CHALMERS

Multi-Channel, Multi-Radio in Wireless Mesh Networks

Master of Science Thesis in Computer Science and Engineering

KRISTOFFER FREDRIKSSON
MATTIAS GUHL

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2011

Multi-Channel, Multi-Radio in Wireless Mesh Networks

KRISTOFFER FREDRIKSSON
MATTIAS GUHL

Examiner: MARINA PAPATRIANTAFILOU

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2011

Abstract

Decreasing hardware prices enables increased performance in wireless networks to a low cost. By adding an extra WLAN radio card to existing single radio network platforms, the possibility to utilize additional frequencies arise and opens the world of multi-channel, multi-radio (MCMR).

In this report, we investigate different approaches to make as good use of MCMR as possible. A hybrid technique using two radios, one receiving data and the other transmitting, is chosen and implemented. This requires a user-space application, modifications to the wireless driver and the development of a Linux kernel bonding module handling the communication between user-space and drivers.

Test results shows that the chosen method increase performance substantially compared to the standard single-channel, single-radio setup, which makes further development interesting as new faster wireless techniques becomes standard.

Sammanfattning

Fallande hårdvarupriser gör det möjligt att till ett lågt pris öka prestandan i trådlösa nätverk. Genom att utnyttja två WLAN-kort istället för ett i befintliga plattformar öppnar sig en värld av möjligheter inom multi-channel, multi-radio (MCMR).

I rapporten undersöker vi olika metoder för att implementera MCMR. Metoden som implementeras går kortfattat ut på att ett av de två radiokorten tar emot data och det andra skickar data. För att göra detta möjlighet krävs en applikation, modifieringar av drivrutiner samt utvecklandet av en bondingmodul i Linuxkärnan vars uppgift är att sköta kommunikationen mellan applikation och drivrutiner.

Testresultaten visar att hastigheten under olika förhållanden ökar påtagligt i jämförelse med vanlig single-channel, single-radio, vilket gör fortsatt utveckling intressant i takt med att nya snabbare trådlösa tekniker blir standard.

Preface

This is a thesis work in computer technology performed at CRL Sweden on behalf of the Department of Computer Science and Engineering at Chalmers University of Technology.

We would like to thank our fellow adviser at CRL Sweden, Anders Lundström, for sharing his knowledge and our examiner at Chalmers, Marina Papatriantafilou, for looking after us. We would also like to thank Christian Svensson for performing the initial multi-channel multi-radio specific enhancements of the GNU/Linux bonding module.

Responsible for developing the MadWifi enhancements and network communication parts of the user space application has been Kristoffer Fredriksson. Mattias Guhl has been responsible for the neighbor- and channel management parts of the user application, performance testing and thesis illustrations. Enhancements and performance optimizations of the GNU/Linux bonding driver has been in collaboration.

Contents

1 Introduction

The ease of deploying wireless networks has resulted in widespread usage in all areas. In everything from small personal networks to large city wide networks there is an advantage to avoid the use of cables. With increased usage comes increased demands on performance; both to support more users and to enable services such as high quality streaming television and audio - these are services which requires high data throughput as well as low end-to-end delays.

Developing new wireless technologies with higher throughput capabilities is time and resource consuming and the deployment often requires replacement of current hardware with new hardware. An alternative solution is to make use of already existing cheap hardware and exploit the available frequency spectrum by simultaneously utilizing multiple wireless network cards tuned to individual channels. Normally the same channel is used throughout a network, if multiple channels are used, it is often only done to separate the client part of the network from the backbone.

The features of the wireless medium is such that only one source can use a specified frequency spectrum to transfer information at a given moment. If a second source tries to use the medium, it will cause a collision and corruption of data. Nodes operating on separated frequencies can't communicate with each another, so by using multiple non cross interfering channels, also referred to as orthogonal channels, it is possible for multiple nodes to concurrently send data and thereby increase the networks overall throughput.

1.1 Design choices and issues

Implementing multi-channel is not trivial [1, 12]. There are numerous possible executions to consider, each suitable for specific types of network layouts and with its own advantages and problems. A few fundamental design choices are:

- Single radio or multiple radios
- Static or dynamic channel assignment
- At which layer in the OSI model to do the implementation

An early choice in the design phase is to choose whether to implement multi-channel with a single radio or multiple radios. Both options have their advantages and their problems. The first solution often requires rendezvous and time synchronization between nodes in the network which can be problematic [16, 9]. A multi-radio approach on the other hand can be realized without complex time synchronization but requires additional hardware. Another choice is between static and dynamic channel assignment. A static precalculated-assignment can be favorable in a static network but will not work well in a network where nodes move arbitrarily.

At which OSI layer to implement the support for multi-channel affects for example the support for current hardware and applications.

Some aspects that must be considered regardless of implementation are:

- *Connectivity*: Unwanted network segmentation should be avoided to ensure that nodes in range of each other can communicate.
- *Broadcast messages*: For instance, how to ensure that local broadcast messages should be able to reach all nodes in the neighborhood even if they are on different channels.
- *Mobility*: How does the network support arbitrary moving nodes?
- *Delay*: It takes time for the hardware to change between channels. Frequent channel switching will introduce delays which impacts network performance.
- *Single card nodes*: How will a solution with multiple radios support single radio nodes?
- *Synchronization*: If nodes should rendezvous to a specific channel to negotiate channel usage and/or deliver broadcasts how will the synchronization between nodes work?
- *Operating system support*: Does the operating system support multiple cards? How does the operating system choose which card to route the traffic through?
- *Routing*: Should routing be aware of multi channel paths to improve performance?

1.2 Purpose

The purpose of this thesis is to identify and implement a suitable multi-channel multi-radio solution to be used in an ad-hoc network. The chosen solution should increase the networks overall throughput without introducing unacceptable end-to-end delays. The implementation should also be able to function in both a static and a mobile network with arbitrary placed nodes and be able to handle nodes leaving and entering the network.

The solution should run on GNU/Linux and support existing generic hardware. E.g no modifications to the current IEEE 802.11-MAC layer. The development and target platform is the Avila GW2348-2 Network Platform by Gateworks Corporation[1] with support for two Mini-PCI wireless network cards. In software there should be no hard upper limit on supported cards.

[1] `http://www.gateworks.com`

1.3 Scope

The solution will support GNU/Linux and no other operating system. To be able to optimize network card drivers, the development will focus on Atheros[2] based cards and the Open Source WLAN driver MadWifi[3]. It is not required for the channel assignment to be optimal and routing optimizations will not be implemented. Single radio solutions will be discussed but not considered for a final solution. How to handle single radio nodes in a multi-radio network will also be discussed but not implemented or solved. The solution should be suitable for omni directional antennas. Specific solutions for directional antennas will not be discussed.

The rest of the report is structured as follows. First the analysis present the functions that must be available in the operating system and wireless drivers to support multi-channel multi-radio and the problems that must be solved to gain satisfactory performance. It will also provide a review of proposed solutions published in scientific reports. Then the implementation of choice will be described in detail in the method followed by a presentation of the results gained while performing tests. In the discussion the chosen solution is evaluated - advantages, drawbacks as well as the future of the technology is talked about. The paper ends with a conclusion, followed by references and nomenclature.

[2]`http://www.atheros.com`
[3]`http://madwifi-project.org`

2 Analysis

This chapter will present the required functions needed in the operating system to implement multi-channel multi-radio. It will also discuss difficulties with handling multiple channels and evaluate proposed solutions to as effectively as possible make use of the advantages given by multi-channel multi-radio.

2.1 Required support

Regardless of implementation strategy, some basic support is needed in both the operating system and the wireless drivers. One such basic function is the ability to choose on a per packet basis which interface and channel the packet should leave on. Another example is the ability to force the hardware to change channel on demand and to do this sufficiently fast. Furthermore it is important to provide the possibility to configure settings for critical parts. The Linux kernel and the WLAN driver MadWifi doesn't provide all these functions natively. However, the source code for both are provided under an Open Source license, hence it is possible to add the needed support.

A specific requirement for the WLAN driver is that it can operate in so called ad-hoc or IBSS (Independent Basic Service Set) mode. This mode allows nodes in the vicinity of each other to communicate with each other without using a common access point. This is desirable in a multi-channel multi-radio scenario where the purpose is to have communicating neighboring nodes to use separated frequency spectrums. Not all drivers provide this mode, however MadWifi does.

2.2 Handling multiple channels

When faced with the opportunity to choose among multiple channels a few questions arise:

- How to assign each interface a channel?
- How much does the delay introduced by hardware and software during channel switch affect performance?
- How to decide when to switch channel?
- Are concurrent transmissions on channels close to each other in frequency interfering?

A network consisting of stationary nodes and clients that do not move around - but may log off and on - is called a static network. The opposite is called a dynamic or mobile network and requires the nodes to be aware of changes in the network topology.

Depending on whether the wireless network is to be static or dynamic, different methods can be used to assign each interface a channel. This can be done statically by hand, by using an algorithm or letting the interfaces dynamically choose between available channels. If the channels are assigned at start-up using an algorithm to find the most optimal setup, the network is more or less static by default since the algorithm has to be run all over again each time a node wants to enter or leave the network. Leaving the network includes node failures, making the network vulnerable to all sorts of hardware or software malfunction. On the other hand, a static channel assignment requires no negotiation among the nodes through messages, hence less CPU and bandwidth overhead. Thus, a dynamic network where nodes may enter and leave unpredictably, cannot employ a static channel assignment. Especially not in a wireless network where traffic may be routed through clients that frequently enters and leaves the network. A static channel assignment occur by default when the number of channels are equal to the number of interfaces. Though, one extra channel is enough to enable a dynamic channel assignment.

Low switching times is not a priority in regular wireless networks and therefor hardware and wireless drivers aren't developed with optimized switching times in mind. Measurements have shown that switching channel on a interface takes several milliseconds (see section 3.2.3) depending on drivers and hardware, which adds up to a significant amount of time if for example a message has to be sent over all available frequencies. This leads to the conclusion that improvements in both hardware and the software are possible. The time it takes for the system to switch channel is therefore not negligible and an important aspect when choosing a dynamic channel assignment scheme. Due to this, a static approach can be preferred in a hardware setup where the cost of switching between channels are high in terms of time.

Another difficulty with switching channel is when to switch. The channel can't be switched as soon as a packet to be sent on a different channel arrives, as this would lead to a frequent channel switching. Instead the packet has to be queued and wait for it's turn. Still, using queues won't solve the problem. A queue can't always be emptied before changing channel either, since this could lead to starvation of other queues if the current queue is filled continuously. Hence, rules regarding when to switch channel and algorithms for how to service queues has to be introduced.

The IEEE 802.11 [2] standard defines a series of channels divided into two separated frequency ranges, 802.11a at 2.4 GHz and 802.11b/g at 5.0 GHz. Depending on hardware limitations and country specific regulations, the user is restricted to a set of channels. Swedish users can for example choose from 13 channels in the 2.4 GHz range and 19 channels in the 5.0 GHz range [25]. Unfortunately, adjacent channels may compete for the frequency spectra, hence interfering and not optimal to use simultaneously in a multi-channel environment. This applies best to the 2.4 GHz range where tests have shown that only 3 of the 13 channels are completely non-interfering or so called orthogonal [1]. In addition to interfering channels, common hardware targeted to consumers -

such as routers used for residential broadband sharing - supports only 802.11b/g, leaving this spectrum crowded. Because of this the 5.0 GHz range is the most suitable spectrum to operate high performance backbones and perform tests in.

One might think that two WLAN cards would perform twice as good as one. However, this is not possible due to the channel assignment issue.

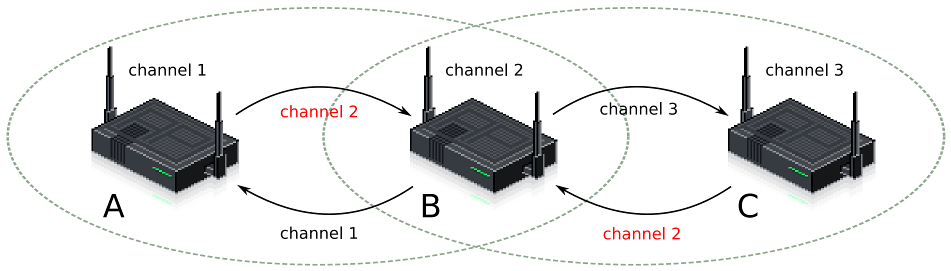


Figure 1: Channel assignment issue

The channels can't be arranged to avoid interference even if both cards have the ability to transmit, the intermediate node will create a bottleneck. The only setup that in theory would double the throughput, using two WLAN cards, would be a static approach where each node uses two separate channels. This would have the same effect as deploying two identical paths of single-channel, single-radio nodes in parallel. Hence, only applicable if creating a high speed path from point A to point B.

The hidden terminal problem

In a multi-hop ad-hoc network, all nodes aren't in range of each other and can therefor not be aware of all ongoing transmissions. For example, node B is in the range of both node A and C, but A and C can't hear each other. That is, node A is hidden from node C. If node A and B are communicating, node C is unaware of this and might try to talk to node B, causing a collision. This scenario is called the hidden terminal problem [18].

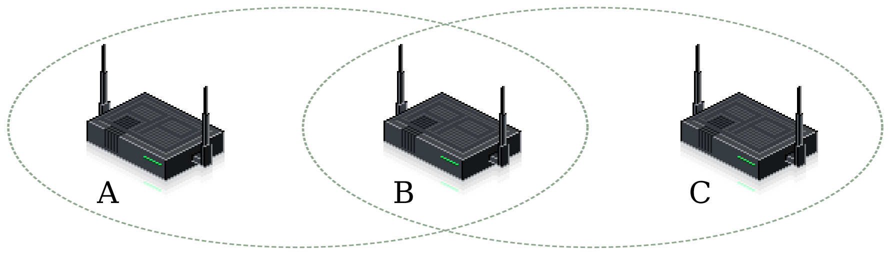


Figure 2: Hidden terminal in single channel networks

The 802.11 standard has a mechanism called DCF to avoid this problem. DCF reserves a channel by exchanging RTC/CTS messages, meaning nodes only sends data when the channel is free. In a multi-channel environment this problem is however more complex due to the fact that nodes in range of each other can be hidden when operating on separate channels [5].

As seen in Figure 3, node D is already transmitting data to node C on channel 2, when node A sends a request to send (RTS) to initiate data transmission on channel 3. B answers with a clear to send (CTS) message and A starts transmitting. However, node C couldn't hear the CTS and is unaware of the communication between A and B. Short after node C sends a request to transfer data on channel 3, using channel 2, to node D that answers with a CTS. When node starts transmitting a collision occurs on channel 3.

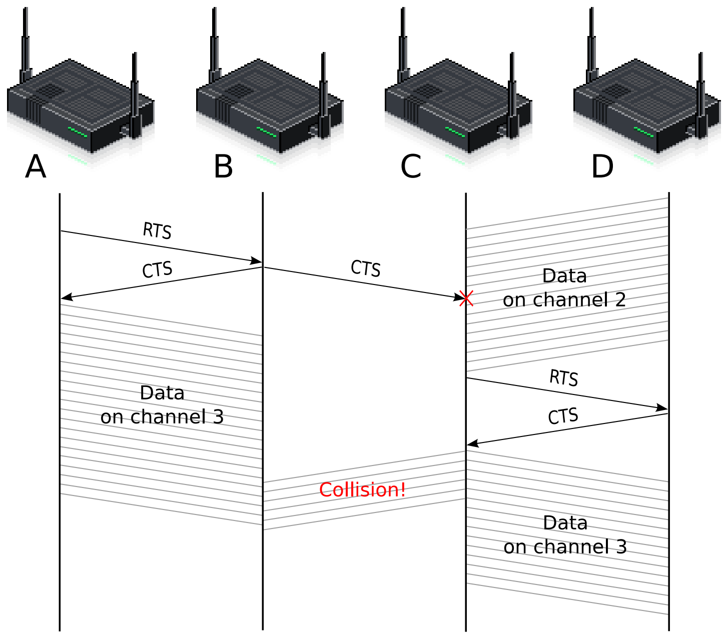


Figure 3: Hidden Terminal in multi channel networks

2.3 Review of proposed solutions published in scientific reports

The following section presents and evaluates already proposed modifications and/or additions to standard protocols and mechanisms to support and/or better utilize multi-channel multi-radio. To understand the basic difference between solutions, the chapter starts of with explaining the Internet protocol stack.

2.3.1 The Internet protocol stack

The communication between software and hardware is divided into different layers to standardize network communication and to abstract the different parts of the communication chain. The most common model of the Internet protocol stack is the seven layer OSI model [23]. The model defines clear rules on how the layers communicate, making it easier to develop new functions at certain layers without having to worry about what's going on in the adjacent layers, as long as the in- and output follows the standard.

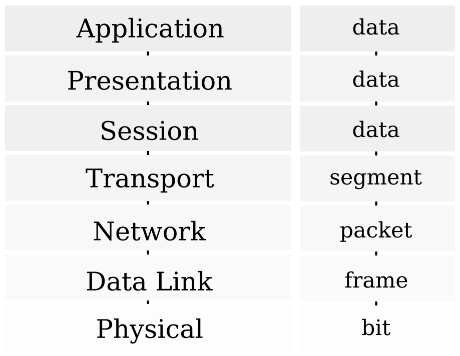


Figure 4: The OSI Model

Information that leaves the application layer, will be processed in turn by each and every layer down to the Physical layer. Each layer performs its tasks and the information leaves the Physical layer as a series of ones and zeros traveling over the physical medium. Data reaching an intermediate node, won't necessarily be processed by each layer. A normal router will process the information at the network layer and decide what to do with the data, for example decide which interface the data must leave on to reach its destination. There are however also more advanced routers which processes the data and makes decisions on the application level.

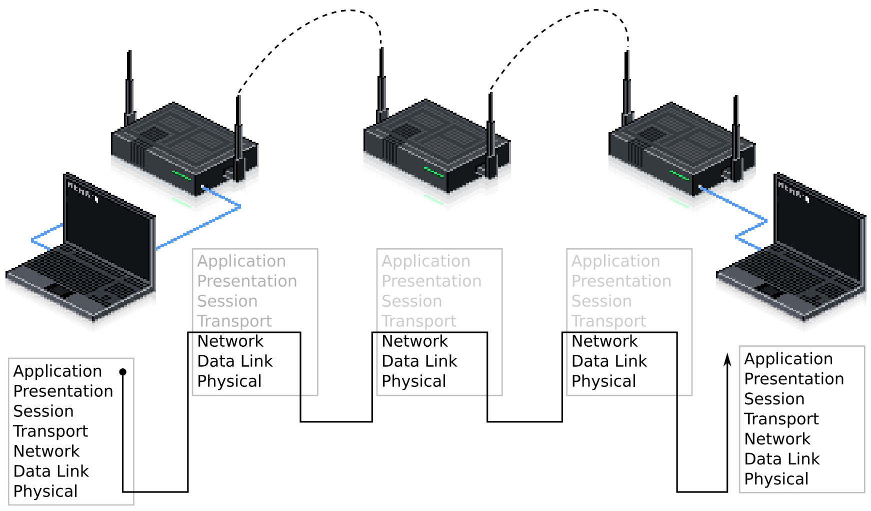


Figure 5: Routing

The following proposed solutions requires either modifications to the Network layer or the Data Link layer, depending on which mechanism or protocol the solution utilizes. The Internet Protocol (IP) that defines addressing is located at the Network Layer, along with routing protocols to determine the optimal route between source and destination in a network. The Data Link layer handles most of the wireless LAN mechanisms, such as the IEEE 802.11 which defines rules for wireless communication.

2.3.2 Data Link layer

The support for multi-channel multi-radio can be added on the Data Link layer. This layer consists of the two sublayers Media Access Control (MAC) and Logical Link Layer (LLC). The latter is sometimes referred to as layer “2.5”. The MAC layer controls the access to the physical medium. It allows for multiple users of the same medium by providing rules for how and when a user is allowed to access the medium for transmissions. The LLC is an interface between the MAC and the overlying network layer. LLC provides for example flow-control and multiplexing [24]. Flow control means controlling the rate at which hosts inject packets into the network to avoid traffic congestion and multiplexing is the act of combining several different streams in such a way that they can be separated later on. Due to the closeness to the physical layer it is beneficial to modify the MAC to support multi-channel multi-radio, but there are also disadvantages. One advantage is that it is possible to modify how the nodes access the common medium and optimize it for a multi-channel multi-radio setup.

Solutions on higher layers must rely on existing MAC-protocols which might not have been designed with this aspect in mind. Modifying the MAC can also introduce problems, for example break compatibility with other common hardware. Some other benefits and drawbacks are discussed in the solutions presented below.

Extended Receiver Directed Transmission protocol Maheshwari et al. [9] presents two new MAC protocols for multichannel operations. The extended Receiver Directed Transmission protocol (xRDT) based on RDT and the Local Coordination-based Multichannel MAC (LMC MAC). In RDT every node has a "well known" channel which it tunes to when it doesn't have any data left to send. When a node has data to transmit, it tunes its interface to the well known frequency of the next hop node and transfers the data. When done, it switches back to its own idle-channel. To solve two problems that arise with the original RDT, xRDT implements two new mechanisms; one mechanism to eliminate the hidden terminal problem and one to resolve the deafness problem. The first problem in solved by implementing a single frequency busy tone. This is done by adding additional hardware, namely a tone interface. Each channel is assigned a specific frequency and when two nodes communicate over a specific channel the sender uses the tone generator to signal the channel is busy. This allows other nodes to know which channels are currently occupied. Deafness arise when a node wants to send data to an occupied node [9]. The node who wants to transmit, but can't, will try again after a specific time period. This back-off time increases exponentially in 802.11 and might result in that the node sleeps during the time the intended receiver rendezvous at the common channel. Before the back-off times out, the receiver becomes busy again and the sender misses its chance. This problem is addressed by using a "wake up" signal using the regular interface. When a node is done transmitting and changes back to its idle channel, it first sends out a "wake up signal" which enables nodes in back-off mode to activate their transmitter and try to transmit data. The idle channel for the nodes can either be static or the nodes can change channels dynamically based on, for example, channel load. I.e. the nodes continuously monitor which channels that are being used by its neighbors and if the node's current channel is used by several nodes in the neighborhood the node choose a new channel with fewer users.

Local Coordination-based Multichannel MAC While xRDT utilizes one packet and one tone interface, LMC MAC is designed for a single interface. To avoid network wide time synchronization, nodes in the network locally coordinate transmission schedules. The schedules consist of two phases. In the first phase the nodes exchange control data over a default channel with the goal to negotiate what channels to use in the second phase. The second phase is a data window where nodes concurrently transmit data over the previously negotiated channels. All nodes who doesn't already know a schedule can propose one during the control window. The proposal is included in a RTS message and consists of two proposed values; control window duration and data window length. It also contains a list of all free channels at the node. To keep track of which channels are being used, LMC MAC extends the normal 802.11 NAV to handle multiple channels by making it a vector with one element for each available channel. The receiving node chooses an available channel and replies with a CTS and then the original initiator responds with a RES packet either confirming the channel or denying. Other nodes overhearing the negotiation mark the channel busy and then start to follow the proposed schedule and negotiate channels. When the control window comes to an end, nodes switch to the negotiated channels and transmit data. Nodes overhearing the schedule but for some reason couldn't negotiate a channel will remain silent until the end of the data window. It is possible to improve efficiency by letting nodes dynamically negotiate the duration of the data window depending of how much data each node have to send.

The paper [9] only briefly mentions how to solve broadcast with the two protocols. LCM MAC can implement it during the control window, while the solution for xRDT in its current form is to send it out on all channels. The authors expects xRDT to be more or less unaffected by mobility thanks to the busy tone and that LCM MAC may have some problems because of its large data window size. Hence, xRDT is heavily affected by the interface's channel switching time and LCM MAC nodes can suffer starvation when a node wants to send data to a receiver ruled by another schedule.

Hybrid Multi-Channel Protocol Kyasanur et al. have proposed a data link layer protocol for multi-channel multi-radio support called Hybrid Multi-Channel Protocol (HMCP) [12]. In [4] the same group of researchers presents "A Hybrid Interface Assignment Strategy". This strategy is further developed in [3] and used in the HMCP protocol. This scheme eliminates the rendezvous problem and doesn't need synchronization between nodes.

The HMCP protocol does not work with only a single interface, it requires at least two radios. The radios are divided into two groups at each node, fixed interfaces and switchable interfaces - hence the name hybrid. The first group is fixed on specified channels permanently or "for a longer period of time", while the second group can switch between channels frequently. A node must at least have one fixed interface to ensure connectivity between neighboring nodes. The channel used is either calculated for a specified node by using a known function with a node specific input or chosen randomly and then advertised through locally broadcasted HELLO-messages. The latter solution makes it possible for a node to switch its used channel on the "fixed interface" if it from a received HELLO-message discovers that the current channel is used by other nodes in the neighborhood. When a node has data to transmit to a neighboring node, it switches one of its switchable interfaces to the channel used by the neighbor's fixed interface.

To ensure that broadcast messages are delivered to all neighbors, each message is sent out over all available channels. Compared to a single channel solution, HMCP requires more packets to be transmitted but the utilization per channel is the same. One possible problem is that broadcasts arrive at the neighbor nodes at different points in time.

In [8], Li et al. presents HMCMP, an improved version of HMCP. In HMCP, the waiting time used to avoid the hidden terminal problem is static, while HMCMP uses a dynamic waiting time to increase utilization. The waiting time in HMCMP depends on the probability that a collision may occur. If the probability is low, the waiting time is short and data can be sent over the channel earlier, thus improving throughput. HMCMP also adapts the transmission time for each channel based on traffic load.

Multi-channel MAC In [5] the authors J. So and N. H. Vaidya propose a solution calledMulti-channel MAC (MMAC) . It is designed to work with the existing 802.11 MAC and consists of nodes equipped with a single radio. In MMAC, time is divided into beacon intervals. Every beacon starts with a so called ATIM window, when all nodes in the network are forced to listen (rendezvous) on a common channel. During this time slot, nodes negotiates channel to use for data exchange. When the ATIM window is over, nodes switch to the selected channel for data communication until the next beacon interval starts. This calls for clock synchronization which has been shown [16] to be non-trivial. Also all nodes must stay in the data window for a specific time which means they cannot utilize channel diversity to its full potential [9].

2.3.3 Network Layer

The Network layer is in charge of supplying each packet with a route through the network from its source to the destination. AODV [15], OLSR [10] and DSR [11] are examples of routing protocols performing this task in wireless ad-hoc networks. The routing protocol uses different metrics for choosing which route that currently is the best for each packet to use to reach it's destination. Some protocols use shortest path, e.g. the fewer number of hops the better. Other routing metrics rely on the link speed and quality. It is beneficial to have the routing protocol aware of the different characteristics introduced by a multi-channel multi-radio network [13]. For example using a metric based on current channel utilization and link quality. Some proposed routing protocols are not only modifying the metric for MCMR but are also responsible for assigning channels.

Below is an overview of a few different routing protocols meant to be used in an multi-channel multi-radio network.

Ad-hoc On-Demand Distance Vector Ad-hoc On-Demand Distance Vector (AODV) [15] is a reactive routing protocol, meaning it will only perform routing when needed. Although AODV supports multi radio multi channel, it is optimized for single-channel single-radio. AODV uses the shortest path metric to find a route between sender and receiver. This metric does not perform too well when deploying multi-channel multi-radio [13]. AODV-HM [6] and AODV-MR [7] are however two extensions to the AODV protocol, created to better suit a multi-radio environment.

AODV-Hybrid Mesh (AODV-HM) uses shortest path as metric but distinguishes between *mesh routers* and *mesh clients*. Mesh routers are nodes used in the networks infrastructure and are usually equipped with several wireless network interfaces. All other nodes are classified as clients. A route involving nodes classified as mesh routers are preferred over routes only involving clients. During route discovery, nodes with several interfaces have the possibility to choose which interface to be used by marking the route discovery packet before broadcasting it to its neighbors. The decision can be based on for example current interface load. The routing protocol itself does not handle channel assignment, it relies on preassigned channels. According to the developer, AODV-HM performs better then regular AODV in networks where mesh routers are equipped with more than one interface. In networks where routers only have a single interface, the preferential treatment of routes including mesh routers may degrade the performance compared to AODV.

AODV-Multi Radio(AODV-MR) is a very simple extension to AODV which basically only barely enables multi radio support by adding a field in the routing message header.Channel Assignment-AODV [17] is another solution to improve AODV, but with a combined routing and channel assignment approach. This protocol performs channel assignment during each route discovery by randomly picking an available channel from all possible channels in the network. Each node that receives the route discovery request chooses a free channel and appends it to the request. If there are several active transmissions in the network, channel conflicts are resolved in the route reply. The solution has low overhead due to using already existing messages for selecting channels and Gong and Midkiff [17] proves the correctness of the channel assignment algorithm mathematically. However, they do not mention how to solve broadcasts or other common channel switching problems. Therefore such functionality must be implemented at lower layers.

Optimized Link State Routing Protocol OLSR [10] is a proactive routing protocol, i.e. regularly updates the topology information based on control messages from neighboring nodes. Each node carefully selects a couple of neighbors as its "multipoint relays" (MPR) such that it can reach two-hop neighbors through these. By only letting the MPR forward control messages, the number of transmissions are reduced when flooding the network.

The Link Quality Optimized Link State Routing Protocol (LQ-OLSR) [14] is a table driven proactive protocol based on OLSR [10]. To preserve scalability it uses a non-cumulative link quality metric which is locally calculated and only deliver broadcasts to neighbors. Due to its proactive nature the protocol is suitable for mesh backbones which seldom go through changes. LQ-OLSR extends OLSR's link quality metric and modifies the algorithm for MPR selection to prefer nodes with high willingness to forward traffic - this can be based on for example local value of power level or traffic. Another modification is to the route selection process. It is modified to utilize the improved link quality metric to chose the intermediate node for traffic destined for a node in the two-hop neighborhood. For packets destined for nodes outside of this range, OLSR's original shortest path metric is used. Just like AODV-HM, LQ-OLSR currently doesn't handle channel assignment among neighbor nodes. The authors testbed uses three interfaces with static channel assignment and the current proposed implementation uses an attribute to indicate which interface to use. According to the authors this attribute can be extended to implement channel assignment.

Dynamic Source Routing Dynamic Source Routing (DSR) [11] is like AODV, a reactive routing protocol. It to, uses the shortest path metric, which might result in reduced throughput in an multi-channel multi-radio setup [11]. Kyasanur et al. propose a Multi Channel Routing metric (MCR) [3] which is similar to the DSR but does not use the shortest path metric. Instead, MCR combines the WCETT [13] link cost metric with the switching cost that occurs in multi-channel architectures. MCR is preferably used on top of the hybrid Link Layer protocol described in chapter 2.3.2.

3 Method

The chosen approach was based on Kyasanur's hybrid solution [12] which divides radios into two types: fixed and switched. A fixed interface is locked to a specific channel during a long time-span while a switchable has the possibility to frequently switch between channels. When a node has data to transmit it performs a look-up to find which fix channel the next hop neighbor uses and tunes one of its own switchable interfaces to that channel.

The implementation is versatile. It was implemented at the link layer without modification to the existing 802.11 MAC protocol. The solution hides the multiple interfaces to the higher layers and supports local broadcasts; hence existing protocols, such as address resolution (ARP) and routing (e.g. OLSR and AODV), functions without any modifications. It allows for predefined channels or dynamic channel assignments and there is no requirement for network wide synchronization for channel negotiations. It is also possible to alleviate the hidden terminal problem.

Some drawbacks with this solution are that it requires at least two interfaces and that broadcasts must be sent out over all channels to reach all neighbors. Hence a broadcast requires channel switches and more data to be transmitted compared to if broadcast could be solved by leaving only on one channel. Broadcasts may also arrive at the neighbor nodes at different points in time and this can cause troubles for routing protocols.

The main function - choosing which interface to send a specific frame on and on what channel - is implemented by adding a new mode to the existing bonding module for the Linux kernel. This bonding module enables one or more network interfaces to act as one with the purpose to provide for example redundancy or increased bandwidth. To support the channel selection for the fixed interface - which is based on channel usage in the two-hop neighborhood - a user space application was created. This application is in charge of choosing the fixed channel based on gathered neighborhood data and advertising its chosen channel to neighbors. It is also responsible for initial setup and configuration. All time critical operations are handled by the bonding driver while the user space application is in charge of less time critical features. In addition, optimizations were made to the open source WLAN driver MadWifi to minimize channel switching time and additional functions to improve performance.

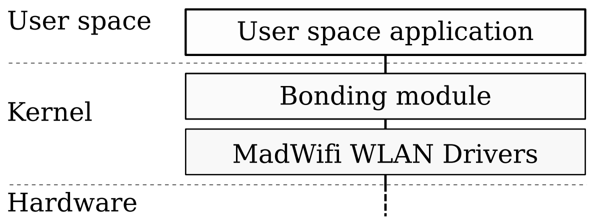


Figure 6: User- & kernel space

The rest of this chapter will present an overview of the provided functions and then describe the different parts in detail - starting with the Linux bonding driver, then the user space application and finally the modifications made to the WLAN driver.

3.1 Implementation overview

Each node has a minimum of two interfaces consisting of at least one fixed and one switchable. A fixed interface is used to receive data and to ensure network connectivity while a switchable is used to transmit data to the neighbors.

Each node keeps a neighbor-table which lists the fixed channels of its neighbors. This is used to look-up what channel to use when transmitting data to a specific neighbor. Information about neighbors is obtained through status notifications called HELLO-messages. Each node periodically broadcast these messages with information about its current fixed channels. Each message also contains information about the nodes current neighbors and their fixed channels. This information enables the nodes to keep track of the channels currently used in the two-hop neighborhood. With this knowledge it is possible for each node to try to optimize the local channel usage by dynamically choose which fix channels to use.

Each interface has a queue for each available channel. After a lookup in the neighbor table, packets are placed in the corresponding queue. Data supposed to be broadcasted is place in all queues. Queues are served in order. A queue is inspected before an actual frequency switch, if the queue is empty the next queue in turn will be inspected. If a queue has packets, the interface is switched to the corresponding frequency and the data transmitted. Normal behavior is to serve each queue until they are empty. It is however possible to define a maximum and a minimum staying time for each channel. By enabling these features, the switching overhead could be decreased and the end-to-end latency optimized.

To minimize the effect of the hidden terminal problem (see chapter 2.2) after changing channel, an interface sleeps for a specified period of time before trying to send data. This allows the Network Allocation Vector to be updated, hence avoiding collisions[3]. The upper limit for this period is the time it takes to transmit a maximum sized packet. It can be lowered by letting the node dynamically calculate the sleep-duration on the number of neighbors currently using the channel - if there are few users the sleep can be significantly lowered without affecting performance negatively[20]. The current implementation only allows a static duration.

3.2 Implementation details

3.2.1 Linux bonding module

The Linux kernel bonding driver makes it possible to aggregate multiple network interfaces and have them appear as a single logical interface [19]. The bonding driver operates at the link layer [23] above the interface device drivers. It offers several modes of operation such as different types of load balancing and hot standby. Multi-channel multi-radio support was implemented by the creation of a new mode.

This new mode bonds physical wireless interfaces together and present them as the logical interface “mcmr0”. The bonding module is in charge of all traffic received and transmitted through this interface. Each physical interface is either configured to be “fixed” or “switchable”. A fixed interface is only used to listen to incoming traffic while all outgoing traffic is disabled. A switchable interface has a reversed role, namely to send traffic, hence incoming traffic is disabled.

All incoming traffic to fixed interfaces is directly delivered without any modifications. The main function of the bonding mode is to handle outgoing traffic. When a packet arrives from higher layers it is first checked if it is supposed to be broadcasted. If so, the packet is copied and placed in all available queues. If it is a unicast packet a look-up in the registry is done to check if the node knows about the destination neighbor and what channel it is currently using. If the destination is found the packet is queued in the corresponding channel queue. If it is not found, the packet is dropped.

The registry consists of tuples with one-hop neighbor information. Each tuple is composed of the neighbors Ethernet address (MAC address) and the channel to use. The registry is maintained by the user space application via IOCTL-calls, which obtains the information by receiving HELLO-messages from neighbors. The registry maps each Ethernet address to one channel only. If one neighbor has multiple fixed interfaces it is up to the user space application to choose which specific channel to use. The decision was placed on the user space application to minimize the work required for each outgoing packet.

Servicing channel queues The channel queues are served by separate threads responsible for each of the switchable network interfaces. The current implementation has support for a maximum of one switchable interface, but can easily be extended to support multiple interfaces. The worker thread iterates over the available channel queues and examines if there is data waiting to be transmitted. If the current queue is empty it tries the next queue without doing an actual channel change in hardware. If a queue has data to transmit, the thread checks if the interface is currently tuned to the channel. If not, it request a hardware channel change via a standard wireless extension[4] IOCTL. Directly after a channel change the thread has the possibility to sleep for a specified interval before starting data transmission. This is done to minimize the impact of the hidden terminal problem, which can degrade network performance in a dense network. This sleep duration is currently set at compile-time.

Then the thread starts to dequeue frames from the queue and hands the responsibility for the frame over to the WLAN driver. If the driver reports an error, the frame is dropped. However, a successful handover to the driver doesn't necessarily mean the frame will be delivered to the destination. The frame can be dropped later on by for example the driver or by the hardware.

The normal behavior for the thread is to service the channel until it is empty. This is a simple solution and can cause starvation if the current queue is continuously filled with new data faster than the hardware can send. Therefore it is possible to set a maximum staying time on each channel. This feature is enabled and configured at compile time. If activated, after each sent frame the thread calculates the expected transmission time for the frame based on size. If the total expected transmission time exceeds the maximum staying time the thread is forced to stop dequeuing data from the current queue and check if another queue has data to send. If no other queue has data to be transmitted, no channel change will occur and the thread will continue to service the interrupted channel.

It is also possible to set a minimum staying time on each channel. When the hardware is changed to a specific frequency the thread is forced to stay on the channel for the specified time and continuously check for new packets even if the channel is empty. This can be enabled to minimize switching overhead by tweaking the total (switch-time / time on channel) ratio.

[4]`(http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Linux.Wireless.Extensions.html)`

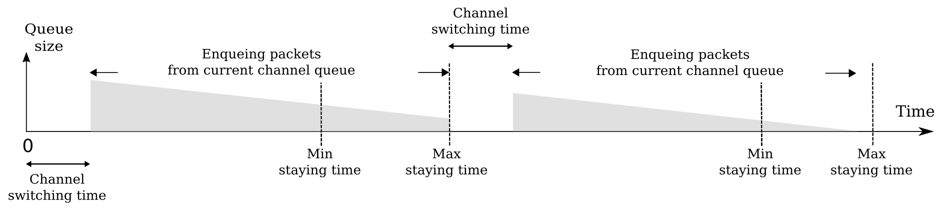


Figure 7: Staying time

When a channel switch is performed all data in the data queues are flushed by MadWifi. Therefore a function was added that enables the bonding module to query the driver if there are data left to transmit. This enables the bonding module to stall while the data is transmitted.

Throughput performance issues Problem with the performance arose when testing an early build of the system. With the MCMR-module enabled and only one channel activated to disable switching, the obtained throughput node-to-node was about 4 Mbit using UDP traffic. This compared to 28 Mbit with the MCMR-module disabled.

After extensive debugging the source of the throughput problem was identified to be caused by the MCMR-module in combination with MadWifi and the features of the 802.11 MAC. More specific, the acknowledgment-process of frames.

Normally after receiving a data frame, the receiver checks the frame for errors. If no errors are found, it sends an ACK-frame to the receiver to signal that the frame was successfully received. The ACK is addressed to the Ethernet address found in the 802.11 frame headers source field.

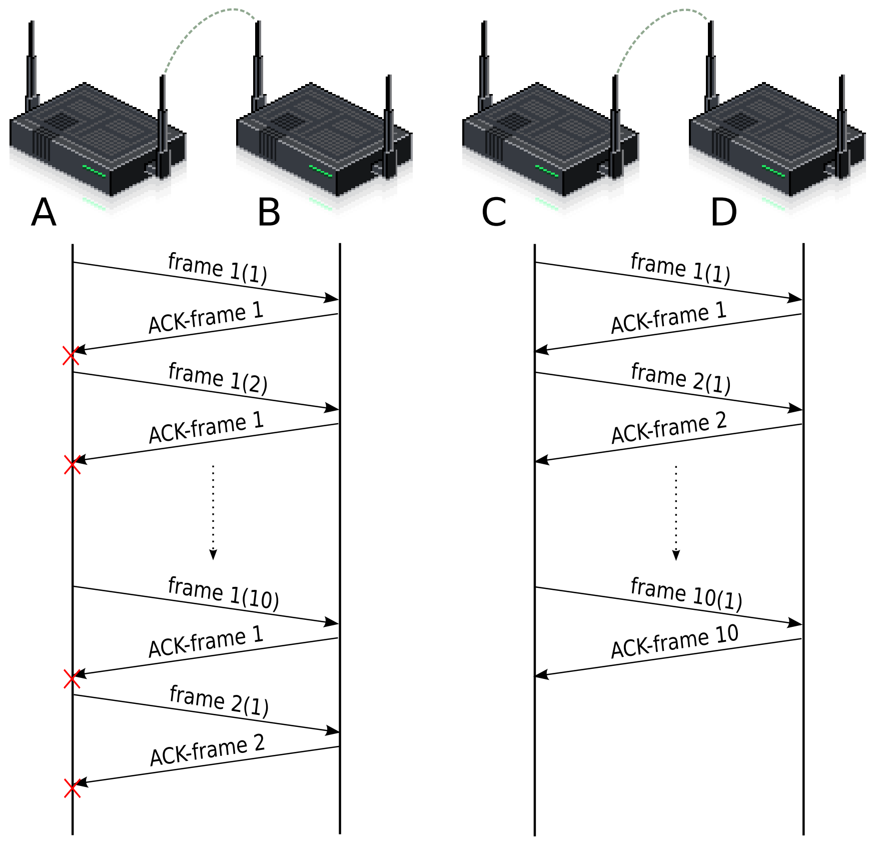


Figure 8: Faulty behavior to the left. Right shows a correct ACK-process.

If the sending node does not receive an ACK addressed to its Ethernet address before a timeout, it will resend the frame until it receives an ACK. This is done up to the specified number of times denoted by the driver or hardware. In Atheros case, the number of retries are permanently set in the hardware and can not be controlled by software or the driver.

The problem had to do with how the bonding module in combination with MadWifi handles Ethernet addresses. For each physical wireless card MadWifi creates a wireless device named “wifiX”, where X denotes the number of the card starting with zero. The first card will for example be named “wifi0”, the second “wifi1” and so on. It is then possible to create and bind virtual interfaces to these devices. The default naming scheme is “athX”, where X is the number of the device starting from zero. The first interface bound to each device clones the Ethernet address from its parent device. After the bonding module is loaded in MCMR-mode, interfaces that should belong to the “mcmr0" interface are enslaved. Enslaving is the process of associating sub interfaces with the bonding interface, i.e. letting the bond interface know which interfaces it is responsible for. The bonding module which needs an Ethernet address for the "mcmr0" device, clones the address from the first enslaved interface. Consecutive enslaved cards will then via wireless extension IOCTLs be assigned the same Ethernet address.

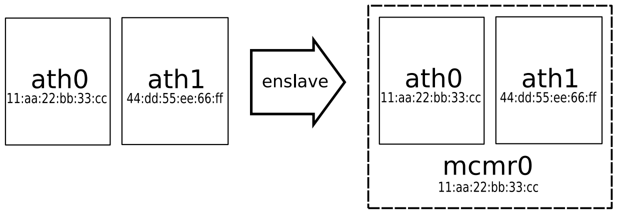


Figure 9: Enslaving

When a frame was sent out, the source address was automatically set by the system to the ethernet address of "mcmr0" and all of its slaves, not the permanent Ethernet address of the dynamic interface actually transmitting the frame. However, an Ethernet address set manually does not propagate correctly to the hardware and therefore the hardware does not listen to frame acknowledgments addressed to this Ethernet address.

This caused each frame to be resent by the hardware until it gave up and proceeded to the next frame. This severely reduced throughput. The chosen solution to solve the problem consists of letting the bonding module thread, which services the channels, set the source address of each frame to the switchable interface's permanent hardware address just before it handles over the responsibility of the frame to the driver, see Figure 10.

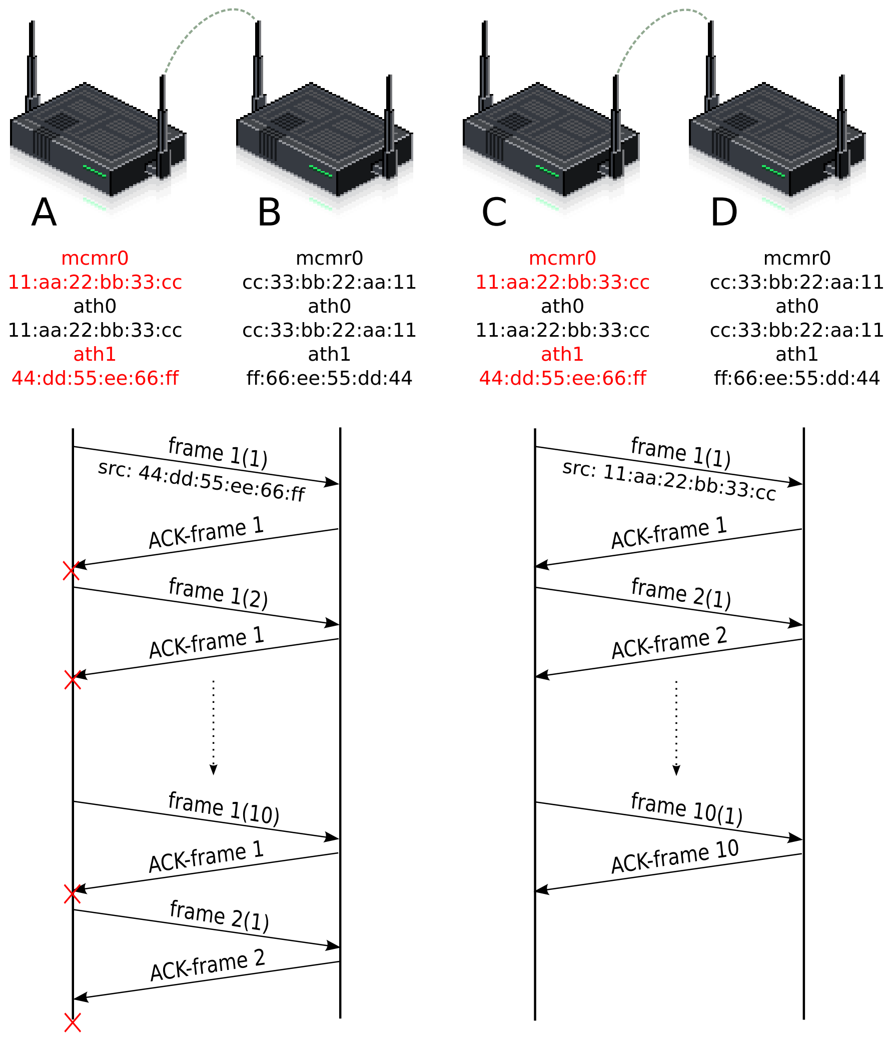


Figure 10: Faulty behavior to the left. Right shows a correct ACK-process.

3.2.2 User space application

The user space application is responsible for initiating and configuring the bonding module. It is also the one managing channel selection for fixed interfaces and handling neighbor information by sending and processing received HELLO-messages. The application was implemented as a module in the existing Mesh-framework created by CRL Sweden[5]. It is also possible to convert it to a standalone application.

[5]`http://www.crlsweden.com/`

The implementation is capable of handling an arbitrary number of bonding interfaces (viz. mcmr0, mcmr1 and so on) each with an arbitrary number of enslaved fixed and switchable interfaces.

During upstart the application reads a configuration file containing all the configurable parameters needed to init the application and the bonding module.

Configuration file example:

```
#Virtual MCMR-interface
MCMRInterface = "mcmr0"
#Enslaved fixed interface
FixedInterface = "ath0"
#Enslaved switchable interface
DynamicInterface = "ath1"
#Interval beween outgoing HELLOs
HelloInterval = 5000
#Port to listen for HELLOs on
ListenPort = 55000
#When to classify a neighbour as old
NeighbourEntryExpire = 11000
#Time between checks for stale entries in the neighbour table
NeighbourExpireCheck = 3000
# Channels not to use
ExcludedChannels = 36,44,48,52,56,64,104,108,112
#Frequency range:  A or G
FrequencyRange = A
```

The application then activates each interface in the bonding module and sets them to the correct mode. This is done by IOCTL calls to the module. It also enables all channels for each card except the ones denoted by the parameter “ExcludedChannels”. In a network with few nodes it is beneficiary to only use a few channels, because this minimizes the number of channels broadcasts must leave on, hence lowering the number of required channel switches. It is important that all nodes in the network have the same set of active channels or network partitions could occur.

Even if it would be possible for the nodes to use the frequencies defined by both 802.11a and 802.11g this is not allowed by the application due to the increased switching cost involved.

A fixed channel is chosen by random from the by hardware supported channels for each fixed interface and assigned via a regular wireless extension IOCTL. After initiation the application starts to send and listen for HELLO-messages.

HELLO-messages

Once during the specified interval the node broadcasts a HELLO-message. The message will in other words be transmitted on all allowed channels. The purpose of this message is to inform neighbors about its presence and what channel or if it has multiple fixed interfaces - channels, the neighbors must use to transmit data to the node. The data consists of each cards permanent Ethernet address and its currently used channel. If the node itself has received HELLO-messages from other nodes it includes data about these one-hop neighbors. This data is comprised of the nodes IP address and Ethernet addresses of each of the fixed interfaces plus the associated channels. With this information it is possible for the receiver to correctly calculate the channel usage in the two-hop neighborhood. The Ethernet address is needed by the receiver to identify if it already knows about the other node as a one-hop or possibly two-hop neighbor.

The HELLO-message interval affects the networks mobility. In a static network it can be sufficient to send out messages for example once every 30 seconds or even less. In a mobile network with nodes moving around plus leaving and entering the network frequently the interval must be decreased to not leave invalid entries in the nodes' neighbor tables.

Before each HELLO-message the node consult the channel usage list to check if it is beneficiary to change one of its fix channels. If one of the current channels is used by more nodes than the node itself, it checks if there are any unused channels or a channel with a lower usage that would be beneficiary to change to. If there is such a channel there is a 50% chance of the node actually changing channel. This is to avoid a state where multiple nodes frequently switches between channels.

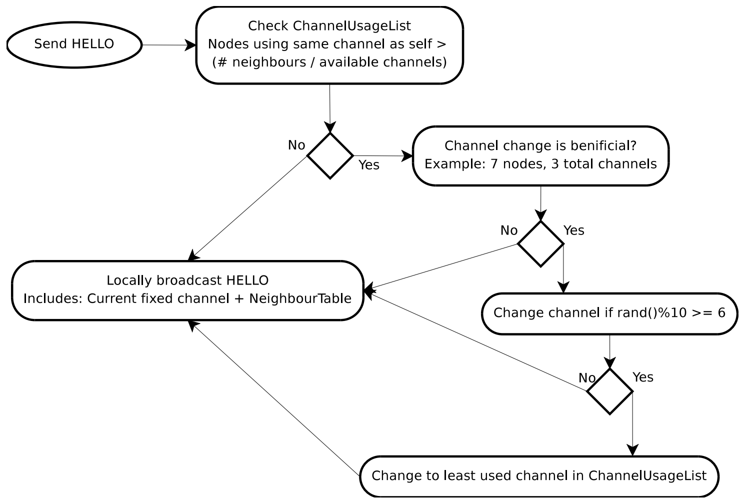

Send HELLO
Check ChannelUsageList
Nodes using same channel as self >
(# neighbours / available channels)
No
Yes
Channel change is benificial?
Example: 7 nodes, 3 total channels
No
Yes
Locally broadcast HELLO
Includes: Current fixed channel + NeighbourTable
Change channel if rand()%10 >= 6
No
Yes
Change to least used channel in ChannelUsageList

Figure 11: Send HELLO-message

Neighbor- and channel management

After receiving a HELLO-message the information is analyzed. The sending node of the message is a one-hop neighbor to the receiver and can be reached directly by the node. The neighbors Ethernet addresses and each corresponding channel is recorded in the nodes neighbor table along with a last-seen time stamp. If the node already exists as a two-hop node, it is promoted to one-hop. Each one-hop tuple is then added to the bonding module via IOCTL as an unicast entry. The channels used is also added to the channel usage list. The purpose of this list is to record all channels used and their usage count in the two-hop neighborhood. If a node changes channel the usage of the old channel is decreased by one and the new channel's usage increased.

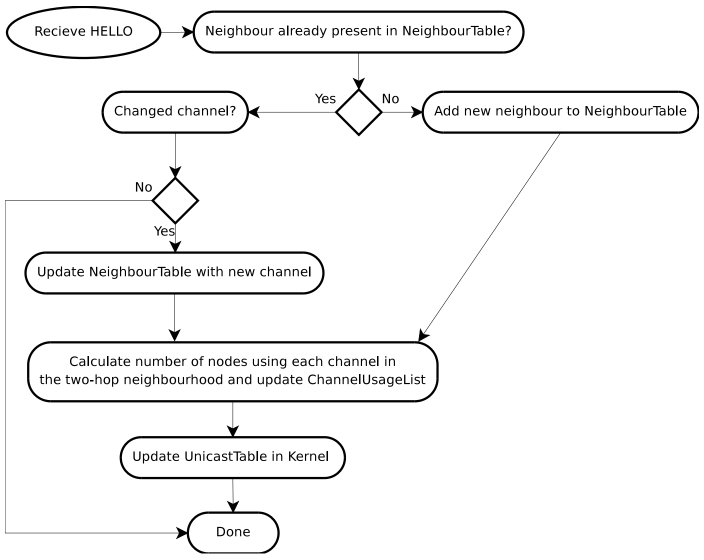

Recieve HELLO
Neighbour already present in NeighbourTable?
Yes
No
Changed channel?
Add new neighbour to NeighbourTable
No
Yes
Update NeighbourTable with new channel
Calculate number of nodes using each channel in the two-hop neighbourhood and update ChannelUsageList
Update UnicastTable in Kernel
Done

Figure 12: Receive HELLO-message

The nodes in the message data - containing the senders one-hop neighbors - are all possible two-hop neighbors to the receiving node, but they can also be one-hop neighbors. Each entry is compared to the entries in the neighbor table. If the node exists and is already known of as a one-hop neighbor the data is disregarded and the last-seen time stamp is not updated. If the time stamp was updated, it could cause none existent nodes to be kept alive in the network as two-hop. If the node exists as a two-hop neighbor the information is updated. If it does not exist, and the channels is supported, it is added as a two-hop neighbor. These neighbors are not added to the bonding module, they are only used for channel usage statistics. The channel usage list is updated accordingly.

Regular checks are done for stale entries in the neighbor table. If a one-hop or a two-hop neighbor has not been heard of for the specified time, it is classified as stale and removed from the neighbor table. One-hop neighbors is also removed from the bonding module. In a highly mobile network the value "NeighborEntryExpire" must be low to not keep invalid nodes in the neighbor table.

3.2.3 MadWifi WLAN driver optimizations

The bonding module and user space application works with all hardware drivers with support for IBSS. However, due to the frequent channel switches the performance can suffer due to hardware and driver taking long time to switch between channels.

Measurements showed that a recent development SVN version of MadWifi had a switching time of about 8-9 ms measured from the time the driver got the IOCTL until the IOCTL returned. The stable 0.9.4 version showed better timings of around 5 ms.

MadWifi version	Average switching time (seconds)
0.9.4	0.004959
SVN rev 3517	0.008580

The higher switching times for the newer SVN version is probably due to code additions added to comply with the IEEE 802.11h-2003 standard [20]. This standard provides Dynamic Frequency Selection (DFS) which mean an access point should avoid channels which is used by satellites and radars. Because of this and the fact that the version has proven to be stable on the Avila platform, the 0.9.4 version was chosen to be optimized.

MadWifi was studied to identify what happened from the moment the driver got the change channel IOCTL until the change was done. Each important function was timed and functions going down to the hardware via HAL were identified. HAL is short for "Hardware Abstraction Layer" and provides an interface which MadWifi must use to control the actual hardware. This is to make sure MadWifi does not change values in hardware registers which it is not supposed to change, e.g. use illicit frequencies or too high transmission powers.

When a call is made to change channel, MadWifi does the following important steps:

First it disables interrupts to avoid being interrupted by the device. Then it stops all outgoing transmissions and drains the current pending transmission queue by dropping all unsent queued packets by calling "ath_draintxq". After this "ath_stoprecv" is called. This function instructs the hardware to stop packet reception and tells above layers to queue pending outgoing packets. Then the hardware is reset, this is where the actual frequency change occurs. After the reset a few things such as transmission power limits, channel flags and, if it is a change between modes, hardware rate maps are updated. Then packet transmissions and reception are enabled. A few of these functions was possible to optimize or completely remove to lower the total switching time.

Some example functions and their approximated execution time:

Function	Exec (sec)	Comment
ath_hal_intrset	0.000008	Disable/enable interrupts
ath_draintxq	0.000063	Disables packet transmission and drain pending out-queue
ath_stoprecv	0.003025	Disables packet reception by disabling PCU and DMA engine
ath_hal_reset	0.000650	Hardware reset
ath_update_txpow	0.000315	Update transmission power limits after channel change
ath_startrecv	0.000195	Start reception of packets

Two important functions are "ath_hal_reset" and "ath_startrecv". The first function is a call to HAL resetting the hardware and doing the actual channel change. All operations is out of MadWifi's control. Hence, not possible to optimize. The second function initiates reception buffers and the like to enable packet reception after a reset. It consists mostly of calls to the HAL and therefore not either possible to optimize without having access to the HAL-code and hardware. Based on this, a hard theoretical minimum value with current hardware and HAL is around 0.000845 seconds (0.845 ms).

The function responsible for over 0.003 seconds during a change is "ath_stoprecv". This function disables packet reception by disabling Packet Control Unit (PCU) and the Direct Memory Access (DMA engine). After disabling DMA it sleeps for 0.003 seconds to let a possibly present DMA-operation finnish. This sleep was removed during optimizations along with functions deemed unnecessary for the operation of MCMR, for example all functions which updates channel flags, transmission power limits and the hardware rate map when changing between nodes. Also the state machine was eliminated. After optimizations the channel change time was on average 0.000997 seconds.

A lot of the optimizations performed were MCMR-specific and broke normal behavior of the driver. Therefore, it was made possible to turn on and off optimizations when loading the MadWifi module.

Another optimization was to disable the 802.11 beacons on the interfaces. In an ad-hoc network beacon frames are normally broadcasted to inform nodes in the vicinity about the nodes service set identifier, supported rates and other parameters. This information is needed to setup communication between nodes. Due to the characteristics of the MCMR setup, beacons are not needed and only cause unnecessary overhead.

Pending out-queue

Any packets waiting to be transmitted in the drivers out-queue are dropped when a channel change is ordered. To enable the bonding module to query the driver for the queues state, the existing function in MadWifi which returns wireless statistics was modified. This function is called by a wireless extension

IOCTL and a variable - otherwise set by MadWifi to zero - was modified to return the number of bytes left in the queue. This information makes it possible for the bonding module to calculate an expected transmission time and stall before issuing a channel change. Allowing packets left in the out-queue to be transmitted instead of discarded.

4 Results

This sections presents the results from the performance tests made with the intention to compare the usual single-channel, single-radio setup to MCMR under fair conditions.

When running MCMR, each node is equipped with two interfaces operating in 802.11a mode. The single interface setup is using the same hardware and the identical settings. Channel 64, 100 and 120 was mainly used when testing to make sure the channels did not interfere with each other. The traffic was generated and measured using Iperf[6] on standard desktop computers to not put any more load than necessary on the Avila Network Platforms.

To test the advantage of using multiple channels compared to a single channel, the first scenario consists of a total of four nodes. The four nodes are neighbors, i.e they all hear each other. Two of the nodes are responsible for transmission of data and two acting as receivers. The two sending nodes simultaneously generate traffic to one receiving node each.

Simultaneously transmissions on the same frequency are bound to interfere, resulting in a decreased throughput. Hence, utilizing two non-interfering channels should increase throughput.

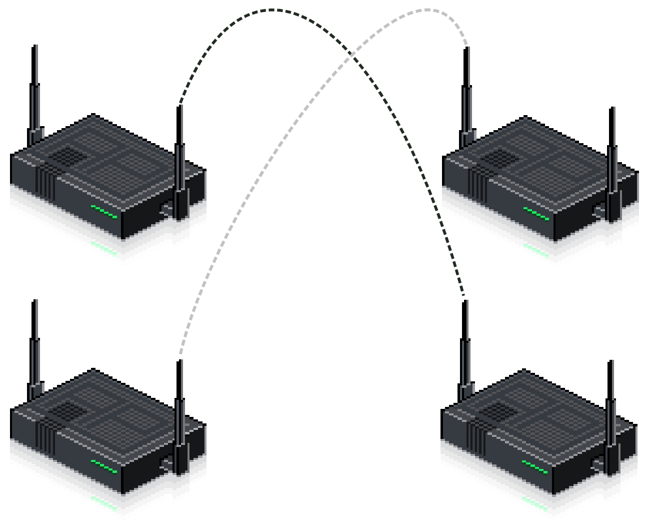


Figure 13: Scenario 1, one-hop, two channels scenario

The first test, Figure 13, confirmed that two separate channels can be used simultaneously without any interference. Transferring data between two nodes,

[6]http://sourceforge.net/projects/iperf/

four nodes in total, using MCMR resulted in doubling the throughput, as shown in Figure 14.

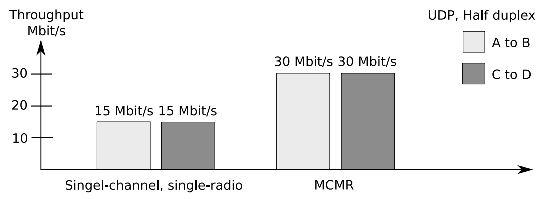


Figure 14: Scenario 1, throughput

When only using one channel, all transmissions have to share the channel bandwidth. In this case the bandwidth is 30 Mbit/s, resulting in 15 Mbit/s on each transmission. When increasing the number of simultaneously transmissions between nodes in a single-channel neighborhood, the bandwidth will decrease according to a simple formula:

$$throughput = \frac{maximum\,channel\,bandwidth}{number\,of\,transmissions}$$

If instead using multiple channels, the throughput will be equal to the maximum channel bandwidth as long as the number of available non-interfering channels are equal or more than the number of simultaneously ongoing transmissions. That is, each transmission will be able to use a unique channel:

$$throughput = \frac{maximum\,channel\,bandwidth}{number\,of\,transmissions}\,number\,of\,used\,channels$$

The second scenario is a simple setup to test the end-to-end one way (half duplex) throughput using one, two and three hops. Transmitting data between sender and receiver using two nodes takes one hop, while four nodes enables a three hop scenario. Since data will only be sent in one direction, no channel switching will be required other than when broadcasts are sent.

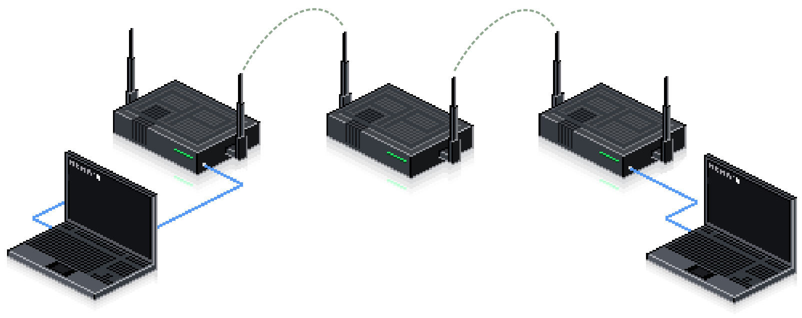


Figure 15: Scenario 2, multi-hop

This test showed that both single-channel, single-radio and MCMR reached the practical limit of 33 Mbit/s over one hop. However, when sending data over two hops, MCMR was able to increase the throughput by 60%.

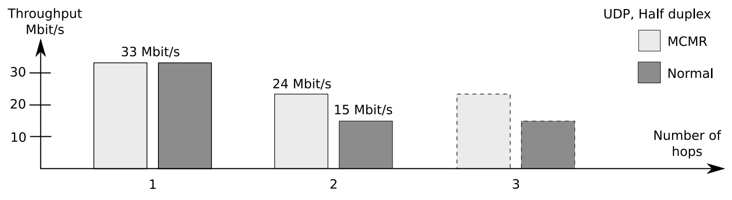


Figure 16: Scenario 2, performance

Even though MCMR uses different channels, the throughput will decrease due to high processor load on the nodes both receiving and transmitting data. A more powerful platform would probably be able to increase throughput.

In a perfect setup where each node only has two neighbors and the distance is enough to avoid two-hop interference, the single-channel single-radio will be able to keep the 15 Mbit/s over 3 hops and more. Performing fair tests on more than two hops, however requires the ability to strictly control the range of each node or a very large testing site, which was not possible within the scope of this report. A few tests using a MCMR setup and the user space application iptables[7] to block certain nodes from hearing each other was however performed and showed that the throughput reached about 24 Mbit/s over 3 hops. This shows that the MCMR solution is capable of improving the bandwidth over multiple hops compared to a single-channel single-radio setup. Even if the measured 24 Mbit/s is significantly higher than 15 Mbit/s obtained with the single-channel single-radio setup, it is far from the 33 Mbit/s measured in previous tests for MCMR over one-hop. The main reason for this is the increased channel switching required in the three-hop setup.

[7] http://www.netfilter.org/

5 Discussion

5.1 Implementation

Implementing the multi-channel multi-radio solution in practice was not without issues. Development of the user space application and the creation of the new bonding module mode were quite straight forward. The problems encountered were mostly quickly solved. A far greater problem was MadWifi and its numerous issues already present in the driver. Issues such as memory leaks which may not be notable in a normal environment, but certainly noticeable when brought to the extreme by the frequent channel switching.

The choice to implement a solution without network wide synchronization or a common shared channel resulted in a solution requiring frequent channel switching. An initial concern was that the switches would introduce an unacceptable latency to the network. In early test builds without any modifications to the WLAN driver this was also the case. However, with an optimized driver the obtained test results show that it is possible to in practice yield an increase in throughput while using a multi-channel multi-radio solution without introducing unacceptable delays due to channel switching.

5.1.1 Bonding driver

The central part of the implementation is the bonding driver mode. It is responsible for the different channel queues and the operation of the switchable interfaces. The choice to utilize much of the already present features of the bonding driver was inspired by Kyasanur [4, 12] and has been successful. It saved a lot of development resources and have created a versatile solution easy to modify and extend. For example CRL Sweden has extended the MCMR mode to handle a network consisting of nodes with directional antennas - a network configuration the original mode was not intended to handle. The current implementation only supports a total of two cards - one fixed and one switchable. Support for more cards, fixed and switchable, is possible to implement without making extensive modifications. For example, to add support for more switchable interfaces a thread for each interface must be created and rules for how the threads are allowed to service the common packet-queues must be added. By adding support for more cards it is possible to adjust the ratio of fixed and switchable interfaces. A node must have at least one of each type, but depending on the nodes role in the network it can be beneficial to have for example three of four cards to be fixed. This particular setup could be used on a gateway node which mostly receives incoming traffic and does not have much outgoing.

One problem in multi channel networks is how to make sure local broadcasts reach all neighbors. To fully utilize multi channel support all nodes should use frequencies which do not overlap or interfere. This means nodes can only hear messages on the specific channel they are currently tuned to. To solve this all broadcast messages are send out on all channels. This unfortunately introduces both channel switching and data overhead. The switching overhead grows with the number of channels available and it is therefore advantageous to disable channels in a network with few nodes or in a network where the nodes' coverage do not overlap. The current implementation lets the user space application enable the channels that should be used during start-up. Once a channel is activated it is not possible to deactivate it during runtime. An advanced solution could be to implement dynamic enabling and disabling based on the number of nodes in the network. Because local broadcasts must reach all nodes in the neighborhood to not introduce network partitions, all nodes must have the same channels available. This means a dynamic configuration of available channels must be done by a distributed vote among all the nodes in the network.

The bonding module allows for a lot of performance fine-tuning. There are a lot of different parameters that can be tuned to optimize general network performance or to suit specific network- and traffic characteristics. The most important part of the bonding driver is the thread or threads servicing the packet queues. The current implementation services the queues in order. But it is possible to modify this behavior and use a more advanced queue strategy. For example is it possible to prioritize the queue with the oldest data or the queue with the most data. Another parameter to optimize is how long a thread services a specific queue. By modifying the minimum staying time for a thread on a queue it is possible to lower the amounts of channel switches and let the interface spend more time servicing queues. The higher the value is, the fewer the channel switches. However, a too high value might affect the latency in the network negatively because data in other queues will have to wait longer before the thread can service them. A high minimum staying time is more beneficial on a system where the time it takes to perform a channel switch is high. On a system with low switching time it might not be beneficial to stay and wait for more data and better to switch directly when the queue is empty.

Another parameter related to the minimum staying time is the maximum staying time on a channel. If a thread is only allowed to switch channel when the current queue is empty, it can cause starvation of the other queues. By adding a maximum duration a thread can service a channel each time, the thread is forced to check the other queues and service them if necessary. A low value will increase the time spent switching but might also decrease the networks latency. An advanced solution would be to dynamically modify the minimum- and maximum staying time based on the networks current traffic characteristics.

Another implementation detail is how to handle the hidden terminal problem. In a dense network with many nodes switching channels this could potentially be a large problem degrading network throughput. The current implementation makes it possible to statically assign a duration each node must wait after a channel switch before trying to send data. In a dense network with many active nodes trying to transfer data the optimal waiting time approaches the maximum transmission time of a maximum sized packet. In a not so dense network it is beneficial to have a low waiting time due to the risk of multiple nodes transmitting simultaneously is decreased. Once again, an optimal solution would be a dynamic solution. For example it would be possible to calculate the waiting time based on known one-hop neighbors. This calculation would not be time critical and therefore beneficial to perform in the user space application.

During the process of solving the initial performance problems described in detail in chapter 3.2.1 the bonding driver and user space application were tested with Intel based cards and the Intel PRO/Wireless 2200BG driver for Linux[8]. This driver behaves different when creating interfaces compared to MadWifi. While MadWifi creates a "wifiX" card representing the physical device and then bind virtual interfaces to these, e.g. "ath0", the Intel driver directly creates an "ath0" interface. After some initial problems getting the Intel drivers to run correctly on the Avila platform they worked without complications in combination with the bonding driver. Although no measurements were performed on channel switching time, throughput were as expected over one-hop and broadcast messages was correctly transmitted on all channels. This showed that the implementation is compatible with different hardware and drivers. The Intel driver unfortunately uses a proprietary firmware which makes it impossible to optimize channel switching.

5.1.2 User space application

All non-time critical functions are placed at user space. Most of the features could also be integrated in the bonding module. This is however not recommended. Common programming practice is to place only the necessary features in kernel space, e.g. time critical and features needing access to system resources such as network interfaces. Developing code at user space is also more convenient, e.g. memory management is easier and the program can crash without bringing down the whole system. The latter is often the consequence if a module crashes due to the module sharing the kernels code space [22].

The application keeps track of the neighboring nodes and updates the bonding driver with information about which channels a one-hop neighbor can be reached on. Each node spread information about its own channels and its known neighbors by regularly broadcasting HELLO-messages. These messages introduce an overhead which grows with the number of neighbors and the lower the interval is set. The minimum size of a HELLO-message for a node with one fixed interface and no neighbors are 17 bytes. Then the size grows 11 bytes for each neighbor added with one fix channel. The size of the message header for each node is 6 bytes and each channel, i.e. each fixed interface at the node, takes an additional 5 bytes. This doesn't take much of the available bandwidth even if HELLOs are sent several times per second to suit a mobile network. The broadcast nature of HELLOs requires that they must be delivered on all available channels.

[8] `http://ipw2200.sourceforge.net/`

This means the total data that will have to be transmitted for each HELLO is: (size of HELLO) * (number of channels). The channel switching overhead must also be taken into account. However, we believe the lack of a demand for a network wide synchronization, which has been proven difficult [16, 9] or a common dedicated channel outweigh the overhead introduced. It is possible to minimize size of the HELLO-messages. For example by implementing a new message structure capable of compressing addresses. Another possibility is to combine the messages with existing routing messages. This would lower the total number of broadcast messages in the network and hence lower switching cost.

The information received from neighbors' HELLO-messages is not only used to get information about how to reach the neighbors, but also to gather information about the two-hop neighborhood. This information is used to calculate which channels to use for the nodes fix interfaces. Each node strives to minimize the total usage number for each channel. No investigation on algorithms choosing the optimal channel was done. For example, if two channels shares the same usage as well as has the lowest usage the node randomly choose which to use. This choice could be modified and based for example on which neighbors that are using the channels. Actual channel utilization could also be measured and used as a basis for the channel choice, e.g. prefer the one with lowest usage. It would also be possible to let the routing protocol choose channels. There are a lot of proposed modifications to existing routing protocols, a few described in the analysis. One benefits with letting the routing protocol choose channels is that it has extended information about the network and can setup both routes and channels.

There are measurements in place to avoid too frequent channel switches on fixed interfaces and to avoid a state in the network where several nodes constantly switches between channels. When a node decides it would be beneficial to change channel, a random function decides if to change channel or not. This is based on a percentage, the current implementation uses 50% chance of changing channel and has worked well in the test setup. This value could however need some modifications in a real network to avoid channel hysteresis.

5.1.3 WLAN driver

MadWifi was the open source driver of choice. It supports a range of Atheros chip-sets often found on common and cheap wireless cards. It proved to be the single area responsible for most troubles during development. The first problem encountered had to do with frame acknowledgments when running in MCMR-mode and is described in more detail in the method. This was an issue not present in for example the Intel wireless driver. The current solution is working and no bad side effects has been identified. The issue could probably be solved in the driver if it had direct access to the hardware and wasn't required to communicate via the hardware abstraction layer.

Later in the development and during testing other issues were encountered. Channel switching is normally something done “once in a while”. In a standard setup the only time channel switching is somewhat frequent is when the card performs a scan. If not user triggered, scans are normally not performed very often. A background scan in MadWifi with the purpose of finding a better AP when in roaming mode will be done once every 5 minutes. WLAN drivers and hardware are therefore normally not optimized to perform fast channel switches. An unmodified version of MadWifi takes from 5 ms up to about 10 ms per channel switch. This duration is too long to perform well when channel switches can occur up to a 100 times a second . Therefore MadWifi was modified to better support frequent channel switches. By identifying and removing steps in the channel change process that is not necessary during MCMR-operation, it was possible to bring the delay down to about 1 ms. It is in theory possible to lower this even further by removing more steps in the driver. However, MadWifi’s only method of telling the hardware to change channel is to request a reset. Measurements shows that this reset takes about 0.650 ms with current hardware. After the reset there are a few calls steps to setup the hardware. These add about 0.100-0.200 ms to the switch time. This shows that it would be beneficial to create hardware with faster channel switching support.

Working with MadWifi has not been trouble free. The driver consists of a lot of code, largely undocumented, which resulted in that the process of optimizing were often trial-and-error. The first version optimized were the stable release 0.9.4. During testing this version had large amounts of memory leaks causing nodes to crash or behave suboptimal. Changes made to the driver could not have introduced such behavior. Instead it is more likely that the frequent channel switching made an existing problem much worse. A study of existing bug tickets for MadWifi showed that other users experienced problems. A lot of leaks seemed to have to do with the management of Linux network buffers and the drivers node references to neighbors being lost during different scenarios. By testing different SVN versions of the driver the problem were alleviated but not fully removed.

The current implementation is based on a SVN revision 3226 from early 2008. Memory leaks still exists but mostly shows during high load on nodes which perform frequent switching. Newer SVN versions were tested but these introduced problems with channel switching, for example not being able to switch while the card was in UP state.

The memory leaks has proved troublesome during testing and must be fixed if the setup would be used outside of a test environment. It is possible that the reversed engineered successor to MadWifi, ath5k might solve a lot of problems when it is ready for deployment. It is at the moment not an option due to being early in development and missing a key set of features.

All data queued by MadWifi but not yet transmitted on the current channel is purged when a channel switch is ordered. This has the possibility to cause unnecessary packet drops. The features added to let the bonding module stall if there are data left proved to be not all that successful. If there are data left to be transmitted the bonding driver has the possibility to calculate an expected transmission time and then stall before telling the driver to change channel. However, due to these queues being in MadWifi, i.e. in software and not the hardware, it is not possible for the bonding driver to do a blockable sleep while waiting for the data to be processed. A blockable sleep means that no other code, such as MadWifi, can be run on the processor. This would however have been possible if the frames were queued in the hardware. Quite often the calculated stall duration proved to be too small in relation to the minimum sleep interval provided by methods to yield the processor to other processes and therefore not beneficial. The minimum scheduling time for a process is 1 jiffie, which in Linux 2.6.19 with default settings converts to 10 milliseconds. Due to this the current implementation of the bonding driver has the feature disabled.

5.2 Limitations

The chosen approach requires at least two wireless cards. There are other multi channel approaches which gain performance improvements by using one card, see chapter 2.3. However these often require modified MAC protocol which was not an option.

The chosen approach also requires low channel switching times to provide acceptable performance. If the driver has poor channel switch characteristics it must be optimized. This is only possible with access to the drivers source code, something which is seldom provided. This limits the choice of WLAN drivers and hence available hardware.

The current implementation also has limitations. For example the bonding driver does not allow more than two cards in total. This support is however already present in the user space application. The current implementation also lacks for support single card nodes in the network. In a typical network setup MCMR nodes would form the high bandwidth backbone and single card nodes would connect to a regular access point that in turn is connected to the MCMR nodes. It would however be possible to connect single card nodes directly to the MCMR network by modifying the user space application and bonding driver. If the fixed interface is capable of both receiving and sending traffic and by enabling the fixed interface to send out beacons to alert single card nodes of its presence, then single card nodes could connect to the fixed interface just like a regular access point.

5.3 Future

The solution/implementation was created to work on the Gateworks Avila 2348-2 Network Platform[9] with support for only two WLAN cards and a 266MHz

[9]`http://www.gateworks.com/products/avila/gw2348-2.php`

CPU. Hence, more powerful platforms with more support could result in a different approach. Gateworks top of the line network platform, the Cambria GW2358-4[10] supports four WLAN cards and runs a 667MHz CPU. If having the possibility to utilize four WLAN cards, three of the cards could for example adapt to the traffic load and help out where ever necessary. Hence, switching between sending and receiving data. Only one of these three cards would have to be responsible for handling control data like HELLO-messages. This would allow the other cards to concentrate on processing data on one channel and not frequently having to change each time the HELLO-messages are to be sent.

Besides the evolution in hardware, more wireless standards are developed. One of the most ground breaking is the IEEE 802.11n [26] standard which improves the wireless technology in both range and throughput. Improvements have been made in the physical layer as well as in the MAC layer compared to IEEE 802.11a/b/g. These changes enable the theoretical throughput to increase from 54 Mbit/s in 802.11a/b/g to 600 Mbit/s when applying 802.11n. Equip a Cambria Network Platform with four 802.11n WLAN cards and the possibilities - in case of performance - are endless.

Most important is however further optimizations to the bonding driver and the wireless drivers. Primarily a more dynamic queue handling with the ability to service queues depending on different conditions and variables.

[10]`http://www.gateworks.com/products/cambria/gw2358-4.php`

6 Conclusion

In this report, we have investigated different approaches and the possibilities to implement multi-channel, multi-radio to increase performance in wireless multi-hop mesh networks. After reviewing a set of proposed solutions, a hybrid method was chosen. This approach had the most advantages compared to its competitors, but was still not trivial to put into effect. The implementation in user-space was fairly straight forward as opposed to adding support in the operating system and wireless drivers. Not only did it have to work, but optimizations had to be done to get a substantial increase in performance compared to a ordinary single-channel, single-radio setup.

When creating a chain of nodes, each node has to be in range of two neighboring nodes - one on each side - for the chain to work. If the nodes uses only a single channel, each intermediate node has to receive and transmit on the same channel causing an interference that will halve the maximum throughput. When applying MCMR, the intermediate node can make use of the advantages with non interfering channels by using two different channels simultaneously.

Consider a wireless network consisting of several nodes - utilizing a single channel - in for example a wide city network. To avoid too much interference in such an environment, the nodes placement would have to be well planned. If the nodes instead deployed MCMR, interference wouldn't be such a big problem due to the implemented intelligence. Furthermore, in a close environment where all nodes are in range of each other, a single-channel, single-radio setup will only be able to reach a total of 33 Mbit/s split between the nodes transmitting data due to interference. The more nodes involved, the more useful MCMR is and exactly by how much the performance would increase can only be answered by performing more extensive testing.

Our implementation has shown to be suitable in different scenarios and with hardware prices decreasing, also a cost effective solution to enhance performance in wireless networks.

The next step would be to use more powerful hardware with support for more WLAN cards and abandon the MadWifi driver in profit for the new and constantly improved ath5k driver [11]. Also switch to IEEE 802.11n which is supported by ath9k [12]. Both ath5k and ath9k are included in the Linux kernel as from version 2.6.25. Also, further investigate optimizations to the drivers.

[11] `http://linuxwireless.org/en/users/Drivers/ath5k`
[12] `http://linuxwireless.org/en/users/Drivers/ath9k`

References

[1] D. Valerio, F. Ricciato and P. Fuxjaeger, “On the feasibility of IEEE 802.11 multi-channel multi-hop mesh networks”, COMCOM 3857, 2008.

[2] “IEEE 802.11,2007 IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications”, 2007.

[3] 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, 10(1):31–43, 2006.

[4] P. Kyasanur and N. H. Vaidya, “Routing and Interface Assignment in Multi-Channel Multi-Interface Wireless Networks”, in WCNC, 2005.

[5] J. So and N. H. Vaidya, “Multi-channel MAC for Ad Hoc Networks: Handling Multi-Channel Hidden Terminals using a Single Transceiver”, in Mobihoc, 2004.

[6] A. A. Pirzada, M. Portmann and J. Indulska, ”Hybrid Mesh Ad-hoc On-demand Distance Vector Routing Protocol ”. In Proceedings of the Thirtieth Australasian Computer Science Conference (ACSC’07), volume Vol 29, pages 49–58, 2007.

[7] A. A. Pirzada, M. Portmann and J. Indulska, ”Evaluation of Multi-Radio Extensions to AODV for Wireless Mesh Networks”, MobiWac ’06, 2006.

[8] C-Y. Li, A-K. Jeng and R-H Jan, ”A MAC Protocol for Multi-Channel Multi-Interface Wireless Mesh Network using Hybrid Channel Assignment Scheme”, Journal of Information Science and Engineering 23, 1041-1055, 2007.

[9] R. Maheshwari, H. Gupta and S. R. Das, ”Multichannel MAC Protocols for Wireless Networks”, SECON ’06, 2006.

[10] P. J. T. Clausen. RFC 3626: ”Optimized Link State Routing Protocol (OLSR)”, Oct 2003.

[11] D. B. Johnson, D. A. Maltz and Y.-C. Hu, “The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)”, Ietf Manet Working Group (Draft 10), 2004.

[12] P. Kyasanur, J. So, C. Chereddi and N. H. Vaydia, ”A Multichannel Mesh Networks: Challenges and Protocols”, IEEE Wireless Communications, 2006.

[13] R. Draves, J. Padhye and B. Zill, “Routing Area in Multi-Radio, Multi-Hop Wireless Mesh Networks”, ACM Mobicom, 2004.

[14] R. de C. Paschoalino and E. R. M. Madeira, “A Scalable Link Quality Routing Protocol for Multi- radio Wireless Mesh Networks ”, ICCCN 2007, 2007.

[15] C. Perkins, E. M. Royer and S. Das, ”Ad hoc On-Demand Distance Vector (AODV) Routing”, IETF RFC 3561, 2003.

[16] H-S. W. So, G. Nguyen and J. Walrand, ”Practical Synchronization Techniques for Multi-Channel MAC”, MobiCom ’06, 2006.

[17] M. X. Gong and S. F. Midkiff, “Distributed Channel Assignment Protocols: A Cross/Layer Approach”, WCNC 2005, 2005.

[18] J. Yoo and C. Kim,”On the Hidden Terminal Problem in Multi-rate Ad Hoc Wireless Network”, ICOIN 2005, 2005.

[19] Linux Ethernet Bonding Driver HOWTO, http://www.mjmwired.net/kernel/Documentation/networking/bonding.txt, 2009-09-25.

[20] C-Y. Li, A-K. Jeng and R-H Jan, ”A MAC Protocol for Multi-Channel Multi-Interface Wireless Mesh Network using Hybrid Channel Assignment Scheme”, Journal of Information Science and Engineering 23, 1041-1055, 2007.

[21] IEEE 802.11h, http://standards.ieee.org/getieee802/download/802.11h-2003.pdf, 2009-09-24.

[22] The Linux Kernel Module Programming Guide, http://www.tldp.org/LDP/lkmpg/2.6/html/x427.html, 2009-09-24.

[23] International Standard ISO/IEC 7498-1:1994, ”Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model”, Second edition 1994-11-15, Corrected and reprinted 1996-06-15.

[24] Douglas E. Comer, “Interworking with TCP/IP. Vol 1: Principles, Protocols, and Architectures”, Fourth Edition, 2000.

[25] Post- och telestyrelsens allmänna råd (PTSFS 2005:4) om den svenska frekvensplanen, http://www.pts.se/upload/Documents/SE/frekvensplanen2005_4.PDF, 2009-09-24.

[26] IEEE 802.11n, http://standards.ieee.org/prod-serv/80211n.html, 2009-09-24.

Nomenclature

802.11	A set of standards for wireless local area networks
ad-hoc	See IBSS
ARP	Address Resolution Protocol - translates an ip address to a hardware address
ATIM	Announcement Traffic Indication Message - announce that there are buffered frames to be sent
CPU	Central Processing Unit
DCF	Distributed Coordination Function
IBSS	Independent Basic Service Set - A peer-to-peer wireless LAN configuration part of the IEEE 802.11 standard.
IEEE	Institute of Electrical and Electronics Engineers - http://www.ieee.org
IOCTLs	Input/output control - allows user-space to communicate with drivers and kernel
kernel	Central component of the operating system
MAC layer	Media Access Control - A sublayer of the Data Link layer (OSI model)
Mini-PCI	Small formfactor extension cards for the PCI bus
multi-channel	Using more than one wireless channel simultaneously
multi-radio	Utilizing multiple WLAN cards on a single node
NAV	Network Allocation Vector - a mechanism to avoid contention
Open Source	Software under licenses which enables users to modify the source code
OSI model	Open Systems Interconnection Reference Model
PCI	Peripheral Component Interconnect - A cumputer bus for hardware devices

rendezvous	For example all nodes in the network tunes to a predetermined channel at the same time
WLAN	Wireless Local Area Network

Appendix A

User space daemon

```
/**
 * Reads user config and sets up sub interfaces
 * for fixed and dynamic interfaces.
 */
module_configure
{
        READ user config
        CONFIGURE_INTERFACE( interface, MCMR_FIXED )
        CONFIGURE_INTERFACE( interface, MCMR_DYNAMIC )
        SET Listen_Port
        SET Hello_Interval
        SET Neighbour_Entry_Expire
        SET Neighbour_Expire_Check
        SET Frequency_Range
        SETUP Enabled channels
}


/**
 * Initiates sockets and interfaces.
 * Starts function mcmr_hello_send for transmitting hello messages
 * and mcmr_input_poll_sockets for receiving.
 * Also starts remove_expired_neighbour to check and remove old neighbour entries.
 *
 */
module_start
{
        IOCTL clear kernel registry for MCMR interface
        IOCTL open socket to bonding module
        READ module configure parameters

        FOR EACH MCMR interface
                SETUP MCMR interface specific data
                INIT Listen_Port
                INIT Hello_Interval
                CONFIGURE Neighbour_Entry_Expire
                CONFIGURE Neighbour_Expire_Check

                FOR EACH slave interface
                        SETUP interface data
                        IOCTL enslave interface

                        IF fixed interface THEN
                                IOCTL enable fixed interface
                                SELECT fixed interface channel randomly
                                IOCTL set channel
                        ELSE IF dynamic interface THEN
                                IOCTL enable dynamic interface
                        END IF
                END FOR
        END FOR

        TIMER_REGISTER( Hello_Interval, mcmr_hello_send )
        TIMER_REGISTER( Interval, mcmr_input_poll_sockets )

        FOR EACH MCMR interface
                TIMER_REGISTER( Neighbuor_Expire_Check, remove_expired_neighbours )
        END FOR
}

/**
 * Prepare and broadcast HELLO messages
 *
 */
mcmr_hello_send
{
                INIT outgoing message

                FOR each interface
```

```
                        IF fixed interface THEN
                                ADD (permanent hardware address, current channel) to message
                        END IF
                END FOR

                FOR EACH known neighbour
                        IF one-hop neighbour THEN
                                ADD (neighbour MAC address, neighbour channel) to message
                        END IF
                END FOR

                SETUP MCMR-header
                SETUP UDP-header
                CALL NET_WRITE_SOCKET( message, broadcast-addr )
                CALL manage_fixed_channel
}


/**
 * Reads socket and checks if the message is a valid HELLO message.
 * Collects one- and two-hop neighbour information.
 */
mcmr_input_poll_sockets
{
        CALL NET_READ_SOCKET
        CHECK for valid ip- and udp-header in message

        IF message is a HELLO message && is valid THEN

                IF sender of the HELLO has at least one valid channel THEN
                        IF neighbour already exist in neighbour list THEN
                                UPDATE neighbour last seeen timestamp
                        ELSE
                                ADD neighbour to neighbour list, marked as one-hop neighbour
                                IOCTL add neighbour to kernel modules registry
                        END IF
                END IF

                FOR EACH neighbour node included in HELLO message
                        IF two-hop neighbour has at least one valid channel THEN
                                ADD neighbour to neighbour list, marked as two-hop neighbour
                        END IF
                END FOR

        END IF
}


/**
 * Removes old entries in the neighbourtable
 *
 */
remove_expired_neighbours
{
        GET timestamp
        FOR EACH interface
                FOR EACH neighbour
                        IF time difference between timestamp and last received HELLO > max expiry time THEN
                                REMOVE neighbour from neighbour list
                                IOCTL remove neighbour from kernel module registry
                        END IF
                END FOR
        END FOR
}


/**
 * Gets the number of neighbours using the channel the interface is currently tuned to.
 * Checks if there is a channel with less usage.
 *
 */
manage_fixed_channel
{
        SET least_used_channel to current channel

        FOR EACH enabled channel
                IF  neighbours using the channel < neighbours using the least_used_channel THEN
                        UPDATE least_used_channel
                END IF
        END FOR

        GENERATE random number between 0 and 10
        IF random number < 5 THEN
                IOCTL set fixed channel to least_used_channel
        END IF
}
```

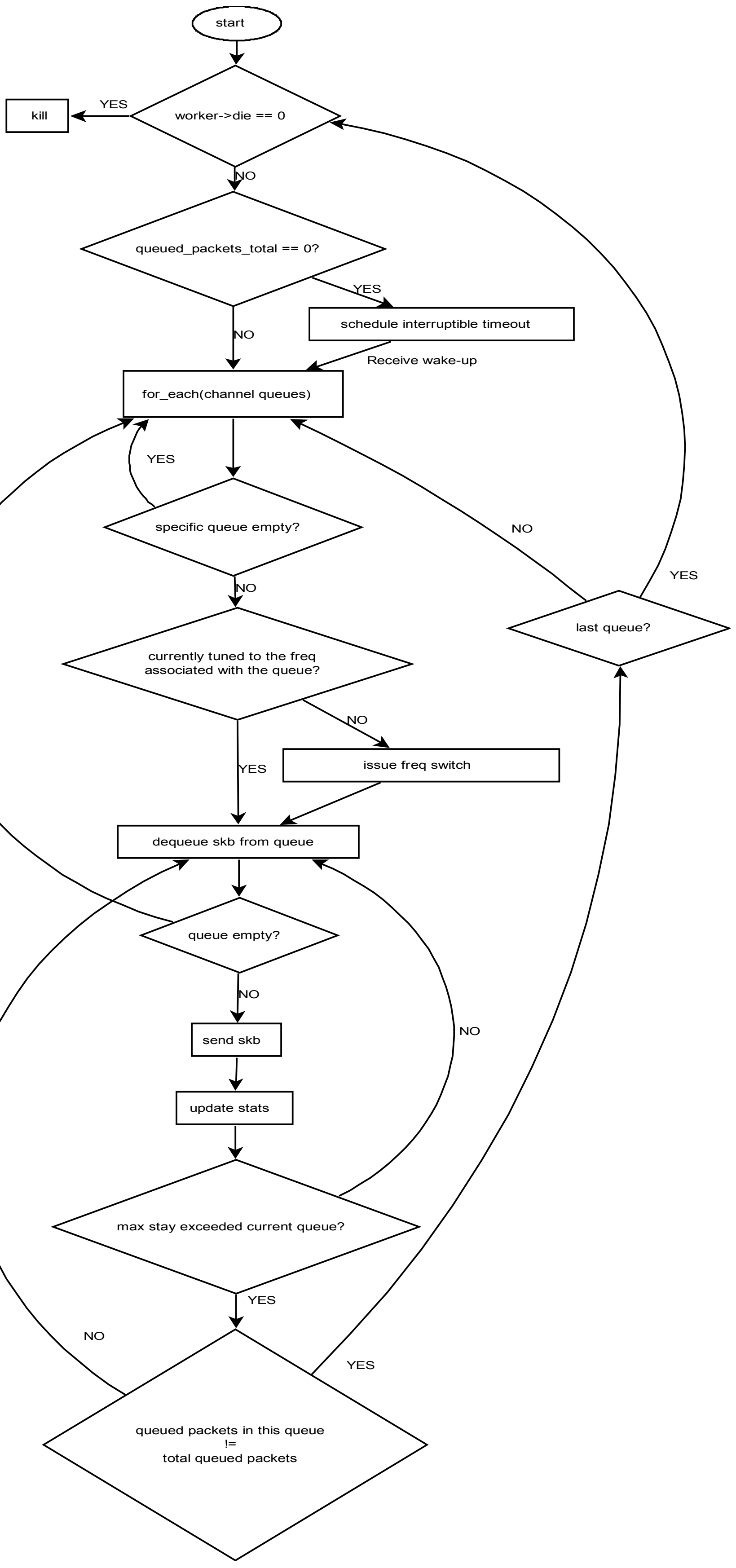
Kernel module

```
/**
 * Initiates the MCMR device.
 * This must be done before any interface
 * is enslaved.
 **/
mcmr_init_bond_interface
{
            SET device name
            CREATE proc directory
            CREATE sysfs
            SETUP wireless handlers
            SETUP ioctl handlers
            INIT neighbour registry as empty
            INIT packet queues
            START MCMR worker
            SETUP xmit function pointer
            CALL register_netdevice
}

/**
 * Binds an interface to the MCMR device and
 * assigns default settings.
 **/
mcmr_enslave_device
{
    STORE device's current flags
    STORE permanent MAC address

    IF first device to be enslaved
            COPY MAC address from slave to MCMR interface
    ELSE
            ASSIGN slave the same MAC as the MCMR interface
    END IF

    SET default MCMR flags
    SET device inactive
}

/**
 * Set the mode of a previously enslaved interface
 * to fixed.
 **/
mcmr_enable_fixed_interface
{
    SET mode fixed
    SET outgoing traffic disabled
    SET device as active
}

/**
 * Set the mode of a previously enslaved interface
 * to dynamic.
 **/
mcmr_enable_dynamic_interface
{
    SET mode dynamic
    SET outgoing traffic enabled

    FOR each enabled channel
            IF current frequency is supported by hardware THEN
                    INIT outgoing channel queue
            ELSE
                    RETURN error
            END IF
```

```
        END FOR

        SET device as active
}

/**
 * Called by the userspace daemon to
 * register neighbours received from HELLO
 * messages. Input parameters is the
 * neighbour's destination MAC and frequency.
 **/
mcmr_io_add_unicast
{
        IF frequency corresponds to a packet queue THEN
                ADD neighbour tuple to registry [MAC, FREQ]
        END IF
}

/**
 * Called by the userspace daemon to
 * remove neighbours.
 **/
mcmr_io_remove_unicast
{
        IF neighbour tuple exists THEN
                REMOVE from registry
        END IF
}

/**
 * Queueing of packages in the packet queues.
 * Broadcasts and unicasts are handled differently.
 *
 * Broadcasts are copied to all packet queues and
 * for unicasts a neighbour lookup is performed to
 * find out which channel the packet should be
 * queued in.
 **/
mcmr_enqueue_skb
{
        IF outgoing packet is broadcast THEN
                FOR each channel queue
                        IF packet not a HELLO message AND no known neighbour on channel
                                CONTINUE to next channel queue
                        END IF

                        IF queued bytes in queue + skb length > maximum queued bytes THEN
                                CONTINUE to next channel queue
                        END IF

                        COPY skb
                        CALL mcmr_worker_enqueue(skb)
                END FOR
        ELSE
                Lookup neighbour information in the MCMR registry
                IF neighbour not found THEN
                        RETURN NET_XMIT_DROP
                END IF

                IF queued bytes in queue + skb length > maximum queued bytes THEN
                        RETURN NET_XMIT_DROP
                END IF

                CALL mcmr_worker_xmit(skb)
        END IF

        UPDATE queue counters and statistics

        RETURN NET_XMIT_SUCCESS;
}

/**
 * Check if it is possible to transmit att packet directly.
 * If not, it is queued and later sent by the worker thread.
 **/
mcmr_worker_enqueue
{
        IF hardware tuned to outgoing channel && all queues are empty THEN
                SET source address in eth header to the cards permanent MAC address
                CALL send function in the MadWiFi driver directly and handover the skb
        ELSE
                QUEUE skb in the given queue
                WAKE UP worker if it's sleeping
        END IF

        UPDATE queue counters and statistics
}

/**
 * Worker Thread:
 * The thread is responsible for servicing
 * each packet queue. If a queue has packets
 * the hardware is tuned to the corresponding
 * frequency and the packet is handed over to
 * MadWiFi for transmission.
 **/
mcmr_worker_thread
{
        WHILE worker running

                IF packets queued THEN
                        CALL SCHEDULE (interruptible timeout)
                END IF

                FOR each packet queue

                        IF queue is empty THEN
                                CONTINUE to the next queue
                        END IF

                        IF wireless card not tuned to the corresponding frequency THEN
                                REPEAT
                                        IF hardware still have frames to send THEN
                                                CALL SCHEDULE (interruptible timeout)
                                        ELSE
                                                CALL the hardware frequency tune function in MadWiFi
                                        END IF
                                UNTIL hardware tuned to the frequency
                        END IF

                        IF avoid hidden terminal is enabled THEN
                                SLEEP (uninterruptible) for the configured duration
                        END IF

                        START timer tracking how long the specific queue has been serviced

                        WHILE queue contains packets
                                DEQUEUE SKB from queue
                                SET source address in eth header to the cards permanent MAC address
                                CALL send function in the MadWiFi driver and handover the SKB
                                UPDATE queue information

                                IF another queue has packets AND current queue serviced for atleast the
                                        maximum specified duration THEN
                                                BREAK to check the next channel
                                END IF

                        END WHILE
                END FOR
        END WHILE
        clean up
        KILL THREAD
}
```

Appendix B

Flow chart – MCMR worker thread


start
worker->die == 0
YES
kill
NO
queued_packets_total == 0?
YES
schedule interruptible timeout
Receive wake-up
NO
for_each(channel queues)
YES
specific queue empty?
NO
currently tuned to the freq associated with the queue?
NO
issue freq switch
YES
dequeue skb from queue
queue empty?
NO
send skb
update stats
max stay exceeded current queue?
NO
YES
queued packets in this queue != total queued packets
NO
YES
last queue?
NO
YES