-hop flows. In the next chapter we will extend REACT and SALT to the multi-hop scenario.



(a) 802.11

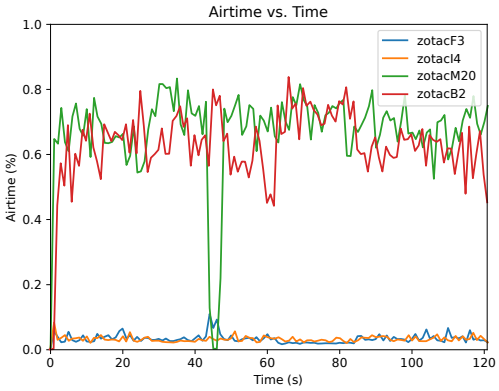


(b) Garlisi et al. Tuning Algorithm

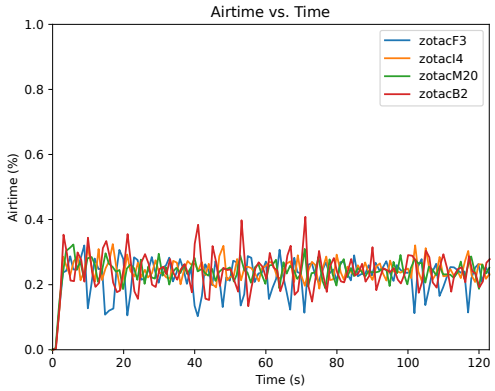


(c) SALT

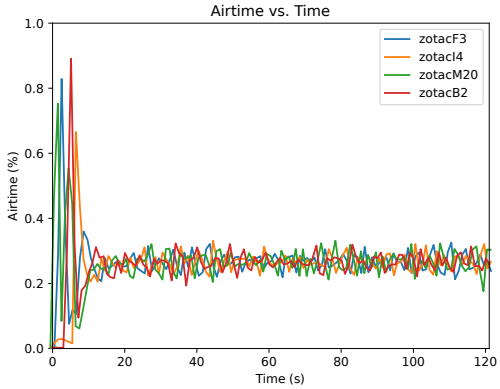
Figure 13: Comparison experiment airtime vs. time graphs for star topology



(a) 802.11



(b) Garlisi et al. Tuning Algorithm



(c) SALT

Figure 14: Comparison experiment airtime vs. time graphs for line topology

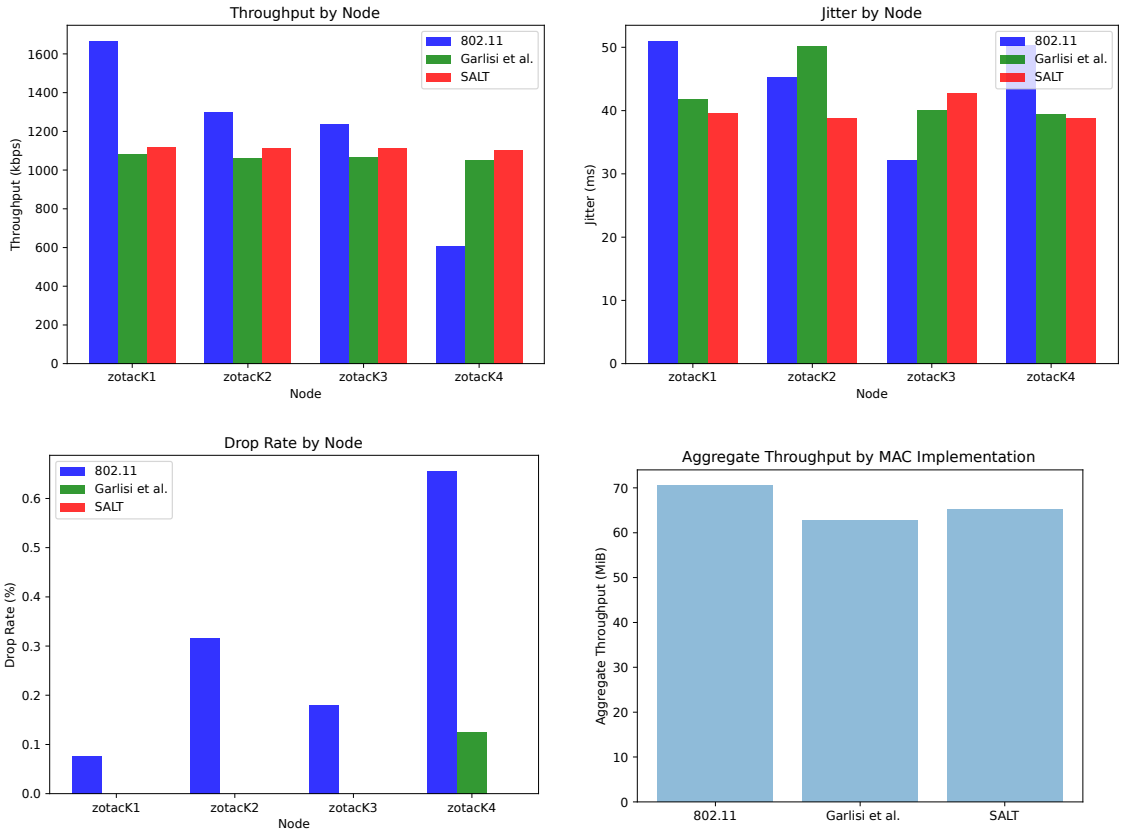


Figure 15: Comparison experiment statistics for complete topology

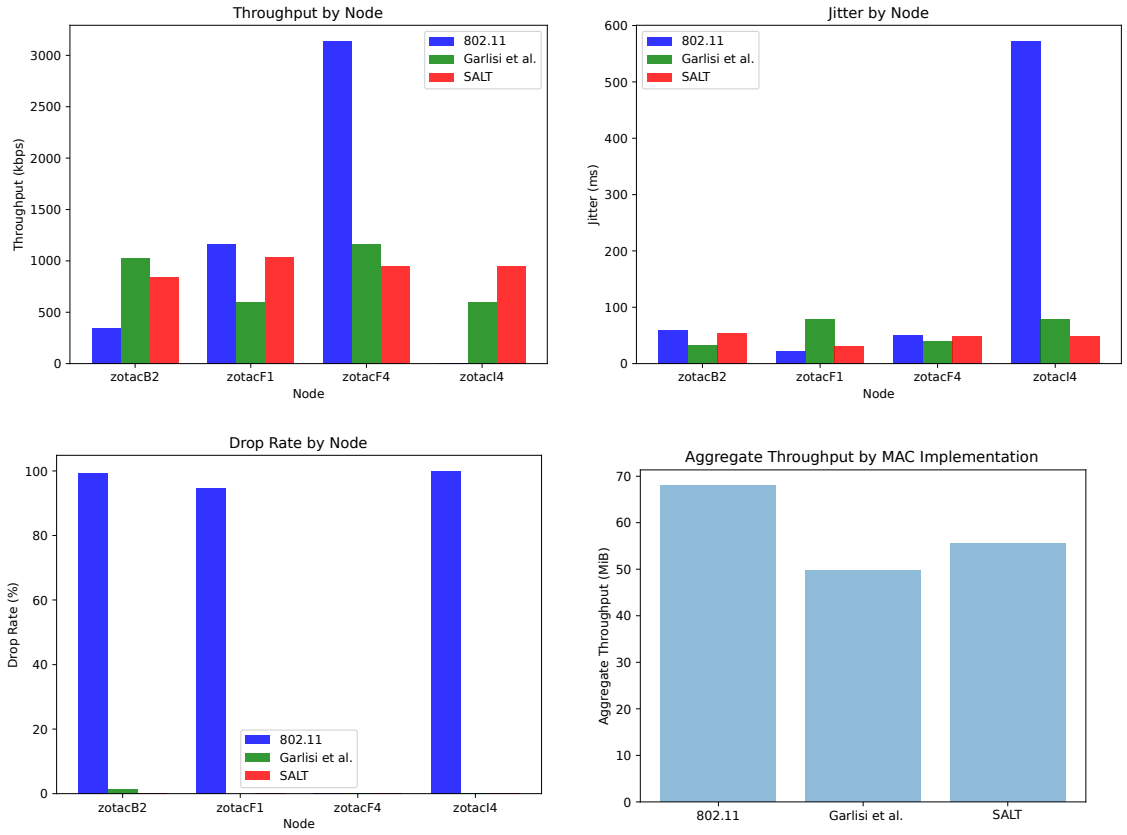


Figure 16: Comparison experiment statistics for star topology

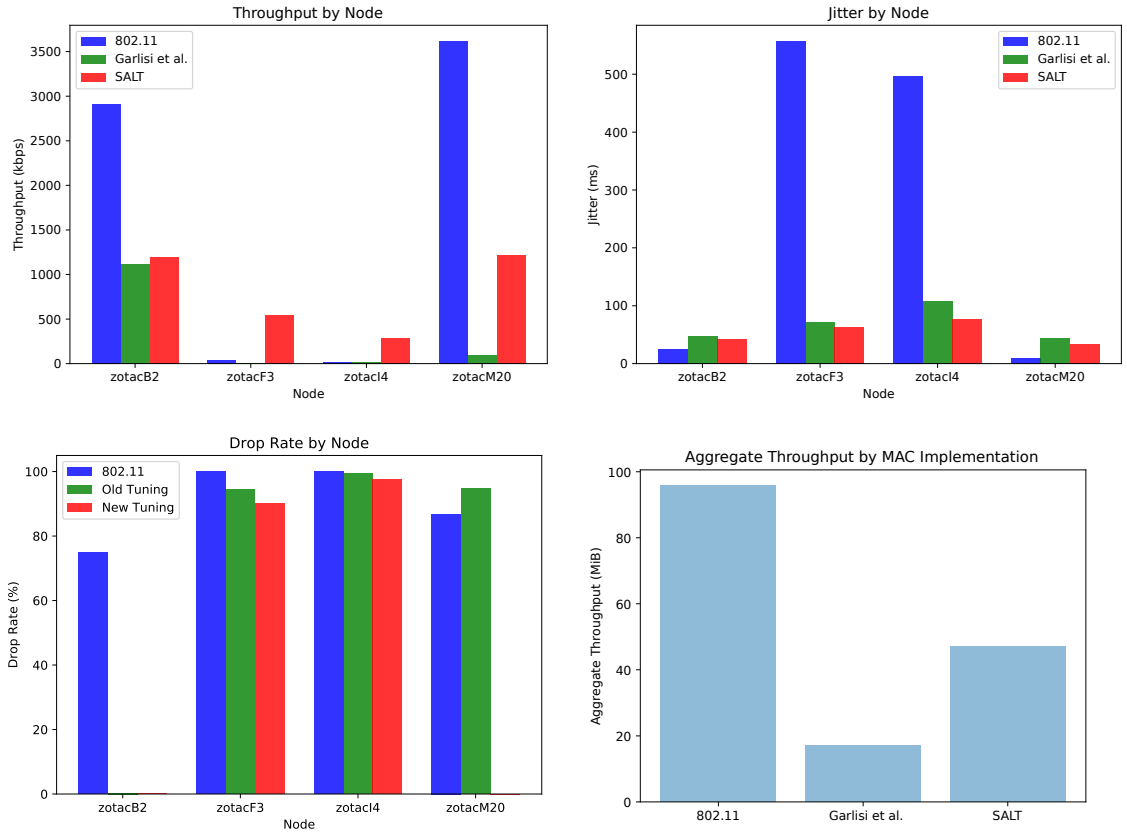


Figure 17: Comparison experiment statistics for line topology

Chapter 5

MULTI-HOP REACT AND MULTI-HOP RESERVATIONS

Until now, nodes running REACT have only taken into account their own traffic needs. In the auction a node's bid secures airtime for itself, but if there are multi-hop flows in a network this is insufficient because it does not take into account the fact that a node might need to forward traffic ultimately destined for others. In this chapter we present a multi-hop airtime reservation protocol that addresses this issue.

Without a reservation algorithm a node could try to predict how much airtime to reserve for multi-hop flows passing through it. Nodes store each of their neighbor's claims and could make a guess based on this regarding what their demand should be to reflect the possibility of multi-hop flows. Unfortunately claims provide no information on the directionality of flows, multi-hop or not. A claim is sent to every node within broadcast range and only informs the receiver that the sender is currently expecting to utilize the amount of airtime claimed. Claims also do not tell a node anything about the demands of nodes beyond their neighbors, where the multi-hop flow could be originating. Multi-hop reservations allow the originators of multi-hop flows to inform nodes of the additional traffic they are expected to forward. Nodes along the reservation path can also inform the originator of resource saturation. The REACT auction itself is a convenient mechanism that can be used for the purpose of making these reservations and one that has no analogue in standard 802.11.

Each auction in the REACT protocol allocates as much capacity at a particular node as that node chooses. In our algorithm, a reservation is made by reducing this capacity by the reservation amount at nodes along the path and their neighbors. Once

the reservation is placed nodes along the reservation path increase their allocations by the reservation amount. This secures airtime for the flows that will be passing through the node while still maintaining the standard REACT auction for allocating airtime in the neighborhood. Section 5.1 provides a more precise description of this process and Section 5.2 presents our evaluation of it.

5.1 Multi-hop Algorithm Description

The multi-hop reservation algorithm is split into two parts. There is an "placement" algorithm for the node placing the reservation, and there is a "confirmation" algorithm for nodes whose capacity is being reserved. When attempting to make a reservation the reservation can fail to be placed if enough capacity cannot be reserved. For simplicity we assume that packets take a static route through the network. We define a reservation as an amount of airtime that a node wants to reserve at each node along a certain path. Presumably that path would be the path of a future multi-hop flow the reserving node initiates (although perhaps there are other uses for reservations we cannot anticipate and of course the flow could have already been started). If the reservation is made successfully then the network has given a guarantee to the reserving node that the airtime allocated to each node along the path is at least as much as the reservation amount, not counting any airtime those nodes have reserved for other reservations or have been allocated in the process of bidding.

Algorithm (Placement)**.** *Let A be the node making airtime reservation r from itself to node Z. Determine the next node on the path to Z and let this node be B. Send (A, r, B, Z) to all neighbors but B. If a single neighbor returns a failure message then the reservation fails. If no neighbor returns a failure message then continue*

and send (A, r, B, Z) to node B. If B returns a failure message then the reservation fails otherwise the reservation was successfully placed. If the reservation fails send a message to all neighbors telling them to unreserve any airtime that was reserved in this node's name for A.

Algorithm (Confirmation)**.** *Receive reservation (A, r, C, Z) from a node B. Check if there is at least r unreserved capacity at this node. If there is enough capacity then reserve it in B's name for A's reservation. If there is not then return a failure message. If this node is not node C (meaning it could be Z or just an ordinary neighbor) then return success to node B. Upon reaching this point the node must be node C. Determine the next node on the path to Z and let this node be D. Send (A, r, D, Z) to all neighbors but D. If a single neighbor returns a failure message then the reservation fails. If no neighbor returns a failure message then continue and send (A, r, D, Z) to node D. If D returns a failure message then the reservation fails. If the reservation fails send a message to all neighbors telling them to unreserve any airtime that was reserved in this node's name for A. Return success or failure to node B depending on if reservation succeeded or failed.*

5.2 Multi-hop Reservation Evaluation

To evaluate the reservation process we tested REACT and 802.11 on the line topology with a multi-hop TCP flow. Figure 18 shows this topology with the multi-hop flow colored green. In the trials that used REACT and the reservation was placed first and then the TCP flow was started. The reservation placed was for 26.6% airtime, approximately the maximum amount of airtime that could be reserved in this scenario. The TCP flow lasted for 120 seconds in both the REACT and 802.11 trials. Multi-hop

routing was done statically with each node programmed with neighbor information before the two minutes started.

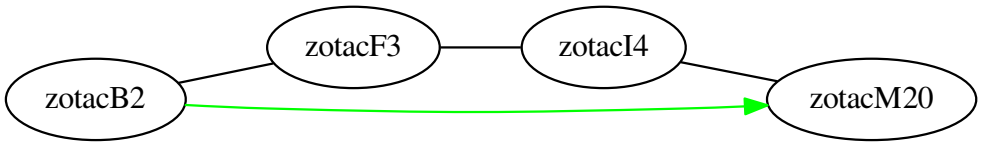
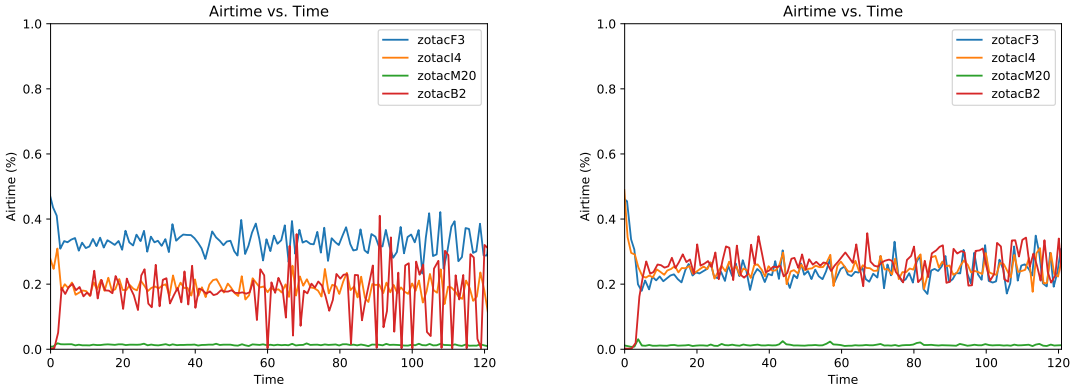


Figure 18: Line topology with multi-hop flow

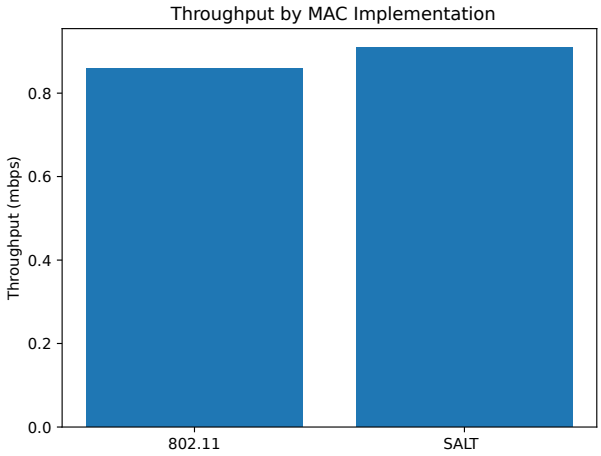
In this experiment the multi-hop flow achieved a higher throughput when REACT was running than when using standard 802.11. This result can be seen in Figure 19 along with airtime graphs for both REACT and 802.11. The reservation of 26.6% airtime was placed after 0.9063 seconds and REACT converged after 9.041 seconds with a CV threshold of 0.15.

Three primary reasons contribute to throughput being higher for REACT. First the tuning algorithm parameters are themselves tuned precisely allowing REACT to converge quickly to the reserved airtimes of 26.6%. This means that for the majority of the experiment nodes are able to access the channel fairly and contend less. Convergence time would be less important in a longer running experiment as the time during which REACT had not yet converged would be shorter in proportion to the rest of the experiment. The second reason has to do with how TCP congestion control is affected by the reduced contention after convergence. TCP was originally designed for wired networks and interprets dropped packets as being caused by network congestion [16]. Wireless networks on the other hand can experience packet loss for a variety



(a) 802.11

(b) REACT with Reservation



(c) Throughput Comparison

Figure 19: SALT/REACT with multi-hop TCP reservation placed vs. 802.11

of reasons including interference from other nodes in the network. This is especially problematic in multi-hop ad-hoc networks that intrinsically have hidden and exposed nodes. However when REACT was running in our experiment nodes along the path of the multi-hop flow were able to access the medium fairly and according to their allocations and thus contend less. Finally, 802.11 nodes did not get fair access to the channel and the TCP flow was severely unstable. These periods of instability reduced 802.11 throughput.

5.3 Summary

While the throughput difference between REACT and 802.11 in our multi-hop scenario was small REACT's throughput was still higher. REACT also provided for much more fair transmission opportunities between nodes. In previous experiments without multi-hop flows we had only seen REACT provide the latter while suffering from throughput degradation. This shows that in the multi-hop TCP scenario the reduction in contention that REACT provides is even more important and leads to gains in network efficient beyond just fairness.

Chapter 6

CONCLUSION

In this thesis we have shown that REACT and contention window tuning are even more effective techniques than anticipated. Previous implementations of REACT required substantial changes to the kernel and modified MAC level headers in a way that made nodes incompatible with ones not running REACT kernels. On the contrary, the implementation of REACT in this thesis exists primarily in user-space, and we demonstrated its effectiveness while maintaining minimal messaging overhead and with only small changes to the kernel. These kernel modifications expose an interface for setting the contention window size but do not prevent REACT nodes from interoperating with nodes not running REACT. We have also introduced a new tuning algorithm called SALT. Our algorithm converges more tightly to REACT's airtime allocations than previous implementations, and tighter convergence leads to further reductions in unfairness and jitter as compared to 802.11 DCF. Finally we took REACT and SALT to the multi-hop scenario by introducing a new multi-hop reservation algorithm that leverages the airtime allocation and realization capabilities of this combination. With a reservation in place we have shown that REACT and SALT do reduce contention enough in a multi-hop topology that a multi-hop TCP flow exhibited higher throughput than in the trial that used 802.11 DCF. There is still much work that could be done to advance REACT and SALT.

One aspect of REACT that has not been addressed in real-world implementations of the protocol is that of nodes leaving an auction. In our experiments this was unnecessary because nodes were statically configured and immobile. There are many

ways this could be accomplished and many reasons for why nodes may leave the auction. The toughest scenario is when a node goes offline unexpectedly (e.g., loss of power). Mobile nodes are a similar scenario. While less catastrophic, mobile nodes will still cease being able to communicate with other nodes as they move away from them, and a mobile node will probably not be able to predict exactly when its transmissions will start to be lost. In simulation [12] used neighbor timeouts to determine when to evict nodes from the auction and this would appear to be the only option for unexpected neighbor loss. The timeout selected cannot be so short that nodes are unduly evicted as this would cause unnecessary fluctuations in the allocation, but the longer the timeout the longer essentially open capacity remains unused. A neighbor timeout several times the length of the period control messages are sent out on would probably be safe enough for our implementation of REACT while still being near to as short as possible. If a node has finished transmitting and would like to forfeit its airtime before the neighbor timeout the easiest way would be to reset its bid to zero.

One aspect of SALT that could be improved is the time that it takes to converge. It takes longer than the Garlisi et al. algorithm and at 7-9 seconds is perhaps a bit too high for some applications of REACT. In fact if nodes are joining and leaving the auction at around once every 10 seconds then SALT may never converge to REACT's airtime allocation. This improvement could be made by finding a β and k that minimizes convergence time, maybe to the disadvantage of throughput of jitter or fairness. A more interesting avenue of exploration would be to combine SALT with the Garlisi et al. tuning algorithm and perhaps leverage the benefits of both.

Adapting the reservation algorithm to a dynamic routing protocol is also left for future work. This would likely require communication between REACT and the routing software. If the route a multi-hop flow takes through the network changes

then a reservation placed for that flow along the old route needs to change. With our current algorithm we could accommodate this, albeit poorly, if the node that placed the reservation has some way to find out that the route has changed. This information could come from the routing protocol software or perhaps the reserving node could periodically perform route tracing. If the route changes the reserving node would need to withdraw its previous reservation and then place a new one, but this is far from optimal. A better solution might be to alter the reservation at only the nodes that changed along the route or to have the routing algorithm take the reservation into account when making routing decisions. In fact, a new routing protocol could be developed that subsumes the placement and confirmation functionality of the reservation algorithm, one which takes into account the reservations it has facilitated placing when determining routes.

Finally, more experiments could be done. Part of the point of REACT and SALT is to reduce jitter, and the applications this is most useful for are real-time audio or video. It would be interesting to send video or audio traffic in a experiment, instead of just UDP flows that serve as a proxy for this type of data, and then listen to or watch the results. One could even use SALT on a WLAN and place a video call to see if there is a noticeable improvement in the consistency of the quality of the call. This last experiment also brings up an area of research enabled by the user-space implementation of REACT: nodes using REACT can co-exist on the same network as nodes not using REACT. However it is not clear that there would be any benefit to using REACT and SALT if some neighbors are not. One would presume that selfish nodes would naturally consume so much of the channel that fair nodes would not be able to achieve their allocations or derive any other benefit from REACT. In the WLAN scenario as well, your competition becomes centrally scheduled MAC protocols

that take advantage of the presence of an access point in way that REACT and SALT do not.

The success of REACT with SALT and other tuning schemes in multiple topologies and with a variety flow configurations shows that our techniques hold a lot of promise. Our reservation algorithm has shown that SALT and REACT can reduce contention in multi-hop ad-hoc networks and that TCP does not have to be avoided in these scenarios. This work has contributed to enabling fair, scalable ad-hoc networks.

REFERENCES

[1] *Cellphone networks overwhelmed after blasts in Boston.* [Online]. Available: https://www.bostonglobe.com/business/2013/04/16/cellphone-networks-overwhelmed-blast-aftermath/wq7AX6AvnEemM35XTH152K/story.html.

[2] Amnesty International News, "Gambia: Communication blackout shatters illusion of freedom during the election," *Amnesty International*, 2016.

[3] N. Cohen, "Hong Kong protests propel FireChat phone-to-phone app," *The New York Times*, 2014.

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

[5] ——, "Does the IEEE 802.11 MAC protocol work well in multihop wireless ad hoc networks?" *IEEE Communications Magazine*, vol. 39, no. 6, pp. 130–137, 2001.

[6] M. Garetto, T. Salonidis, and E. W. Knightly, "Modeling per-flow throughput and capturing starvation in CSMA multi-hop wireless networks.," in *INFOCOM*, 2006.

[7] C. Chaudet, D. Dhoutaut, and I. G. Lassous, "Performance issues with IEEE 802.11 in ad hoc networking," *IEEE Communications Magazine*, vol. 43, no. 7, pp. 110–116, 2005.

[8] P. Bahl, R. Chandra, and J. Dunagan, "SSCH: slotted seeded channel hopping for capacity improvement in IEEE 802.11 ad-hoc wireless networks," in *Proceedings of the 10th annual international conference on Mobile Computing and networking*, ACM, 2004, pp. 216–230.

[9] S.-L. Wu, C.-Y. Lin, Y.-C. Tseng, and J.-L. Sheu, "A new multi-channel MAC protocol with on-demand channel assignment for multi-hop mobile ad hoc networks," in *Parallel Architectures, Algorithms and Networks, 2000. I-SPAN 2000. Proceedings. International Symposium on*, IEEE, 2000, pp. 232–237.

[10] J. Camp, J. Robinson, C. Steger, and E. Knightly, "Measurement driven deployment of a two-tier urban mesh access network," in *Proceedings of the 4th international conference on Mobile Systems, applications and services*, ACM, 2006, pp. 96–109.

[11] 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 (ToN)*, vol. 8, no. 6, pp. 785–799, 2000.

[12] 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, 2014.

[13] 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*, IEEE, 2015, pp. 48–55.

[14] S. Bouckaert, W. Vandenberghe, B. Jooris, I. Moerman, and P. Demeester, "The w-iLab.t testbed," in *Testbeds and Research Infrastructures. Development of Networks and Communities*, T. Magedanz, A. Gavras, N. H. Thanh, and J. S. Chase, Eds., Springer Berlin Heidelberg, 2011, pp. 145–154.

[15] G. Bianchi and I. Tinnirello, "Remarks on IEEE 802.11 DCF performance analysis," *IEEE Communications Letters*, vol. 9, no. 8, pp. 765–767, 2005.

[16] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving TCP performance over wireless links," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 756–769, 1997.