A FRAMEWORK FOR DYNAMIC CONTENTION WINDOW ADAPTATION FOR THE NET-X SYSTEM

BY

SHEHLA SALEEM RANA

B.Eng, National University of Science and Technology, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Adviser:

Professor Nitin H. Vaidya

ABSTRACT

Network designs with multichannel and multi-interface capabilities are fast emerging as the new solution to the high interference and low throughput problems faced by wireless mesh networks today. Multichannel capability, ideally, isolates transmissions over different channels and hence interference is expected to be minimized. Multi-interface capabilities aim to solve network partitioning problems by making sure that the wireless network remains connected at all times during operation.

Practically, however, there are limitations to performance improvement because of channel contention and adjacent-channel interference effects. Moreover, an intelligent channel allocation algorithm has to make sure that it utilizes all the channels effectively and in such a way as to minimize interference to other wireless nodes and also to balance the amount of contention over different channels.

In this thesis, we take another approach to improving the performance of wireless networks from the perspective of providing better fairness. Specifically, we build driver-level support in Net-X, a multichannel, multi-interface system for adjusting the contention window adaptively through knowledge of network topology and the level of channel contention sensed by a node.

To my parents, my brother, and my sisters, for their love and support

ACKNOWLEDGMENTS

I would like to thank my adviser, Dr. Nitin H. Vaidya, for his guidance and encouragement throughout this work. I would especially like to thank Vijay Raman for always helping me with understanding the Net-X system and with all my experiments. I would also like to thank all members of my research group for their help and support through discussions and suggestions. Finally, I would like to thank my husband for his constant support and help throughout this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AC	Access Category
BEB	Binary Exponential Backoff
CAL	Channel Abstraction Layer
CAM	Channel Abstraction Module
HAL	Hardware Abstraction Layer
IOCTL	Input-Output Control
KMCR	Kernel Multichannel Routing
MAC	Medium Access Control
UDP	User Datagram Protocol
WME	Wireless Multimedia Extensions
API	Application Program Interface

CHAPTER 1

INTRODUCTION

This thesis presents a framework to provide better fairness in the operation of the Net-X project. Specifically, to achieve this objective, we modify the wireless driver used by Net-X to implement control over the contention window on demand while the driver is running. The Net-X project is a system that provides multichannel and multi-interface support and has been completely and cleanly integrated into the Linux network protocol stack. Originally, the project was aimed at providing and exploiting multichannel and multi-interface capabilities, but several extensions have been proposed to it since then. These include a framework for rate and power control [1] and also for intelligent channel assignments in order to minimize co-channel and adjacent channel interference effects [2]. We propose yet another enhancement, by providing control over the contention window such that the wireless nodes can adapt to the changing levels of contention in real time. We have implemented this through appropriate interaction between user layer protocols and the device driver.

The IEEE 802.11 wireless networking standard offers support for the use of multiple channels. This ability can be easily exploited by infrastructure-based networks such that they assign different, noninterfering channels to access points that are close by. But in the case of infrastructure-less networks, there is no such coordination. Here the channels are to be assigned either totally randomly and independently or through some intelligent but distributed algorithm. Whichever approach is taken, a common objective is to maintain network connectivity. The

interface capabilities available in wireless systems today allow them to be switched to different channels over time, but at any one time, a single interface can only be tuned to a single channel. This means that for any two wireless nodes to be able to communicate with each other in an adhoc network, they must be tuned to the same channel at the time of communication. This capability of making sure that different wireless nodes can communicate with each other whenever they want is provided by Net-X. It is achieved by exploiting the fact that many wireless devices available and in use today are equipped with one or more interfaces. Therefore, the Net-X solution is to tune one of the available interfaces to a fixed channel and dynamically switch the other interface appropriately, depending on the fixed channel being used by the intended receiver for the communication. Transmission to a neighbor is always done by tuning the switchable interface to the fixed interface of the neighbor. All nodes receive packets on their fixed channel. This design allows for any node with at least two interfaces to be able to communicate with all others. IEEE 802.11 provides for the use of 12 non-overlapping channels and it would appear that if these channels are assigned to different nodes, then interference between transmissions over different channels should be minimal. However, in practice, signal leakage between adjacent frequency bands and imperfect cutoff of filters at the hardware level are reasons why non-negligible interference is still experienced during multichannel operation utilizing all the channels. Several solutions have been proposed for this. Chereddi and Vaidya [3] propose to only use a subset of the available channels such that the interference between them is minimized. Raman and Vaidya [2] propose an intelligent channel assignment strategy such that adjacent channels are not assigned to nodes in the vicinity of each other. Furthermore, the intelligent channel assignment algorithm would also ensure a balanced use of the channels so that the whole spectrum is utilized efficiently.

We build our extensions on top of the current Net-X implementation that already includes the intelligent channel assignment. Traditionally, the IEEE 802.11 MAC protocol specification dictates the use of a binary exponential backoff (BEB) mechanism in case of a collision. This would mean that if a node fails to receive an acknowledgement for the packet it just sent out, it assumes that a collision has happened. In response to this, the node doubles its contention window and chooses a new backoff counter randomly from this larger window. This action reduces the chance of a future collision, but this behavior essentially also reduces the channel access probability of the node that experienced the collision. However, this effect can be improved upon in such wireless networks where the nodes have more information about, for example, their neighborhood or the load on the network. The approach to minimizing collision probability is termed *contention resolution*. Contention resolution based on the knowledge of network topology has been shown to work better than a totally random approach [4]. For the particular case of Net-X, we observe that since the wireless nodes in the Net-X system are not very mobile, a knowledge base of the network topology can be built over time. However, notice that this information is not static, because the neighbors of a node with whom it contends for the channel are not necessarily those that are geographically close to it, but rather those that use the same channel for reception as the node under question. Therefore, if the channel assignment is dynamic, the topology information, consisting of the contenders for transmission and reception, would also be dynamic. We base our contention window adaptation on this knowledge of the number of neighbors on the same channel as the node under consideration. This gives us an idea of how heavily or lightly loaded the channel is; therefore, appropriate backoff windows are chosen by our design and are enforced by the device driver dynamically. To implement this, we needed that the driver receive

more information. We provided support such that the user space can provide the driver with information about the channel usage in the neighborhood of a node so that the driver can choose a contention window accordingly.

The Net-X project uses systems where the wireless interfaces are designed according to a thin MAC approach. In this approach, all the time-insensitive parts of the MAC protocol are implemented in software and time critical functionality is implemented in hardware. A hardware abstraction layer (HAL) is provided as a means of communication between hardware and software components of the interfaces. We made modifications to the Mad-Wifi driver used by Net-X so that it can make run-time changes in the contention window depending upon its knowledge of the network as received from the user space.

CHAPTER 2

RELATED WORK

There has been a lot of work on modifying the backoff behavior of IEEE 802.11 wireless systems. This is because more than one characteristic of the network is affected by this behavior. Specifically, changing this behavior boils down to changing the contention window size, i.e., values of CW_{min} and CW_{max} used by the wireless nodes. Cali et al. [5] did a detailed analysis for the IEEE 802.11 wireless LANs and derived average contention window sizes to maximize the throughput. Their conclusion, that proper tuning of the IEEE 802.11 backoff algorithm can be used to drive the protocol close to the theoretical throughput limit, led many researchers to focus on this problem. So for example, Mishra and Sahoo [6] propose to change the contention window sizes based on traffic type in order to meet some quality of service (QoS) requirements. They also propose to use non-contiguous contentions windows in order to provide service differentiation. Moreover, they also advocate a linear increase in the contention window as a consequence of collision, rather than an exponential increase.

Pang et al. [7] propose yet another variation of a linear change in the contention window parameters. They call their algorithm Multiplicative Increase, Multiplicative/Linear Decrease (MIMLD), where again they make decisions about whether to do a multiplicative or linear increase in the contention window depending upon certain network parameters like the current contention window and recent results of transmission attempts, i.e., success or collision etc. Hussain et al. [8] also propose a similar mechanism for dynamically

changing the contention window in order to meet some priority constraints.

Byungjoo and Haniph [9] propose an Efficient Contention Window Control (ECWC), which again modifies the contention window in order to solve the burstiness that happens as a result of a node undergoing consecutive successes. They also claim that the probability of packet failures due to hidden terminals is higher than the probability of collisions. They therefore choose to set the CW size to a fixed value by setting both CW_{min} and CW_{max} to be the same value. They claim that this would eliminate the burstiness caused by too many successes for one flow and also remove the unfairness in the throughputs achieved by different flows.

Yet another work is by Nafaa et al. [10], where the authors propose the concept of a Sliding Contention Window in which the contention window parameters are adjusted dynamically but they must remain within some specified boundaries. They define these boundaries separately for different service classes and claim that their algorithm can provide fairness while not compromising on throughput under conditions of high congestion.

Another work for the case of sensor networks is by Kim and Kang [11]. Here the authors propose a Distance Adaptive Contention Window (DACW) for the IEEE 802.15.4 standard. Their idea is to dynamically adjust the contention window according to the hop count distance to the sink and some traffic conditions. They claim that with DACW, each sensor node can achieve self-routing capability with reduced overhead and improved performance.

A similar work is by Ksentini et al. [12], in which the authors propose to use an adaptive range of contention window parameters and change not only CW_{min} but also CW_{max}. They call this scheme the Determinist Contention Window Algorithm (DCWA) and their algorithm also considers the current network load and the past history before choosing a new range and size of the contention

window.

In [13], the authors propose an algorithm that considers provision of QoS guarantees and also provides energy conservation. They introduce a new scheme for backoff where the backoff counter is decreased after sensing the channel as idle for one time slot. Therefore, the time between consecutive backoff decrements becomes a stochastic random variable depending on previous channel probabilities. They then set contention windows in separate ranges for different traffic classes. They claim that their work provides better QoS performance because it takes the stochastic nature of the channel into account. One effort to specifically address the problem of unfairness in IEEE 802.11 wireless networks was by Ozugar et al. [4]. They use another variant of contention resolution mechanisms: the access probabilities. These access probabilities are pre-calculated based on some information that every node broadcasts. These link access probabilities are modified every time new information is exchanged. They propose two ways to do this information exchange: connection-based and time-based. In the connection-based approach, each node broadcasts the number of logical connections or average contention time to stations within communication range. The nodes also send out information about the last backoff window size they used in the RTS packet. They claim that this increased information exchange can provide about an order of magnitude improvement in terms of throughput.

Ganu et al. [14] is probably the only work where a modification of CW_{min} and CW_{max} has been done on a real test bed. But they have also done this off-line; i.e., they choose different values of these parameters for different experiments and then reset and reload the driver again with the new values. They do not do this while the driver is working.

We have mentioned only some of the many approaches taken to modify the

contention resolution behavior of the IEEE 802.11 protocols. While all these schemes have their strengths and weaknesses, we would like to mention here that *none* of these schemes offers a dynamic control over the contention window in an experimental implementation. All of them are based on simulations, and so we chose to develop a framework where the contention resolution behavior of actual physical devices may be modified dynamically and during regular network operation.

CHAPTER 3

BACKGROUND

Since we have implemented contention management for the Net-X system in particular, we give an overview of the architectural and design details of the Net-X system. Although we talk about the fundamentals of the design, a more detailed and in-depth explanation can be found in [15].

3.1 IEEE 802.11 Collision Avoidance

The IEEE 802.11 MAC protocol specification includes a binary exponential backoff mechanism for collision avoidance. Since the wireless nodes share the space over which they transmit and receive, it is imperative that we minimize the chance that two or more interfering nodes end up transmitting concurrently. The 802.11 MAC is a random access protocol where nodes use a carrier sensing mechanism to sense if the channel is busy or idle. If they can sense a carrier signal of strength greater than a carrier sense threshold CS_{thresh}, they conclude that the channel is busy and defer their own transmissions. The deferring is such that each node maintains a contention window and then chooses a random backoff interval from that window. It then keeps on sensing the channel in every slot and if the channel is found to be idle, the node decrements its backoff counter by 1. When the counter reaches zero and the node senses the channel as idle, it transmits its packet. If, however, an acknowledgement for the packet is not received within a certain amount of time, the node assumes that the packet

has collided with another transmission and has been lost. All nodes in this situation are required by the protocol to double their contention window and choose another backoff interval randomly and start the process of counting down again. The motivation behind this is that, since the nodes would be picking a number uniformly randomly from a larger window, the probability that any two or more nodes choose the same number will decrease. Many improvements have been proposed to this naive implementation, some of which are mentioned in Chapter 2.

3.2 The Net-X Testbed

Net-X architecture is a system that implements multichannel protocols on wireless nodes with multiple interfaces. The system was developed by the Wireless Networking Group at the University of Illinois at Urbana-Champaign (UIUC). It consists of more than 20 Soekris net4521 boxes spread over various rooms on the fourth floor of the Coordinated Science Lab at UIUC. Each node has a 133 MHz miniprocessor, a compact flash (CF) card slot, two PCMCIA slots and one mini-PCI slot. Linux kernel 2.4.26 is run as the operating system on these Soekris boxes [2]. For all Net-X experiments, the nodes are equipped with one mini-PCI and one PCMCIA wireless card. These wireless cards are Atheros chipsets running the Mad-Wifi driver. The cards can operate on all the 12 channels provided by 802.11a. The mini-PCI cards have a pair of external antennas and the PCMCIA card has its own internal antenna. All these details can be found in [2].

3.2.1 Net-X multichannel operation

A new routing protocol was also designed and implemented for Net-X which takes into account the fact that it is to work for a multichannel and multi-interface system. The two wireless interfaces present in each node are used such that one of them keeps switching over different channels as needed and the other one remains fixed on a channel for longer intervals of time. The former is termed a *switchable* interface and the latter is termed a *fixed* interface. All data is received over the fixed interface and transmissions are done by the fixed interface if it is tuned to the channel of transmission. If however, a transmission is to be done on a channel different from the fixed one, the switchable interface is tuned to the channel of transmission and then data transmission is started. The motivation behind this design is that with such a system in place, two nodes do not necessarily have to be on the same channel at all times in order to communicate with each other. Instead, whenever a node A wants to talk to node B, node A first tunes its switchable interface to the channel used by the fixed interface of B. This way A can communicate with B whenever A wishes, and once communication is complete, A may tune its switchable interface to some other channel if it wants to transmit to some other node in the neighborhood that is using a different channel. This operation is depicted pictorially in Fig. 3.1.

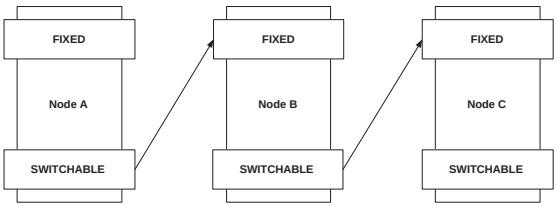

Figure 3.1: Multichannel Operation of Net-X

The interface switching approach allows for dynamic channel assignment, such

that nodes can tune to different channels depending on the experienced interference, fading or traffic load conditions. However, the currently available hardware was not designed with frequent switching in mind and therefore switching times for most hardware are such that very frequent switching may not be feasible from the perspective of delays incurred. If, however, sufficient hardware support becomes available, such that switching times are minimized in comparison with actual transmission times, then better protocols and systems can be built based on this interface switching capability.

Net-X, however, makes sure that channel switching decisions are made intelligently such that the resultant packet delays may be minimized and any packet losses during the transition time are eliminated completely. Several modifications have been made to the device driver operation in order to minimize switching delays. These are explained in detail in [15].

Referring back to Fig. 3.1, the fact that a node tunes its switchable interface to the fixed channel interface of the neighboring node implicitly assumes knowledge of the fixed channel of the neighbor. Of course, some mechanism has to be put in place in order to make sure that this assumption is true. For this purpose, hello messages are exchanged between nodes by using a multi-channel broadcast. Support for multichannel broadcast was integrated into the Linux kernel stack when the system was originally designed in [15]. These hello messages contain information about the fixed channel of a node and that of all its neighbors. So for example, in Fig. 3.1 when node A sends out a hello message, it includes information about its own fixed channel. Then, when node B sends out a hello message, it includes the fixed channel information not only about itself but also about all its one-hop neighbors from whom it received a hello earlier: in this case from node A. So now C receives a hello message from B which provides C with information about both B and A; i.e., C is now able to

build a database of the channel usage of both its one-hop and two-hop neighbors.

A modified version of AODV is used as the routing protocol as described in [16] and [17]. The modified routing protocol tries to find a route that minimizes the expected transmission time by incorporating this criterion into a new routing metric called the Multichannel Expected Transmission Time (MCETT).

3.2.2 Net-X system design

Here we briefly describe the main components of the Net-X system architecture. The components and functionalities relevant to this thesis will then be described in more detail later. A detailed diagram of the architecture is given in Fig. 3.2.

3.2.3 Channel abstraction layer

The Linux protocol stack implementation does not directly allow multichannel and multi-interface operation. To integrate this capability with the stack, a channel abstraction module was defined as in [15]. The channel abstraction layer (CAL) operates between the network layer and the device driver. Its main function is to hide the underlying mechanics of multiple interfaces from the higher layers. It provides support for multichannel operation and abstracts the details of multiple interfaces so that a single virtual interface is exposed to the higher layers. CAL was implemented as part of the *bonding driver* available in the Linux kernel. The user space protocol can communicate with the CAL via input-output control (IOCTL) calls. Up to 16 private IOCTL calls may be defined in the Linux kernel. The main components of the CAL include the following:

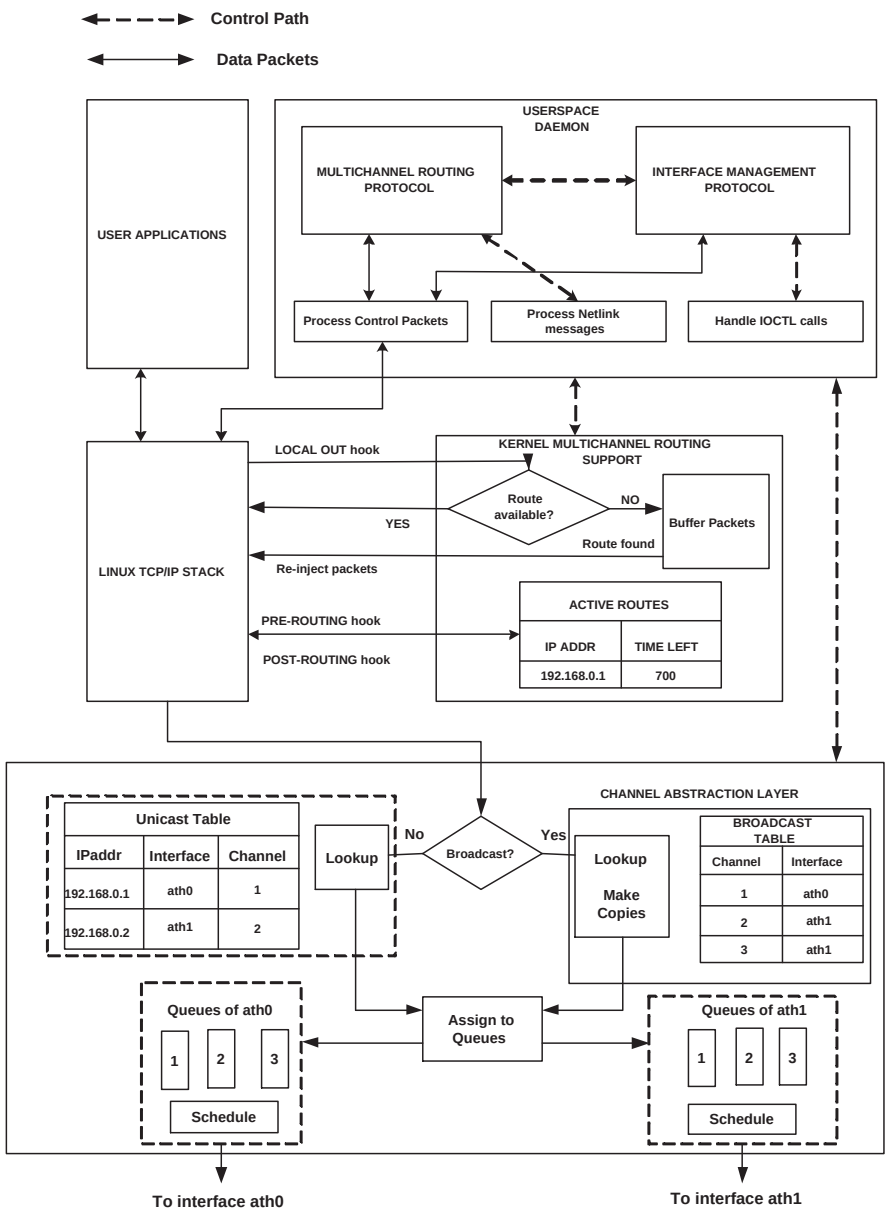

Figure 3.2: Net-X Architecture with Multichannel and Multi-Interface (ath0 and ath1) Operation

1. **Unicast Component:** The unicast component maintains a table structure with information about the channel and interface to be used in order to reach a neighbor. The table is of the form shown in Table.3.1. This table is important from the perspective of this work because we make modifications to this table to provide channel usage information from the

Table 3.1: Unicast Table Structure

Destination IP	Interface	Channel
192.168.30.6	ath0	2
192.168.30.8	ath1	3

user space protocol to the CAL, which can then hand it over to the device driver for making decisions about the appropriate contention window to be used. The way it works is that whenever there is a unicast packet to be transmitted, it is handed over to this component. The table is used to look up the channel to be used for the desired destination IP address and the packet is sent out to the appropriate queue.

2. **Broadcast Component:** The broadcast component makes sure that broadcast packets are sent out on all the channels so that all the neighboring nodes receive them no matter what channel they are currently listening to.

3. **Scheduling Component:** The scheduling component implements policies about queuing the packets unless they are ready to be transmitted and also makes decisions about interface switching. Obviously, this had to be more intelligent than a naive implementation where an interface channel was switched for every packet. Instead, this component efficiently enqueues packets on a per-channel basis for subsequent transmission.

3.2.4 Kernel multichannel routing support

This module, abbreviated as the KMCR module, provides on-demand routing support for multichannel routing. More details on this can be found in [17].

3.2.5 Device driver

Mad-Wifi is used as the driver for the Atheros-based devices used for Net-X. It is a partially open-source driver and provides functionality to cleanly run Atheros-based devices in the Linux kernel. More details of the driver will be discussed when we talk about the modifications made to the driver in Chapter 4.

3.2.6 User space protocol

As mentioned in Section 3.2.1, periodic *hello messages* are exchanged between the wireless nodes for learning the network topology and reachability information. The user space protocol deals with the communication of these hello packets. It also processes the hello packets to extract neighborhood information and communicates with the CAL through IOCTL calls to update CAL tables. The user space protocol also interacts with KMCR for multichannel routing support. For the purpose of this work, we are only interested in the hello message processing done by this protocol and also the information it exchanges with CAL. We will make changes to these components of the user space protocol and will explain them in Chapter 4.

Now that we have addressed the main architectural components of the system, we will take a look at the design and implementation of support for dynamic adaptation of contention window for the Net-X system.

CHAPTER 4

SYSTEM DESIGN AND IMPLEMENTATION

In this chapter, we describe the various design and implementation aspects that have been incorporated in Net-X in order to deal with changes in the level of contention within the changing wireless environment. We start by identifying the goals of the system, then we discuss the approaches to achieve these goals, and finally we discuss how they have been implemented in the system.

4.1 Design Highlights

We wanted our system to be able to make better use of the knowledge of its neighborhood when making transmission decisions. To make this work, we implemented a new design whose main functionality can be described in a step-by-step manner as follows:

1. Gather topology and channel usage information for the two-hop neighborhood.

2. Pass this information to the driver efficiently.

3. Modify the driver so that it dynamically adapts its contention window based on the received channel occupancy information.

We now describe how these steps have been implemented in the actual system. As we know, the nodes in the Net-X system are not very mobile;

therefore, a high-level knowledge base of network topology can be built at each node. This topology, however, keeps changing as the nodes switch to different available channels in order to fully utilize the availability of multiple channels. As discussed earlier, *hello messages* are exchanged periodically between nodes to keep track of this dynamically changing topology. We exploit the information obtained through these messages for our dynamic control of the contention window through the wireless device driver. For this purpose, several changes were made to how the Net-X system works. We briefly identify the main changes below:

1. **User-Space Hello Message Protocol** The hello message exchange allows nodes to discover the identities of their two-hop neighbors and also to receive information about the channels used by their two hop neighbors. This information is then used to populate routing tables used by the Linux kernel for communicating with these neighbors on the appropriate channels. In the current implementation, the hello message protocol, running at a node, passes information to the kernel, about the channel and interface to use in order to reach each of its neighbors. We modified it so that now it also gathers and provides information about the number of neighbors the node has on each channel.

2. **Channel Abstraction Layer (CAL)** The channel abstraction layer consists of the bonding module used to hide the existence of multiple interfaces from the higher layers. We modified the unicast component maintained by this module such that it now contains an entry for the number of neighbors on each channel and passes that to the wireless device driver along with other reachability information, i.e., the channel and interface to be used for transmitting the packet.

3. **Wireless Device Driver** The driver is responsible for actual transmission and reception of packets through the physical interface. It is called by CAL whenever a packet is received by CAL. With our changes, CAL hands the packet down to the driver along with information about population on the channel of use. We modified the driver so that it dynamically chooses an appropriate contention window for each packet depending upon the perceived channel contention.

4.2 Implementation

We now discuss the implementation of each of the above mentioned changes to the system.

4.2.1 Hello message protocol

This is the user-space protocol for exchanging periodic *hello messages* containing neighborhood and channel information. This protocol deals with gathering information for broadcast and unicast messages. It is also responsible for maintaining information about channel statistics and load balancing. An intelligent channel assignment policy has also been implemented which dynamically makes channel switching decisions for the wireless nodes in order to minimize adjacent channel interference and make sure that the number of nodes on each channel remains balanced at any point in time. This protocol helps build up neighbor tables by gathering information from the hello packets and passing them on to the channel abstraction module where routing entries are made corresponding to the calls made by the user space protocol. It uses IOCTL calls to communicate with the channel abstraction module. A total of five new IOCTL calls have been implemented for this user space multichannel protocol to

interact with the CAM. These are tabulated in Table 4.1.

Table 4.1: IOCTL Calls for Interaction Between User-Space and Channel Abstraction Module

IOCTL Call	**Purpose**
AddValidChannel	Declare a channel as valid for use
AddUnicastEntry	Add, Update or Modify Unicast Entry in Unicast Table
AddBroadcastEntry	Add, Remove Broadcast Entry in Broadcast Table
SwitchChannel	Command for a channel switch
GetStatistics	Obtain Channel statistics

We are specifically interested in the *AddUnicastEntry* IOCTL call since it adds a new unicast entry into the unicast table maintained by CAM. It is called by the hello message protocol in the user space every time the protocol discovers a new neighbor and needs to enter it into the kernel routing table. Currently, this IOCTL takes four arguments including the IP address of the new destination, the interface and channel to be used for reaching this destination. We augmented the protocol such that now, it also maintains the number of two-hop neighbors on each channel. We then modified the *AddUnicastEntry* IOCTL call to take this measure of population on each channel as a new argument. For each of the IOCTL calls mentioned in Table 4.1, appropriate data structures and methods are defined inside the bonding module of the CAM. Each IOCTL call has a unique code that is used to identify the call and the set of actions to be taken in order to implement the particular call. We modified the data structures to contain the additional information and reserved memory for it along with the rest of the unicast routing table. This way, channel usage information now becomes available to CAM for further use as explained in the following sections.

4.2.2 Channel abstraction module

The channel abstraction module sits between the network layer and the interface device drivers. Since our main aim is to use the number of nodes on a particular channel for choosing an appropriate contention window, we wanted this information to somehow be made available to the driver. There were several options for achieving this. We analyze them below in terms of their pros and cons.

1. We could have programmed a new IOCTL call to work directly between the user space and the device driver. This would have appeared to be an easy option, but it has several shortcomings. First, the driver operates at the link layer and uses information understandable to the link layer, e.g., the MAC addresses of the nodes. This information is not available at the user space and hence it is hard for the two to communicate information that depends on different identifiers for the nodes, i.e., IP addresses and MAC addresses.

2. Another option would have been to include this information as part of each packet header so the driver can read it directly. But this approach requires modification to the packet headers in the Linux kernel, and we want our implementation to be general so that it will not become specific to a modified kernel. Therefore we chose not to do this, and to keep the kernel as is.

3. Our final option, and one that we chose in this work, was to use the channel abstraction layer as an intermediary between the user space and the device driver. This way, we can cleanly integrate this new support because we already have a way to communicate between the user space

and the CAL through the IOCTL calls. For the next step in the communication, i.e., between CAL and the device driver, we use the functionality of *export_symbols* which will be explained along with the driver modifications in Section 4.2.3.

The bonding driver mentioned in Section 3.2.3 is part of the channel abstraction module and functions to bond various physical interfaces into one exposed virtual interface. We now look at it in more detail to understand the changes made to this module.

The channel abstraction layer has three main components as mentioned in Section 3.2.3. Our goal is to modify the behavior of the system whenever it sends out a unicast packet in such a way that it chooses an appropriate contention window depending upon the knowledge of the neighborhood topology. Whenever a unicast packet is to be sent out, it is passed to the bonding module in the CAL which maintains the relevant unicast routing table. This table originally looked like Table 3.1. We wanted to add more information in this table about the channel population for each channel corresponding to the next hop towards a destination in the routing table. This information is now passed from the user-space multichannel protocol through the modified *AddUnicastEntry* IOCTL call as explained in Section 4.2.1. The modified IOCTL call now enters more information such that a modified unicast table looks like that shown in Table 4.2.

Table 4.2: Modified Unicast Table Structure

Destination IP	Interface	Channel	No of 2-hop Neighbors on this Channel
192.168.30.6	ath0	2	3
192.168.30.8	ath1	3	4

With this modification to the table, we now have information about channel population available to the CAM. Whenever a unicast packet is received, it is

passed to the unicast component in the CAL to look up the appropriate channel and interface to send the packet. After our modification, the channel population information is also selected. The packet is then passed to the scheduling and queuing component to wait for subsequent transmission. Moreover, the driver needs to be updated with the information about the population on the channel so that it can choose a suitable contention window.

We had multiple options to consider about when to update the driver with this information. These are as follows:

- Update the driver with the channel population information each time a new routing entry is made in the unicast routing table. This would mean the driver would know about the channel population as soon as it becomes available or is updated. But this would not be an intelligent approach and might even be redundant in many cases. To understand this, recall that hello messages are periodic and hence the population of nodes on a channel will vary with time. We explain this by an example. Consider the simple scenario in Fig. 4.1.

  Consider a network initialization phase where the nodes are first building a map of the network. In this scenario, consider how Node 1 will populate its routing table based on the hello messages. Suppose the sequence of events is such that Node 1 first receives a hello message from Node 2, informing it that Node 2 uses channel 40 as its fixed channel. At this time, Node 1 would make a routing entry for destination Node 2 with fixed-channel 40. Node 1 would also update the neighbor population on this channel as 1. Its unicast table at this stage would look like that shown in Table 4.3.

  Next, Node 1 may receive a hello message from Node 3 informing it that Node 3 uses fixed channel 60 and that Node 5 uses fixed channel 40. After

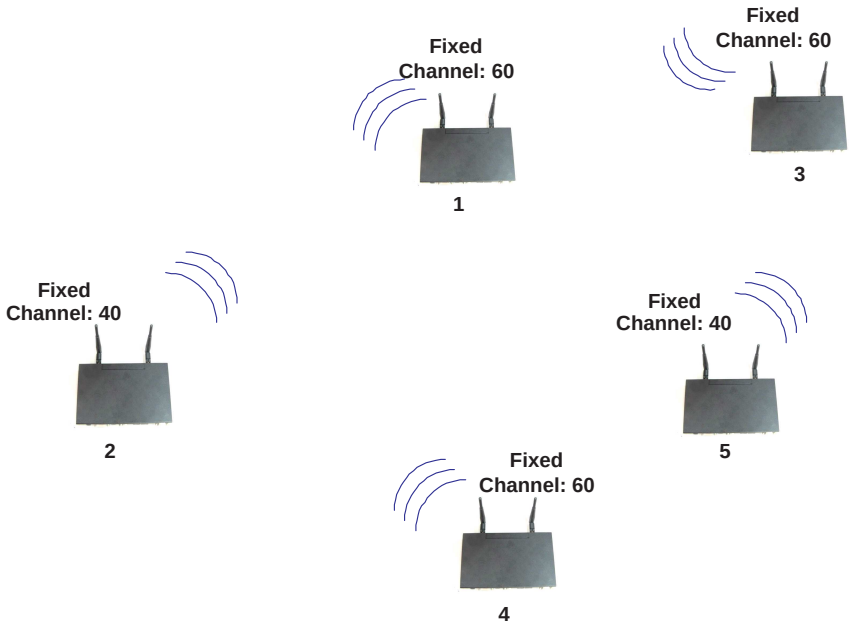

Figure 4.1: Building a Neighborhood Map Based on Hello Messages

Table 4.3: Unicast Table at Node 1

Destination IP	Interface	Channel	No of 2-hop Neighbors on this channel
192.168.30.2	ath0	40	1

this second hello message, Node 1 now makes two new entries in its unicast routing table: one for Node 3 and one for Node 5. Also, since Node 1 now knows of *two* neighbors on channel 40, it should update its routing table entry for Node 2 to reflect the most recent number of neighbors on channel 40. After this step, the unicast table at Node 1 should look like that in Table 4.4.

Table 4.4: Updated Unicast Table at Node 1

Destination IP	Interface	Channel	No of 2-hop Neighbors on this Channel
192.168.30.2	ath0	40	2
192.168.30.3	ath1	60	1
192.168.30.5	ath0	40	2

Therefore, unless Node 1 had a packet to send on channel 40 to Node 2 before the reception of the second hello message, there is no need to

update the driver right after receiving the hello message from Node 2. This is because a second update would have to have been done after receiving the hello from Node 3. So as this example shows, we do not need to update the driver as soon as a new routing entry is created or updated. Also, notice that if a node does not receive a hello message from any of its neighbors for a specific interval of time, the link to that neighbor is considered as broken and the corresponding entry is deleted from the routing table. The node population on that channel is updated as well.

- The second option is to update the driver only when we have a unicast packet to send out. As also motivated by Figure 4.1, we chose this option. With this choice, when a unicast packet is available at CAL, it picks up the interface and channel information required to reach the next hop and updates the driver by passing to it the most recent channel population value. This way, we avoid any unnecessary updates to the driver. Another optimization is that when the node population on a channel is received, all the entries of the unicast table are sifted through to see if there are other destinations which would also be using the same channel. For all such nodes found in the unicast table, the entry for node population on that channel is updated to reflect the most recently received number. This can also be seen in how the neighbor entry for Node 2 is updated when making a new entry for Node 5. This is because Node 5 and Node 2 both require packet transmission on the same channel.

Since both the bonding module and the device driver operate as loadable modules in the Linux kernel, we needed some form of intermodule communication for passing information between these two modules. We chose to use the functionality of *export_symbols* provided by the Linux kernel. The

bonding module uses a function *bond_xmit_muic* for transmitting packets. We made changes to the component of this function dealing with unicast packets. Whenever a packet is recognized as a unicast packet, the bonding module makes a call to the Mad-Wifi driver through a new function *adapt_cw_in_driver* and passes on the information about the number of nodes on the channel that the current unicast packet will be sent on.

4.2.3 Mad-Wifi driver modifications

In this section we explain the modifications we have made to the Mad-Wifi driver. The Mad-Wifi driver is a partially open source system. It consists of the following main components:

- **Hardware Abstraction Layer (HAL)** This is the closed source module of Mad-Wifi. It comes as a compiled binary and although some understanding can be derived about its functions, no modifications to this module were possible in the original Mad-Wifi. However, newer drivers like the ath-5k [18] and ath-9k [19], provide HAL as an open source component and allow far more research flexibility. All access to the hardware must go through HAL. The HAL performs some transmit and receive functions, e.g., setting a maximum limit on the allowable transmit power, etc. It also handles some beacon management functions as well as some interrupt and cache handling functions. Since Mad-Wifi provides several modes of operation— e.g., monitor mode, BSS mode, etc.— HAL provides control over selection of these modes and also some chip reset functions. More details about the list of main functions and methods used by HAL can be found in [20].

- **ieee80211 Protocol Stack** This is the net80211 stack from FreeBSD. It

contains simple generic 802.11 functions and some callback functions. It is used to support the Atheros device and provide WLAN functionality and some cryptography functionalities.

- **ath Module** This module defines Atheros specific functions and callbacks to interact with the IEEE 802.11 layer and to interact with the actual device hardware through HAL.

- **ath_rate module** This module provides some rate control algorithms. Currently Mad-Wifi supports three different rate control algorithms and the user can choose between them.

The fact that some part of the driver was closed source made it impossible to know the feasibility of this project at the beginning. We knew that it was possible to make changes to the contention window parameters offline, i.e., by resetting the driver and then reloading it again with new values for CW_{min} and CW_{max}. But such a mechanism does not offer much potential for use in any algorithms that require a dynamically adaptable contention window. Our main goal was to be able to make what are termed "hot" changes to these values, i.e., to change them without disrupting the operation of the driver. With this in mind and with the design as explained above, we established that the main objective, was to enhance the driver so that it has the following capabilities:

- Interact with the channel abstraction module to receive information about the number of perceived contenders for a channel.

- Depending on this received number, dynamically adapt its chosen CW_{min} for the current packet transmission at run time.

We achieve the first goal here by making the driver expose a new API function, namely *adapt_cw_in_driver*, to the CAL. It is made available to the

CAL by using *export_symbols*. The function, *adapt_cw_in_driver* takes as its arguments, the appropriate packet information, from which we extract a parameter we call *cCount*. This quantity represents the number of neighbors using this specific channel for their reception in a two-hop neighborhood of the node in question and is sent to the driver by the same CAL that received it from the user space. For the Net-X system, it would signify the number of two-hop neighbors of the node, whose fixed interfaces are tuned to this channel at the time of packet transmission. Since the nodes keep on changing their fixed channels depending on the channel quality and the throughput they achieve on that channel, the number *cCount* also changes dynamically.

Once the channel population becomes available to the driver, the driver has to perform two tasks:

- Choose an appropriate CW_{min} for this packet transmission.
- Use this chosen value when picking its backoff counter at the hardware level.

We achieve the first objective here, by doing extensive simulations in NS-2 for networks of different sizes. We run the simulations for different values of CW_{min} and find out which particular value of CW_{min} gives the best results in terms of a combination of overall throughput, individual flow throughputs, and the number of collisions encountered by the individual flows. The details of these experiments and some results are presented in Chapter 5.

The second objective is more device implementation dependent. Before going into the details of how this is done, we first look at a capability of Mad-Wifi which enables this change.

4.2.3.1 Wireless multimedia extensions

Mad-Wifi has support for Wireless Multimedia Extensions (WME), more recently termed as Wifi Multimedia (WMM). This is a subset of the IEEE 802.11e wireless LAN specification, which is introduced to provide some quality of service (QoS) guarantees to the wireless traffic. This is done by assigning different priorities to packets of different classes. The classification of packets is done according to one of the following four categories, termed as *access categories (AC)*:

1. **Voice (VO):** This access category is given the highest priority. This is to ensure acceptable latencies to the increasingly popular Voice Over Internet Protocol (VoIP) traffic.

2. **Video (VI):** This is considered less stringent in terms of its latency requirements and hence comes second on the priority list. This can provide QoS to streaming video applications.

3. **Best Effort (BE):** As the name indicates, best effort traffic is not promised any QoS provisions. The regular IP data traffic we see on the Internet today falls into this access category and it comes third in the priority list.

4. **Background (BK):** This is the lowest priority category with no strict requirements on delay or throughput. File download, print jobs, etc., can be categorized in this class.

WME provides these different priorities to each access category by assigning them different contention window sizes. This is implemented in Mad-Wifi as well. Table 4.5 shows how these priorities are enforced in Mad-Wifi through selection of different ranges of CW_{min} and CW_{max} for different access categories.

Table 4.5: Mad-Wifi WME Parameters

Access Category No	AC Descriptor	CW_min	CW_max
0	BE-Best Effort	2^3	2^{10}
1	BK-Background	2^4	2^{10}
3	VI-Video	2^3	2^4
3	VO-Voice	2^2	2^3

Since Net-X uses the default system configuration, its traffic is classified as best effort and so now we get an idea of how we can add support for changing contention window in the driver. We need to be able to change the WME parameters for the queues containing best effort traffic inside the driver. There were several functions already existing in the *ath* module inside the Mad-Wifi driver that dealt with transmission queues. Let us look at them to see which one would be more feasible for our purpose.

The first is that of *ath_tx_setup*, which sets up a hardware data transmit queue for a specified access category. The function *ath_tx_setup* then calls *ath_txq_setup*. This function is used by the driver to set up a hardware transmit queue for the given traffic class. The *ath_txq_setup* function starts off by setting the values of CW_{min} and CW_{max} to the default values for each traffic class. It then calls the closed source HAL function *ath_hal_setuptxqueue* for actual queue initialization in the hardware. When these queues are properly set up with their appropriate contention window parameters, control returns to the calling function, i.e., *ath_tx_setup*.

At first, we thought of incorporating a dynamically changeable CW_{min} as part of this function. But this function is called once the device initializes, and calling this function again runs the risk of resetting the transmit queues altogether, which may result in some unpredictable losses for the packets already queued.

So we turned out focus on another function, which is the *ath_txq_update* function. This function allows us to make updates to WME parameters of a

specific access category. This is the perfect place for us to incorporate our changes since this function does not need to reset the device and it only changes the WME parameters for the queues while they are still operational. This function then calls a HAL function *ath_hal_settxqueueprops*. This HAL function takes as its argument, the new WME parameters and then updates the queue with the newly received parameters. Upon success, it returns control to the calling function.

We chose to use *ath_txq_update*, as the function which we use, when we want to make changes to the CW_{min}. In the following, we explain the step-by-step operation of the driver for updating its contention window.

- The driver exports a function *adapt_cw_in_driver* to the CAL.
- CAL calls this function whenever it has a unicast packet to send. It passes the channel population metric as an argument to this function.
- Based on this channel population metric, the driver picks an appropriate CW_{min}
- Upon choosing a new CW_{min}, the function *adapt_cw_in_driver* calls *ath_txq_update*, and passes this CW_{min} for updating the queue properties.

The overall flow of information from the userspace to the driver is shown in Fig. 4.2. This figure clearly shows the sequence of actions that need to be taken in order for a new value of CW_{min} to be chosen and applied to a transmission. Shown in blue arrows in the figure are the modes of communication between different Net-X system components. IOCTL calls are used for user space communication with other kernel modules. For intermodule communication, *export_symbols* functionality is used. This is part of the Linux kernel. The Linux

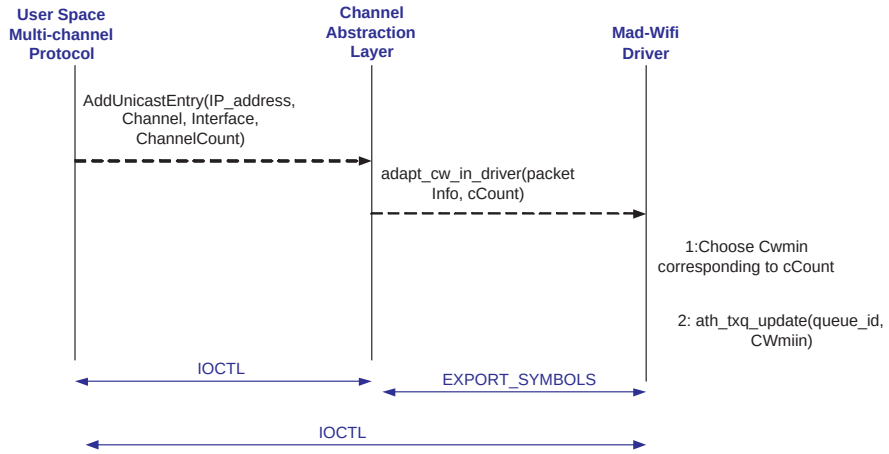

Figure 4.2: Communication between NetX Components for Dynamic Contention Window Adaptation

kernel maintains a table called the *ksym* table which contains different functions accessible to all kernel modules. The function *export_symbols* is from *kernel/ksyms.c*, and adds the exported function to the kernel symbols table so that it then becomes accessible to other kernel modules. All kernel modules can then access the exported function or variable directly from the kernel table and make calls to the exported functions. This architecture protects the different kernel modules from modifying each other's variables or causing conflicts to memory access reads or writes.

With this design and implementation put in place, we call the Net-X system the *modified* Net-X system. We present some background work for building up a database of appropriate contention windows for networks of different sizes in Chapter 5 and then show experiment results and performance comparisons of the original and modified Net-X systems in Chapter 6.

CHAPTER 5

NS-2 SIMULATION RESULTS

We start by simulating wireless networks of different sizes in NS-2 and study the effect of changing contention window, specifically CW_{min}, in them. First, we start by simulating only one flow between two nodes. We send UDP traffic at 6 Mbps, with packet size of 1470 bytes. We also fix the channel transmission rate to 6 Mbps and then vary the contention window in powers of 2 from 2^0 to 2^{10}. These experiment settings remain the same for simulations of all network sizes discussed in this chapter. The throughput of the single flow, as a function of its CW_{min}, is shown in Fig. 5.1.

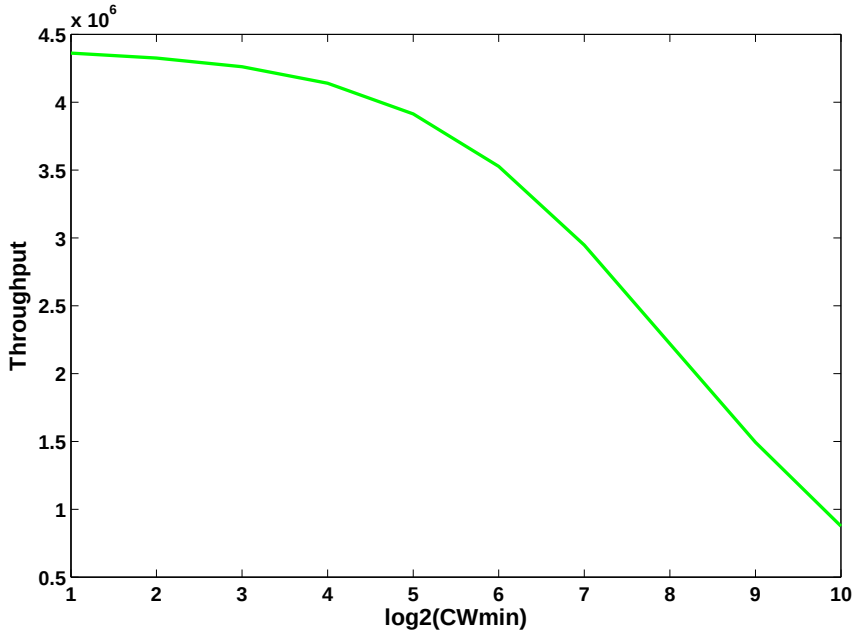

Figure 5.1: Throughput Variation as a Function of CW_{min} for 1 Flow

Whereas this experiment does not say much about an appropriate contention window, we performed it only to be able to visualize the trend in the changing throughput when CW_{min} changes. We have done a similar experiment for the Net-X testbed as well, and its results in Chapter 6 show a similar trend.

Next we did experiments for up to 20 flows and analyzed how the aggregate throughputs change when the contention window is varied from 2^0 to 2^{10}. This range is also selected because this is the allowable range for the device driver that we use. Also, there is not a lot of flexibility in choosing this number since it is restricted by the wireless device driver to be a power of 2. Therefore, in some cases, the difference between two successively available CW_{min} values can be large, especially for higher powers of 2. If we can achieve more fine-grained control over setting this value, we can probably do much better tuning of the CW_{min} and make it more appropriate to changing contention in the environment.

We also remark here that with multiple flows, the aggregate throughput alone should not be the only metric considered for evaluating network performance. There are other, equally important performance metrics, that need to be considered when characterizing network performance. These may include individual flow throughputs, i.e., a measure of fairness, packet delays, packet delivery ratios, etc. We specifically considered the case of fairness in addition to the aggregate throughput when doing suitable contention window calculations. Wireless networks suffer from severe unfairness problems especially because of random access. The way IEEE 802.11 Binary Exponential Backoff (BEB) works is that there is potential for one node to capture the channel while the others might suffer because of their backoff. This may happen, for example, if a node happens to randomly pick a small initial backoff; then while this node transmits, other nodes will wait because they found the channel to be busy. When the

node finishes transmission, it again chooses a backoff from a comparatively smaller window because it just experienced a transmission success and so it may get access to the channel yet again before another node has had a chance to count down to zero. Therefore, it can be concluded that under some conditions, BEB can further aggravate the achievable fairness in the network and give rise to short-term unfairness.

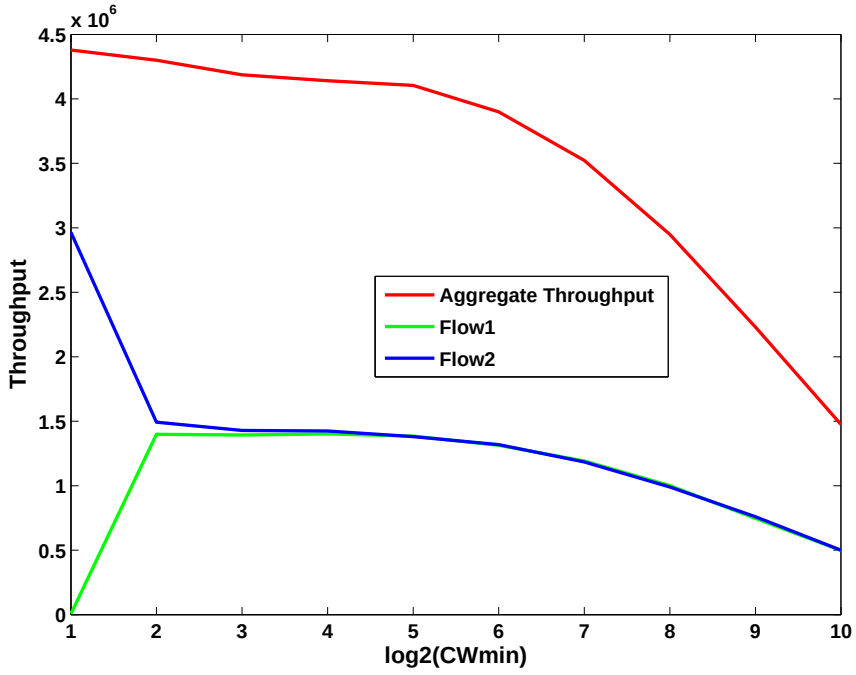

Figure 5.2: Throughput Variation as a Function of CW_{min} for 2 Flows

We plot the values of the aggregate throughputs and the per-flow throughput on the same graphs in Figs. 5.2 to 5.6. These figures are for 2, 3, 4, 5 and 15 flows, respectively. As we can see from the figures, there are some values of the contention window that result in higher aggregate throughput but poor fairness; i.e., one flow might capture the channel and starve others. This is clearly seen in the case when the value of CW_{min} is 1 in most cases. However, as the size of the contention window increases beyond 1, the individual throughputs achieved by the flows start to become closer to each other.

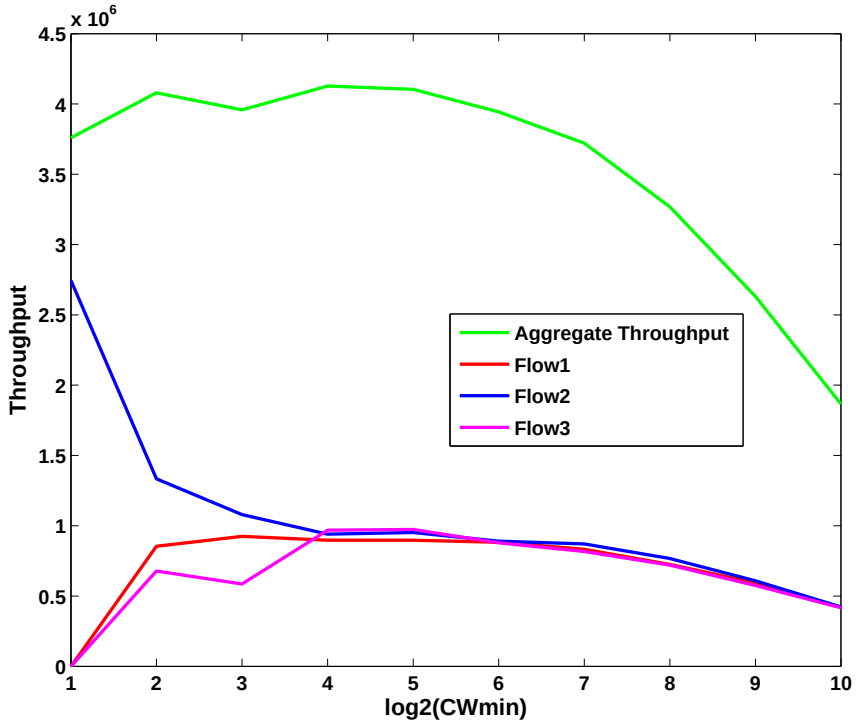

Figure 5.3: Throughput Variation as a Function of CW_{min} for 3 Flows

Our interest lies in the range of CW_{min} where we get better fairness between the flows while still not compromising too much on throughput. For this reason we chose different appropriate values of contention windows for different numbers of flows in the network under consideration. These values are ones for which all the flows achieve more or less the same individual throughput. These chosen CW_{min} values (shown in Table 5.1) are then incorporated into the Mad-Wifi wireless device driver running on the actual Net-X test bed, and the contention window is adapted dynamically based on the number of flows perceived to exist on a specific channel.

We also looked at another parameter as a function of the changing contention window. This was the number of collisions experienced by each flow. Intuitively, we can imagine that the number of collisions would decrease as the contention window grows in size. But of course there is a clear trade-off here: We cannot

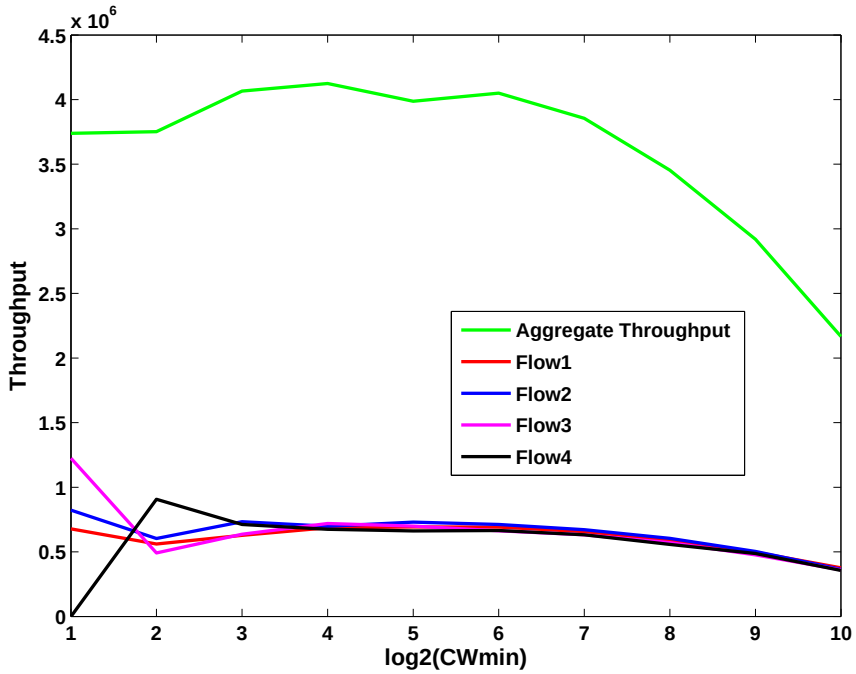

Figure 5.4: Throughput Variation as a Function of CW_{min} for 4 Flows

expand the contention window by a large amount because, although it would decrease the number and probability of collisions, it would also degrade throughput. And so we wanted to see if our chosen contention windows were the ones which also strike a balance between the number of collisions experienced by a flow and the overall throughput it achieves.

Figure 5.6 shows the aggregate throughput for a network of 30 nodes with 15 UDP flows at the rate 6 Mbps. Figure 5.7 shows the individual flow throughput obtained by each flow (left). The figure on the right plots the number of collisions experienced by each flow as CW_{min} is changed from 2^0 to 2^{10}. Figures 5.6 and 5.7 demonstrate the existence of a suitable contention window near 2^8. This is the point at which the aggregate throughput is near its peak, the individual throughputs are very close to being equal, the number of collisions is reasonably small for all flows, and all the flows experience about the same absolute number of collisions. Therefore, at this point, the network is fair

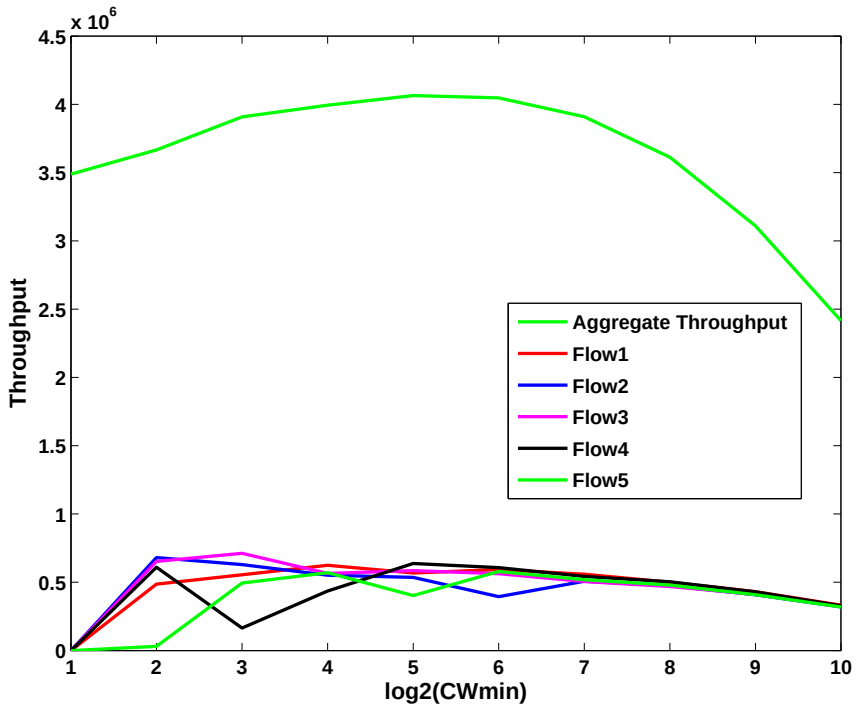

Figure 5.5: Throughput Variation as a Function of CW_{min} for 5 Flows

both in terms of useful throughput and in terms of collision losses, while aggregate throughput is not compromised.

Table 5.1: Appropriate CW_{min} Values for Different Numbers of Flows

No Of flows	CW_{min}
1	2^2
2	2^3
3	2^4
4	2^5
5	2^6
6-8	2^7
$\geqslant 9$	2^8

These NS-2 simulations served as the basis for our contention window adaptation algorithm used in the actual Net-X testbed. The appropriate contention windows for networks of different sizes were obtained from these simulations and are tabulated in Table 5.1. These values are fed into the driver

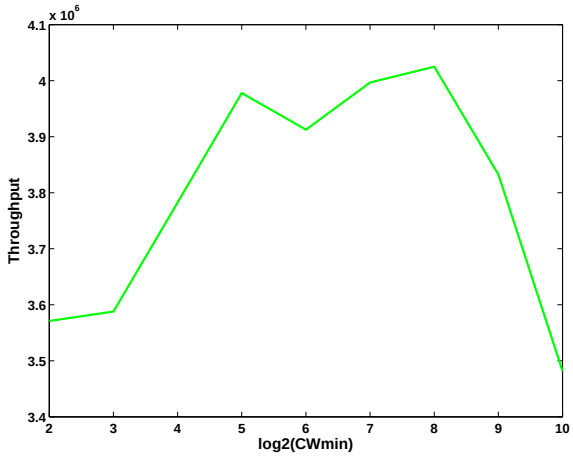

Figure 5.6: Aggregate Throughput Variation as a Function of CW_{min} for 15 Flows

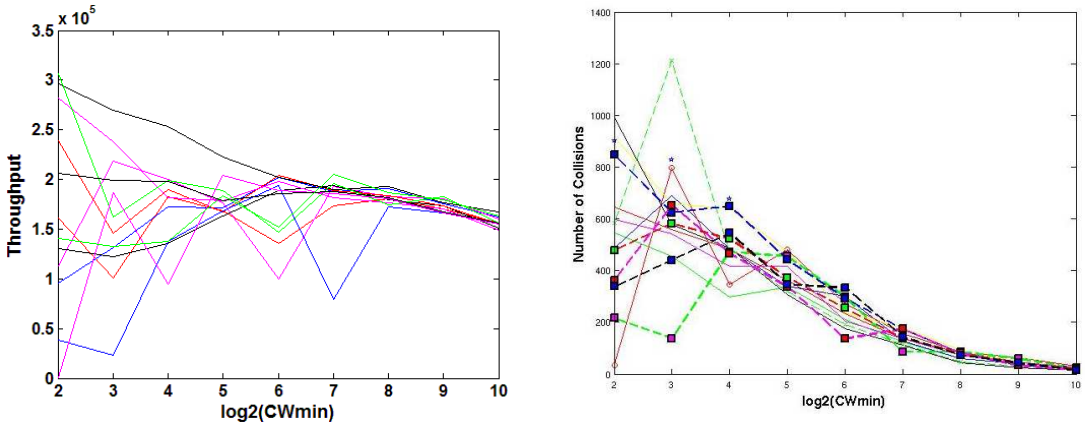

Figure 5.7: Individual Throughput as a Function of CW_{min} for 15 Flows (Left), Collisions per Flow with respect to CW_{min} (Right)

database for eventual dynamic adaptation of the driver. This is explained in the next chapter.

CHAPTER 6

NET-X EXPERIMENT RESULTS

In this chapter we present and discuss experimental results of dynamically changing the contention window, more specifically the CW_{min} in our test bed. Our main goal in this work was to provide a framework with support for changing the CW_{min} dynamically at run time during network operation. We have now integrated support for this in the Mad-Wifi driver's operation in the Net-X system.

6.1 Experiments for Functional Verification of the Modified Driver

Our main goal in this work was to implement the capability of dynamically adapting the contention window in our testbed. There was already a way to change the contention window for the Mad-Wifi driver, but for any changes to the contention window to take effect, the driver had to be restarted and reloaded in the system. This involves delays on the order of tens of seconds and is clearly not desirable for adapting the contention window in real time. We made changes to the driver so that it should now support modifications to the contention window even at run time, and we wanted to verify proper operation of this capability. Therefore, we chose to do an experiment to verify that we can in fact correctly make changes to the CW_{min} used by the driver at run time. To begin with, we start by a simple experiment, much as we did in NS-2 simulations.

Here we set up a single flow between two nodes in our test bed, running the modified Net-X system with adaptable contention window. The flow carries UDP traffic at the rate of 6 Mbps. The channel transmission rate is also fixed at 6 Mpbs. We use *iperf* [21] to generate UDP traffic and to measure throughput in all our experiments discussed in this chapter. We start with the smallest possible contention window allowed after our modifications, i.e., $CW_{min} = 1$, and keep increasing it in powers of 2 up to $CW_{min} = 2^{10}$. We make these increments at 10 second intervals. The corresponding throughput achieved by the single flow is shown in Fig. 6.1.

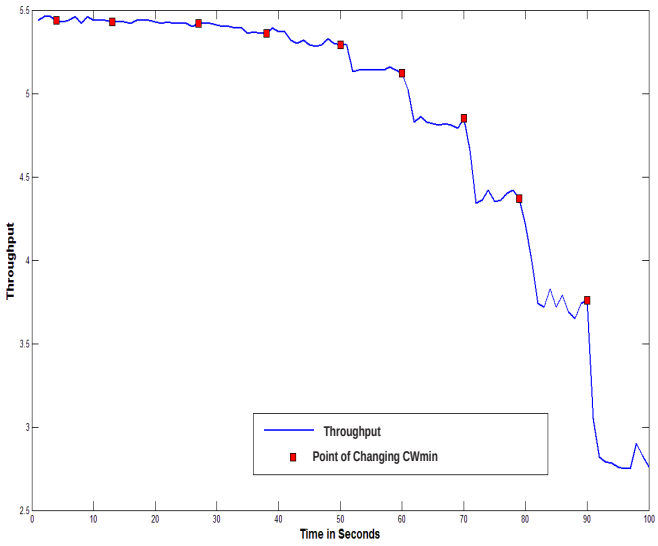

Figure 6.1: Effect of Dynamically Increasing CW_{min} in Net-X on Throughput for 1 Flow

Every time we increase the contention window size, there is a corresponding drop in the throughput. Also, the magnitude of the drop continues to increase as we continue increasing the value of CW_{min} with higher powers of 2. This is intuitive because we are increasing the window exponentially. Therefore, since the figure shows only the trends expected of the algorithm, we take it as an indication that we are now able to change the contention window in a real system at run time.

We also did another simple experiment where we changed CW_{min} without following a sequence of increasing from 2^0 to 2^{10}. We chose randomly from this range and again recorded the throughput obtained by one flow. The results are shown in Fig. 6.2.

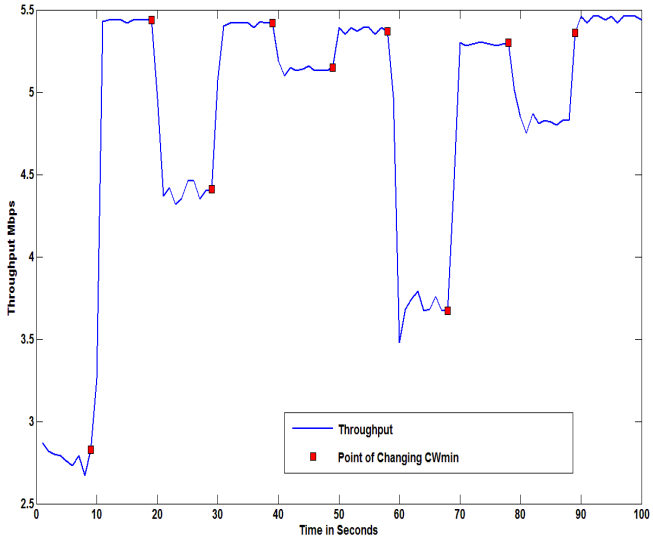

Figure 6.2: Effect of Dynamically Changing CW_{min} in Net-X on Throughput for 1 Flow

Again, the point of the above two experiments was to verify the correctness of the implementation rather than its performance. Because the Mad-Wifi driver has some closed source modules, we needed to make sure that the incorporated changes were not nullified or overridden somewhere in the closed source functionality. The above two experiments validate our claim that we have indeed integrated support in the Mad-Wifi device driver to allow a dynamic change in the contention window size and range as required by any user application.

6.2 Experiments for Performance Evaluation of the Modified Driver

We now examine the fairness behavior of the original Net-X system with the built-in IEEE 802.11 Binary Exponential Backoff (BEB) mechanism. We design these experiments in such a way that we have control over the level of contention in the network so that we can study the contention window adaptation of the system in isolation from other effects. To achieve a changing level of contention, we perform experiments where some flows start in the middle of an ongoing experiment. We will explain more details as we present the experiment results.

6.2.1 Evaluation based on qualitative fairness

In our experiments, all the flows carry UDP traffic at 6 Mbps. The packet size is set to 1470 bytes. We used iperf to generate UDP traffic and the server reporting feature of iperf to measure throughput. The server/receiver makes a record of the throughput achieved at 10 second intervals. The channel transmission rate is also fixed at 6 Mbps. All experiments last for 100 seconds and we do several runs of the same experiment and plot the average of the results. We use these experimental settings in all experiments discussed in this section.

We start with a simple experiment where we consider a small network of 8 nodes placed in each other's interference ranges. Moreover, we fix the channels so that all of them now tune to the same channel. Then we start 4 flows between them, carrying UDP traffic. We set the data rate to 6 Mpbs. We do this by starting 2 flows in the very beginning of the experiment and 2 more flows in the middle. This was done to observe how the network would adapt to changing contention in the neighborhood. Since all the nodes are in the vicinity of each other, all of them must contend for access to the channel. Figure 6.3

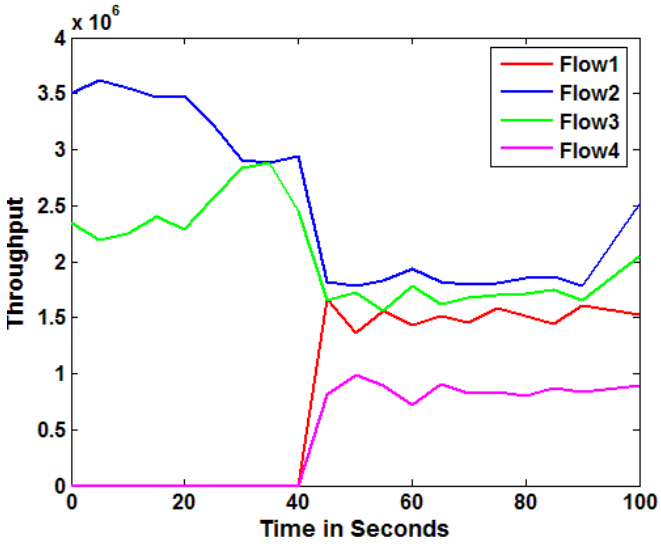

Figure 6.3: Effect of Changing Channel Contention in Net-X on Throughput for 4 Flows

shows the individual throughputs achieved by 4 flows in the original Net-X system. Here we start Flow 2 and Flow 3 at the beginning of the experiment, and Flow 1 and Flow 4 join 40 seconds into the experiment. From the figure, we can see that even in the beginning of the experiment, i.e., when there are only 2 flows, the original Net-X system is quite unfair, with Flow 2 getting about 3.5 Mbps while Flow 3 receives less than 2.5 Mbps. And after 40 seconds, when the other two flows start, the first 3 flows still get more throughput than Flow 4. This behavior can be explained by the fact that Flow 2 may have encountered some successful transmissions in the beginning of the experiment and therefore kept on reducing its contention window size. On the other hand, since only one of either Flow 2 or Flow 4 can be successful at any one time, the success of Flow 2 implies that Flow 4 found the channel to be busy more than once and so could not count down its backoff counter to zero, thereby worsening the unfairness even more. This simple example clearly illustrates that the IEEE 802.11 BEB can be quite unfair at times. This also motivates the development of more intelligent algorithms for contention resolution rather than a purely random

access method. We then did the same experiment for the modified Net-X where

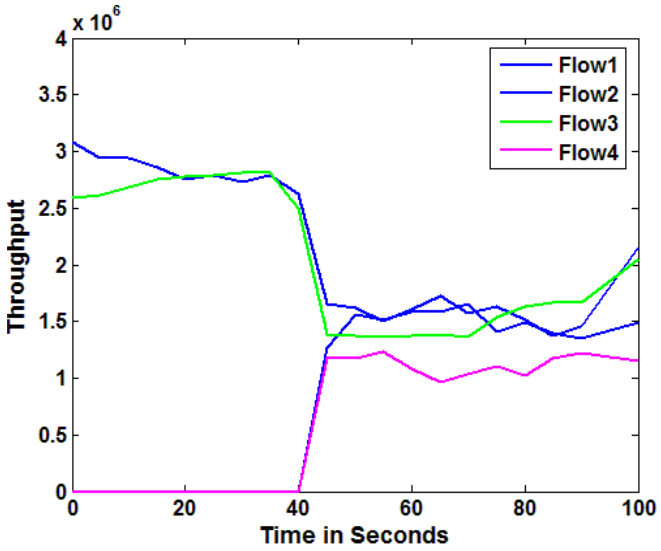

Figure 6.4: Effect of Dynamically Changing CW_{min} in Net-X on Throughput for 4 Flows

the contention window, more specifically, the CW_{min}, of nodes is changed dynamically depending upon the number of neighbors around them. We have found suitable contention window sizes for different network populations, as mentioned in Chapter 5. The modified driver now chooses a CW_{min} value from those appropriate values; i.e., as a node obtains more knowledge about its neighbors, it accounts for the neighbors when choosing its own contention window. This is desirable because it also facilitates the newly joining nodes in being able to access the channel. Otherwise, there is a chance that newly joining nodes will find the channel to be busy and will not be able to transmit. Therefore, with the feature of dynamically choosing an appropriate CW_{min}, the new results are shown in Fig. 6.4.

As the figure shows, the system is now much more fair than in the original case. The highly unfair regime of the original system from t=0 sec to t=30 sec is eliminated with a balanced throughput for both flows, and even after the 2 new flows start at time t=40 sec, they converge pretty well and fast to a more fair

throughput for all. This happens because as soon as the new nodes join the network and set up new flows, the old nodes realize this addition and from then onwards, when choosing a CW_{min}, they account for the fact that they are now contending with 3 other flows rather than just 1, as in the beginning of the experiment.

We then did a similar experiment in a network of 10 nodes with a total of 5 flows. Here again, in the beginning, there are only 2 flows in the network and then 3 more flows are set up during the course of the experiment. We wanted to see how the original and modified Net-X systems behave in the face of changing network population and increasing channel contention. Once again, we made all the nodes use a fixed channel in order to be able to control the contention in the network. Also, the flows carry UDP traffic and we fix the data rate to 6 Mbps. We start the experiment by setting up only 2 flows, Flow 2 and Flow 3. Then about 40 seconds into the experiment, we set up three more flows: Flow 1, Flow 4 and Flow 5. With these settings, the results of throughput achieved by each flow as a function of time in the original Net-X system are shown in Fig. 6.5.

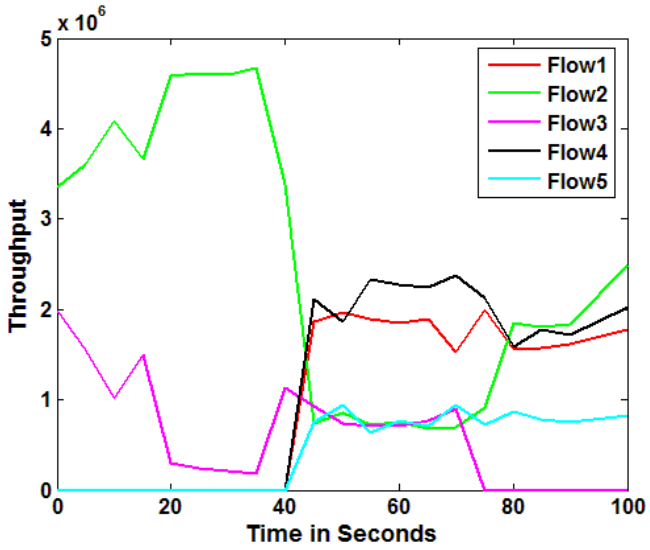

Figure 6.5: Effect of Changing Channel Contention in Net-X on Throughput for 5 Flows

This figure again manifests the same weakness of the original system: its unfairness. As can be seen from the figure, even though Flow 2 and Flow 3 were set up at the same time, they achieve highly different throughputs. At times, Flow 2 is receiving throughput even higher than 4.5 Mbps, whereas Flow 3 appears to be starved, with throughput falling even below 500 kbps. Of the three new flows, Flows 1 and 4 are in each other's neighborhood, whereas Flow 5 is in the neighborhood of Flows 2 and 3. Again we can see that soon after the new flows are set up, the network becomes highly unfair, with the throughput of Flow 3 falling to zero and Flow 2 again close to starving Flow 5.

Next, we repeat the same experiment for our modified Net-X system, and the results are shown in Fig. 6.6. We can see the huge improvement in terms of fairness in the beginning of the experiment where Flows 2 and 3 achieve close to equal throughput right from the start. Even after the new flows are set up, the system remains fair to all the flows. Since Flows 1 and 4 are in each other's neighborhood whereas Flows 2, 3 and 5 are in a separate physical neighborhood, we observe two sets of bandwidth allocations. Flows 1 and 4 achieve throughput of about 2 Mbps each whereas the other three flows obtain around 1 Mbps each.

Our next experiment was with a set of 12 nodes and 6 flows. The difference this time was that all the flows interfered with each other and hence were contending for the channel together. The rest of the settings remain the same. We set up flows to carry UDP traffic at 6 Mbps. Three flows start at the beginning of the experiment and three more join them in the middle of the experiment. The results for the original and the modified Net-X are shown in Figure 6.7.

From Figure 6.7, we can see how the original system fails to allocate bandwidth fairly to Flows 2, 3 and 4 in the beginning of the experiment. In the second half of the experiment, Flow 2 seems to remain dominant with a major

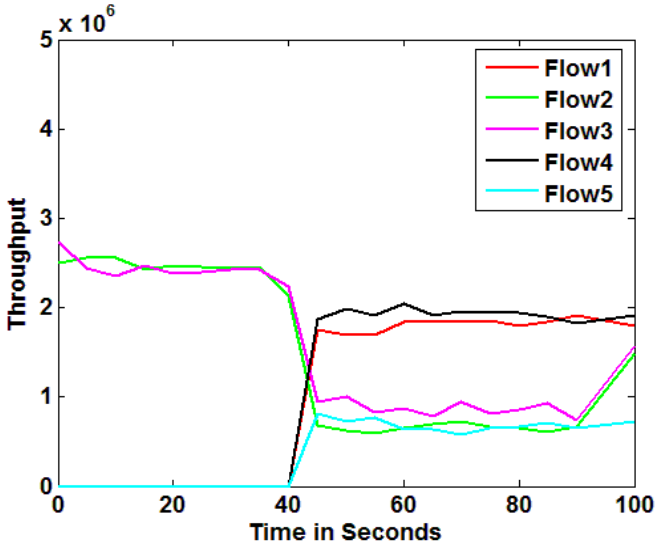

Figure 6.6: Effect of Dynamically Changing CW_{min} in Net-X on Throughput for 5 Flows

share of the bandwidth while others get much smaller shares. The modified system, however, shows an improvement right from the beginning. Flows 2, 3 and 4 are very close in terms of bandwidth shares and when new flows are set up, all of them converge quickly to more or less equal shares of bandwidth.

The next experiment is similar except that now all 6 flows start at the beginning of the experiment and then 3 of them go down at about 55 seconds into the experiment. The results are shown in Fig. 6.8.

We can see how the modified system adapts very well when contention in the system decreases and the flows that remain quickly converge to larger but fair throughput allocations. We next did a similar experiment for a set of 14 nodes by setting up 7 flows between them. The flows carry UDP traffic at 6 Mpbs. Again we start 4 flows at the beginning of the experiment and let the other 3 join in halfway through. Again the purpose of this setting is to be able to observe how the Net-X system adapts to changes in network load and resultant changing channel contention in the environment. Figure 6.9 (left) shows how the original system behaves. Here again, in the beginning of the experiment we can

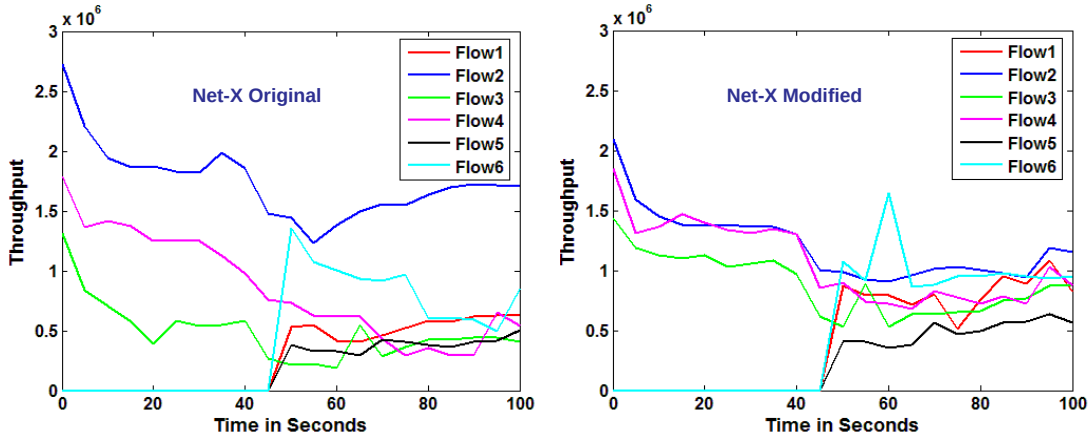

Figure 6.7: Net-X Performance for 6 Flows Experiment 1

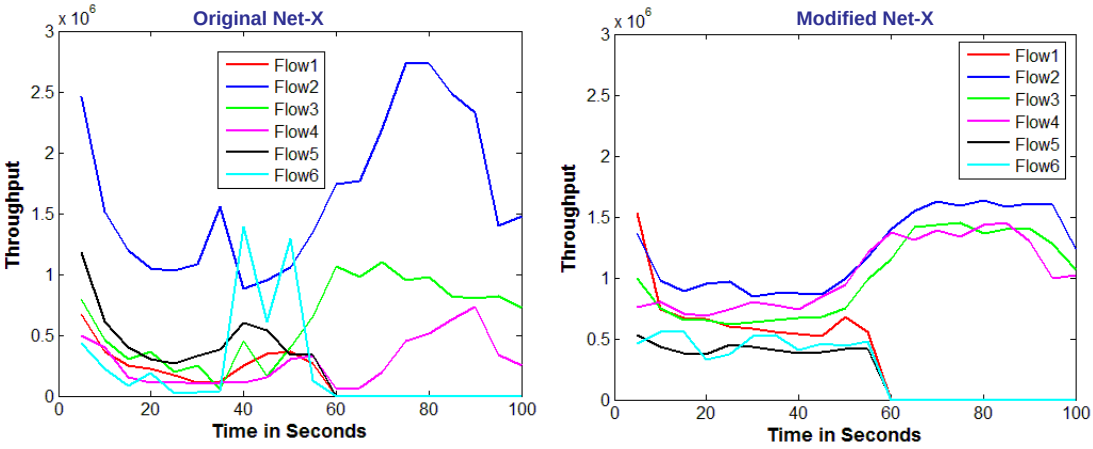

Figure 6.8: Net-X Performance for 6 Flows Experiment 2

see the unfairness between the shares of Flow 1 and Flow 3. Later on, when the new flows join in, throughput achieved by Flow 3 goes down much lower than the other neighbors, whereas a similar unfairness can be seen in the throughput shares of Flow 4 and Flow 6.

We now repeat the same experiment with 7 flows for the modified Net-X system with contention window adaptation. The results are shown in Fig. 6.9 (right). Again we can see the improvement in fairness. Flow 1 and Flow 3 obtain the same throughput right from the beginning of the experiment. The same is true for Flow 4 and Flow 5. Even after the new flows join in, the

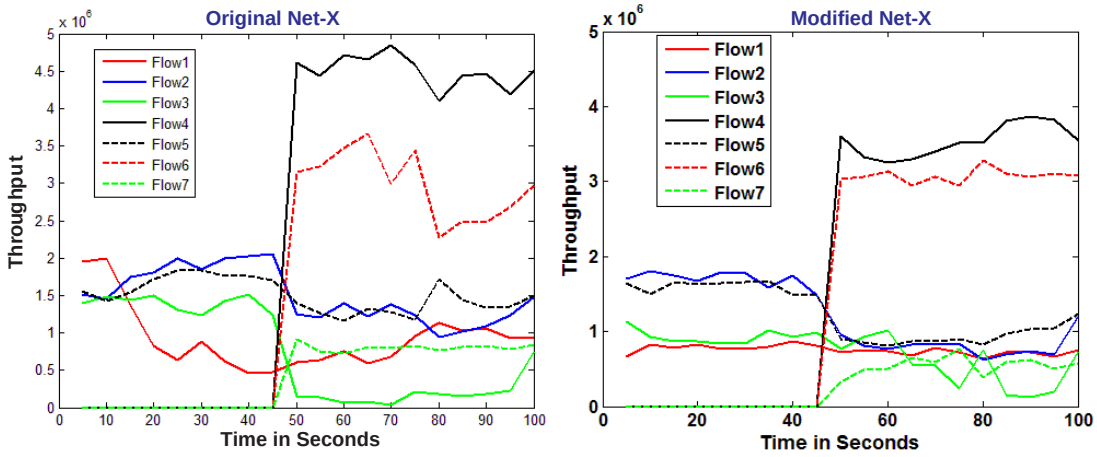

Figure 6.9: Effect of Changing Channel Contention in Net-X on Throughput for 7 Flows

network adapts reasonably well, with Flows 4 and 6 also getting a close enough share of bandwidth.

6.2.2 Evaluation based on quantitative fairness

We also did a quantitative analysis of the fairness achieved as a result of operation of the modified Net-X system. For this, we chose to use Jain's fairness index as our metric of fairness comparisons. We chose this very popular index as our metric for fairness comparison because, reportedly it is not unduly sensitive to atypical network flows [22]. It is defined as

$$fairness = \frac{\sum(x_i)^2}{(n \times \sum x_i)^2} \tag{6.1}$$

where n is the number of flows and x_i is the throughput of flow i. The result of this fairness calculation lies between 1 and $1/n$, with the former being the best and the latter being the worst. With this definition of fairness, we now compare the Jain's fairness index value for the original Net-X and the modified one.

The Jain's fairness index results for the 5 flows experiment are plotted in

Fig. 6.10. This accounts only for the first half of the experiment, i.e., from time 0 to 40 sec. In this interval, only two flows exist in the system.

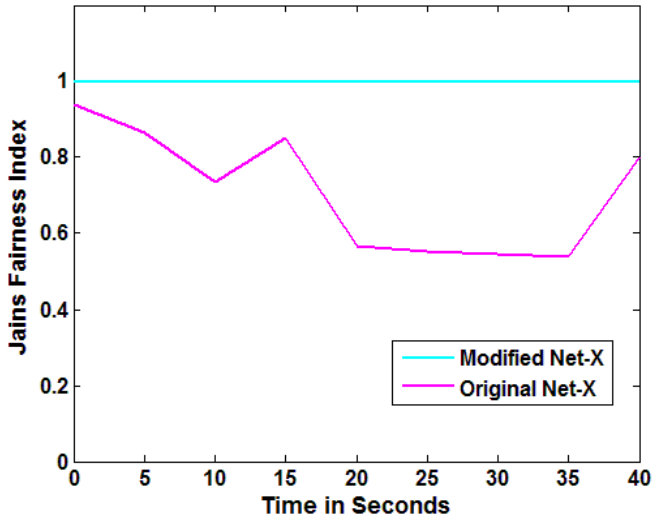

Figure 6.10: Jain's Fairness Index for 5 Flows Experiment (t=0 to t=40)

As we can see from Fig. 6.10, the modified Net-X system performs *much* better than the original one. The Jain's fairness index for the modified system is always very close to the best-case value of 1, whereas the original Net-X falls to as low as 0.55. We should also notice that since we have only two flows in this part of the experiment, the worst-case Jain's fairness index value is 1/2, and the original Net-X system is very close to the worst-case scenario.

As for the second half of the experiment, Jain's fairness index is plotted in Fig. 6.11. Here we can see that both the algorithms perform well enough for Flows 1 and 4 (shown in the left). However, for Flows 2, 3 and 5, the original Net-X system again lags far behind the modified one in terms of fairness. The modified Net-X system almost always maintains a Fairness index of 1, which is the best-case value.

Next we plot Jain's fairness index for the first experiment with 6 flows, all contending for access to the channel. This figure again shows the strength of the

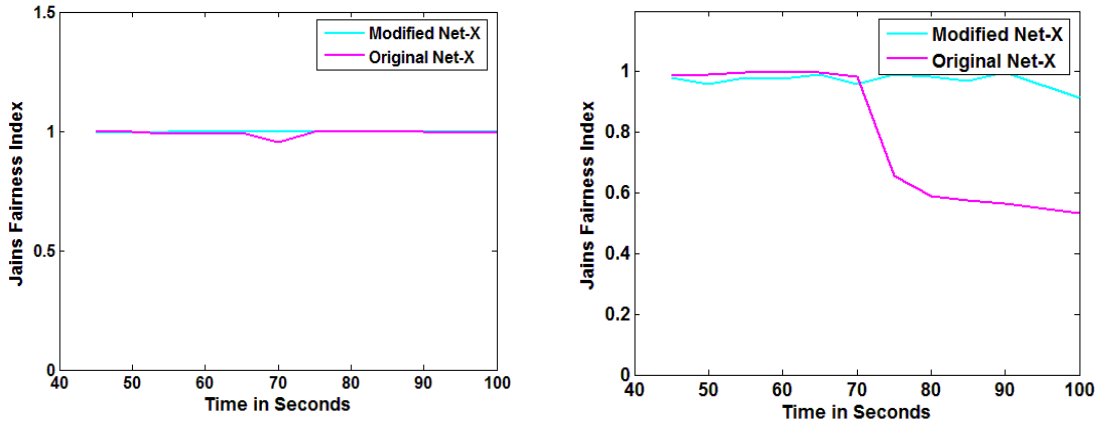

Figure 6.11: Jain's Fairness Index for 5 Flows Experiment (t=45 to t=100)

contention resolution capability of the modified Net-X system. We can see that the modified system maintains a fairness of index of 1, i.e., the best-case value for the major portion of the experiment, whereas the original system's fairness index degrades to as low as 0.65. These results are shown in Figure 6.12.

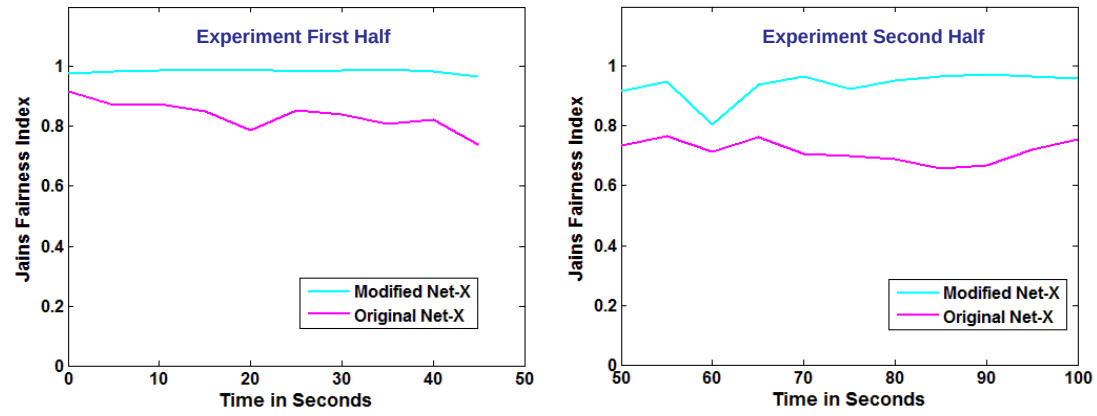

Figure 6.12: Jain's Fairness Index for 6 Flows Experiment 1

The next result is that fairness index for the second experiment with six flows. This experiment started with 6 flows and then 3 of these were stopped in the second half of the experiment. The results are shown in Fig. 6.13. We also plot Jain's fairness index for the 7 Flow experiment. For the first half of the experiment, Fig. 6.14 plots the fairness index for Flows 2 and 5 on the left and

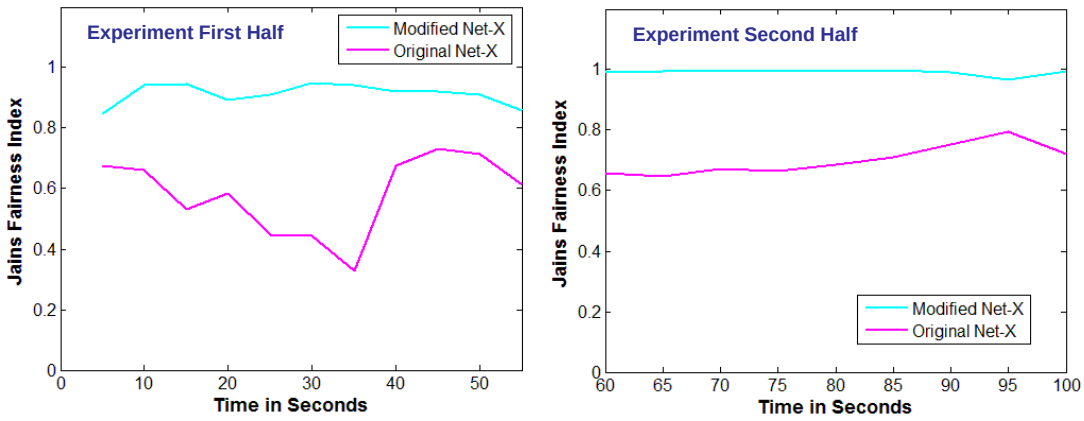

Figure 6.13: Jain's Fairness Index for 6 Flows Experiment 2

shows the same for Flows 1 and 3 on the right. Similarly for the second half of the experiment, Fig. 6.15 shows the fairness index for Flows 4 and 6 on the left and for Flows $1, 2, 3, 5$ and 7 on the right. We can again see the clear improvement achieved by the modified system.

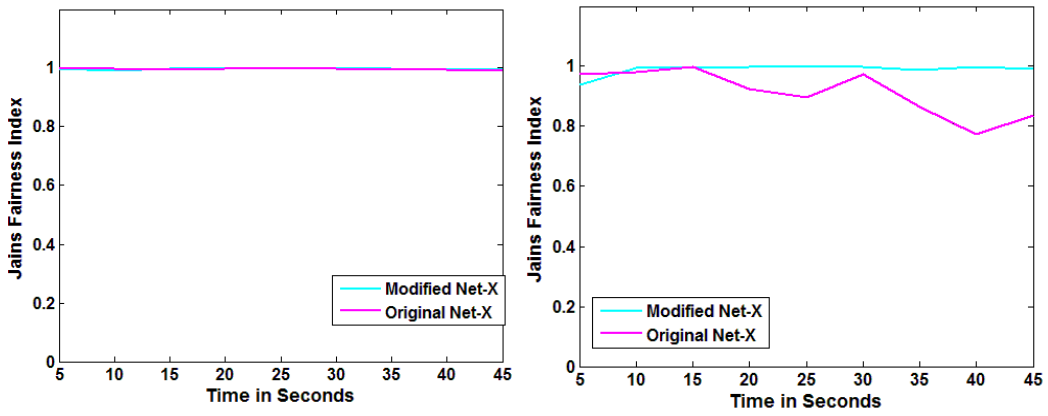

Figure 6.14: Jain's Fairness Index for 7 Flows Experiment (t=0 to t=45)

6.2.3 Evaluation based on aggregate throughput

Aggregate throughput is usually used as a metric of how well the channel resources have been utilized. The higher the aggregate throughput, the better the resource utilization. However, a higher aggregate throughput does not

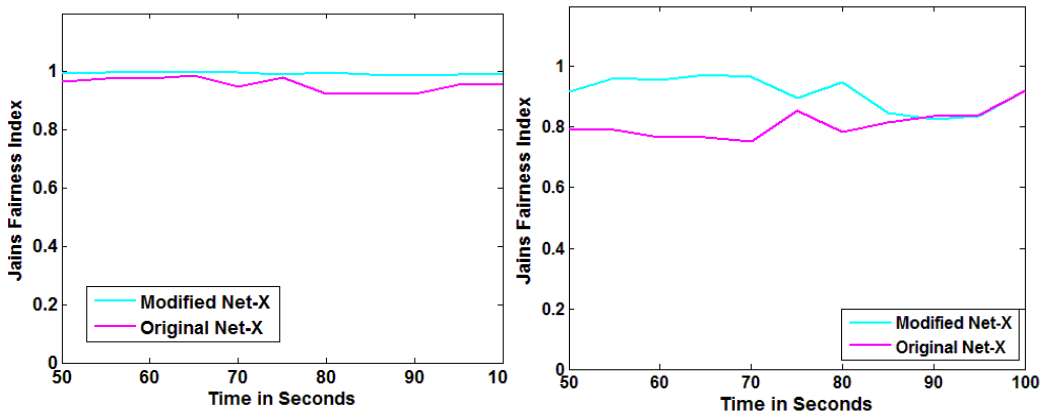

Figure 6.15: Jain's Fairness Index for 7 Flows Experiment (t=50 to t=100)

necessarily imply better fairness. There might be circumstances where a higher aggregate throughput is the result of one flow capturing the channel and starving all other flows. Therefore, while aggregate throughput is a metric that should ideally be maximized, for reasons of practicality, it should not be maximized in isolation; that is, we should consider a combination of several performance metrics when evaluating system performance. That is why, in this work, we chose to consider a combination of aggregate throughput and individual flow throughputs as our performance metric. In this section, we plot the aggregate network throughput achieved by the two systems being studied in this work: the original Net-X system and the modified one. The results are shown in Figs. 6.16 to 6.18.

As we can see, even the aggregate network throughput has not been compromised considerably by the modified algorithm. In fact, Figs. 6.17 (left) and 6.18 show that the modified Net-X system sometimes *improves* the aggregate throughput as well. We therefore gained fairness without losing much network resource utilization. This was one of the main objectives of this simple algorithm.

The experiments discussed in this chapter identify the potential for

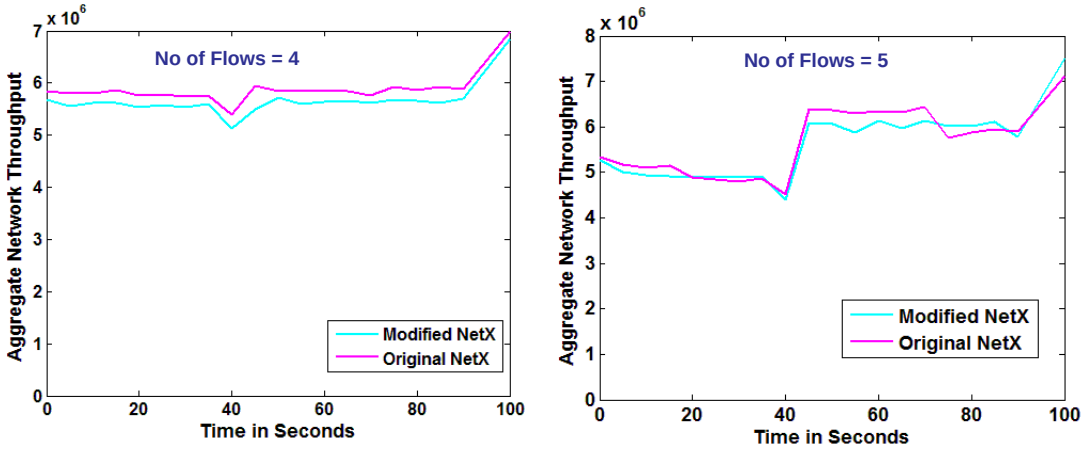

Figure 6.16: Comparison of Aggregate Throughput for 4 and 5 Flows

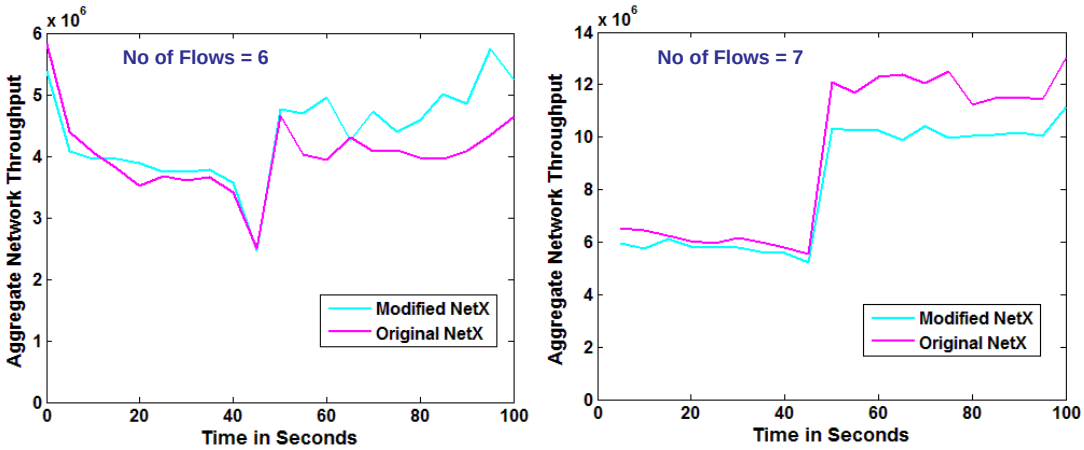

Figure 6.17: Comparison of Aggregate Throughput for 6 and 7 Flows

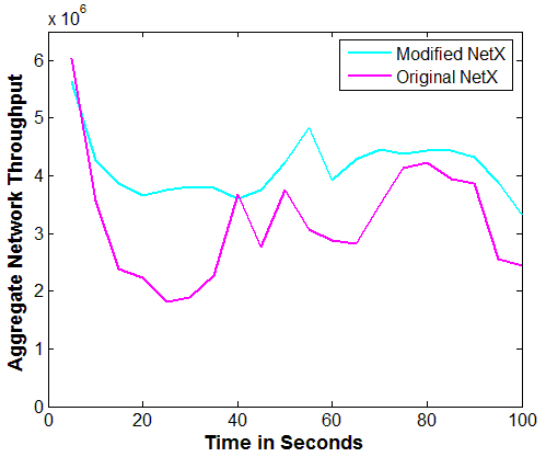

Figure 6.18: Comparison of Aggregate Throughput for 6 Flows Experiment 2

throughput unfairness that exists in the original random access method with exponential backoff used for contention resolution. They also motivate the development of more sophisticated ways of dealing with contention in the network. With our simple modifications, we have shown that there is much room for improvement in the area of contention resolution and that even some simple mechanisms can offer many benefits.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis, we have designed and implemented a framework for adaptive control of the contention window used by wireless devices for contention resolution. We have integrated support for this functionality in the Net-X test bed, which is a real test bed of more than 20 wireless nodes using Atheros-based chip sets and using Mad-Wifi as the wireless device driver. We started by describing the problem and identifying the weaknesses of the simple IEEE 802.11 congestion avoidance mechanism. We also looked at the many different variations and proposals for improving upon the inherent contention resolution of IEEE 802.11. Through this background study, we were able to recognize that a real world system that provides the capability to experiment with different contention resolution schemes was all but absent. We therefore stressed the need for a real physical system that provides a framework for implementing different algorithms for contention resolution so that more research opportunities can turn up in this area. With this goal in sight, we started this work on enabling the Net-X test bed to allow the user to implement different contention resolution protocols and study their performance.

One of the major hurdles for this work was the fact that the Mad-Wifi driver is only partially open source. Therefore, it could not be ascertained beforehand whether such a capability is even possible for this driver. However, as we worked through the design and implementation phase, we were able to identify the sections of code that we could modify and use to serve our purpose. Also, once

we had completed the implementation, we first did a set of experiments to verify the correctness of our implementation. Once we could demonstrate the correctness, we then set out to do more experiments with the new design.

With some simple experiments, we have been able to show improvements in the new system with regard to throughput fairness in the wireless scenario. The results look promising, although the algorithm still needs to mature. We would like to remark here that, rather than implementing a specific contention resolution protocol, with the new system we provide a general framework for trying out new contention resolution protocols. Because this work provides the user with control over how the driver modifies its contention window at run time, any number of schemes proposed in the literature can be implemented and their performance evaluated on a real test bed. We plan to do performance comparisons of such schemes as part of our future work.

REFERENCES

[1] N. Tripathi and N. H. Vaidya, "Rate control framework for Net-X," M.S. thesis, University of Illinois at Urbana-Champaign, 2007.

[2] V. Raman and N. H. Vaidya, "Adjacent channel interference reduction in multichannel wireless networks using intelligent channel allocation," University of Illinois at Urbana-Champaign, Tech. Rep., March 2009.

[3] C. Chereddi and N. H. Vaidya, "System architecture for multichannel multi-interface wireless networks," M.S. thesis, University of Illinois at Urbana-Champaign, 2006.

[4] T. Ozugur, M. Naghshineh, P. Kermani, and J. Copeland, "Balanced media access methods for wireless networks," in *Proceedings of the 4th Annual ACM/IEEE Conference on Mobile Computing and Networking*, 1998, pp. 21–32.

[5] 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, pp. 785–799, Dec. 2000.

[6] M. Mishra and A. Sahoo, "A contention window based differentiation mechanism for providing QoS in wireless LANs," *Ninth International Conference on Information Technology*, vol. 18, pp. 72–76, 2006.

[7] Q. Pang, S. C. Liew, and J. Lee, "A TCP-like adaptive contention window scheme for WLAN," in *IEEE International Conference on Communications*, vol. 6, 2004, pp. 3723–3727.

[8] S. Hussain, K. Zia, M. Khan, S. Ahmad, and S. Farooq, "Dynamic contention window for quality of service in IEEE 802.11 networks," in *National Conference on Emerging Technologies*, 2004, pp. 12–16.

[9] P. Byungjoo and L. Haniph, "Performance improvement of TCP with an efficient contention window control mechanism (ECWC) in IEEE 802.11 based multi-hop wireless networks," in *Fifth International Conference ADHOC-NOW*, Aug. 2006, pp. 365–375.

[10] A. Nafaa, A. Ksentini, A. Mehaoua, B. Ishibashi, Y. Iraqi, and R. Boutaba, "Sliding contention window (SCW): Towards backoff range-based service differentiation over IEEE 802.11 wireless LAN networks," *IEEE Network Magazine, Special Issue on Wireless Local Area Networking: QoS Provision and Resource Management*, pp. 45–51, July/Aug. 2005.

[11] Y. Kim and W. Kang, "Distance adaptive window mechanism for wireless sensor networks," in *23rd International Technical Conference on Circuits/Systems, Computers and Communications*, 2008, pp. 1693–1696.

[12] A. Ksentini, A. Nafaa, A. Gueroui, and M. Naimi, "Determinist contention window algorithm for IEEE 802.11," in *Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2005, pp. 2712–2716.

[13] N. Saxenaa, A. Royb, and J. Shin, "Dynamic duty cycle and adaptive contention window based QoS-MAC protocol for wireless multimedia sensor networks," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 52, pp. 2532–2542, Sept. 2008.

[14] S. Ganu, K. Ramachandran, M. Gruteser, I. Seskar, and J. Deng, "Methods for restoring MAC layer fairness in IEEE 802.11 networks with physical layer capture," in *International Symposium on Mobile Ad Hoc Networking & Computing Proceedings of the 2nd International Workshop on Multi-hop Ad Hoc Networks: From Theory to Reality*, 2006, pp. 7–14.

[15] P. Kyasanur, C. Chereddi, and N. H. Vaidya, "Net-X: System extensions for supporting multiple channels, multiple interfaces, and other interface capabilities," University of Illinois at Urbana-Champaign, Tech. Rep., August 2006.

[16] P. Kyasanur and N. H. Vaidya, "Routing and interface assignment in multi-channel multi-interface wireless networks," in *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, 2005, pp. 2051–2056.

[17] P. Kyasanur and N. H. Vaidya, "Routing and link-layer protocols for multi-channel multi-interface ad hoc wireless networks," *Sigmobile Mobile Computing and Communications Review*, vol. 10, pp. 31–43, Jan. 2006.

[18] http://linuxwireless.org/en/users/Drivers/ath5k. Accessed on April 10, 2009.

[19] http://linuxwireless.org/en/users/Drivers/ath9k. Accessed on April 10, 2009.

[20] "Hal usage notes," http://madwifi-project.org/wiki/DevDocs/HAL-usage. Accessed on April 10, 2009.

[21] http://iperf.sourceforge.net/. Accessed on April 10, 2009.

[22] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared systems," Digital Equipment Corporation, Tech. Rep. 301, 1984.