Channel Switching Overhead for 802.11b

Michael Harris <mharris@siu.edu>, Sarah Harvey <shharvey@siu.edu>

***Abstract*—IEEE 802.11b devices operate in the 2.4 GHz ISM band the United States. Because of this, 802.11b devices are susceptible to interference from a multitude of sources such as microwave ovens and cordless phones. Interference from other devices can lead to transmission failures and degraded performance. Wireless devices may be configured to communicate on different channels at different times as necessary to mitigate the effects of interference. However, the determination of the best possible channel at any given time, and the act of channel switching, introduces a delay. This delay can lead to certain transport-level protocols failing and restarting (i.e. TCP streams timing out, thus disconnecting). Furthermore, the 802.11b protocol does not specify a mechanism by which channel switching may be coordinated between communicating devices. Without such a mechanism, communication would be interrupted until both devices are again communicating on the same channel. We propose to design a protocol by which channel switching behavior may be coordinated between communicating 802.11b devices. This protocol will be implemented using open source drivers on Linux, and specifically target devices with drivers that work with the mac80211 subsystem present in the Linux kernel. We will measure the overhead incurred by channel switching and suggest methods by which this effect may be mitigated.**

I. Assignment

In this project, using open source drivers for 802.11b, the channels will be changed on the fly and the change time overhead will be recorded when two wireless LAN cards are talking to each other.

II. Plan of Execution

A protocol is required to coordinate channel switching behavior between two communicating nodes. Our ultimate objective is to design and implement this protocol, to take measurements of the time overhead incurred by channel switching, and to derive the best possible solutions by which this may be mitigated.

Our basic plan of execution consists of the following steps:

1) Design a channel switching technique to be integrated in the existing IEEE 802.11 MAC protocol.
2) Standardize our operating environment. This will entail the act of determining and documenting the configuration requirements of a kernel on which our protocol may be implemented.
3) Begin development of a proof-of-concept program using `nl80211` to implement the channel switching protocol.
4) Test the protocol and record the time overhead incurred from the channel switching behavior.
5) Evaluate our findings and make changes as necessary.
6) Present a report with our findings.

III. Assumptions

A. Wireless devices, drivers, and firmware

We focused our testing only on 802.11b devices that depend on the `mac80211` framework present in the Linux kernel. Our available resources limit us to testing only the `iwlcore` drivers (supporting Intel Wireless WiFi chipsets) and `p54usb` drivers (supporting Intersil's Prism54 chipset) [3].

Two of the laptops in the experiment made use of the Intel Wireless WiFi 3945ABG Wireless miniPCI card supported by `iwlcore` & `iwl3945` [4].

One of the laptops in the experiment made use of a Dell Computer Corp. Wireless 1450 Dual-band USB 2.0 [8] device with the ISL3887 chipset. This device required external firmware to be loaded. This was supported by `p54` & `p54usb`.

For monitoring 802.11 communications, two additional laptops were used making use of Intersil's Prism2.5 chipset. These were supported with the `orinoco` and `hostap` drivers, however due to limitations of the firmware, no custom packets could be sent without severe modification of both the driver and the firmware, thus these devices were used only to verify the correctness of the custom protocol from an outside viewpoint.

B. Operating environment

We are relying on `mac80211`/`nl80211` framework API due to its modular design and level of abstraction [2]. Arch Linux [7] and Slackware Linux [6] were chosen as testing environments due to their technical simplicity and lack of unneeded bloat. This is crucial as many contemporary desktop Linux distributions bundle in a variety of utilities for configuration (some graphical), and the effects of any external applications on these devices, directly or indirectly, must be minimized.

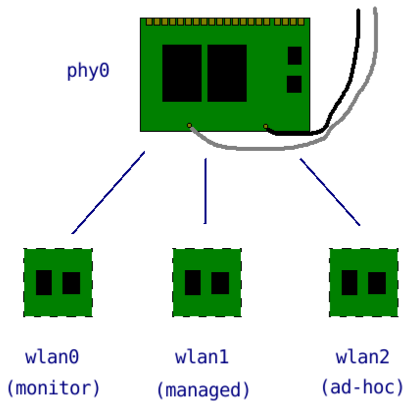
C. Timeline

We have allocated 7 weeks of work for this project. The timeline for our project is as follows:

- (10/18 - 10/31) The first 2 weeks will consist of the initial protocol design.
- (11/01 - 11/07) The next 1 week will be spent standardizing our operating environment.
- (11/08 - 11/28) The next 3 weeks will be spent doing protocol implementation/module development and testing.
- (11/28 - 12/07) The next 1 week will be spent aggregating our data and preparing a final report.

Last modified December 12, 2009

Fig. 1. Conceptual representation of `mac80211` framework



IV. MAC80211/CFG80211/NL80211

The `mac80211`/`cfg80211` stack is a set of modules that attempt to move the management of IEEE 802.11 frames out of proprietary firmware and into some place more accessible, namely kernelspace or userspace. This serves as an abstraction layer to which both the hardware-level driver and the software-level kernelspace or userspace can access, and thus can interact each other without making use of direct low-level system calls (as done with `ioctl` in Wireless Extensions). As the IEEE 802.11 standard is continually evolving, it is vital that the software changes can keep up with the latest specifications, as to be compliant with regulations. Furthermore, the prevalence of SoftMAC devices in the consumer market continue to prove the point that it is both cheaper and easier to make changes in the software stack on barebone, skeletal hardware as opposed to continually revising hardware design of chipsets. Direct results of this effort have resulted in implementation of 802.11s (mesh networking) across all `mac80211` devices, in addition to generic implementations of WPA, WPA2, IBSS (ad-hoc networking) etc.

`nl80211` is a set of tools/libraries in userspace that allow for direct interaction with `mac80211`. One of the most obvious implementations is the `iw` package, which essentially serves as a replacement for Wireless Tools. One of the more interesting points to note is the fact that `mac80211` imposes a new scheme for managing wireless interfaces. Each wireless device is represented as a `phy` physical device, from which one or more virtual interfaces may be created. The `phys` are never seen as Ethernet interfaces (e.g. via `ifconfig`); only the virtual interfaces are technically exposed to the rest of the system without `nl80211`. Although there are physical limitations to this scheme (all virtual devices must be on the same channel, etc.), this does allow for the creation and manipulation of additional interfaces to the same device without necessarily disrupting any concurrent communications on wireless interfaces, making sharing of resources possible across a variety of applications.

V. PROTOCOL DESIGN

Efficiently measuring the time between channel switches mandated the creation of a new temporary protocol. The basic scheme of the protocol involves a node or station continually gathering data on communication latency (low-level), noise, signal strength, and (link) quality. Should it find that any or all of these suddenly deviate for the worse from the average values, then it sends a channel switch request to either its neighbors (ad-hoc/IBSS) or to the host access point (infrastructure/BSS). Based on the situation of the other nodes, the host AP or ad-hoc network will either approve or deny the request.

Upon denial, the node waits for some contention period while continuing to observe its communication situation, and should the situation continue to be worse, it will make another channel switch request. Upon approval, the node then switches to a designated channel defined by the ad-hoc network or host AP, and attempts to reauthenticate with the network.

The nature of the IEEE 802.11 protocol forced us to consider channel switching in both station/AP and ad-hoc network situations. As such, we have divided up the different test cases in which we consider channel switching viable based on whether the node is connected to an access point, or is part of a decentralized ad-hoc network.

A. *Channel Switching in infrastructure networks*

The basic protocol in this situation is as follows:

- Data communication
- One or more nodes detect interference (increase in latency, decrease in signal strength, etc.)
- One or mode nodes send CHSW_REQ packet
- AP responds with an CHSW_ACK, signaling to all stations in vicinity that they need to switch
- Measurement of channel switch begins, measured by node A
- AP forcibly disassociates all connected nodes, and resets its buffer
- AP switches channels; nodes switch channels
- AP is ready for reassociation
- Nodes reconnect to AP
- Measurement of channel switch ends

From here, we can divide it up into several test cases in which the protocol would be applicable.

1) 1 station, 1 AP: In this case, as node A is the sole client served by AP, any requests for channel switching are granted immediately.

2) Multiple stations, 1 AP: In this case, the access point must have some way of determining when channel switching should occur, given the fact there are more than two nodes connected at any given time. In our protocol, we determined that the AP should only grant a channel switch once more than 50% of the nodes make a channel switch request within a specified timeframe, e.g. 10 seconds.

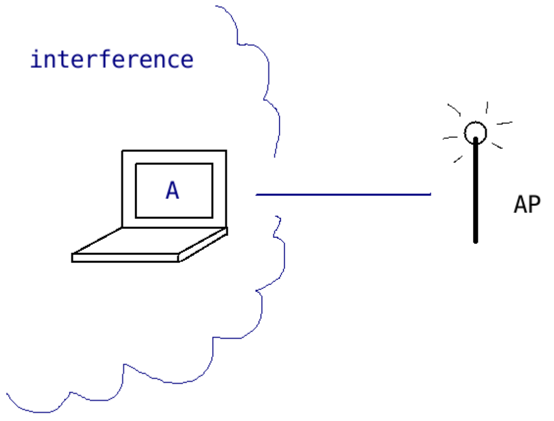


Fig. 2. One node, one AP

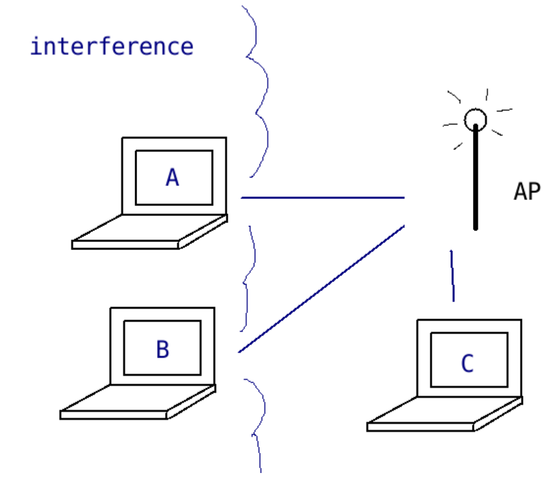


Fig. 3. Three nodes, one AP

B. Channel Switching in Ad-hoc Networks

Channel switching in ad-hoc networks[1] becomes much harder to coordinate as there is no centralized structure to the network. As such, there is no easy way to figure out when it becomes viable to make a channel switch, as it is not possible to keep track of how many nodes are suffering from interference at any given time, or is it possible to switch any region of nodes, as that would fragment the network. Our solution to this problem is through the designation of temporary "virtual APs" to coordinate the channel switching for the entire network. By adding a centralization element to a decentralized network, we can thus easily apply the above protocol with minor modifications.

The challenge then is how to determine the virtual APs in an ad-hoc network that would be able to coordinate a channel switch for the entire ad-hoc network. This is not an easy task. We can only devise potential solutions to this problem, and hope that future research yields better results.

[1]Also known as IBSS

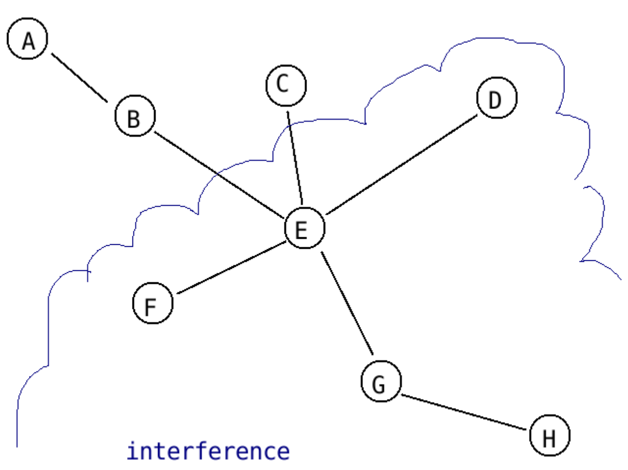


Fig. 4. Sample ad-hoc network

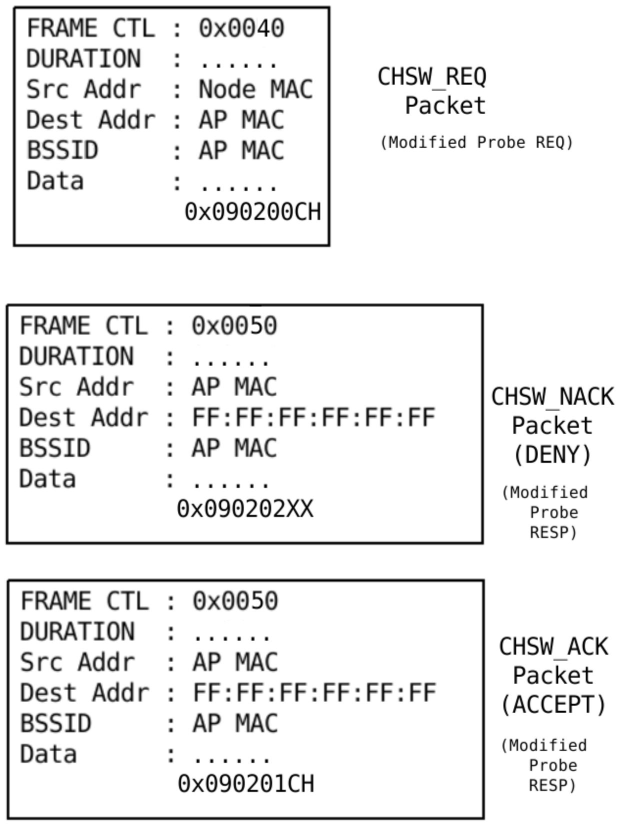
One basic protocol in this situation would be as follows:

- Data communication
- Node H detects interference (increase in latency, decrease in signal strength, etc.)
- Node H polls all other nodes to determine a suitable temporary AP
  - This is chosen as node with most links, and closest (by number of hops) to client
- Act of polling updates all other nodes' CHSW tables
- If node H is aware that other nodes wish to switch, it sends CHSW_REQ to E, the virtual AP
- E responds with an CHSW_ACK, signaling to all stations in vicinity that they need to switch
- CHSW_ACK is propagated through the network
- Measurement of channel switch begins, measured by node H (upon reception of CHSW_ACK)
- All nodes disassociate, reset their buffers
- All nodes switch channels, prepare for reassociation
- Nodes reconnect
- Measurement of channel switch ends

Unfortunately due to time and resource constraints, we were unable to fully test this protocol. The current implementation involves channel switching overhead only between two ad-hoc nodes.

VI. Implementation

Central to our protocol design was the creation and/or modification of two existing IEEE 802.11 management frames to facilitate coordination of channel switching. While it was preferable to have two dedicated frame types for the implementation of this protocol, due to limitations of wireless hardware this is simply not possible; we found that custom frames would either be dropped or generate errors in the software whenever we attempted to queue them for transmission. Instead, Probe Request/Response frames were modified by appending an additional 4 bytes containing channel switch protocol information (reserve byte, size byte, REQ/ACK/NACK byte, channel byte) to the data portion of the frame, as to



```
FRAME CTL : 0x0040
DURATION  : ......
Src Addr  : Node MAC
Dest Addr : AP MAC
BSSID     : AP MAC
Data      : ......
           0x090200CH
```

```
FRAME CTL : 0x0050
DURATION  : ......
Src Addr  : AP MAC
Dest Addr : FF:FF:FF:FF:FF:FF
BSSID     : AP MAC
Data      : ......
           0x090202XX
```

```
FRAME CTL : 0x0050
DURATION  : ......
Src Addr  : AP MAC
Dest Addr : FF:FF:FF:FF:FF:FF
BSSID     : AP MAC
Data      : ......
           0x090201CH
```

Fig. 5. Modified Probe REQ/RESP to become CHSW REQ/RESP

remain compliant with the existing IEEE 802.11 protocol. Thus any non-channel-switch capable AP or node could safely respond and/or discard the packet as the CHSW data could be regarded as extraneous. We thus created/modified programs to listen for these frames and initiate the channel switch via `nl80211`.

Driver implementations already include mechanisms to access noise levels, signal levels, and link quality, such as that through wireless-spy (`iwspy`) or `radiotap` headers. To measure latency, we made use of a built-in mechanism in the 802.11 protocol, namely Probe Request/Response. At various intervals, a burst of 30 or so Probe Request packets would be sent, and upon reception of Probe Response packets, minimum, maximum, and average latencies could be calculated. As this burst of activity generally floods the network, it is almost always guaranteed to have slightly higher latencies than regular traffic unless the network happens to be extremely saturated.

Ultimately the software portion of this project consisted of 2-3 components: modified `hostapd` (Host AP Daemon), and a custom proof-of-concept program `inject`, one making use of `nl80211` (requiring `mac80211` drivers), and the other making use of Wireless Extensions.

A. *Modified* `hostapd`

HostAPD [5] is a program which replicates the functions of dedicated access points through the use of consumer hardware. As of now, it supports the following drivers: `prism54`, `madwifi`, `hostap`[2], wired, and `nl80211`. For the purposes of this project, we made use of the `nl80211` driver as it is generic and can thus interface with a variety of different hardware, providing the hardware supports interfacing with `nl80211`/`mac80211`, thus it was fairly trivial to modify HostAPD in order to add recognition for CHSW_REQ and CHSW_ACK.

When using the `nl80211`/`mac80211`, HostAPD makes use of two virtual devices on the physical `phy`. The first device, e.g. `wlan0` is the original virtual device that is set to Master or AP mode to respond to clients. A second device is created, e.g. `mon.wlan0` to monitor and thus capture any packets coming in to the device. This provides a fairly high amount of control as each device has a dedicated task.

B. `inject`

`inject` is a proof-of-concept program that implements dynamic channel switching either in Ad-hoc mode[3] (with another node also running `inject`), or in Infrastructure mode with a modified `hostapd`. Upon startup, the program will:

- Determine the physical `phy` used for managing the virtual wireless interfaces
- Create a monitor interface for directly capturing 802.11 frames
- Send bursts of probe requests to measure latency
- Gather data on signal level, noise level, quality, etc.
- Implement the node-end of the channel-switching protocol.

C. *Implementation in non-*`mac80211` *drivers*

Implementation of dynamic channel switching was explored in other non-`mac80211` drivers (e.g. `orinoco` and `hostap`) however these drivers proved hard to modify as frame management was done in firmware as opposed to the actual driver itself. This makes it extremely hard to be able to make any sort of trivial modification to frame management without having any prior knowledge of how to modify the firmware. As such, we made use of userspace `inject` to control and interact with this device simply in monitor mode. It should be noted that this is not reflective of real-world situations, as the driver does not have support for concurrent modes on a single device, thus there is no way to do external monitoring without disrupting existing communications. In the end, while it was possible to monitor all packets using either `orinoco` and `hostap`, the use of firmware essentially made it impossible to use `inject` without severe modification (i.e. using hardware-specific system calls, modifying the firmware, etc.), which would remove all aspects of generality within the project.

[2]It is important to note the difference between `hostap` and `hostapd`; `hostap` is a driver that provides Master or AP mode capability in Prism/2.5/3 devices, while `hostapd` is the actual daemon userspace program that serves as the AP software in Linux.

[3]Full IBSS capability in `mac80211` is present in Linux kernels 2.6.30+

D. Implementation in `mac80211` drivers

The design of `mac80211` is such that there is no theoretical bound to how many virtual devices may exist on a single `phy`. As such, it is fairly easy to create an additional virtual device to monitor communications to and from the node without disrupting existing communications. Thus, the version of userspace `inject` in this implementation made use of `nl80211` functions exclusively to create a virtual monitoring device, listen to that device for the appropriate CHSW frames, and initiate the channel-switching protocol as necessary.

VII. Results

Results of experimentation and implementation of the protocol may be divided into the following sections:

- Initial implementations
- Case study: One station, one AP
- Case study: Two nodes in Ad-hoc

A. Initial Implementations

It is interesting to note that initial testing to verify that channel switching worked caused a number of interesting problems to surface. Running `inject` with one station and one AP with constant channel switch requests/responses (every second or so) revealed a randomized problem where the two devices would appear to get "stuck" while in the middle of a channel switch. This would be the case regardless of whether the channel switches were sequential or randomized. Further analysis revealed that occasionally, the CHSW_ACK would be lost in transit to the station, thus the AP would switch channels and listen on the new channel, while the station was stuck waiting for the CHSW_ACK to verify initiate the switch. One potential explanation for this is derived from the inherent nature of the implementation of drivers in Linux: once data is sent to a device, there is no guarantee that the device will actually transmit the data in a timely fashion, i.e. control of the packet/frame is at that point, exclusively under the control of the physical device. It was found that a combination of sending multiple CHSW_ACKs on approval and only a single CHSW_NACK, and microsleep after queueing each frame for transmission, mitigated this effect. It is also probable that at the time, constant rapid channel switches would cause the device to reset, thus clearing its buffer before the CHSW_ACK would actually be transmitted. It is also hypothesized that a greater difference in channel switch (5 channels or more) may take slightly longer (in terms of microseconds) to process in the device, than a smaller difference (4 or less).

VIII. One station, one AP

The first case study performed was measuring the channel switch made between one station connected to one AP. In this case, a channel switch request is sent by the station to the AP, and based on the AP's status, it either approves or denies the request. The initial implementation mandated strict channel-switch policies in the event that several stations would be connected to the AP at once. In such a case, the AP would examine its local list of authenticated and associated stations, determine how many of them recently made a channel switch request, and if 50% or more of the stations have made the request within a specified recent contention period, the AP would approve the request by broadcasting a CHSW_ACK. If less than 50% of the stations fall within that contention period, then the AP would send a specified CHSW_NACK to the node who initially requested the switch.

While this implementation would be useful for real-world situations, it proved to be impractical for our tests. As such, a benevolent channel-switch policy was implemented in addition to the strict policy in `hostapd`, in the form of a macro flag to trigger "friendly" mode. This policy essentially allowed for constant channel switches requested by any node at any time. For each type of test, 100 channel switches were made, to mitigate the effect of any outliers present in the data. Data collection was made on channel switch overhead with and without the CHSW protocol devised above.

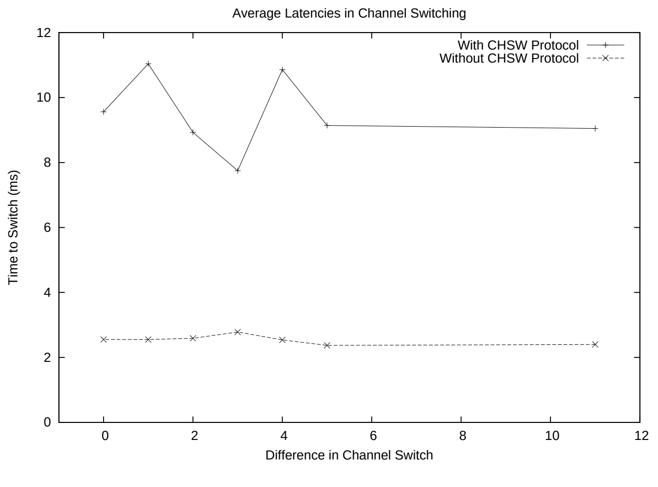
A. Data

TABLE I
Channel Switch Overhead with CHSW Protocol

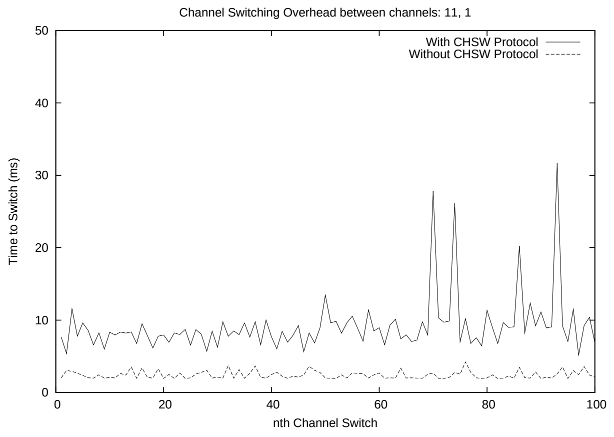
Channel Jump	Avg Latency (ms)	Std Dev (ms)
11	9.05	4.05
5	9.14	3.62
4	10.86	5.03
3	7.75	2.89
2	8.93	3.46
1	11.04	7.50
random	9.56	6.07
AVG	9.48	4.66

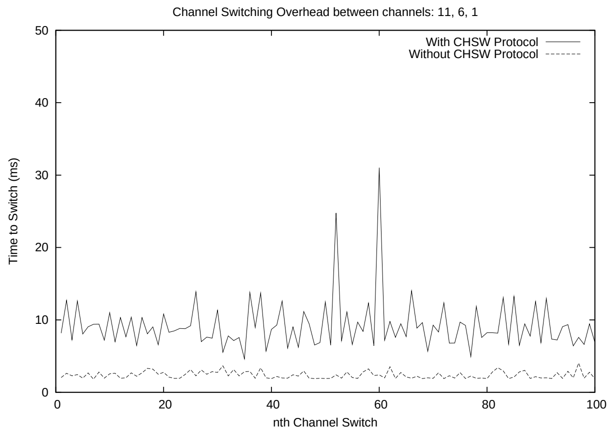
TABLE II
Channel Switch Overhead without CHSW Protocol

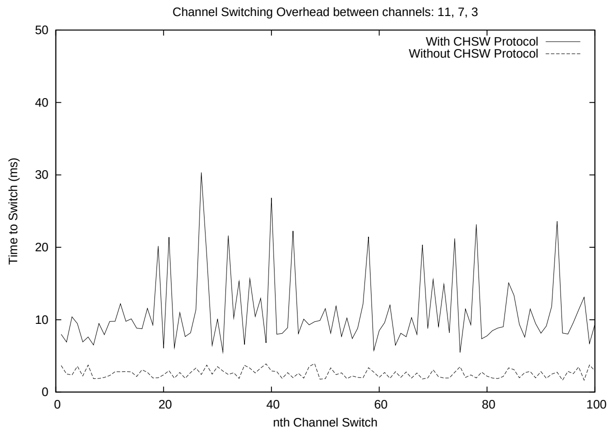
Channel Jump	Avg Latency (ms)	Std Dev (ms)
11	2.40	0.58
5	2.37	0.55
4	2.54	0.65
3	2.78	0.77
2	2.59	0.72
1	2.55	0.72
random	2.55	0.65
AVG	2.54	0.66

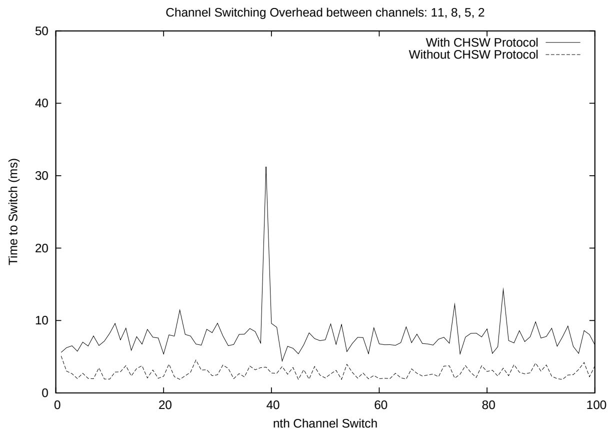


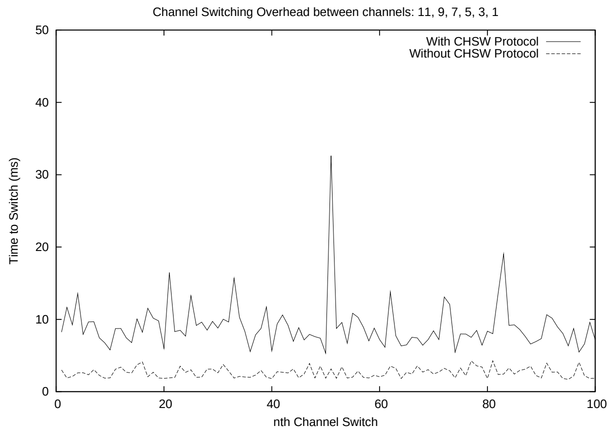
B. Graphs

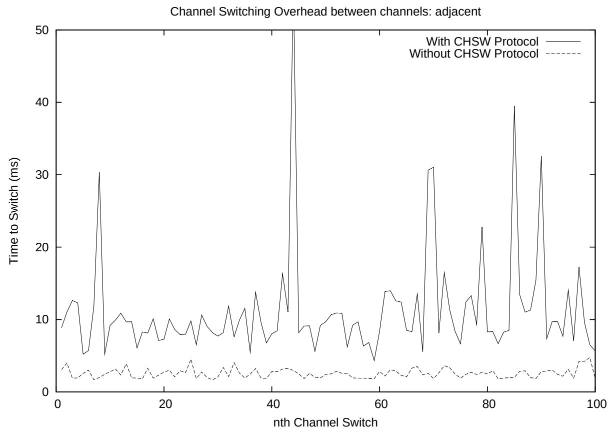


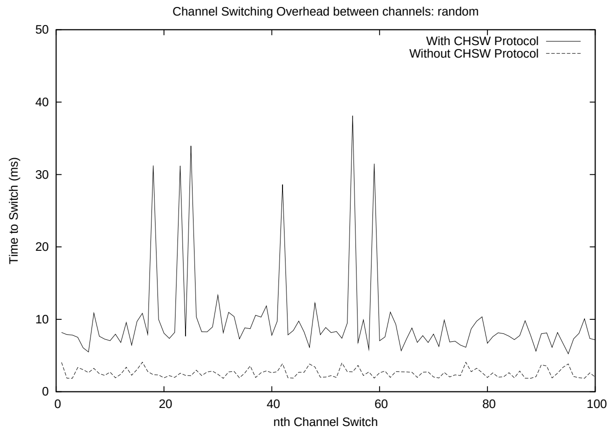












C. Analysis

In general, with the inclusion of the CHSW protocol, the average channel switch time was between 7.83ms (3-channel difference) and 11.15ms (1-channel difference), with an overall average of 9.57ms. Without the inclusion of the CHSW protocol, the average channel switch time was between 2.37ms (5-channel difference) and 2.78ms (3-channel difference), with an overall average of 2.54ms. Different types of channel switches were conducted, ranging from switching from opposite sides of the 802.11b spectrum, to simply switching to an adjacent channel. It should be noted that a channel in 802.11b has overlap with 4 channels on either side, thus we measured whether the fact that the overlap exists affected the latency of the channel switch. As it turns out, the actual channel switch time is very low, around 2-3ms, as most of the latency time occurs in CHSW negotiation (70+%). It seems that on average, there is no difference in the type of channel switch (i.e. how many channels exist between the starting frequency and the ending frequency), although it should be noted that the standard deviation was much higher when measuring latencies including the protocol as opposed to latencies measured without the protocol. Randomizing the channel "jump" had a negligible effect on the latency measured.

As seen in the graphs, data in Table II was fairly consistent (with standard deviation < 1ms), whereas there were several outliers for the data in Table I. This can be attributed to a number of different reasons:

1) No Guarantee of transmission: As mentioned above, once a data frame/packet is sent to the actual device, there is no guarantee that it will actually be transmitted. The kernel at that point has lost control of the data, giving sole control of it to the wireless device.

2) External overhead/latencies: An attempt was made to reduce all external effects by background programs and daemons on the data capture. However, some daemons could simply not be stopped as they were required for the running of the operating system. These include the recent introduction of `udev` into Linux-based systems; `udev` is the replacement background device manager that polls all devices, checks for the insertion of new devices, and dynamically loads any modules into the kernel as necessary. It is theoretically possible that `udev` may have been in the middle of one of its polling cycles for a partial duration of the data capture.

3) Protocol Overhead: It is quite possible that latencies were introduced due to delay in both transmission and reception of the packet. In this case, actual channel switching only accounted for 30% of the total latency measured, while the remainder of the time was spent with the device waiting for an ACK/NACK and processing the result as necessary.

Overall, it was found that regardless of difference in channel, latency in physical switching including the CHSW protocol is small, around 10ms or so, and actual channel switching takes 3ms.

IX. Two nodes in Ad-hoc Mode

The second case study performed was measuring the channel switch made between two nodes connected in ad-hoc mode. In this case, one of the nodes is designated to be the virtual AP, thus given the role to accept/deny channel switch requests for the entire network. Due to resource (lack of laptops) and time constraints, the full protocol could not be implemented and tested, thus each laptop was given a manual designation as to whether it would serve as a virtual client or virtual AP. Then a similar protocol to the one above is carried out between the virtual AP and the virtual client nodes. For each type of test, 50 channel switches were made, to mitigate the effect of any outliers present in the data. Data collection was made on channel switch overhead with and without the CHSW protocol devised above.

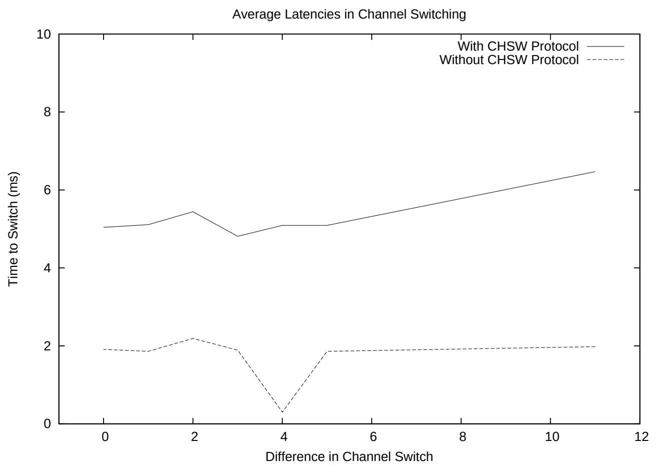
A. Data

TABLE III
Channel Switch Overhead with CHSW Protocol

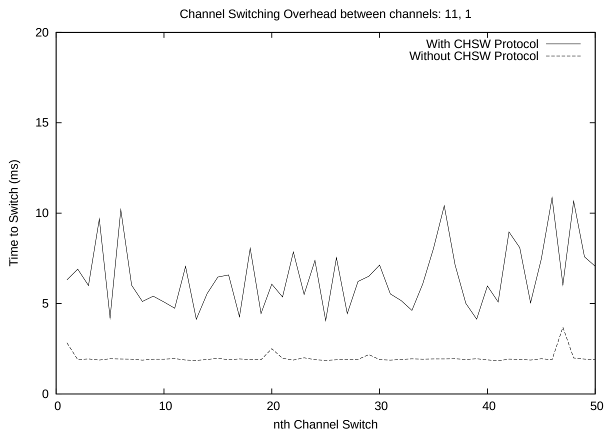
Channel Jump	Avg Latency (ms)	Std Dev (ms)
11	6.47	1.79
5	5.09	1.32
4	5.09	2.62
3	4.81	1.07
2	5.44	1.86
1	5.11	1.38
random	5.04	1.20
AVG	5.29	1.43

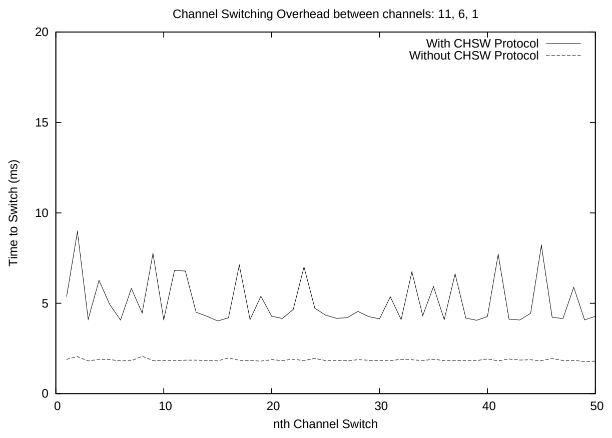
TABLE IV
Channel Switch Overhead without CHSW Protocol

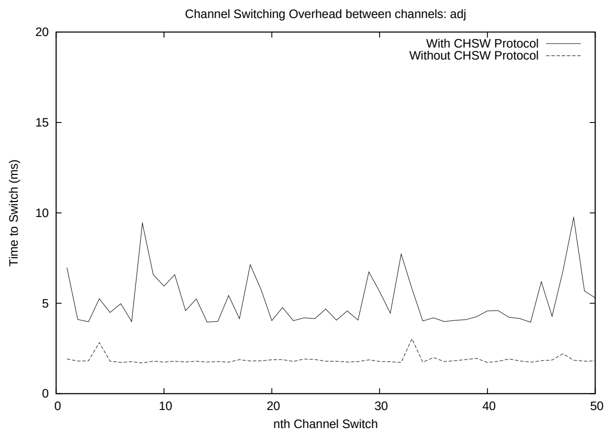
Channel Jump	Avg Latency (ms)	Std Dev (ms)
11	1.98	0.29
5	1.86	0.06
4	1.95	0.30
3	1.89	0.09
2	2.10	1.15
1	1.86	0.24
random	1.91	0.28
AVG	1.66	0.34

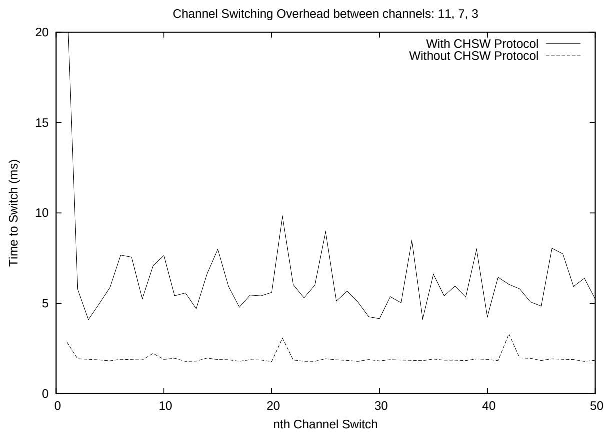


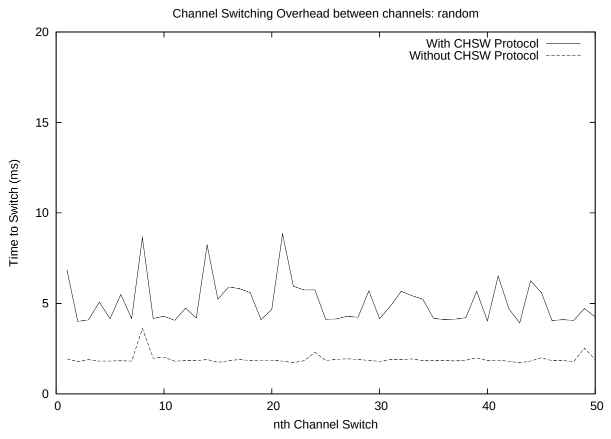
B. Graphs

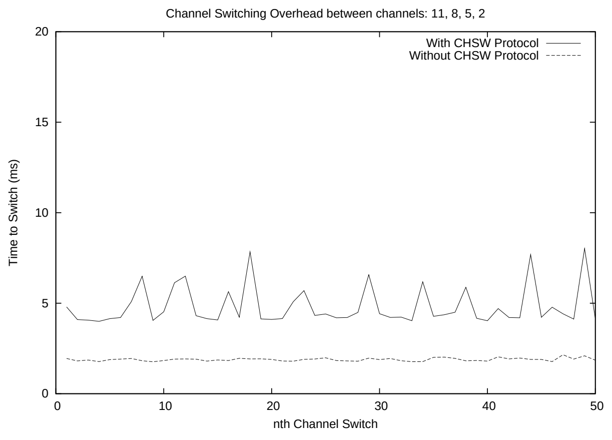


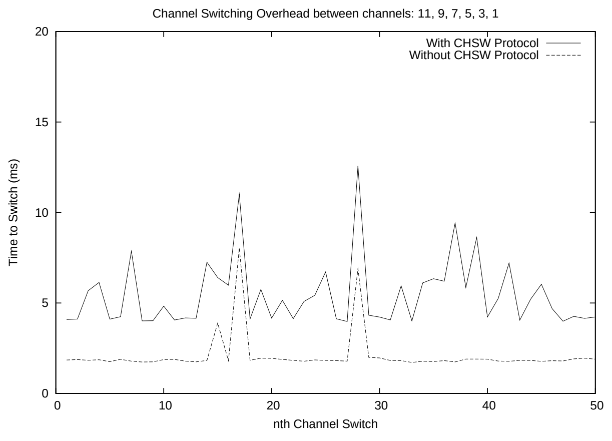












C. Analysis

In general, with the inclusion of the CHSW protocol, the average channel switch time was between 4.81ms (3-channel difference) and 6.47ms (11-channel difference), with an overall average of 5.29ms. Without the inclusion of the CHSW protocol, the average channel switch time was between 1.86ms (1,5-channel difference) and 2.10ms (2-channel difference), with an overall average of 1.66ms. Like the previous experiment, different types of channel switches were conducted, ranging from switching from opposite sides of the 802.11b spectrum, to simply switching to an adjacent channel. As in the previous experiment, the actual channel switch time is very low, around 1-2ms, suggesting that the time needed for an actual channel switch is negligible. Again, most of the latency time occurs in CHSW negotiation (70+%). It seems that on average, there is no difference in the type of channel switch (i.e. how many channels exist between the starting frequency and the ending frequency), although, switching channels 3 at a time proved to have the lowest latencies. Again, the standard deviation was much higher when measuring latencies including the protocol as opposed to latencies measured without the protocol. Randomizing the channel "jump" had a negligible effect on the latency measured.

As seen in the graphs, data in Table IV was fairly consistent (with standard deviation < 1ms), whereas there were several outliers for the data in Table III (although these are less prominent than that in Table I). This can be attributed to a number of different reasons:

1) No Guarantee of transmission:

2) External overhead/latencies:

3) Protocol Overhead: It is interesting to note that overall latencies for Ad-hoc networks was significantly less than that of Infrastructure networks, by a factor of 2. This suggests one of two possible outcomes: a) each chipset handles 802.11 management frames significantly differently such that it incurs unneeded latency; or b) the protocol in Infrastructure mode adds a significant amount of unneeded overhead, possibly due to the fact that additional beacon frames are broadcasted (and thus processed by any listening nodes in the area).

4) External Interference: One particular unique point of this experiment was the initial trouble to get the two nodes to communicate with each other acceptably on channel 3. As the experiment was conducted in a civilian setting, this problem in communication could be attributed to any consumer devices within the area that happened to be emitting electromagnetic radiation on a harmonic close to 2.4GHz. In this case, the offending device was localized to be a digital television set/box, as when the test equipment was moved to another area, the interference problem was mitigated.

Overall, it was found that regardless of difference in channel, latency in physical switching including the CHSW protocol is small, around 6ms or so, and actual channel switching takes 2ms.

X. Future Work

Due to resource and time constraints, not all aspects of the protocol devised above could be implemented. As such, a full implementation of the protocol remains in the area of future work, as the basis for channel negotiation and switching have been laid down.

Additionally, the current protocol design calls for deauthentication/disassociation of all stations in Infrastructure mode when the AP approves a channel switch request. For purposes of mobility, this proves extremely convenient as the act of disassociation will forcibly disconnect all network streams using the wireless interface. However, as the act of association/disassociation merely establishes a virtual link between station and AP, it seems possible that a channel switch may be made without having to disassociate the nodes, thus preserving all existing connections. As such, design and implementation of a protocol that will preserve network streams above the MAC layer while negotiating and fulfilling a channel switch amounts to another area of future work.

XI. Conclusion

We implemented a mechanism to mitigate problems in wireless communication incurred by interference. Our scheme assesses the health of a channel in real-time by monitoring signal strength, noise, and average latency between communicating nodes, and coordinates a net-wide channel switch when the interference level crosses a programmable threshold. The actual time required to switch channels is negligible in both infrastructure and ad-hoc modes given the relative advantage of communicating on an inteference-free channel, however any coordination protocols add a significant amount of overhead (triples or more the initial latency). However, it appears that the protocol negotiation for Infrastructure networks adds significantly more overhead than that for Ad-hoc networks.

In addition to this, we provided a background into the `mac80211` framework, and subsequently the `nl80211` libraries used for interfacing with the framework. This provides any user with full control of management of 802.11 frames, making it trivial to modify an/or add to existing schemes within the 802.11 protocol, while remaining generic enough to apply to a wide variety of hardware.

In this project we have laid a basic framework in the method and manner of implementing an advanced channel-switching protocol. Future work remains to fully implement the protocol devised, utilizing the full capabilities of the `mac80211` framework present in the Linux kernel.

References

[1] ”IEEE 802.11-2007: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications”. IEEE. 2007-03-08. http://standards.ieee.org/getieee802/802.11.html

[2] ”mac80211 - Linux Wireless”. Linux Wireless. http://wireless.kernel.org/en/developers/Documentation/mac80211

[3] ”p54 - Linux Wireless”. Linux Wireless. http://wireless.kernel.org/en/users/Drivers/p54

[4] ”Intel Wireless WiFi Link Drivers”. Intel Corporation. http://intellinuxwireless.org/

[5] Jouni Malinen. ”hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator”. 2009-11-28. 2009-12-05. http://hostap.epitest.fi/hostapd/

[6] ”The Slackware Linux Project: General Information” 2009-11-28. http://slackware.com/info/

[7] ”Beginners’ Guide - ArchWiki”. ArchWiki. 2009-10-15. http://wiki.archlinux.org/index.php/Beginners_Guide

[8] Dell Wireless 1450 WLAN USB 2.0 Adapter Product Specifications. http://www.dell.com/downloads/us/products/optix/dell1450_spec.pdf

[9] Jean Tourrilhes. ”Wireless Tools for Linux”. http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html

[10] Jean Tourrilhes. ”Wireless Extensions for Linux”. http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Linux.Wireless.Extensions.html