Faculty of Economic Sciences, Communication and IT

Mats Persson, Rickard Karlsson

TOM++

Estimating One-Way Delay in Wireless Mesh Networks

Computer Science
C-level thesis

Date/Term: 10-01-22
Supervisor: Andreas Kassler
Examiner: Martin Blom
Serial Number: C2010-02

Computer Science

Mats Persson, Rickard Karlsson

TOM++ - Estimating One-Way Delay in Wireless Mesh Networks

Bachelor's Project

C2010:02

TOM++ - Estimating One-Way Delay in Wireless Mesh Networks

Mats Persson, Rickard Karlsson

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Mats Persson

Rickard Karlsson

Approved, 2010-01-22

Advisor: Andreas Kassler

Examiner: Martin Blom

Abstract

Wireless mesh networks is a relatively new concept that is being developed for, amongst other things, providing Internet access to rural areas. They are cheaper because no physical connections between the routers are needed and thus also easier to deploy. However, wireless links are slower and more unstable than wired network links, which is especially notable when streaming media such as Voice Over IP or movies. This can be improved with routing protocols should be able to find a path which minimizes packet delay and schedulers that can prioritize packets that have been in the wireless mesh network too long. For such a routing protocol to work, the delay a packet experiences in the network has to be monitored.

To solve this problem this thesis presents TOM++, an improved implementation of the Tool for One-way delay Measurement (TOM). The improvements take queuing time and channel switching into consideration when performing one-way delay measurements in a multi-radio, multi-channel wireless mesh network based on Net-X. This has been accomplished by approximation of the sending time of packets in the transmission queue at each intermediate node. TOM++ has been tested and evaluated on the KAUMesh testbed, and this resulted in an increase of approximately 5% in the one-way delay measurement accuracy. A prototype of a Kalman filter has been implemented to see if it can be used to predict the approximation error of the sending time. No results for the Kalman filter prototype is presented in this thesis, but suggestions on further development are.

Acknowledgements

We would like to thank our mentors Andreas Kassler and Peter Dely with all the help they have provided. We would also like to thank Marcel Cavalcanti de Castro for the help with the mesh nodes. Finally we want to thank Christina Zell, Sverker Zadig, Therese Axelsson and Robin Wikström for their help with proofreading the thesis, and our families and friends for their help and support.

Contents

List of Figures

List of Tables

Listings

1 Introduction

TOM, Tool for One-way delay measurement, was made to monitor packet delays. It estimates the one-way delay, which is defined as the time it takes for a packet to travel through a wireless mesh network from when it enters the mesh backhaul until it is leaving at the mesh gateway or, in the case of intra-mesh traffic, the last mesh router. This is shown in figure 2.1 in section 2.1.

This thesis describes how to improve the estimation that TOM performs. TOM measures the time from when a packet arrives at a router until the packet is put into the transmission queue in the hardware, and then adds an estimation to that time to get the delay for the packet. The estimation is done by averaging the time it takes from when a packet enters the hardware transmission queue until an acknowledgment is received. The main improvement compared to TOM is made by approximating the sending time of packets in the transmission queue of the wireless mesh routers which leads to less time to estimate, and also taking channel switches into account. A prototype of a Kalman filter will also be implemented to test if it can improve the sending delay estimation of each packet. The implementation will be evaluated on the KAUMesh, Karlstad University Mesh, testbed.

The thesis has the following structure:

Section 2 states the background information necessary for understanding the detailed sections.

Section 3 gives an overview and detailed description of the architecture from an abstract point of view.

Section 4 is a detailed description of what happens to a packet as it makes its way through a mesh router.

Section 5 contains a description of the tests that where performed.

Section 6 sums the key points of this thesis, the result and suggests possible future improvements.

2 Background

This section gives an overview of wireless mesh networks, a wireless mesh network testbed set up at Karlstad university, different approaches of delay monitoring and a description of the Kalman filter algorithm.

2.1 Wireless Mesh Networks

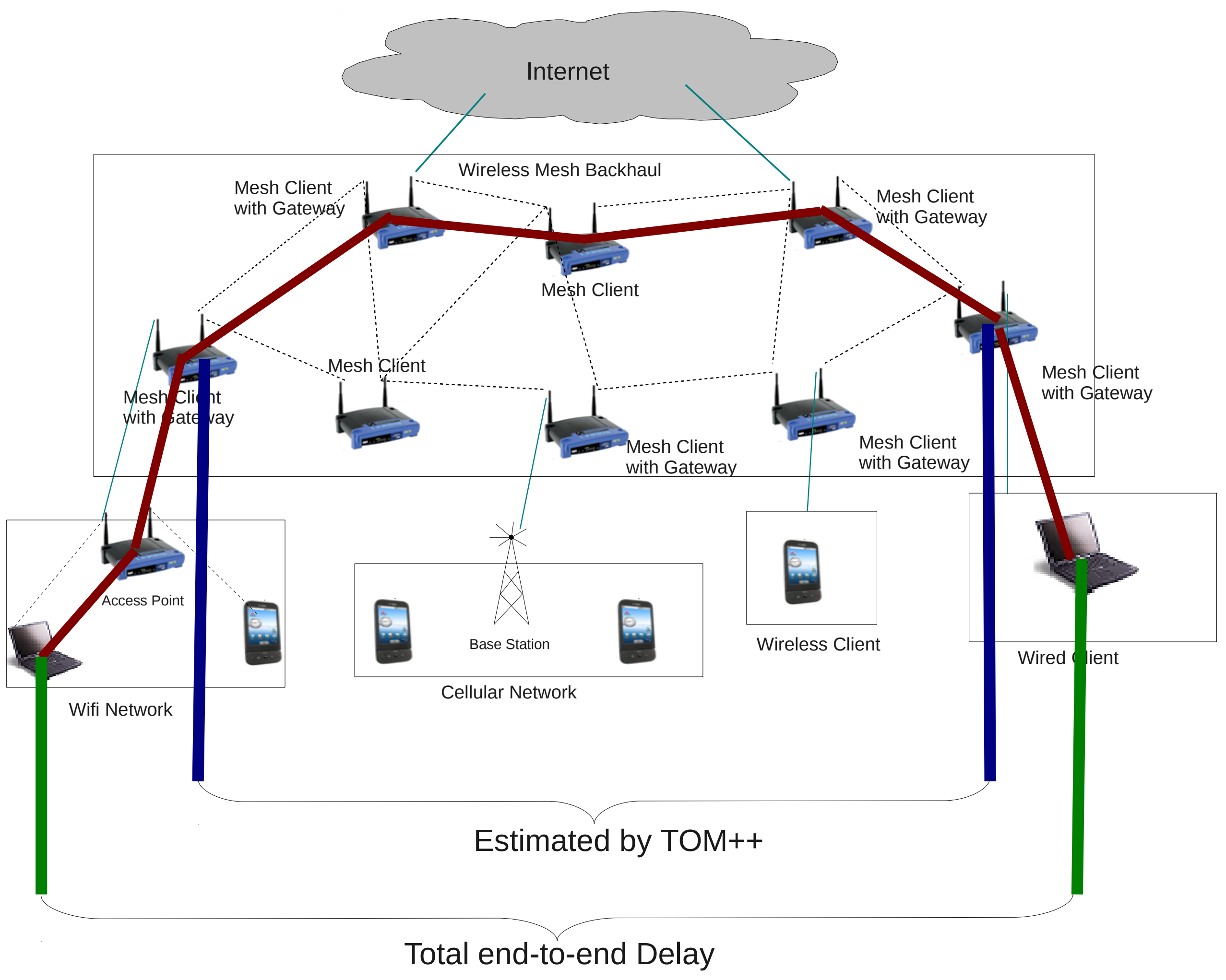

Figure 2.1: One-way delay overview

A wireless mesh network[22], figure 2.1, is a network that is structured like a mesh with radio links as data bearers. The mesh is constructed of mesh routers which connect wirelessly with each other. For Internet connection every mesh network needs to have dedicated mesh routers with gateway functionality relay packets between the networks. The mesh routers are the backhaul of a wireless mesh network, the same as a backbone is for a wired network. The radio links can be implemented with any kind of wireless technology or a mix of several technologies, for example 802.11, 802.16 or W-CDMA.

Communication between nodes in a wireless mesh network is broken down into a series of hops. This allows data to be sent over a large distance. Intermediate nodes route the data dynamically through the mesh, making the network more stable since packets can be routed around nodes that are disabled or have a high load. Traffic in wireless mesh networks is often forwarded to and from a wired network through a gateway.

There are three generations[18] of wireless mesh networks; Single-Radio, Dual-Radio and Multi-Radio Mesh.

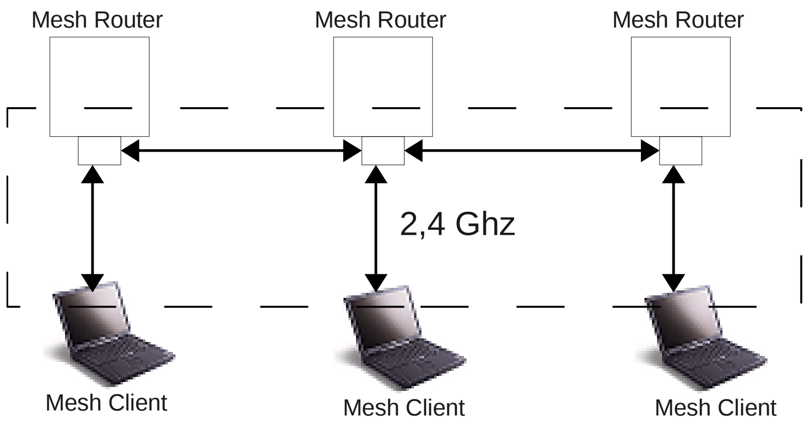

Figure 2.2: 1-Radio

Single-Radio Mesh (figure 2.2) uses only one radio channel for both clients and backhaul, therefore providing merely low performance.

In a Dual-Radio Mesh (figure 2.3) the mesh routers are equipped with two radios; one

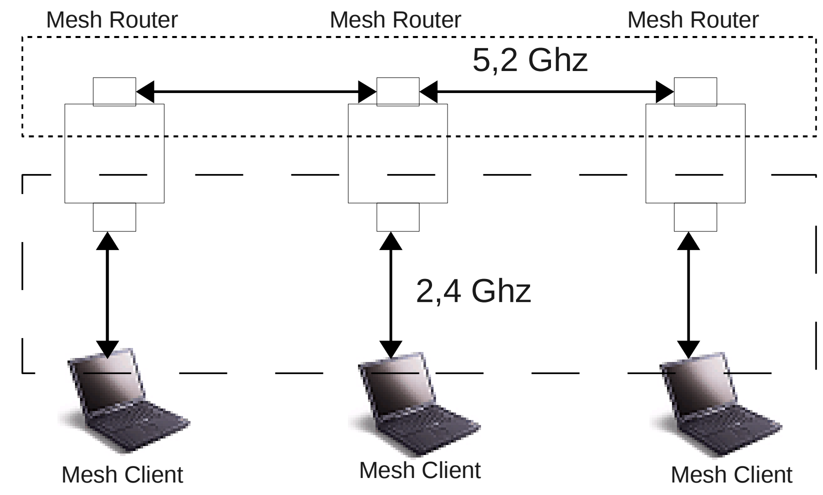

Figure 2.3: Dual-Radio

for clients and one for the backhaul. The radios can operate at different frequencies so that sending and receiving can be done in parallel. Although the performance is improved from Single-Radio Mesh the backhaul is still operating at the same frequency on all mesh routers.

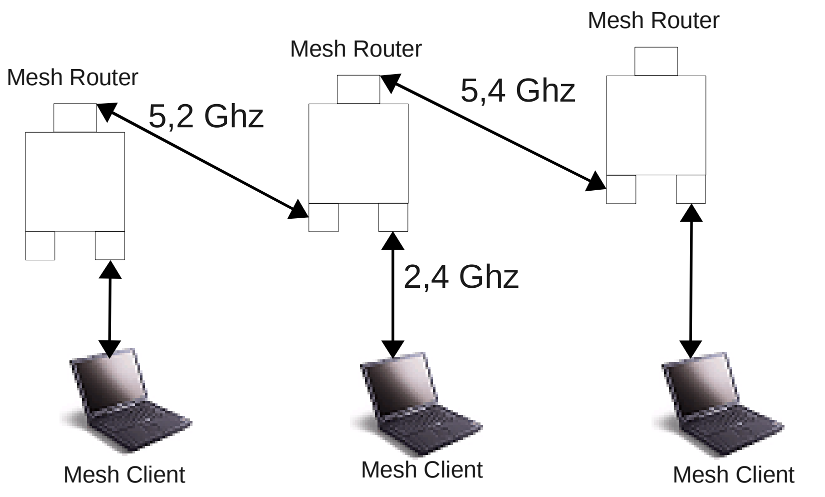

Figure 2.4: Multi-Radio

Multi-Radio Mesh (figure 2.4) uses different channels for the links in the backhual that does not interfere with adjacent links and one channel for servicing clients on a different interface. This increases the performance significantly since each link can be used in parallel.

2.2 Multi-radio, Multi-channel Wireless Mesh Networks

The connection to the mesh backhaul is through mesh nodes that transmit packets wirelessly between each other towards the gateway nodes, which connects the mesh network to the Internet. The number of interfaces that are used to access the backhaul to create a mesh network limits the number of data connections that can happen at the same time. By increasing the number of interfaces, you can also increase the upper limit of the capacity in the network. More channels will be needed to be able to have more interfaces sending at the same time. Depending on the country there can be up to 14 channels available in 802.11b/g and up to 13 channels in 802.11a[10].

One of these interfaces is called the Fixed Interface, as it is allocated at a fixed channel over rather long periods of time compared to the switchable interface. The fixed channel is also selected by a fixed channel selection protocol and can automatically adapt in accordance with for example the interface situation and the traffic load.

The other interface on each node is called the Switchable Interface. It can switch dynamically between any of the remaining channels according to traffic demand, and the maximum and minimum channel switch interval can be configured with command line parameters.

All the data that is sent to the node must go through the fixed channel of the receiving node since that is the only channel the node always listens on. A node can send data through the fixed interface if the receiver shares the same fixed channel; or else the switchable interface of the sender will change to the receivers fixed channel, allowing the data to be sent through the switchable interface instead. Because of this, the mechanisms do not require coordination channels.

Each node periodically sends out a "Hello"-message on all channels. The message carries the fixed channel of all 1-hop nodes. Through this a node can learn the fixed channels of all 2-hop nodes and then use the information to balance the assignment of the fixed interface. For monitoring purposes each node has a wired Ethernet connection to an administration network, making it possible to send monitoring data to the monitoring server in the wired network. It also enables the transmission of configuration information and the download of new kernel versions to the mesh nodes.

Net-X[1][11] is a structure that has been developed by the Wireless Networking group at UIUC, University of Illinois at Urbana-Champaign. The purpose of this structure is to allow for instance more channels, more interfaces and higher transfer speed / power levels in a wireless network. In the next generation of networks there will probably be even more interfaces with different abilities available; such as setting the channel to work on, setting different data rates on different frequencies and equipping the nodes with more radio interfaces and antennas.

By carefully designing the protocols and by using these interface potentials the network performance can increase a lot. However, since most operating systems today do not support the implementation of protocols necessary to these abilities, changes in the operating systems will be needed to make.

The goal for the Net-X group is to develop a common interface, that is equal to the interface of the operating system, so that Net-X can be integrated in the network stack. As a first step in this direction, an architecture that supports the use of several channels, interfaces and interface switching has been developed in Linux. The current structure can easily be extended with other abilities, and these extensions are also a part of the future for the Net-X group. A wireless test environment that uses the Net-X structure has been developed and so far a set of multichannel protocols, developed earlier by the group, has been implemented.

[1]A Wireless Networking Framework providing System eXtensions forSupporting Multiple Channels, Multiple Interfaces, and Other InterfaceCapabilities

2.3 Delay Monitoring

There are several things of interest to monitor in a wireless network; amongst others the one-way delay of a packet, which is the focus of this thesis. Approaches for monitoring the delay of a packet in a wireless path can be divided into two different types, passive and active monitoring. The goal in this project is to estimate the time it takes for a packet to travel through a network as shown in figure 2.1. This will be done with passive monitoring but examples of active monitoring will be presented to get an understanding of the difference between them.

2.3.1 Active Monitoring

Active monitoring approaches generate probing traffic. This results in an overhead and lowers the capacity of the network. However, this active monitor is flexible as it gives control over the generating of packets. There are two different ideas of active monitoring that can be used which are widely spread: the Round Trip Time(RTT) and Packet Pairing(PP).

The PP[14] idea is to send two packets adjacent to each other and should be queued after each other as well. The time when the first packet has been received is then compared to the time when the second packet starts to be received. This is called the dispersion or transmission time. If it is possible to add a timestamp just before the transmission is made, it will result in a very accurate estimation of the transmission time. The downside of this idea is that the clocks at the nodes have to be synchronized, since hardware clocks in general are not coordinated and not even equally fast, synchronization is difficult.

There have been two different approaches on PP. One made by Kapoor et al.[7] that only worked well in simulations and showed that it only functioned well in lightly loaded networks. Sun et al.[8] did another approach with ad hoc probes that were based on the previous mentioned method and the results they got from this were accurate in wireless networks.

In the RTT idea, no synchronized clocks are necessary, it only needs to record the time until a packet returns. In a wireless environment the downside of this idea is that it assumes that the receiver can answer very fast and that the links are symmetric. It would not give an exact time of the one-way delay in a multi-radio multi-channel wireless mesh network with asymmetric links that also have interfering traffic. Another disadvantage of this idea is that additional traffic needs to be generated to update the measurements. Information gathered this way will only be available at the node that triggered it. Further, the probing packets are very small, and the consequence is that the measurements may underestimate larger packets.

There have been experiments on both of these approaches and also on a combination of the two, but none of them gave good results. The probing packet experiments were made by Draves et al.[6]. He compared the ideas and PP was suffering from that the probes will cause additional overhead and are sent at intervals not correlated to the payload traffic. RTT was suffering from problems with clock drifts, skewness and also failed to give a good time in asymmetric links.

2.3.2 Passive Monitoring

Passive monitoring[4] uses a device to monitor something in a network, in this case a change in the network driver that the routers use. This approach collects information and therefore does not generate any extra traffic. Deep packet inspection could be used to gather the information needed but this would result in privacy and security risks. In this case, the only data collected is of how long a packet has been in a network and as a result it does not check the actual data in the packet. The major drawbacks of passive monitoring are to get the delay that is calculated in one node to the next node and to estimate the time it takes for a packet to travel over the medium between two nodes.

Since none of the active monitoring methods gave any good results or suffered from at least one drawback, an algorithm named Adaptive Per Hop Differentiation(APHD)[13] scheme, was introduced. The basic idea of APHD is to update the delay a packet has experienced so far on a multi-hop path, to estimate the transmission time to the next node at each intermediate link and to prioritize packets with a high delay. By saving the *delay so far* there is a possibility to compare that value to a total delay value and thus temporarily change the priority of packets in the network. However, this has only been simulated so no real results have been presented.

To be able to do this, four fields has been added in the Internet Protocol(IP) header option field. The first two fields, *end-to-end delay requirement* and *end-to-end hops*, are set by the sender and specify the time and the hops for a packet between the sender and receiver. The last two fields, *delay so far* and *hops so far*, will be updated at each forwarding node. When a packet arrives to the MAC layer at node B a timestamp $t_{in}(B)$ is recorded, then when the packet is to be transmitted another timestamp $t_{out}(B)$ will be recorded, as shown in figure 2.5. Then the queuing and processing time at node B will be calculated exactly as:

$$t_B = t_{out}(B) - t_{in}(B) \tag{2.1}$$

To avoid any problems with clock synchronization the time the packet will spend on the link between node B and node C is estimated in node B as well. That time is calculated as the packet length p_L divided by the transmission rate R. This will not include backoffs and retransmissions.

$$t_{BC} = p_L/R \tag{2.2}$$

One problem with this calculation, since the backoffs and retransmissions are excluded, is that this is also not included in the packets in the transmission queue. Therefore the delay for a packet which is not head of the transmission queue when it arrives will have a smaller value than expected.

Both of these are assumed to be known when the transmission starts. Just before the packet is sent the hops so far will be increased and the delay so far (dsf) will be calculated and updated as

$$dsf = dsf + t_B + t_{BC} \tag{2.3}$$

With this information the nodes can set the priority level of the packet in order to send packets with high *delay so far* before packets with low *delay so far*.

2.3.3 TOM

TOM is a passive monitoring tool that is used to measure the one-way delay of packets in a wireless mesh network.

Two APHD ideas are used to implement TOM. The first one is an estimation of the one-way delay as the sum of the inter-node queuing and processing delay and the intra-node transmission delay.

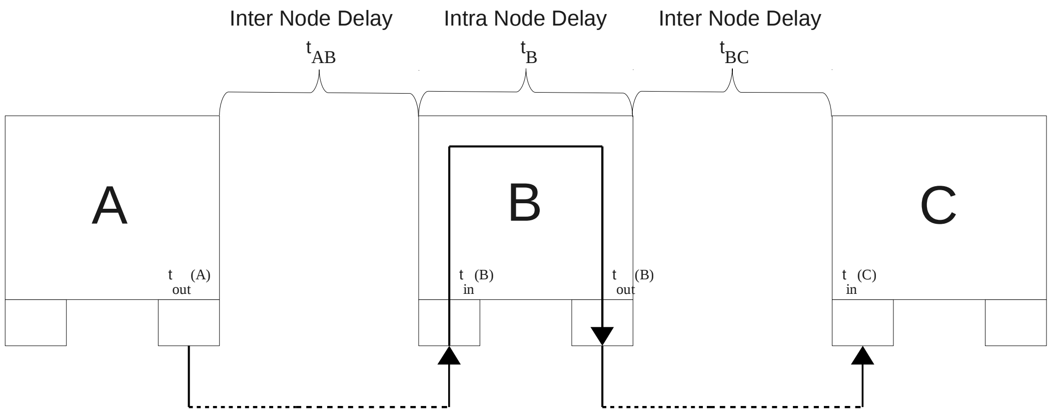

Figure 2.5: APHD - Inter- and Intra-Node Delay

As shown in figure 2.5, the inter-node delay is the delay in the network between two nodes and the intra-node delay is the delay a packet experiences within a node. The second idea is that each packet carries delay information in its IP header option field to make it immediately available. With the second idea one of the drawbacks of this kind of monitoring is solved, however TOM still needs to estimate the approximation error.

2.4 Kalman Filter

A problem with one-way delay measurements is that it is hard to predict the traffic load in a wireless network. For this a Kalman filter[9] can be used as it is a recursive filter that is good for estimating the true state of a linear dynamic system where only noisy observations or measurements are available. It is used to solve a wide variety of engineering problems, such as computer vision and control theory, and is modeled on a Markov chain built on linear operators perturbed by Gaussian noise. The filter updates the estimation at each measurement, it applies a linear operator to the filter state to generate the next state. One advantage about the Kalman filter is that it only needs the previous state and current measurement to predict the next state.

It is a flexible algorithm, since there are many different matrices to tune and there is no fixed number of parameters to filter, so it is tunable for many different systems. Another reason to use the Kalman filter is because it is fairly easy to implement; it does not require much memory or CPU and is well tested with good results. However, since there are many different values to tune it is good to have experience in matrices and the filter itself to be able to tune it right.

2.4.1 Discrete Kalman Filter

The Discrete Kalman filter can roughly be separated into two phases, the predict phase and the update phase, as shown in figure 2.6.

The predict phase uses the previous estimate $\hat{x}_{n-1}$ to make a new estimate $\hat{x}_n^-$, not include any observations(true measurement) from the current timestep. In the update phase, the current predict state will be combined with the new observations y_n and result in $\hat{x}_n$.

The predict phase equations for timestep n are:

$$\hat{x}_n^- = A \times \hat{x}_{n-1} + B \times u \tag{2.4}$$

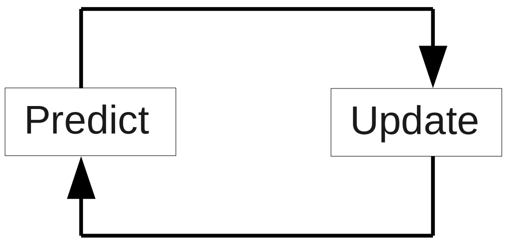

Figure 2.6: Kalman Filter Predict/Update cycle

$$P_n^- = A \times P_{n-1} \times A^T + S_w \tag{2.5}$$

In equation 2.4 and 2.5 the $a \times a$ matrix A is the state-transition model, meaning that it relates the state at the previous timestep $n-1$ to the state at the current timestep n. In practice, A might change at each timestep, but here it is assumed to be constant.

In equation 2.4 the $a \times h$ matrix B relates the optional control input u to the current state at timestep n.

In equation 2.5 the $a \times a$ matrix P_n^- is the error covariance matrix and P_{n-1} is the previous timestep error covariance matrix. The $a \times a$ matrix S_w is the process noise covariance; this matrix could change at each timestep, but here it is assumed to be constant.

The observation y_n is made between the predict phase and the update phase. The equations for the latter phase at timestep n are:

$$K_n = \frac{P_n^- \times C^T}{C \times P_n^- \times C^T + S_z} \tag{2.6}$$

$$\hat{x}_n = \hat{x}_n^- + K \times (y_n - C \times \hat{x}_n^-) \tag{2.7}$$

$$P_n^- = (A - K \times C) \times P_n^- \tag{2.8}$$

In equations 2.6, 2.7 and 2.8 the $a \times m$ matrix C is the observation model and relates the state to the measurement y_n. Practically, C might change at each timestep, but here it is assumed to be constant.

In equation 2.6 the $a \times m$ matrix K_n represents the Kalman gain for the current timestep n. The $m \times m$ matrix S_z is the observation noise covariance and is assumed to be independent from the process noise matrix. It could practically change at each timestep, but is assumed to be constant.

In equation 2.7 the estimation is updated with the observation y_n. The difference $(y_n - C \times \hat{x}_n^-)$ is called the residual, and reflects the difference between the predicted measurement $C \times \hat{x}_n^-$ and the actual measurement y_n. If the residual is zero, it means that there is no difference between them. The weighting of K is important to this equation, since it decides how the actual measurement and the predicted measurement should be weighted. If the observation noise covariance S_z approaches zero, the actual measurement y_n is increasingly important while the predicted measurement is decreasingly important. On the other hand, if the estimate error covariance P_n approaches zero the actual measurement y_n is decreasingly important, while the predicted measurement $C \times \hat{x}_n^-$ is increasingly important.

In equation 2.8 the error covariance is updated.

The observation noise variance is usually measured by taking samples before the use of the Kalman filter which is used to initialize the observation noise covariance matrix S_z.

The process noise covariance S_w is usually more difficult to determine, since the ability to directly observe the process that is being estimated is most often not possible. If there is a relatively simple process model it can produce acceptable results, provided that enough uncertainty is inserted into the process via the selection of S_w.

In either case, no matter how the parameters are chosen, superior performance can generally be obtained by tuning the filter parameters S_z and S_w. When a parameter is tuned it is often done before the Kalman filter is applied, and most times with the help of another Kalman filter, in a process referred to as system identification. If S_w and S_z are constant, the covariance P_n and the Kalman gain K will stabilize and also become close to constant. If this should be the case, these parameters can be tested in another test system or by running the system off-line[2] to initialize them right.

2.4.2 Scalar Kalman Filter

If the Kalman filter is to be used in a system that only has one variable to estimate. The scalar Kalman filter[2] has a predict phase and an update phase too;

The predict phase equations for timestep n is:

$$\hat{x}_n^- = A * \hat{x}_{n-1} + B * u_n \tag{2.9}$$

$$P_n^- = A^2 * P_{n-1} + S_w \tag{2.10}$$

In equation 2.9, $\hat{x}_n^-$ represents the predicted state and P_n^- represents the error covariance, at timestep n without any observation.

The observation y_n is made between the predict phase and the update phase. The equations for the latter phase at timestep n are:

$$K_n = \frac{C * P_n^-}{C^2 * P_n^- + S_z} \tag{2.11}$$

$$\hat{x}_n = \hat{x}_n^- + K * (y_n - C * \hat{x}_n^-) \tag{2.12}$$

$$P_n = P_n^- * (1 - C * K_n) \tag{2.13}$$

A, B and C are constants in equations 2.9 through 2.13. The input to the system is u_n and the output is y_n

[2]Off-line refers to the time before the Kalman filter is applied.

3 Design

The goal with this project is to develop an one-way delay measurement in a wireless mesh network, based on the TOM algorithm. There are five requirements to accomplish this for the approach used in this project. The first requirement is to measure the time spent in a mesh node due to node internal packet handling, referred to as the measurement. The second requirement is the calculation of the time it takes to send a packet over the medium between the mesh nodes to determine the transmission queue waiting time, this is referred to as the approximation. These two requirements are referred to as the measurement process described in section 3.2. The third requirement is the estimation of the unpredictable time which includes packet resends, backoffs and channel switches. This requirement is referred to as the estimation process that is described in section 3.3. The fourth requirement is to transfer the delay measurement values between the nodes. This is done by extending the IP header with accumulated per hop delay estimates described in section 3.1. The last requirement is that the software in clients connecting to the network is to remain unmodified. This means that the one-way delay measurement will be done in the wireless mesh network backhaul as shown in figure 2.1. Therefore the transmission time between a client and the first mesh router will not be included.

The difference between TOM and TOM++ is that the approximation, estimation and channel switches. TOM++ and TOMKalman differs in the estimation. This will be described in detail in following sections.

3.1 IP header

The IP header[19] contains information about the packet, such as source and destination address for the packet, total length, options and padding. The IP header can roughly be divided into two parts, a fixed part and an optional part as shown in figure 3.1. The optional part contains the options and padding fields as they variable in size or nonexistent.

Consequently the fixed part is all fields up to the options field since they are required.

TOM uses the option field to transfer the delay information. This is because the other options are to either modify the IP header or store the values with the packet data. Modifying the IP header can cause problems if a packet is sent to a network that does not support this modification. Storing the values with the packet data cannot be done because of privacy reasons. TOM++ and TOMKalman uses the same approach as TOM so two additional delays will be stored in the option field. The reason why the TOM delay information is not replaced is because measurements will be done to compare the algorithms in the evaluation.

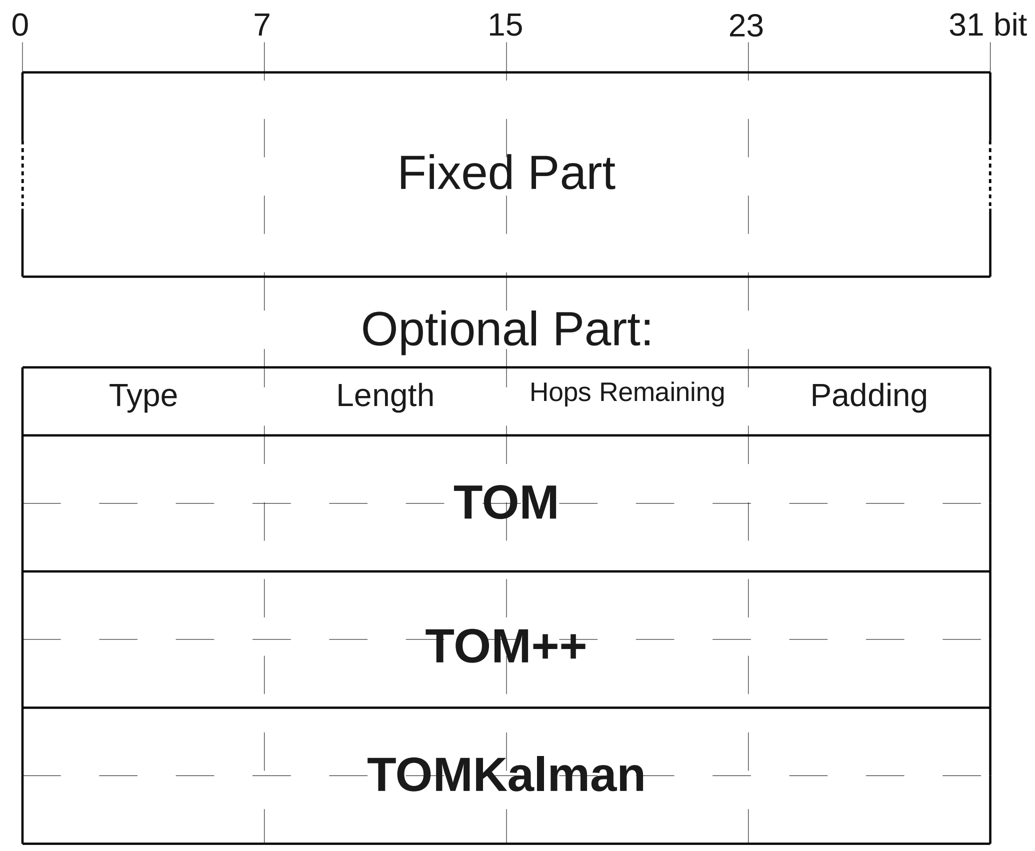

Figure 3.1: The IP header

The fixed part of the IP header (figure 3.1) is not described since it is not important for this project.

The optional part contains seven fields; *type* is the IP option type; *length* is the total IP option size in bytes, this includes all seven fields; *hops remaining* keeps track of the maximum hops until the packet should leave the backhaul; *padding* is used for aligning the fields 32-bit chunks; *TOM*, *TOM++* and *TOMKalman* are the one-way delay measurements for the respective algorithms.

3.2 Measurement Process

The one-way delay measurement will be updated at every node within the wireless mesh backhaul. The measurement process is from when a packet arrives at one node until it is transfered to the hardware transmission queue.

As shown in figure 3.2, there are five key points. Point A and E are also shown in figure 3.3.

At point A a packet arrives at a mesh router, in this example a mesh gateway, and the packet is marked with a timestamp by the hardware.

At point B the IP header option field is checked for the delay information. If it is present the delay fields are updated with the timestamp from point A. Otherwise the delay information is added to the option field and initialized. The measurement is started.

At point C the packet is put into the correct channel queue where it resides until the scheduler switches channel to the channel associated to the queue, then transfers the packet to the MadWifi driver. At the channel switches the TOM++ and TOMKalman estimations, described later, are reset. This point is only relevant if Net-X is used.

At point D the sending time t_n, where n is the position of the packet in the queue, is approximated by dividing the packet size s of this packet and the possible packets in

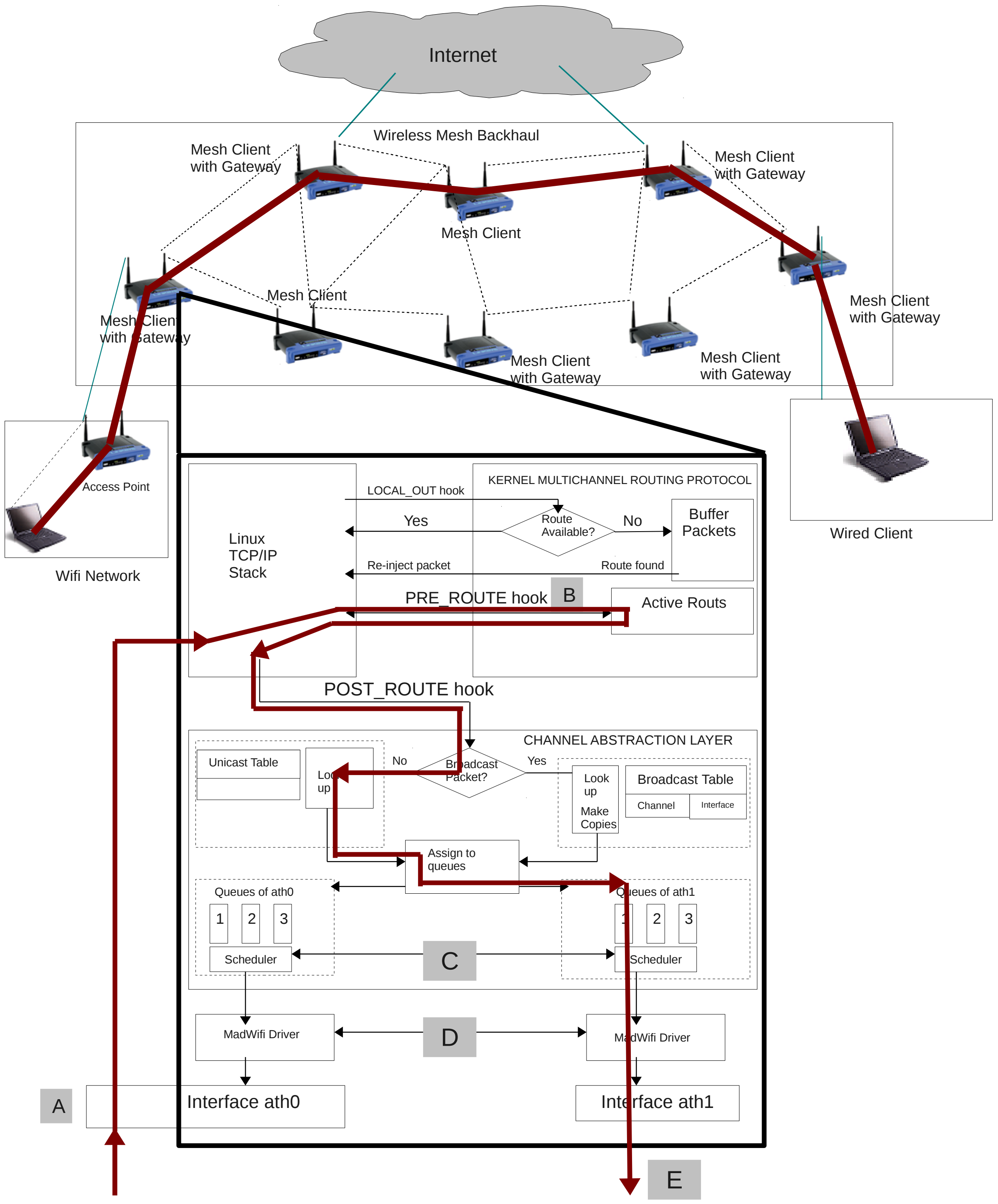

Figure 3.2: Packet Path Through a Node

the transmission queue with the transmission rate r in equation 3.1.

$$t_n = \frac{\sum_{i=0}^{n} s_i}{r} \tag{3.1}$$

Then the measurement is ended by updating the delay fields TOM_i (equation 3.2), $TOM++_i$ (equation 3.3) and $TOMKalman_i$ (equation 3.4) where i is the current one-way delay and $i-1$ is the previous one-way delay. This is done with the current time $currTime$, the approximation a and the respective estimation e. The packet is then transfered to the hardware transmission queue.

$$TOM_i = TOM_{i-1} - currTime + e_{TOM} \tag{3.2}$$

$$TOM++_i = TOM++_{i-1} - currTime + a + e_{TOM++} \tag{3.3}$$

$$TOMKalman_i = TOMKalman_{i-1} - currTime + a + e_{TOMKalman} \tag{3.4}$$

At point E the estimation is updated as described below.

3.3 Estimation Process

The estimation process includes the backoff, possible packet resends and acknowledgment, which can be seen at point E in figure 3.3.

The backoff is a random time that the hardware waits before checking if the medium is clear to send on. This time is determined by the hardware and the range of values that the time can obtain increases in case of collisions in the medium. It is used to avoid packet resending.

A resend can be caused by for example packet collisions and interference in the medium, thus it is impossible to predict.

The acknowledgment is used to determine when to end the estimation and thus its

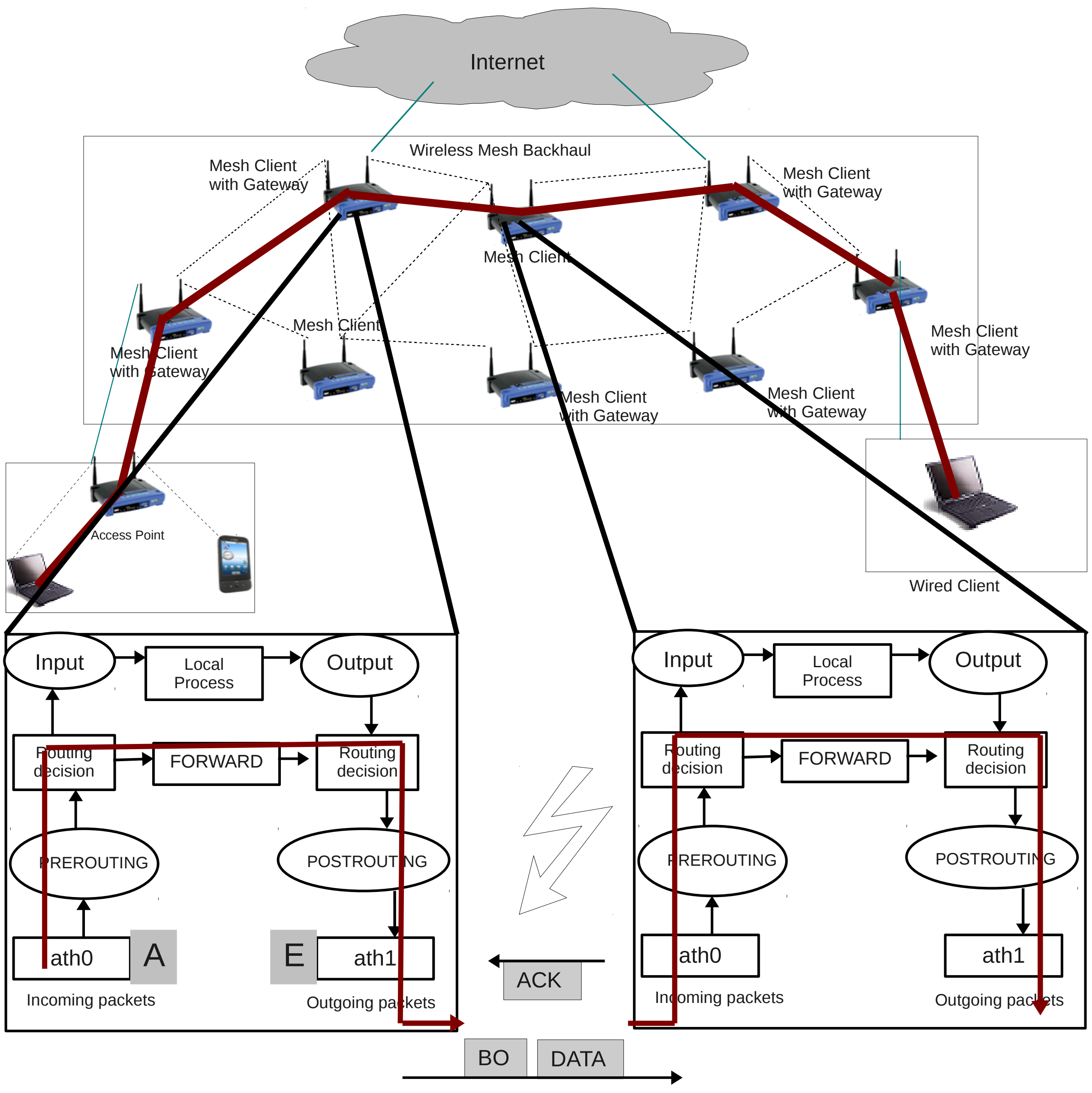

Figure 3.3: Estimation Overview

sending time is included in the estimation. Hence the estimation of a packet will be used on the next packet to be sent.

In TOM++ the estimation is calculated with an Exponential Weighted Moving Average (EWMA)[1] filter that uses weighting factors which decrease exponentially, giving more importance to recent measurements while not discarding the old measurements completely (equation 3.5).

$$e_n = \alpha * e_{n-1} + (1 - \alpha) * t, \alpha \in [0, 1] \tag{3.5}$$

where e_n is the new estimation; e_{n-1} is the previous estimation; α is the weighting factor which could change over time, but here it is assumed to be constant; and t is the measured time from when a packet becomes head of the transmission queue until an acknowledgment is returned.

TOMKalman uses a scalar Kalman filter instead of the EWMA used in TOM++ to calculate the estimation. In the scalar Kalman filter the weighting factor is K which is dynamically updated. TOMKalman uses the same measurement to update the estimation as TOM++ does.

4 Implementation

As described roughly in section 3 the one-way delay measurement includes several steps, measuring the time a packet spends in the node until it is put in the hardware queue, approximating the sending time for the packet and the hardware transmission queue, and estimating the approximation error, which includes the backoff and retransmission time, and the time for the acknowledgment to return.

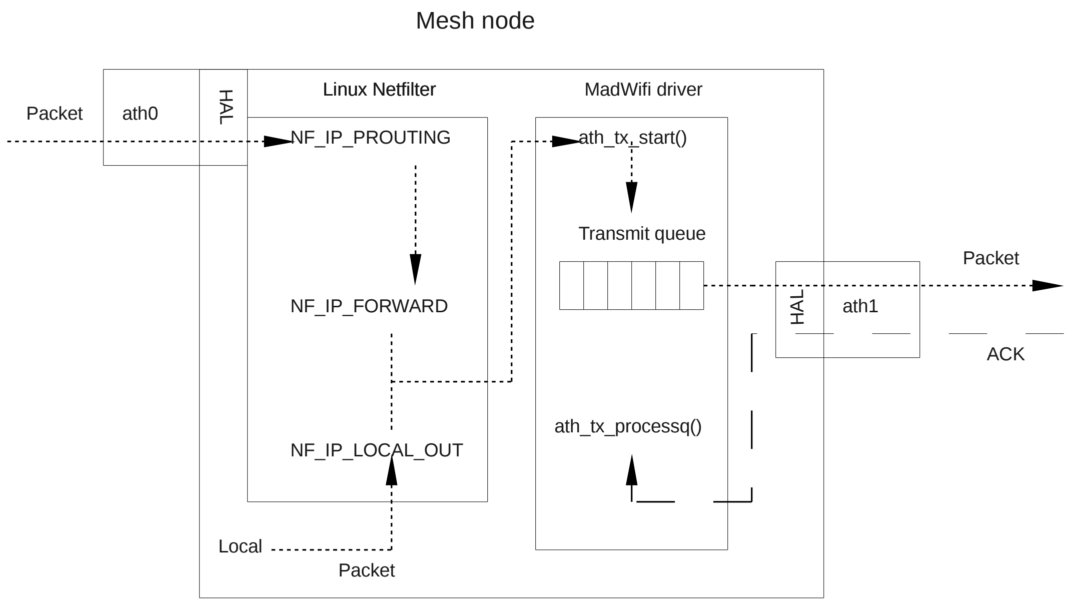

Figure 4.1: Implementation Overview

Here is a short summary of what this chapter contains. When a packet arrives at a node it goes through the Linux Netfilter as described in section 4.1, and if the packet arrives from outside the wireless mesh network the three delay fields are added to the IP header, initialized and the measurement is started, otherwise the fields are updated to start the measurement. Then in the MadWifi driver (section 4.2.1) the measurement is completed, the approximation is calculated, delay fields and estimations are updated. The approximation is the same for TOM++ and TOMKalman. The TOM algorithm, however, has no approximation and therefore only uses an estimation. Estimations for TOM++ and TOMKalman are different in that TOM++ uses a simple averaging function, described in section 4.2.1, and TOMKalman uses a Kalman filter, described in section 4.3. All variables referring to time are in nanoseconds.

4.1 Linux Netfilter

The Linux Netfilter contains five different hooks in the network stack as shown in figure 4.2. Depending on from where the packet arrives and its destination it passes through different hooks. Three of these hooks are used for delay estimation: NF_IP_PREROUTING, NF_IP_FORWARD and NF_IP_LOCAL_OUT. The NF_IP_LOCAL_IN hook is not used because the packet has reached its destination and the NF_IP_POST_ROUTING hook is not used because the IP header option field has already been initialized.

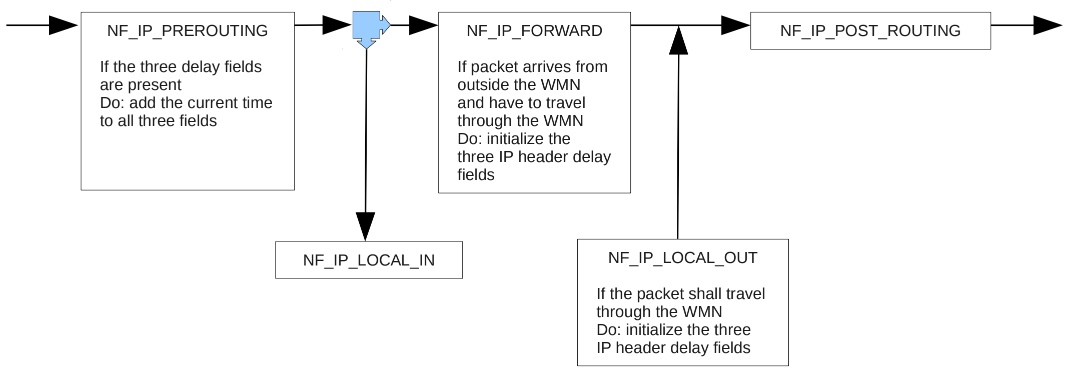
NF_IP_PREROUTING
If the three delay fields are present
Do: add the current time to all three fields
NF_IP_FORWARD
If packet arrives from outside the WMN and have to travel through the WMN
Do: initialize the three IP header delay fields
NF_IP_POST_ROUTING
NF_IP_LOCAL_IN
NF_IP_LOCAL_OUT
If the packet shall travel through the WMN
Do: initialize the three IP header delay fields

Figure 4.2: Linux Netfilter Hooks

When a packet arrives at a node it goes through the NF_IP_PREROUTING hook where the IP header is checked to see if the three delay fields are present by checking if the length of the IP header is equal to the length of the fixed part of the IP header and the length of the three delay fields. The IP header structure is described in section 3.1. If they are not, nothing further is done here because the header fields will be added in the NF_IP_FORWARD hook. Otherwise the current time is added to all three fields to start the measurement. If this node is not the destination of the packet it is routed through the NF_IP_FORWARD hook. Here the packet is checked where it was before entering this node and where it is heading by looking at which interface it was received and which interface it is to be sent on. If it comes from outside the wireless mesh network heading into it the three delay fields are added and initialized to $10000000000LL + timestamp$. The *timestamp* is stored in the socket buffer of the packet when it is received by the network interface and is used for starting the measurement. If a packet is sent from a node it passes the NF_IP_LOCAL_OUT hook where if the packet is to travel into the wireless mesh network the same as at the NF_IP_FORWARD hook occurs. Then the MadWifi driver takes over.

4.2 MadWifi

MadWifi[17], Multiband Atheros Driver for Wireless Fidelity, are drivers for WLAN devices with the Atheros chipset. These drivers are also the most advanced ones available for Linux. The driver themselves are open source but they depend on the Atheros hardware abstraction layer[15] which, for this project, is in binary form. The reason why all the communication between the MadWifi driver[23] and the Atheros card is routed through the hardware abstraction layer is to prevent users from tuning the card to frequencies or transmission powers which violate regulatory restrictions. This generates some problems for the estimation calculation since the last timestamp that can be taken before the packet is sent is just before the packet is transfered to the hardware through the hardware abstraction layer and not when the packet is actually sent.

Changes done in the MadWifi driver[16] are done in the functions in listings 1, 2 and 4. Only the changes made to the functions are shown the rest is cut out. Changes has also been made in the data transmit queue (struct ath_txq) handling. The changes are described further in section 4.2.1.

4.2.1 MadWifi Driver

The changes to the transmit queue handling revolve around the variable axq_totaltime which has been added to the struct ath_txq. This variable contains the approximated total time it will take to send the queue. Each time a packet is added to or removed from the transmit queue the approximated time it will take to send that packet, explained below, is added to or subtracted from axq_totaltime respectively.

Listing 1: IP Header Field Update

```
static int ath_tx_start() {
  /* CUT */
  bf->delayEst=tomppGetDelayEstimation();
  bf->sendEst =
    (bf->approx = 1000 * athff_approx_txtime(sc, an, skb));
  mh_update_outgoing_aphd_opt(skb, (txq->axq_totaltime ?
    txq->axq_totaltime : bf->sendEst/2) + bf->sendEst);
  bf->tomppTimestamp=bf->timestamp=mh_timespec_now();
  /* CUT */
}
```

At the end of the function ath_tx_start() the packet is added to the hardware queue from there on no changes to the packet can be made. Thus, near the end of that function, listing 1, the IP header delay fields are updated using the following formula.

$$d_i = d_{i-1} - t - a - e \tag{4.1}$$

where d_i is the new delay; d_{i-1} is the old delay with the measurement starting time added to it; t is the current time; a is the approximated sending time of this packet and the packets before it in the transmission queue; and e is the current estimated approximation error. Note that TOM does not have an approximation as the queuing time is estimated.

Listing 2: Sending Time Approximation Function

```
static u_int32_t athff_approx_txtime(struct ath_softc *sc,
  struct ath_node *an, struct sk_buff *skb);
```

The function in listing 2 from the MadWifi driver is used to approximate the sending time of a packet. The parameters to the function includes channel configuration (*sc*), node state (*an*) and a packet (*skb*). The function takes the packet length and adds the sizes of the 802.11 encapsulation and cyclic redundancy check, encryption overhead, fast-frame header and padding, and tunneling overheads. It then uses the hardware abstraction layer to calculate the time it will take to send that amount of data with the current state and configuration. The value returned by athff_approx_txtime() is in microseconds so it is converted to nanoseconds as seen on line 5 in listing 1.

Listing 3: Pseudo code of mh_update_outgoing_aphd_opt()

```
mh_update_outgoing_aphd_opt(in: packet, in: approximation) {
  subtract current time from TOM, TOM++ and
    TOMKalman IP header fields
  subtract the TOM estimation from TOM IP header field
  subtract the TOM++ estimation and the approximation parameter
    from TOM++ IP header field
  subtract the TOMKalman estimation and
    the approximation parameter from TOMKalman IP header field
}
```

Packets arriving to an empty queue got low delay values. To compensate for this the approximation for a packet arriving to an empty queue is set to $\beta * calculatedApproximation$ and for other packets it is $calculatedApproximation + axq\_totaltime$. In this project β is set to 1.5. The approximation and estimation are stored in the driver packet buffer (struct ath_buf) and subtracted from the TOM++ IP header delay field. Then the measurement is calculated by subtracting the current time from the delay fields as shown in listing 3, since the time when the packet arrived at the node where added to the delay fields as described in section 4.1. A timestamp, *bf-¿tomppTimestamp*, is also stored in the driver packet buffer which is used for updating the estimation for both TOM++ and TOMKalman. This timestamp starts the measurement for the estimation as seen on line 8 in listing 1. The *bf-¿timestamp* is used by TOM.

Listing 4: Estimation Update

```
static void ath_tx_processq() {
  /* CUT */
  {
    struct ath_buf *next;
    mh_updateCurrentDelayEstimation(bf->timestamp);
    tomppUpdateDelayEstimation(bf->tomppTimestamp, bf->approx);

    next = STAILQ_FIRST(&txq->axq_q);
    if (next != NULL) {
      next->tomppTimestamp=mh_timespec_now();
    }
  }
  /* CUT */
}
```

The estimation is updated in the function ath_tx_processq() in listing 4 which is called by an interrupt when an ACK is received. TOM++ and TOMKalman estimations are measured from when a packet is head of the transmission queue until an ACK for that packet is received. To measure the estimation from when a packet becomes head of the transmission queue the timestamp for the next packet is set to the current time. This is done on line 10 in listing 4. Both TOM++ and TOMKalman estimations are initiated to zero when the driver is loaded into the Linux kernel. TOM++ uses an EWMA filter to update the estimation

$$newEstimation = \alpha * oldEstimation + (1 - \alpha) * measuredTime \tag{4.2}$$

which is located in the function tomppUpdateDelayEstimation() in the file mh_shared.c. This function also calls the function kalmanUpdate() that updates the TOMKalman estimation, described in section 4.3.

Listing 5: Channel Switch

```
static int ath_chan_set() {
  /* CUT */
  tomppReset();
  kalmanReset();
  /* CUT */
}
```

The estimation algorithms are reset when a channel switch is made in the function ath_chan_set(), listing 5. Both TOM++ and TOMKalman are reset to zero, because the state of the new channel is unknown, since no measurements are made for the estimation on the idle channels.

4.3 Kalman Filter

The Kalman filter is the algorithm that updates the estimation for the TOMKalman IP header field, in the function kalmanUpdate() that is located in mh_shared.c. A code library with matrix arithmetics and the Kalman filter calculations were provided online[20] through the tutor of this thesis. The matrix algorithm code uses floating point operations. How this was implemented is described in section 4.4.

The Kalman filter algorithm uses several matrices as described in detail in section 2.4. The matrices B and u are not used since there is no outside controller. Following matrices are initialized as described below.

$$\hat{x} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{4.3}$$

where a is the estimation and b is a state used by the Kalman filter algorithm. Both are initialized to zero.

$$Sw = \begin{bmatrix} \frac{4}{1000} & \frac{16}{1000} \\ 0 & \frac{63}{1000} \end{bmatrix} \tag{4.4}$$

The algorithm adapts faster to change if the values of S_w are low and the delays vary much from packet to packet.

$$P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.5}$$

$$A = \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & 1 \end{bmatrix} \tag{4.6}$$

Matrices P and A was initialized with very simple values since there was not enough time to study the Kalman filter in this project. The solution to this was simply to use a brute force approach to get an initiation that will generate roughly correct values.

The implementation of the Kalman filter was later discovered to be wrong, since there was a scalar version of the Kalman filter which is what this project approach needs. The scalar Kalman filter equations is similar to the normal Kalman filter that had been implemented thus the implementation of the new version went fast and smooth. The scalar

Kalman filter is described in detail in section 2.4.2 and is initialized as following:

$$\begin{aligned} A &= 0.85 \\ S_w &= 100 \\ S_z &= 0.1 \\ C = P = X &= 1 \end{aligned} \tag{4.7}$$

The values chosen in the equations 4.7 are randomly chosen due to the lack of time and knowledge of the Kalman filter.

4.4 Floating-point Operations

Floating-point operations are not supported by the Linux kernel. This means that no floating-point operations can be done in kernel code per default. Furthermore, the ARM architecture used on the mesh nodes does not have a floating-point unit, a coprocessor specialized for floating-point operations[21]. Instead, the ARM Linux kernel has a floating-point emulation software which can be loaded into the kernel as a module to be used in user space programs. This will not allow floating-point operations in the kernel code but it can be modified to do so.

The solution was to compile parts of the floating-point emulation software as a kernel module. All floating-point operations needed resides in the file softloat.c. This file and the header files it relies upon were copied to the working directory, and the symbols for the functions that would be used were exported. Then softfloat.c were added to the project Makefile.

The matrix algorithm code was modified to use the new module floating-point operations instead of the C standard library calls. The floating-point emulator uses packing and unpacking functions to store or load the floating-point values in 32, 64 or 80 bits. This means that the normal C operators for addition, subtraction, multiplication and division cannot be used and neither can normal value assignment. For example a calculation with the ordinary C operators on 64 bit integers may look like this:

Listing 6: Standard C Code Calculation Example

```
a = b + 42;
```

With the floating-point operators it is:

Listing 7: Floating-point Library Calculation Example

```
struct roundingData rd = {float_round_nearest_even , 64, 0};
a = float64_add(&rd , b, int32_to_float64 (42));
```

The first line of the above example contains the struct roundingData. This struct is defined in fpall.h as:

Listing 8: Rounding Data Structure

```
struct roundingData {
    int8 mode;
    int8 precision;
    signed char exception;
};
```

The members of this structure are used as follows.

mode: controls how the rounding of floating-point values should be done. This can take one of the values in listing 9.

Listing 9: Rounding Mode Constants

```
enum {
    float_round_nearest_even = 0,
    float_round_to_zero      = 1,
    float_round_down         = 2,
    float_round_up          = 3
};
```

precision: specifies if the rounding should be with 32, 64 or 80-bit precision.

exception: determines which exceptions should be able to be thrown, if any. The exception member can be set to any of the values in listing 10.

Listing 10: Rounding Exception Flag Constants

```
enum {
    float_flag_invalid   =  1,
    float_flag_divbyzero =  2,
    float_flag_overflow  =  4,
    float_flag_underflow =  8,
    float_flag_inexact   = 16
};
```

The rounding configuration used in the example listing 7 above is used for all floating-point calculations in this project.

The calculation on the second line of listing 7 shows how to do calculations and how to convert integers to floating-point values.

Listing 11: 32-bit integer to 64-bit float Conversion

```
float64 int32_to_float64( signed int );
```

The conversion is done with the function in listing 11. It takes a 32 bit integer as parameter and returns a 64-bit float.

Listing 12: Floating-point Operations

```
float64 float64_add( struct roundingData *, float64 , float64 );
```

```
float64 float64_sub( struct roundingData *, float64, float64 );
float64 float64_mul( struct roundingData *, float64, float64 );
float64 float64_div( struct roundingData *, float64, float64 );
```

Calculations are done with the functions shown in listing 12 which use a common interface of three parameters and a float64 return value. The first parameter is a pointer to rounding data object described in listing 8, the second is the left operand and the third the right operand.

Listing 13: 64-bit Float to 32-bit Integer Conversion

```
signed int float64_to_int32( struct roundingData *, float64 );
```

Listing 14: 64-bit Float to 32-bit Integer Conversion with Rounding to Zero

```
signed int float64_to_int32_round_to_zero( float64 );
```

There are two functions to convert from a 64-bit float to an integer. The first one seen in listing 13 takes two parameters; a pointer to a rounding data object and the 64-bit float that is to be converted. The second function in listing 14 only takes the 64-bit float as a parameter as it rounds to zero.

4.5 Example

This example describes what happens at Node 2 in figure 4.3 when it receives traffic from Node 1 that is to be sent to Node 3.

At A the first packet, P_1, is received and the packet delay is updated.

$$d_{P_1} = m_1 + a_1 + e_0 \tag{4.8}$$

where d_{P_1} is the packet delay; m_1 is the measured time spent in the node; a_1 is the approximated sending time; and e_0 is the estimated approximation error.

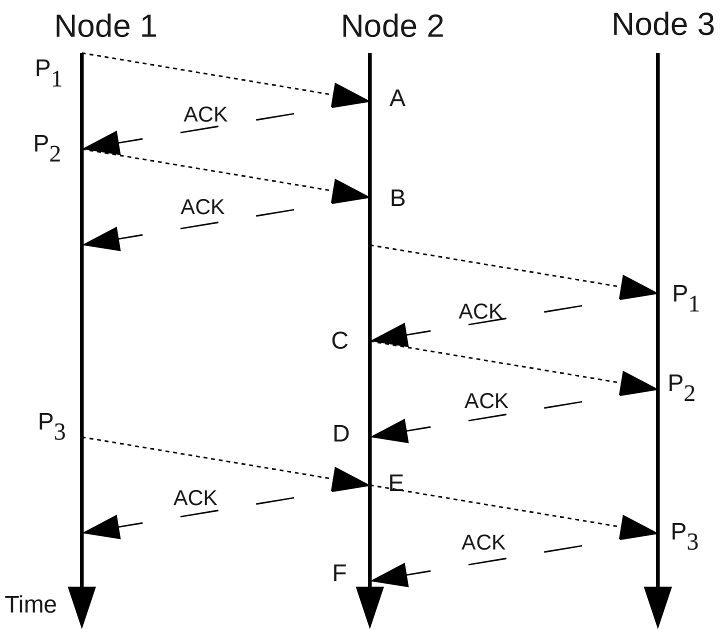

Figure 4.3: Example - Delay Estimation with 2 Packet Queue

At B the second packet, P_2, arrives and its delay is updated as d_{p_1}.

$$d_{P_2} = m_2 + a_2 + e_0. \tag{4.9}$$

Note that the same estimated error, e_0, as in the calculation of d_{P_1} was used.

At C the packet P_1 has been sent and an ACK has been received, thus the estimated approximation error is updated as

$$e_1 = \frac{e_0 + t_1}{2} \tag{4.10}$$

where e_1 is the new estimated error; e_0 is the previous estimated error; and t_1 is the actual time from when P_1 entered the transmit queue to that the ACK was returned.

At D the ACK for packet P_2 has been received and the estimated approximation error is updated.

$$e_2 = \frac{e_1 + t_2}{2} \tag{4.11}$$

At E packet P_3 has arrived and its delay is updated as

$$d_{P_3} = m_3 + a_3 + e_2. \tag{4.12}$$

The estimated approximation error e_2 was used since packet P_2 was the last packet to be acknowledged. If P_3 had arrived between C and D the estimated error e_1 would have been used instead.

At F the ACK for packet P_3 is received and the estimation is updated to

$$e_3 = \frac{e_2 + t_3}{2}. \tag{4.13}$$

5 Evaluation

This section contains an introduction to the evaluation environment (5.1). After that are presentations and discussions of the three test cases that were made; one-channel two-hop scenario without interference; one-channel two-hop scenario with interference; and multi-channel two-hop scenario.

All traffic in the tests are generated by using MGEN[12], Multi-Generator. MGEN is an open source traffic generator. It generates UDP datagrams of configurable size and rate. When MGEN generates a packet it marks it with the local system time. Furthermore, when MGEN receives a packet, it writes the sending timestamp and the receiving time into a log file. To have an accurate measurement of a one-way delay, the clocks at the sender and receiver needs to be synchronized. To synchronize the clocks the start node continuously do NTP updates against the end node via the wired interface. It is done with the help of the script shown in listing 15. The mesh routers are configured to send at a rate of 6Mbps and the packet size for all packets are set to 1000 bytes during all tests. More details on mesh node configuration are given in each test case section.

Listing 15: sync.sh

```
killall ntpd
while true; do
        ntpdate -b 192.168.31.$1 2>&1 >/dev/null
        sleep 1
done
```

For a better understanding of the graph descriptions the terms *offset*, *spike* and *shift* are explained here. When the term *offset* is used, it means the difference between the delay measured by MGEN and the delay measured by the algorithm. The term *spike* is used for delays noticeably higher than the average delay. If the delay curve of the modules shows the same course as the delay curve of MGEN but both curves are out of sync, the packet difference is referred to as *shift*.

TOM++ has been evaluated in the KAUMesh testbed to determine the accuracy of the improved delay measurement, compared to the TOM implementation. This is done on a stream of packets with a normal percent error formula:

$$e = \frac{|\sum_{i=0}^{n}(a_i - m_i)|}{\sum_{i=0}^{n} m_i} \tag{5.1}$$

where e is the percent error of the data stream; n is the total number of packets; a_i is the delay measured by either TOM or TOM++ for packet i; and m_i is the reported MGEN time for packet i. Note though that the actual delay time is smaller than that reported by MGEN and thus these values are not completely accurate but can be seen as a guideline. This is due to that MGEN executes in user space and thus an amount of time is taken to transfer the packet to the kernel.

In this project α in the EWMA filter used by TOM++ is set to 0.5 to get an equal weight between the old estimations and the new estimation measurement. A few tests were done with different values of α but there was not enough time to get any valuable results.

The Kalman filter is not part of this evaluation, since it is not tuned correctly and consequently the results are erroneous. There were both negative and positive offsets of up to several seconds between the Kalman filter graph and the MGEN graph.

5.1 KAUMesh

At Karlstad University there is a wireless mesh network testbed called KAUMesh[5], which is based on 18 mesh nodes that are permanently installed to cover a large area of the newly built House 21. The mesh nodes are based on 802.11a/b/g WLAN, Wireless Local Area Network, but other 802.11b/g based clients can also connect to the wireless mesh network.

KAUMesh is a multi-radio, multi-channel mesh solution that is based on Net-X, suited to run on a high-performance network processor platform and extended by using several extra functionalities. An example of a functionality that is used is that each node has several interfaces. One interface is made for transparent client access, generally by using 802.11b/g mode. The mesh backhaul connection is made by using several radio interfaces per mesh node.

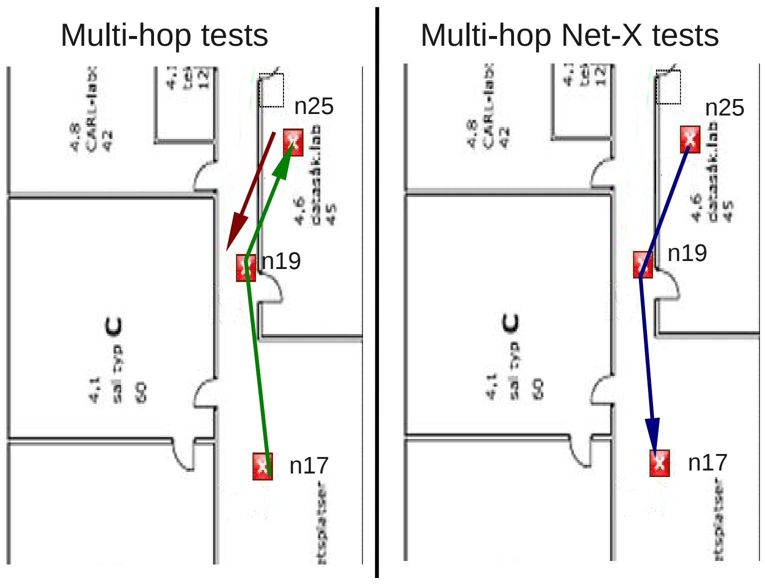

Figure 5.1: Traffic Route for the Tests

The positions of the nodes and how the traffic is routed in these tests are shown in figure 5.1. The following tests includes 3 Gambria GW2358-4 nodes[3], deployed in building 21, that are running the Linux operating system with the customized kernel of version 2.6.22.2.

5.2 2-hop Tests with no Interfering Traffic

This is a simple setup with node 17 sending traffic in one stream to node 25 via node 19, as indicated by the green arrow in figure 5.1. No interfering traffic is generated. Two tests with different transfer rates were performed with this setup. The first test was at 100 packets per second and the other at 200 packets per second, both for a period of 60 seconds.

As seen in the graph in figure 5.2 TOM++ has a lower offset and follows the reported

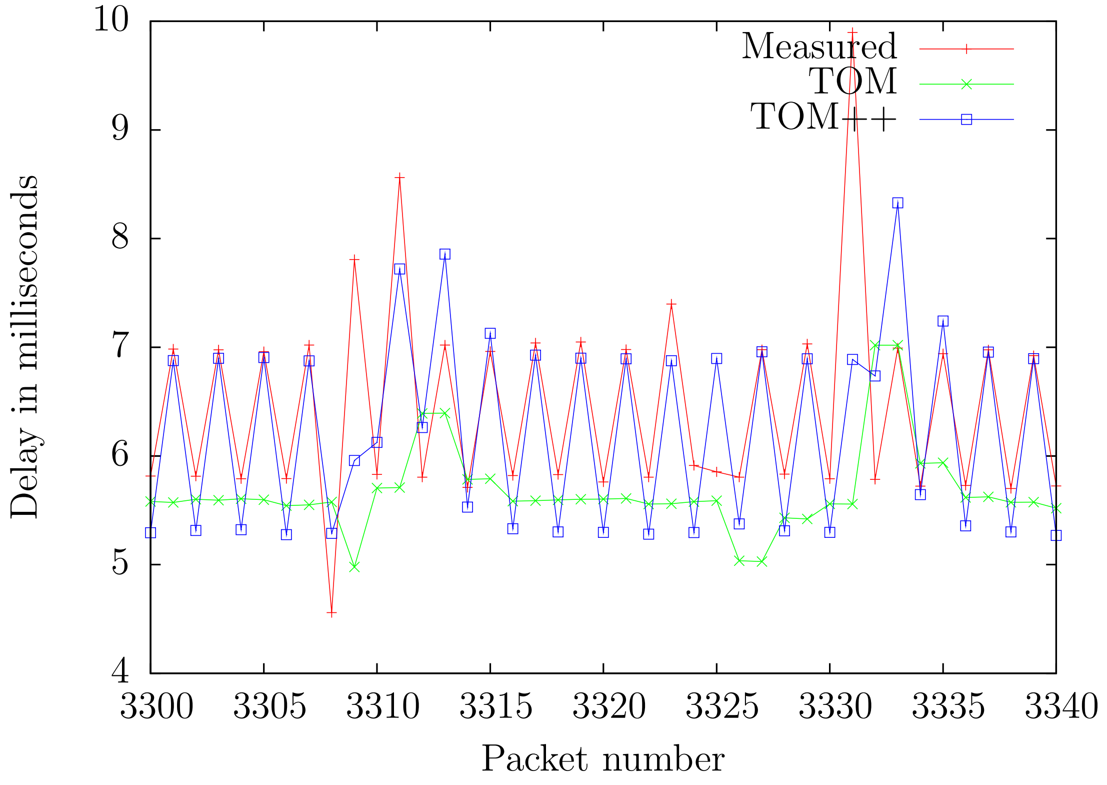

Figure 5.2: 2-Hop 200packets/sec - no Interfering Traffic

MGEN time better than in the TOM algorithm as the queue sending time is included in the TOM++ one-way delay measurement but not in TOM. The spikes that can be seen in the figure are due to delays in the hardware, caused by for example interrupts, DMA data transfer delay or resent packets. This can be seen by the shift of one packet that occurs from the MGEN spike to the TOM++ and TOM responses.

Algorithm	100pkt/s	200pkt/s
TOM	16.4%	12.5%
TOM++	3.5%	5.3%

Table 5.1: Percent Error for 2-hop Traffic with no Interfering Traffic

The results in table 5.1 shows the percent error for the data streams in this test. They show that TOM++ is better than TOM. This is because there is little interference in this test, so the estimation part of the TOM++ delay is almost zero.

5.3 2-hop Tests with Interfering Traffic

For a more realistic scenario these tests are done with interfering traffic. This leads to collisions in the medium which results in increased backoff times and packet resending. In this setup node 17 is sending traffic in one stream to node 25 via node 19 at 100 and 200 packets per second for 60 seconds, while node 25 is generating 140 packets per second to node 19 to get interfering traffic. The interfering traffic is using interface ath2 while the normal traffic is using ath0 but they are sending on the same channel. The green arrow in figure 5.1 is the generated measured traffic and the red arrow is the interfering traffic.

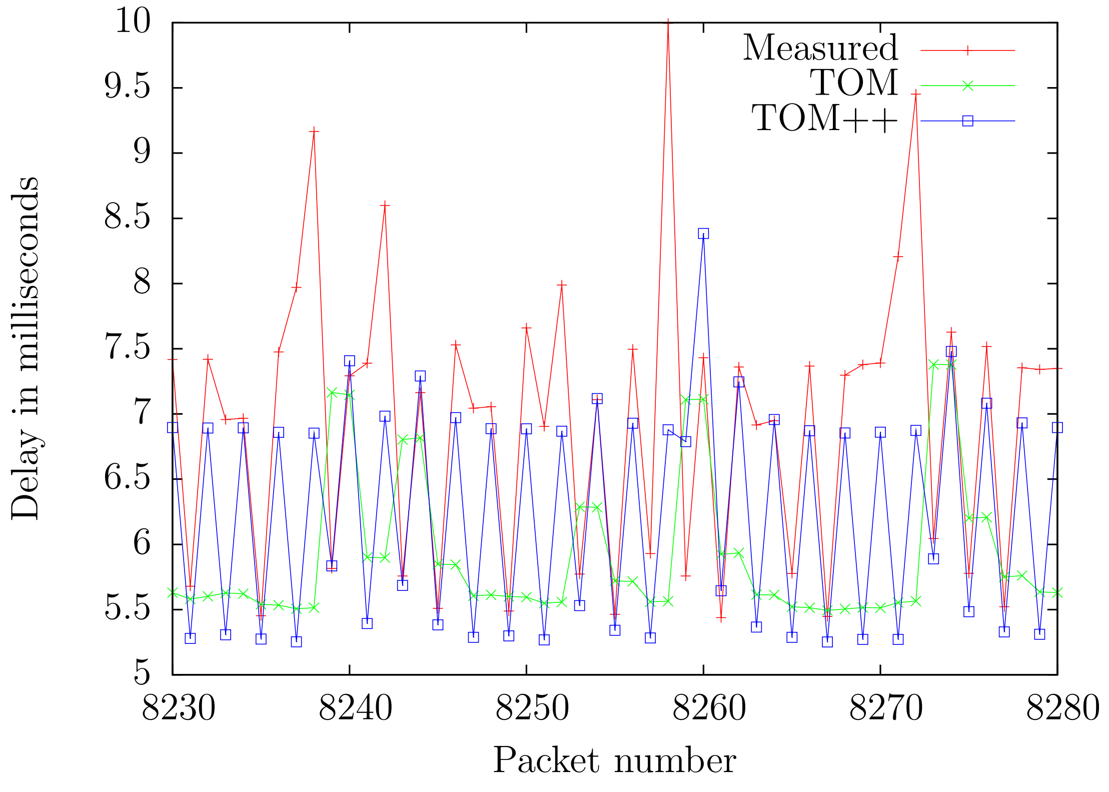

Figure 5.3: 2-Hop 200packets/sec - 140packets/sec Interfering Traffic

The graph in figure 5.3 shows that TOM++ has a lower offset and follows the graph better then TOM.

The one-way delay measurement at packets 8268 through 8271 does not follow the reported MGEN time, this might be because the packet can be stuck in user space where no measurements are made, neither for TOM nor TOM++. This can be seen in other places in the graph but at these packets it is most obvious.

Algorithm	100pkt/s	200pkt/s
TOM	14.8%	17.2%
TOM++	9.8%	11.6%

Table 5.2: 2-hop Traffic with 140pkt/s Interfering Traffic

Table 5.2 shows that TOM++ is 5.0% better than TOM in the 100pkt/s case, and in the 200pkt/s case TOM++ is 5.6% better. In the 100pkt/s case there is rarely any queue buildup at all, whilst there is almost always a transmit queue buildup of two packets, that are consequently transmitted simultaneously, in the 200 pkt/s case. Therefore the TOM uses the same delay estimation in both packets that are put into the queue while TOM++ adds the transmission time of the first packet to the second packet.

5.4 2-hop Tests with Net-X

In this setup node 25 is sending traffic in one stream through node 19 to node 17 with Net-X enabled. All nodes use two interfaces ath0 and ath1. They are set up to listen on a fixed channel on ath0 and to send on two switchable channels on ath1. The specific channels for each node are set up as described in table 5.3. The blue arrow in figure 5.1 represents the traffic from node 25 to node 17. This is set up to see how the delay estimations react to the channel switches that Net-X perform. Channel switches are forced by sending broadcast messages on node 19. The minimum time to stay on a channel after a channel switch is set to 25ms and the maximum time is set to 60ms.

As can be seen in table 5.3 node 25 will be sending on channel 44 when sending the test stream and node 19 will be sending on channel 36, these channels will be called the test channels. When the broadcast is sent the nodes switch to every switchable channel and send the broadcast message on them. The channels not used for the test stream are 64 and 36 for node 17 and 25 respectively. These channels will be called the alternate channels.

	Channel		
		Switchable	
Node	Fixed	Test	Alternate
17	36	44	64
19	44	64	36
25	64	36	44

Table 5.3: Net-X Test Node Channel Setup

Two tests were performed with this setup as well. In the first test the stream was sending at a rate of 50 packets per second, and then 100 packets per second in the second test.

The three spikes in figure 5.4 occurs when there is a channel switch from the test channel to the alternate channel.

The last of these spikes represents a channel switch on node 19. Only packet 3270 arrives at node 19 while it is sending on the alternate channel during this channel switch, so this packet is put into the bonding queue until the node switches back to sending on the test channel. The time spent in the bonding queue counts as a part of the measurement described in section 4, thus the time reported by both TOM and TOM++ will be near accurate for that packet.

Packet 3271 arrives when node 19 has started to transmit on the test channel again. The delay measurements for this packet is not nearly as accurate. The reported MGEN time is about 6.5ms greater than the TOM++ and almost 9ms greater than the TOM delay. This might be because of delays in the hardware after the channel switch since this would not be covered by the measured or the approximated part. The figure also

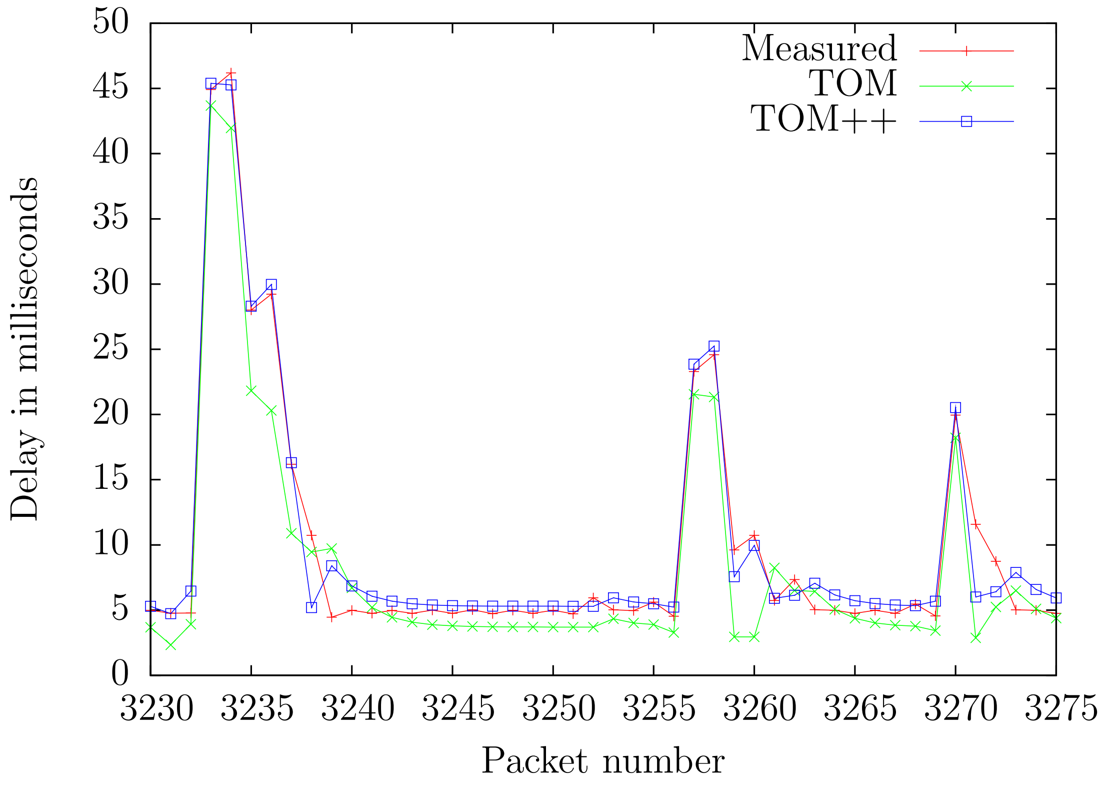

Figure 5.4: 2-Hop 100packets/sec Net-X

suggests that packet 3271 waits in the hardware transmit queue for a brief time since the TOM++ delay is 3.1ms higher than the TOM delay and the increase in delay to packet 3272 is greater in TOM than in TOM++. The reason the TOM delay increase is higher might be because the TOM estimation includes the hardware transmit queue time while the TOM++ estimation only includes the time from when the packet becomes head of the queue until the acknowledgment that it has been sent is returned. At packet 3274 the channel has started stabilizing after the channel switch.

The second spike is a channel switch in node 25. Here two packets, 3257 and 3258, arrive while node 25 is sending on the alternate channel. Packet 3259 arrives when node 25 has switched to the test channel again and therefore it is put directly into the transmit queue with one or two packets ahead of it in the queue as shown by the slight increase of delay relative to the normal delay of about 5ms.

At the first spike both node 25 and node 19 switches channel before packet 3233 resulting in a total delay of almost 45ms at packet 3233. Both TOM and TOM++ keep up with this rapid change since the packets mainly reside in the bonding queue. Node 25 is sending on the alternate channel for two packets thus at packet 3235 the delay drops. However, node 19 is still transmitting on the alternate channel until packet 3237 has arrived.

As seen in the area between the spikes, the TOM++ delay is greater than the reported MGEN time. This is because there is no queue buildup and thus the approximation for the packets are 50% higher which is explained in detail in section 4.2.1.

As described above in section 4.2.1 the estimation for TOM++ is reset at each channel switch thus the packets in the spikes does not get an estimation. This proves to be good as the TOM++ delay gets close to the reported MGEN time. However packets 3238, 3259 and 3271 after each spike has a lower value compared to the reported MGEN time which can be because the TOM++ estimation part was reset to zero during the channel switch.

Algorithm	50pkt/s	100pkt/s
TOM	20.5%	22.0%
TOM++	11.2%	8.8%

Table 5.4: 2-hop Traffic with Net-X

From table 5.4 can be read that TOM++ is 9.3% and 13.2% better than TOM in the 50pkt/s and 100pkt/s cases respectively. The reason the TOM++ measurement improves with the transmission rate is because more packets are received while the nodes are sending on the alternate channel.

6 Conclusion

This thesis presented a way to improve the TOM algorithm to be more accurate with little added system load and a prototype of a Kalman filter.

The improvements from the TOM algorithm to the TOM++ algorithm is the calculated approximation of the transmission queue and packet sending times and that the estimated approximation error is reset at each channel switch. This was done because the TOM algorithm was too inaccurate and estimated the above mentioned sending times. Therefore, by calculating the sending time it is only the backoffs and packet resends left to estimate which leads to less inaccuracy in the one-way delay measurement. This improvement resulted in an increased accuracy of about 5% as described in section 5.3, which is the most realistic case of the tests.

The Kalman filter prototype was implemented to see if it could replace the current estimation process, since these types of filters are good at predicting states. There was not enough time in this project to finish the implementation of the Kalman filter and thus a conclusion to whether it can replace the estimation which is now used could not be made. In section 6.1.1 there are some thoughts on how to further develop the Kalman filter prototype.

The difficulties in doing this has been the hardware abstraction layer and the implementation of the Kalman filter prototype. The hardware abstraction layer prevents changes to the packets after transferring them to the transmission queue. The problems with the Kalman filter prototype was that the original version of it used floating-point operations which the Linux kernel does not support and that the Kalman filter is complex and therefor takes time to fully complete.

6.1 Future Work

There is room for further development in this project. Here are some ideas.

6.1.1 Kalman Filter

The Kalman filter was first implemented and tested as a discrete Kalman filter. It was later discovered that it was the wrong algorithm for this projects approach. Then the scalar Kalman filter was implemented and tested but there was not enough time to tune the filter correctly nor to make proper tests. The current implemented version needs to be tuned to work properly.

Another improvement might be to change the current implementation from using floating point calculation to fixed point. By doing so there might be some time to gain on the packing and unpacking that is described in detail in section 4.4.

6.1.2 Low Delay Compensation

The time added for the first packet in the transmit queue, $1.5 * approx$, is not correct since that calculation is based on the reported MGEN sending time. It includes the time it takes to transfer the packet data from user space to the kernel. It should not be included in the one-way delay calculation since this represents the transfer time before entering the wireless mesh network.

6.1.3 Improved Tests

The tests done in this project are done by synchronizing the clocks on the nodes with NTP, this can be done by using GPS instead which is more accurate. If the clocks are better synchronized the final MGEN delay measurement time will be more accurate, if the MGEN time is more accurate the calculations to get the accuracy for the algorithm will also be more accurate.

Another thing which can be improved on the tests are to calculate the average time it takes for MGEN to send a packet from user space to the kernel, since this time should not be included in the total one-way delay calculation. This time then needs to be removed from the final reported MGEN time two times, since it needs to travel from kernel to user space on the receiver node.

6.1.4 Channel Switch Reset

When a channel switch happens the delay for TOM++ and TOMKalman are reset to zero in the current implementation, the question is if it should be reset to zero or another value. In the results from the tests it is hard to tell if this should be changed or if it should stay at zero, since the new channel a node is switching to in the tests did not have any traffic at all. A more realistic scenario would be to set up a larger Net-X tests with more nodes and traffic on several channels, so when a node switches channel the new channel will have initial traffic.

6.1.5 Intra-node Measurement

The intra-node measurement starts when the packet enters the node and stops when the packet is put into the hardware transmit queue. If there is a way to alter the packet while it resides in the hardware queue the measurement could be extended to include the queuing time which would increase the overall one-way delay estimation. There might be some way to do this through the Atheros hardware abstraction layer or a more direct approach with later versions of the MADWifi driver since they use an open source version of the hardware abstraction layer.

References

[1] Moving average. http://en.wikipedia.org/wiki/EWMA#Exponential_moving_average, Jan 2010. Exponential Weighted Moving Average.

[2] Erik Cheever. Scalar Kalman filter. `http://www.swarthmore.edu/NatSci/echeeve1/Ref/Kalman/ScalarKalman.html`, Jan 2010. Description of how the scalar Kalman filter works.

[3] Gateworks Corporation. Gateworks, Cambria GW2358-4. `http://www.gateworks.com/products/cambria/datasheets/gw2358-4ds.pdf`, Dec 2009. Product manual for the KAUMesh nodes.

[4] Gupta D., Wu D., Mohapatra P., and Chen-Nee Chuah. Experimental comparison of bandwidth estimation tools for wireless mesh networks. *Proceedings - IEEE INFOCOM*, pages 2891–2895, Apr 2009. The 28th Conference on Computer Communications. IEEE.

[5] Peter Dely and Andreas Kassler. Kaumesh a multi-radio multi-channel mesh testbed. *Proceedings of 9th Scandinavian Workshop on Wireless Ad-hoc & Sensor Networks*, 2009.

[6] Draves et. al. Comparison of routing metrics for static multihop wireless networks. *ACM*, 34(4):133, 44, 2004. M1: Copyright 2005, IEE; T3: Comput. Commun. Rev. (USA).

[7] Kapoor et. al. Capprobe: a simple and accurate capacity estimation technique for wired and wireless environments. *ACM*, 32(1):390, 2004. M1: Copyright 2005, IEE; T3: Perform. Eval. Rev. (USA).

[8] Sun et. al. Adhoc probe: end-to-end capacity probing in wireless ad hoc networks. *Kluwer Academic PUBLISHERs*, 15(1):111–126, Jan 2009. Compilation and indexing terms, Copyright 2009 Elsevier Inc.

[9] Mohinder S. Grewal and Angus P. Andrews. *Kalman filtering: Theory and Practice Using MATLAB*. John Wiley & Sons, 2008.

[10] IEEE. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE Standard, Jun 2007. Revision of IEEE std 802.11-1999.

[11] Pradeep Kyasanur, Chandrakanth Chereddi, and Nitin H. Vaidya. A wireless networking framework providing system extensions for supporting multiple channels, multiple interfaces, and other interface capabilities. http://www.crhc.illinois.edu/wireless/netx.html, Aug 2006. Technical Report.

[12] Naval Research Laboratory. Multi-generator. http://cs.itd.nrl.navy.mil/work/mgen/, Dec 2009. Description of how MGEN works.

[13] Jian Lia, Zhi Lia, and Prasant Mohapatra. Adaptive per hop differentiation for end-to-end delay assurance in multihop wireless networks. *Ad hoc networks*, 7(6):1169–1182, 2009.

[14] Hyuk Lim1, Kyung-Joon Park, and Chong-Ho Choi. Stochastic analysis of packet-pair probing for network bandwidth estimation. *Aug*, 50(12), 2006. ISSN: 1389-1286.

[15] The MadWifi Project. Hardware abstraction layer. http://madwifi-project.org/wiki/About/HAL, Oct 2009. Description of how the HAL in MadWifi works.

[16] The MadWifi Project. Madwifi driver. http://madwifi.org/, Sept 2009. Description and code for the MadWifi driver.

[17] The MadWifi Project. Madwifi project. http://madwifi-project.org, Sep 2009. Description of the MadWifi project.

[18] MeshDynamics. Wireless mesh network generations. http://www.meshdynamics.com/mesh-network-technology.html, Jan 2010. Describes the three WMN generations.

[19] J. Postel. Internet protocol. RFC 791 (Standard), Sep 1981. Updated by RFC 1349.

[20] Dan Simon. Kalman filtering. http://www.embedded.com/story/OEG20010529S0118, Oct 2009. The code used in the Kalman implementation.

[21] Robert Bruce Thompson and Barbara Fritchman Thompson. *PC Hardware in a Nutshell*. O'Reilly & Associates, Inc, 2003.

[22] Yan Zhang, Jijun Luo, and Honglin Hu. *Wireless Mesh Networking: Architectures, Protocols and Standards*. Taylor & Francis Group, 2007.

[23] Wenhau Zhao. Madwifi driver summary. http://mesh.calit2.net/whzhao/madwifi_summary.pdf, Aug 2007. Describes the MadWifi structure.