**MULTIHOP RENDEZVOUS ALGORITHM FOR FREQUENCY HOPPING COGNITIVE RADIO NETWORKS**

THESIS

John A. Pavlik, CPT, USA

AFIT-ENG-MS-16-M-039

**DEPARTMENT OF THE AIR FORCE**
**AIR UNIVERSITY**

***AIR FORCE INSTITUTE OF TECHNOLOGY***

**Wright-Patterson Air Force Base, Ohio**

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-16-M-039

# MULTIHOP RENDEZVOUS ALGORITHM FOR FREQUENCY HOPPING COGNITIVE RADIO NETWORKS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

John A. Pavlik, B.S.C.S.
CPT, USA

March 2016

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-16-M-039

# MULTIHOP RENDEZVOUS ALGORITHM FOR FREQUENCY HOPPING COGNITIVE RADIO NETWORKS

THESIS

John A. Pavlik, B.S.C.S.
CPT, USA

Committee Membership:

Dr. K.M. Hopkinson
Chair

Dr. D. D. Hodson
Member

Maj J. M. Pecarina, PhD
Member

AFIT-ENG-MS-16-M-039

## Abstract

Cognitive radios allow the possibility of increasing utilization of the wireless spectrum, but because of their dynamic access nature require new techniques for establishing and joining networks, these are known as rendezvous. Existing rendezvous algorithms assume that rendezvous can be completed in a single round or hop of time. However, cognitive radio networks utilizing frequency hopping that is too fast for synchronization packets to be exchanged in a single hop require a rendezvous algorithm that supports multiple hop rendezvous. We propose the Multiple Hop (MH) rendezvous algorithm based on a pre-shared sequence of random numbers, bounded timing differences, and similar channel lists to successfully match a percentage of hops. It is tested in simulation against other well known rendezvous algorithms and implemented in GNU Radio for the HackRF One. We found from the results of our simulation testing that at 100 hops per second the MH algorithm is faster than other tested algorithms at 50 or more channels with timing ±50 milliseconds, at 250 or more channels with timing ±500 milliseconds, and at 2000 channels with timing ±5000 milliseconds. In an asymmetric environment with 100 hops per second, a 500 millisecond timing difference, and 1000 channels the MH algorithm was faster than other tested algorithms as long as the channel overlap was 35% or higher for a 50% required packet success to complete rendezvous. We recommend the Multihop algorithm for use cases with a fast frequency hop rate and a slow data transmission rate requiring multiple hops to rendezvous or use cases where the channel count equals or exceeds 250 channels, as long as timing data is available and all of the radios to be connected to the network can be pre-loaded with a shared seed.

# Table of Contents


Page

Abstract ... iv

List of Figures ... viii

List of Tables ... xi

I. Introduction ... 1

- Background ... 1
- Motivation ... 2
- Research Objectives ... 3
- Research Questions ... 3
- Proposed Multihop Algorithm ... 3
- Organization ... 4

II. Literature Review ... 5

- Centralized Algorithms ... 6
- Decentralized / Blind Algorithms ... 7
- Probabilistic Algorithms ... 7
- Deterministic Algorithms ... 9
- Random Number Generators ... 11

III. Multihop Algorithm Analysis ... 12

- Multihop Algorithm Design ... 12
- Timing Synchronization Analysis ... 14
- Window Search Size ... 19
- Security ... 19
- Run time analysis ... 20
- Conclusions ... 22

IV. Simulation Setup and Results ... 23

- Objectives ... 23
- Assumptions ... 23
- Response Variables ... 24
- Control Variables ... 24
- Constant Factors ... 24
- Simulation Environment ... 26
- Initial Tests ... 26
- Symmetric Test Results ... 26
- Asymmetric Test Results ... 36

Page

Timing Validation . . . . . 39
Conclusions . . . . . 44

V. GNU Radio Hardware . . . . . 45

Transmitter . . . . . 45
Receiver . . . . . 46
Hardware . . . . . 46
Problems . . . . . 47
Results . . . . . 47
Hardware conclusions . . . . . 49

VI. Conclusions . . . . . 50

Objective . . . . . 50
Criteria for success . . . . . 50
Significance . . . . . 50
Possible Future Work . . . . . 51

Appendix A. Java Simulation Source Code . . . . . 52

MultiHop.java . . . . . 52
TextInterface.java . . . . . 58
ConfigurationLoader.java . . . . . 60
Simulation.java . . . . . 63
Channel.java . . . . . 65
ChannelListener.java . . . . . 67
Spectrum.java . . . . . 67
AlgorithmFactory.java . . . . . 69
Radio.java . . . . . 70
RendezvousAlgorithm.java . . . . . 71
EnhancedJumpStay.java . . . . . 72
JumpStay.java . . . . . 73
ModifiedModularClock.java . . . . . 74
RandomAlgorithm.java . . . . . 76
ShortSequenceBased.java . . . . . 76
DRSEQ.java . . . . . 77
GeneratedOrthogonalSequence.java . . . . . 78
ModularClock.java . . . . . 79
RadioProtocol.java . . . . . 80

Appendix B. GNU Radio Source Code . . . . . 81

channel_selector.h . . . . . 81
channel_selector.cc . . . . . 81
channel_to_freq.h . . . . . 83

Page

channel_to_freq.cc ........ 83
csprng.h ........ 85
csprng.cc ........ 86
freq_to_channel.h ........ 87
freq_to_channel.cc ........ 88
freqhop_signal_source.cc ........ 90
freqhop_signal_source.cc ........ 91
max_cf.h ........ 92
max_cf.cc ........ 93
mode_ii.h ........ 94
mode_ii.cc ........ 95
multihop_rendezvous.h ........ 97
multihop_rendezvous.cc ........ 98
modular_clock_rendezvous.h ........ 101
modular_clock_rendezvous.cc ........ 103
Bibliography ........ 107

# List of Figures

Figure Page

1. State Machine Diagram for Multiple Hop Algorithm using Network Master . . . . . 15

2. Logarithmic scale comparison of mean rounds to rendezvous simulation results for 10-100 channels in a symmetric environment (100% channel overlap). The Multihop algorithm is tested with window sizes of 10, 100, and 1000 channels. . . . . 27

3. Logarithmic scale comparison of mean rounds to rendezvous simulation results for 100-2000 channels in a symmetric environment (100% channel overlap). The Multihop algorithm is tested with window sizes of 10, 100, and 1000 channels. . . . . 28

4. Tukey HSD comparison of Multihop against JS showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster. . . . . 30

5. Tukey HSD comparison of Multihop against EJS showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster. . . . . 31

6. Tukey HSD comparison of Multihop against DRSEQ showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster. . . . . 32

7. Tukey HSD comparison of Multihop against MC showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster. . . . . 33

Figure Page

8. Tukey HSD comparison of Multihop against MMC showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster. . . . . . 34

9. Tukey HSD comparison of Multihop against Random showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster. . . . . . 35

10. Channel correlation values of MH algorithm for 1000 channels in an asymmetric environment using randomly generated noise values in the range of $[0, 1)$ with 1, 5, and 10 non-overlapping channels. . . . . . 37

11. Logarithmic scale comparison of mean time to rendezvous of the Multihop-100 algorithm with a window size of 100 compared against the Random and MMC algorithms at 1000 channels. . . . . . 39

12. Tukey HSD comparison of Multihop against MMC showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster. . . . . . 40

13. Tukey HSD comparison of Multihop against Random showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster. . . . . . 41

Figure | Page


14. Comparison of the calculated vs simulated results of the multihop algorithm using a window size of 100 in a fully symmetric environment. Error bars show 95% confidence intervals. The rounds to rendezvous converges towards 60 because the MH100 takes 50 rounds on average ($n/2$) to seek the timing and 10 rounds ($\alpha$) to complete synchronization. . . . . . . . . . . . 42

15. Logarithmic comparison of the calculated vs simulated results of the multihop algorithm using a window size of 100, fixed at 1000 channels, for an asymmetric overlap ranging from 25%-100%. Error bars show 95% confidence intervals. . . . . . . . . . . . . 43

16. The setup of the hardware transmitter as implemented in the GNU Radio Companion software. The software selects a channel, converts it to a frequency, and then the signal source generates a sin wave to be transmited to the osmocom Sink block which feeds those values to the HackRF One radio for transmission over the air. . . . . . . . . 45

17. The setup of the hardware receiver as implemented in the GNU Radio Companion software. The osmocom source reads from the HackRF One radio, then the results are processed through an FFT to find the strongest signal source, converts that value back into a channel index, and feeds that into the rendezvous algorithm for comparison to see if the right values were generated. . . . . . . . . . . . . 46

# List of Tables


Table Page

1. Response Variables . . . 24

2. Control Variables . . . 25

3. Constant Factors . . . 25

4. Example of two different asymmetric difference rate measurements . . . 36

5. T-test comparison of mean rounds to rendezvous for simulation vs hardware results for the Multihop Algorithm in a symmetric enviroment with 2000 channels at 100 hops per second. . . . 47

6. T-test comparison of mean rounds to rendezvous for simulation vs hardware results for the Modular Clock (MC) Algorithm in a symmetric enviroment with 50 channels at 1 hop per second. . . . 48

MULTIHOP RENDEZVOUS ALGORITHM FOR FREQUENCY HOPPING COGNITIVE RADIO NETWORKS

# I. Introduction

## Background.

Radios have been experimented with since 1887, when Heinrich Rudolf Hertz confirmed Maxwell's electromagnetic theory and was the first to transmit and receive radio waves [15]. Radios have since then evolved to become sophisticated communication devices, among other purposes, and underlie all modern U.S. military tactical communications. From the private talking to his squad leader over a hand-held FM transceiver to the brigade commander utilizing radio waves to communicate with satellites and bring access to the entire internet down to the battlefield, radio waves are essential to military communications [10]. However, wireless spectrum is a limited resource in which all of the prime real estate has already been allocated for purposes such as TV broadcasting and cellular phones [34].

Cognitive radios were first introduced by Joseph Mitola [25, 26] as smarter, more capable radio system. In particular, he defined them to be self-aware and to have a cognition cycle. The self-awareness means each radio can know about its own processes, functions, and current state. This allows a network to ask a question such as "How many distinguishable multipath components are in your location?" [26, p.45] and actually be able to receive an answer from each of its components. The cognition cycle is defined as a way of interacting with the environment and responding to external stimuli, through the OODA loop for example [26, 22].

**Motivation.**

Cognitive radios today offer the potential to increase utilization of the wireless spectrum to extend the reach and bandwidth of radio communications. The basis for this increased utilization is to allow currently licensed spectrum to be shared with secondary users whenever that space is available. Users are categorized as primary and secondary users. Primary users are those who have licensed the spectrum and have priority access to use that spectrum whenever they want without interference. Secondary users are anyone else that wants to use that space in the spectrum. The secondary user can utilize a channel whenever the primary user is not currently using it, but must vacate the channel whenever the primary user begins using it again. There is a lot of spectrum currently assigned but not being utilized fully, such as the VHF and UHF bands currently occupied by broadcast television [1]. The cognitive radio enables this primary/secondary user paradigm by observing the local wireless spectrum and choosing the channels that are best at that particular moment for communication, i.e. channels that are currently unoccupied and free of interference [22]. This is also known as dynamic spectrum access [37, 8].

Before any two radios can communicate, they must first find each other among all of the possible channels; this is known as rendezvous [7]. For traditional radios the problem is easy; the frequency is decided upon before use and all of the radios agree on which channels to use. However, in a cognitive radio used by secondary users the set of available channels can change frequently and the radios must be able to adapt quickly in order to avoid interfering with the primary user and to take advantage of better channels as they become free. Therefore, any radio that attempts to join a cognitive radio network must be able to find or predict the current channel being used by the network. Most existing rendezvous algorithms assume that two radios can search for each other and that rendezvous synchronization is complete when both

radios arrive at the same channel at the same time [18, 6, 32, 31, 19, 4, 17, 5, 23, 29, 20]. However, if the network is utilizing frequency hopping with a fast enough rate that the rendezvous cannot be completed in a single hop, then those algorithms will not work.

Many rendezvous algorithms also require a completely deterministic hopping sequence on both the part of the joining radio and the network for rendezvous to occur [18, 31, 17]. This allows a hostile radio to potentially predict and intercept or jam communications and rendezvous attempts [27, 23].

**Research Objectives.**

Specifically, this thesis is focused on developing an algorithm that allows a cognitive radio to join an already established frequency hopping network in which synchronization requires more than one hop. Additionally, the algorithm must create a hopping sequence that is secure against interception and jamming from a potential attacker.

**Research Questions.**

There are two specific measures of success that this thesis will use to judge results:

1. Does the proposed algorithm successfully allow a cognitive radio to rendezvous and join a frequency hopping network under real world conditions?

2. Does the proposed algorithm have a faster mean time to rendezvous than existing algorithms?

**Proposed Multihop Algorithm.**

To solve that problem, this thesis introduces the Multihop (MH) rendezvous algorithm designed to allow a radio to join an operating frequency hopping cognitive

radio network without the network having to explicitly attempt a rendezvous and for the synchronization process to occur over multiple hops. It is based on generating an identical sequence of random numbers in each radio to allow choosing similar channels in the same order on each radio and using bounded timing data to limit the search space required for a radio to locate the network. The cost of the MH algorithm is that it is no longer blind; it requires a random number sequence or a random number generator with a seed to be shared to all of the radios that will be in the network before rendezvous is attempted. Additionally, the radios must having timing data with a known bound on the maximum difference in timing data between any two radios, e.g. they must all be within 1 second of the current time. The envisioned use case for the MH algorithm is when symmetric key encryption is already being utilized for protecting the privacy of communications over the network, and the additional data required by MH can be loaded at the same time as the radios are receiving their encryption keys, such as a military network.

### Organization.

This thesis is organized as follows. Chapter II covers previously published rendezvous algorithms for cognitive radios from the literature. Chapter III covers the full explanation of the Multihop algorithm and an analysis of its running time. Chapter IV has the simulation setup and results. Chapter V contains the results of implementing the algorithm into GNU Radio based hardware. We discuss the results of my analysis and consider future work in chapter VI.

# II. Literature Review

Current cognitive radio rendezvous algorithms can be broken down into several categories. The first is whether they are centralized or decentralized. Centralized algorithms use either a centralized controller or common control channel (CCC) or both [9]. These have the advantage of being simple to implement; just contact the known server over the pre-defined control channel and request to join the network, for example. However, centralized algorithms are problematic for cognitive radios because a central controller creates a single point of failure for the entire network, and common control channels require either licensing expensive frequency bands or using unlicensed bands that are already heavily congested with other traffic. Decentralized algorithms are simply those which do not utilize a central controller or common control channel, and are also known as blind rendezvous.

Decentralized algorithms break down further based on whether they are probabilistic or deterministic and whether they support symmetric or asymmetric environments [19]. Probabilistic algorithms will rendezvous eventually, but do not provide a bound on the maximum time to rendezvous. Deterministic algorithms do provide a bound, but only on the assumption that the environment is static [23].

Radio environments can be modeled as either symmetric or asymmetric. Symmetric environments are those in which all radios see the same set of channels while asymmetric allows each radio to have its own set of channels. In an asymmetric environment, some subset of the channels must be overlapping between all of the radios, otherwise it would be impossible for any sort of communication to occur at all. All asymmetric algorithms can function in a symmetric environment but the reverse does not hold true.

However, all of the described algorithms below implicitly or explicitly assume the frequency hopping rate is slow enough that several packets can be transmitted on

each channel before hopping to the next. Most algorithms do not define any sort of handshaking mechanism at all and assume that rendezvous has been completed once two radios are on the same channel at the same time.

The most important metric for measuring the performance of a rendezvous algorithm is the *time-to-rendezvous* (TTR). All algorithms attempt to minimize both the average and worst case TTR values, but many algorithms are probabilistic in nature and have an unbounded TTR. The TTR is usually measured in discrete time slots or rounds. For example, taken from the Jump-Stay algorithm:

> Since time synchronization is not available in the network, the same time slot of two CH sequences means that the overlap of two time slots is sufficient to complete all necessary steps for rendezvous. [18]

This assumption is either implicit or explicit in nearly all other published rendezvous algorithms as well, with the important point that a time slot is large enough "to complete all necessary steps for rendezvous". The proposed MH algorithm explicitly assumes that time slots are too small for rendezvous to occur and therefore multiple time slots in a short period of time will be required for rendezvous to complete.

**Centralized Algorithms.**

The HC-MAC algorithm proposed by Jia et al. assume that a CCC is available for exchange of control messages and coordination between secondary users [12]. Ma et al. proposed the Dynamic Open Spectrum Sharing MAC protocol for wireless ad hoc networks which utilizes a common control channel to simplify the design of the radio receiver [21]. Both of these approaches are specifically global CCCs, which are the least feasible option due to the congestion of unlicensed bands. Other designs establish local CCCs with cluster systems. In particular, the distributed coordination approach by Zhao et al. [36] and Spectrum-Opportunity Clustering by Lazos et al. [16]. The

difference between these two algorithms is that the distributed coordination approach minimizes the total number of control channels required to link the entire network while the Spectrum-Opportunity Clustering attempts to balance maximizing the set of common channels within each cluster and maximizing cluster size. Local CCCs are more reliable than a global CCC, but there is a substantial overhead required to establish and maintain the CCC [19].

**Decentralized / Blind Algorithms.**

The majority of published algorithms fall into this category, largely because they do not assume any sort of common control channel is available and the algorithm can be applied to a much larger range of situations. Of those, there are a couple of algorithms that utilize time synchronization like the proposed Multihop Algorithm. The first is the Slotted Seeded Channel Hopping (SSCH) designed for 802.11 Ad-Hoc Wireless networks, but it also works for symmetric models of cognitive radio networks [2]. The second is the set of quorum systems proposed by Bian et al., M-QCH and L-QCH. M-QCH minimizes the maximum TTR while L-QCH minimizes the channel load [4]. However, the M-QCH and L-QCH systems require exact time-synchronization, as opposed to the proposed MH algorithm which only requires a bounded time difference. Bian et al. also introduced the A-QCH system which is asynchronous and does not require any clock synchronization, but it only works for a pair of searching radios and requires both radios to be searching the sequence.

**Probabilistic Algorithms.**

The simplest probabilistic algorithm is to be purely random; pick a channel, check to see if anyone is there, and then pick a new random channel [32]. As long as at least one channel overlaps between the radios attempting to rendezvous then they will all

eventually land on the same channel at the same time; however, this does not provide any guarantees that rendezvous will occur in a finite quantity of time. Another probabilistic algorithm by Theis et al.[32, 33] is the Modular Clock (MC) algorithm and the Modified Modular Clock (MMC) algorithm for symmetric and asymmetric models, respectively. Being probabilistic means that they have a deterministic design based on the Chinese Remainder Theorem, but choose initialization variables with a pseudo-random generator. If two radios choose the same rates of advancement and prime value for modulus arithmetic then they will be unable to synchronize until timeout occurs and different variables are chosen. The MC and MMC algorithms both work based on modular arithmetic, in particular by picking a prime number $p_i$ and random rate of advancement, then channel skipping forward and using the mod operation whenever the prime number is reached to loop back to the beginning. The MC algorithm can guarantee rendezvous will occur in less than $p_i$ steps and the MMC algorithm in less than $p_1 * p_2$ steps, assuming that the two radios choose different random rates of advancement. If rendezvous doesn't occur within $2p$ steps or $2p_i^2$ respectively then both radios pick new rates of advancement and try again. Because of the possibility of picking the same rate of advancement repeatedly, neither MC nor MMC can provide a guaranteed bound on runtime. However, the larger the number of channels, the less likely that issue is to occur.

Misic et al. introduced an adaptation of a cognitive Media Access Control (MAC) protocol [24, 23]. The MAC protocol and rendezvous algorithm function by having a coordinator that reserves time blocks to transmit extra packets on regular intervals, which include necessary network function information, including the next channel in the hop. This allows any radios wishing to join the network the ability to either wait on a channel for the network to hop there or search channels looking for the network; once the joining radio sees at least the trail end of a sequence of packets, it knows

where the network is hopping to next and can follow and send a request to join. Additionally, if no network is found, then the radio can start its own by beginning to broadcast the necessary control packets with each hop. This algorithm is particularly advantageous in that it incorporates rendezvous capability into normal operation of the radio with no need to stop data transmission to initiate a rendezvous search phase to allow additional radios to join.

AMRCC [6] is a random based algorithm which allows for weighting the channels so that preferred channels are more likely to be selected using a pseudo-random number sequence. This algorithm operates in two phases: sensing and handshaking. During the sensing phase each radio analyzes the noise level of each channel and builds a channel ranking table where the best channels (least primary user activity) have a lower index and worse channels have a higher index. Then the radio generates a pseudo-random sequence containing values ranged [1,n] with a bias towards the lower values. Therefore, channels with less interference are more likely to be selected. The output of this routine is an adaptive hopping sequence with pseudo-random values mapped 1 to 1 over the ranking table. Then the algorithm begins phase 2, handshaking. During this phase the radio sends a request to send (RTS) packet on each channel it hops into, and if the intended receiver is on the same channel at that time then the receiver responds with a clear to send (CTS) packet and rendezvous is complete. Then the two radios can exchange SYNC packets to synchronize future hops, which would contain ranking tables and the seed for the pseudo-random sequence.

**Deterministic Algorithms.**

Two well-known deterministic algorithms are Jump-Stay and Enhanced Jump-Stay. Both algorithms work by generating channel hopping sequences in rounds, where each round consists of a jump pattern and a stay pattern. The jump pattern

is similar in principal to the Modular Clock algorithm but is constructed so that overlap between two radios is guaranteed on every channel without being dependent on choosing different prime values. Enhanced Jump-Stay is a variation which improves the asymmetric performance by increasing the time spent in jump mode rather than stay mode, but it has slightly worse performance in symmetric models relative to Jump-Stay. During the jump pattern, the user is constantly hopping to different frequencies and attempting rendezvous, which lasts three times longer than the stay pattern during which the user just waits to see if another user jumps there. For Enhanced Jump-Stay under the symmetric model, rendezvous is guaranteed to occur at most $4P$ timeslots where $P$ is the smallest prime number greater than the number of channels. And under the asymmetric model, rendezvous is guaranteed to occur in at most $4P(P + 1G)$ timeslots where G is the number of channels in common to both users [18, 17]. This algorithm is advantageous in that it operates under an asymmetric model but has a symmetric level of performance in a symmetric model; however, the disadvantage is that this algorithm implicitly requires a shared global labeling of channels.

In addition, an enemy attempting jamming who knows that Jump-Stay is being utilized can easily defeat Jump-Stay due to the entirely deterministic nature of the algorithm. It has been shown in simulations that Jump-Stay can be reduced from a 100% to less than a 20% chance of rendezvous if being actively jammed by just one listener [27]. Rezaei et al. have studied the Nested Grid Quorum-Based Frequency Hopping (NGQFH) algorithm and found that it is also susceptible to jamming but can be enhanced to be jamming resistant by adding a probability of performing a random channel selection. Furthermore, Misic et al. [23] have argued that any deterministic blind rendezvous algorithm can fail to rendezvous in bounded time if rendezvous is interrupted either through intentional jamming or accidental interference by other

users.

The Deterministic Rendezvous Sequence (DRSEQ) by Yang et al. [35] works by generating a *k-shift-invariant* sequence shared by all of the radios in teh system. For any two $k$ and $k'$ starting positions, after some increment $i$ the sequence will overlap itself: $f(k+i) = f(k'+i)$. The sequence is generated deterministically based upon the total number of channels available, with no randomness utilized at all. A related algorithm is the Channel Rendezvous Sequence (CRSEQ) [31] which is based on the properties of triangular numbers and modulus to generate a sequence of subsequences that is guaranteed to overlap a common channel if such exists as long as the other radio is also running a similarly generated search sequence.

**Random Number Generators.**

For algorithms which rely on a random number generator to select channels the pseudo-random number sequence must be cryptographically secure if the algorithm wants to prevent interception and possible jamming. Common random number generators based on a linear congruential method as described by Knuth[13], such as found in Java.util.Random and C# System.Random, are not cryptographically secure as shown by Krawczyk [14]. The most well known source of cryptographically secure pseudo-random number generators (CSPRNG) is the U.S. Department of Commerce National Institute of Standards and Technology publication NIST SP 800-90A [3]. It contains several different random number generators, among them DUAL_EC_DRBG has been found to be insecure [30], but HMAC_DRBG has been studied and found to be secure, so far [11].

# III. Multihop Algorithm Analysis

The premise of the MH algorithm is that each radio can generate an identical deterministic sequence of random numbers from a shared seed, allowing each radio to independently select the same channel if they share the same list of channels to choose from and share timing information. In an asymmetric environment there is no guarantee that both radios have the same list of channels, but as long as the list is similar and the generated numbers the same then the algorithm can still successfully rendezvous. The algorithm synchronizes the time of the joining radio to the network by searching a fixed window size of possible channels in which the network could be based on the current time.

The state diagram of the algorithm is provided in figure 1, and pseudo-code for each of the states is provided by Algorithms 1 and 2. Full Java code for the simulator used to test the algorithm is provided in Appendix A and the C++ code used to demonstrate the algorithm on GNU Radio Hardware is provided in Appendix B.

## Multihop Algorithm Design.

The MH algorithm operates in four phases: sensing, seeking, synchronizing, and network communication.

### Sensing.

During the sensing phase each radio looks at each channel in its local environment and builds a channel ranking table based on any preferred factors, the specific factors used do not matter so long as there is a small difference between the ranking table of each radio. Once that is complete each radio begins creating a hopping sequence utilizing their shared seed, such that every radio generates the same hopping sequence but each over their unique channel ranking table. Each radio creates a sliding window

of channels, with the size of the window determined by a fixed maximum difference between radio clocks. The starting index is at the 50% position of the window. One radio can be designated by the operator as a master to the network and it immediately jumps to the network communication phase in order to allow other radios to join this radio and create a network, but this is not required if timeout fail-over intervals are utilized in the seeking phase. All other radios immediately move to the seeking phase.

**Seeking.**

The joining radios begin seeking for the network by scanning through the sliding window of channels. If a radio in the seeking phase detects a transmission from the network, then that radio knows the timing of the network based on its current position in the sliding window and immediately begins the synchronization phase. If a fail-over timeout is reached then there are no networks to join and this radio transitions to network communication to establish its own network.

**Synchronization.**

The synchronization phase consists of frequency hopping forward at the same rate and generating the same RNG values as the network. Over the course of the next series of hops the joining radio will broadcast a request to join and the network responds with a synchronization transmission containing the channel ranking table of the network. If the timing synchronization of seeking was successful and it was not a false positive, then as long as the differences between the channel ranking tables of the joining radio and the network are smaller than the ratio of successful to total hops then the rendezvous will be successful with high probability and the radio can move to the network communication phase.

The synchronization phase assumes that some sort of error correcting code such

as Reed-Solomon [28] will be utilized at the data link levels of the radio for data transmission. This allows the MH algorithm to assume that transmitting $x$ successful packets out of $y$ total packets will be sufficient for rendezvous to complete. Otherwise the MH algorithm would require $z$ consecutive successful rendezvous attempts. This would not break the MH algorithm but would make rendezvous much more difficult.

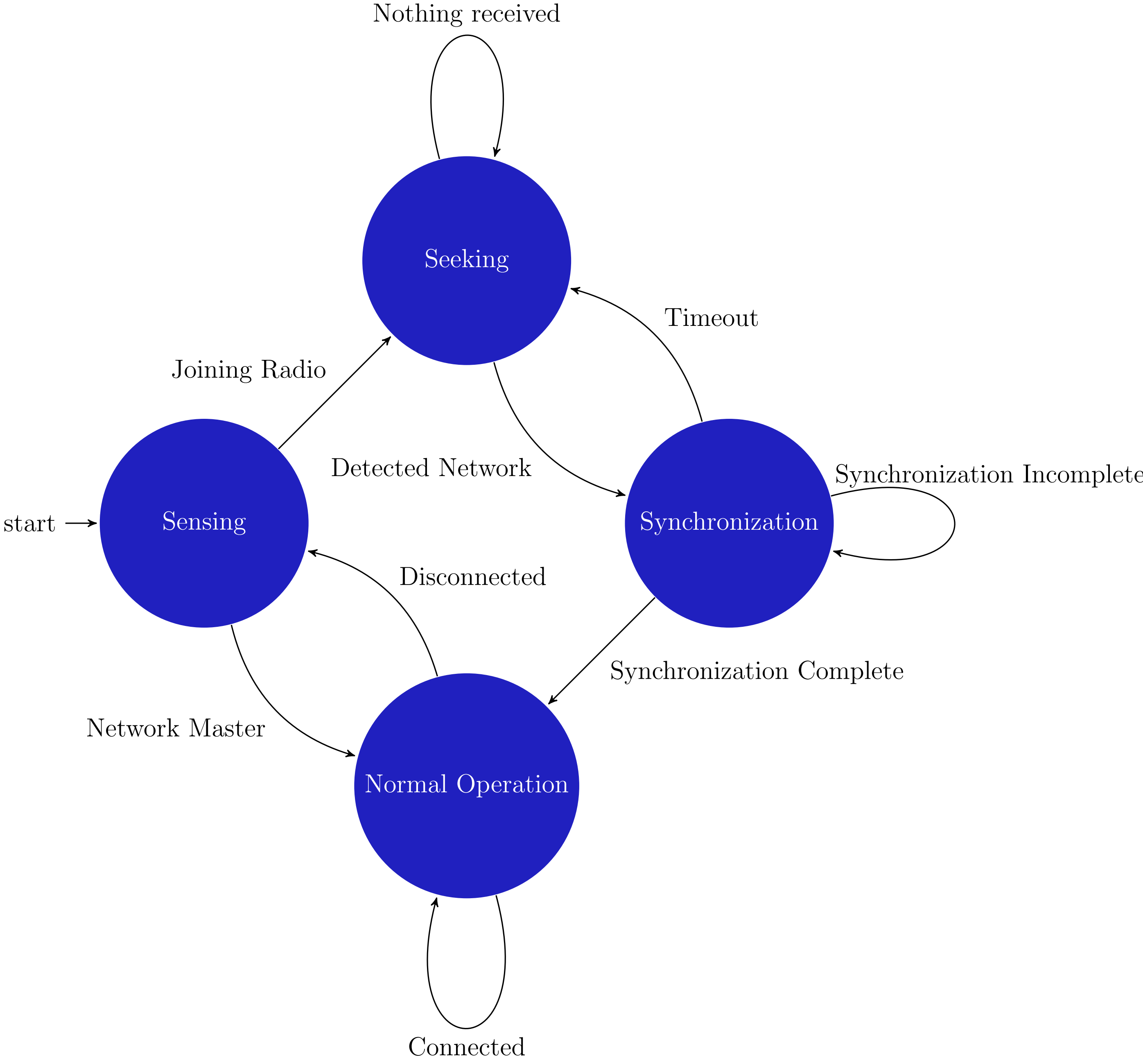
#### Network Communication.

Once the synchronization packet is received the joining radio can transition to the network communication phase and begin passing user data over the network. The joined radio has now established timing and synchronized to the communication channels the network is using, allowing the joined radio to accurately predict future network hops for as long as conditions remain static.

### Timing Synchronization Analysis.

During the Seeking Rendezvous phase of the MH algorithm, the joining radio will be offset in timing from the network by up to $M$ milliseconds. The current number of milliseconds since beginning the search is represented by $t$. The network is hopping at a fixed rate of $S$ ms/hop and the joining radio is searching at a rate of $S'$ ms/hop. The search is complete when the value of $t$ is such that the current network hop is

**Figure 1. State Machine Diagram for Multiple Hop Algorithm using Network Master**



**Algorithm 1** Multihop Rendezvous Algorithm

```
function Sensing(seed)
    for Each channel in the available spectrum do
        Measure noise
        if Primary user not present then
            Add channel to a sorted list
        end if
    end for
    Initialize CSPRNG with seed value
    hops ← current time / hop rate          ▷ Calculate how many hops are needed
    while hops > 0 do                        ▷ to bring the CSPRNG up to now
        hops ← hops − 1
        Cycle CSPRNG
    end while
    Initialize sliding_window array with a size based on hop rate and clock accuracy
    sliding_window_index ← sliding_window.size()/2
    round ← 0
    last_window_update ← 0
    timeout ← 0
    Call Seeking
end function
function Seeking
    Call HopForward
    if Network activity detected on channel_list[sliding_window[sliding_window_index]]
then
        Call Syncing
    else
        Call Seeking
    end if
end function
function HopForward
    round ← round + 1                        ▷ Slide the sliding window forward
    while last_window_update ≤ round do
        sliding_window[last_window_update] ← CSPRNG.next_int() mod channel_list.size()
        last_window_update ← (last_window_update + 1) mod sliding_window.size()
    end while
end function
```

**Algorithm 2** Multihop Rendezvous Algorithm Continued

```
function SYNCING
    Call HOPFORWARD
    sliding_window_index ← sliding_window_index + 1
    if Network activity detected on channel_list[sliding_window[sliding_window_index]]
then
        Attempt or Continue Rendezvous
    end if
    if Rendezvous Complete then
        Update channel_list with network channels
        Call NETWORK
    else
        timeout ← timeout + 1
    end if
    if timeout exceeded threshold then
        timeout ← 0
        Call SEEKING
    end if
end function
function NETWORK
    Call HOPFORWARD
    while Still connected to network do
        Communicate with network
    end while
    Call SENSING                ▷ Disconnected, begin rendezvous from beginning
end function
```

equal to the joining radio hop:

$$\frac{t}{S'} = \frac{t}{S} + \frac{M}{S} \tag{1}$$
$$\frac{M}{S} = \frac{t}{S'} - \frac{t}{S} \tag{2}$$
$$\frac{M}{S} = \frac{tS}{S'S} - \frac{tS'}{S'S} \tag{3}$$
$$\frac{M}{S} = \frac{tS - tS'}{S'S} \tag{4}$$
$$\frac{M}{S} = t * \frac{S - S'}{S'S} \tag{5}$$
$$t = \frac{MS'S}{S(S - S')} \tag{6}$$
$$t = \frac{MS'}{S - S'} \tag{7}$$

Therefore, the search will be complete in $t$ ms, where $t$ is equal to the offset times the joining radio search speed divided by the difference in search speeds between the joining radio and network. Furthermore, this analysis highlights the fact that if $S = S'$ then it is impossible for the radios to close the gap in timing to synchronize. Additionally, if $S' = 0$ then the joining radio can search all channels instantaneously or if $M = 0$ then the joining radio has already synchronized. However, this analysis assumes that no false positives can occur during the seeking process and that both radios are hopping in exactly the same pattern. A violation of the first assumption occurs when both radios land on the same channel but at different points in the hopping sequence. This probability is increased if the number of channels available is small. Every occurrence of such a violation creates a delay in timing synchronization as both radios will begin hopping at the $S$ rate until timeout occurs.

Violations of the second assumption only matter if they occur at time $t$, if that occurs then timing synchronization has failed during this iteration but the search occurs over a finite period of time $W$, where $W \geq M$, and will therefore loop around

and can be solved using the same equation with $W$ replacing $M$. However, this means that synchronization time is unbounded if violations of the second type occur at every point where the hops are equal.

**Window Search Size.**

The previous section assumes that the network is always ahead of the joining radio when searching, and the algorithm utilizes a sliding window to allow for a wraparound effect during the search in order to support that assumption. However, the radios can start up to $M'$ milliseconds apart, therefore up to $M'/S$ indices in the sliding window can be different due to the offset in timing between the radios and if $W = M'/S$ then it is possible for synchronization to never occur. Therefore, the sliding window size is $2 * M'/S + 1hops$ with a starting index in the center of the sliding window, allowing the joining radio to search for both forward and backward offsets.

The use of two different search rates over a finite space guarantees that any two radios will eventually intersect at a single point in that search space, but if each radio has mapped that point to a different channel then the sliding window fails.

**Security.**

The algorithm follows Kerchoff's principle in design by publicizing everything except the key. The key in this case is the secret pre-shared seed used for the CSPRNG. As long as the algorithm uses a cryptographically secure random number generator then an adversary will not be able to predict future hops based on available information without access to the private seed, because the numbers generated by the CSPRNG directly translate into channels selected for hopping. The specific CSPRNG used is not important to the algorithm, only that it generates random numbers securely.

**Run time analysis.**

**Sensing.**

During the sensing phase the algorithm requires performing an examination of each possible channel to determine if it will be used or not. This is an $O(c)$ search where $c$ is the total number of channels and it is limited solely by the speed of the hardware in tuning and examining each channel.

**Seeking.**

The network is frequency hopping forward in time at a fixed rate. Therefore, assuming the joining radio requires an entire hop round to identify the network, the joining radio can search at the maximum possible rate by holding a single position in the sliding window and waiting for the network to hit that position. This is a $n/2$ required time to search on average where $n$ is the size of the sliding window. If there is a false positive then there is a delay until the timeout in the next stage is hit and this stage can pickup where it left off, this value is defined as $\beta$. The probability of a false positive on each hop is $1/c$, creating a $\beta/c$ additional expected delay on each hop due to false positives. Creating a total run time of $n/2 + (n/2) * (\beta/c)$ for a symmetric environment, which suggests that the runtime will actually decrease with a large channel count but increase linearly with the window size and $\beta$ value.

Furthermore, if the environment is asymmetric then there is a probability that the search will align on an index in which the joining radio has the wrong channel and this will cause a delay of size $n$ additional hops. The probability of this missed channel case occurring on any given attempt at rendezvous is $1 - p$ where $p$ is the overlap percentage of channels between 0.0 and 1.0, with a repeat chance for every failure. Therefore, the total number of expected cycles is $\frac{1}{p}$, e.g. if $p = .25$ then it will take 4 cycles on average for timing synchronization to succeed. The total delay caused

by this is therefore $\frac{n}{p}$. As $p$ approaches zero, the total expected rounds will approach infinity. This gives a final asymmetric seeking runtime of $n/2p + (n/2p)(\beta/c)$.

The asymmetric missed channel case could be mitigated by setting the joining radio to search the sliding window as well. For example, if the joining radio searches at half the network frequency hop speed then the joining radio will have two chances to rendezvous instead of one but will also take $n$ instead of $n/2$ hops for the joining radio to match the index of the network. The closer the joining radio gets to the network hop speed, the worse the non-missed channel case becomes. At $(n-1)/n$ the search is $n^2/2$, and if timing is performed in real time as opposed to integer frequency hop rounds, then as the joining radio search speed approaches the network hop speed the time to rendezvous approaches $\infty$.

**Syncing.**

This process is limited by the specifics of the hardware and actual transmission data mechanisms. Some number of successful rendezvous attempts ($\alpha$) will be required out of a total number of attempts ($\beta$) before failure and timeout. In the symmetric environment, these values are a constant, except in the false positive case dealt with in the seeking phase analysis above.

In the asymmetric environment there is a probability that rendezvous will fail even though timing was successfully synchronized. Again $p$ represents the overlap percentage between channels. The joining radio will generate $\beta$ attempts, a required $\alpha$ number of successes, and a $p$ probability that any given attempt will succeed. This leads to a Bernoulli trials formula of $b(\alpha, \beta, p) = \frac{\beta!}{\alpha! * (\beta-\alpha)!} * p^{\alpha} * (1-p)^{\beta-\alpha}$. However, that only gives the probability of exactly $\alpha$ successes and $\beta - \alpha$ failures out of $\beta$ trials, where this actually only requires at least $\alpha$ successes. Therefore, the easiest way to calculate it is to compute $b(0, \beta, p)$ through $b(\alpha - 1, \beta, p)$ to find the probability of

failure then subtract that from 1. Giving a final probability of $1 - \sum_{x=0}^{\alpha-1} b(x, \beta, p)$. Therefore, this phase will require $n/(1 - \sum_{x=0}^{\alpha-1} b(x, \beta, p))$ total hops.

**Network.**

In this phase the algorithm is considered complete as rendezvous has already occurred.

**Conclusions.**

The algorithm is guaranteed to rendezvous eventually so long as there are at least $\alpha$ overlapping channels and the window size is correctly set to twice the maximum bound of timing offset between any two radios in the network.

On average, in a symmetric environment, the MH algorithm will take $n/2 * (1 + \beta/c) + \alpha$ hops to rendezvous, or simply $O(n)$ where $n$ is the size of the sliding window. There is an additional $O(c)$ initialization factor, but this can run at the maximum speed of the radio processor as opposed to being limited by the frequency hop rate and will have minimal impact relative to the $O(n)$ timing for any realistic value of $c$.

In an asymmetric environment the MH algorithm will take

$$\left(\frac{n}{2} + n * \left(\frac{1}{p} - 1\right) + \frac{n}{p} * \left(\frac{1}{1 - \sum_{x=0}^{\alpha-1} b(x, \beta, p)} - 1\right)\right) * \left(1 + \frac{\beta}{c}\right) + \alpha$$

hops where $n$ is the size of the sliding window, $p$ is the overlap percentage of channels between $0.0 < p < 1.0$, $c$ is the total number of channels, and $b$ is a function representing a Bernoulli trial as described in Syncing above. As the overlapping number of channels approaches zero, the asymmetric rendezvous time approaches infinity. That formula is backwards compatible with symmetric environments as well where $p = 1.0$. The formula is validated successfully against simulation results in chapter IV.

# IV. Simulation Setup and Results

### Objectives.

This simulation based experiment measures the mean time to rendezvous of the MH algorithm against other rendezvous algorithms from the literature in different environmental conditions. Additionally, the simulation determines the failure rate of the algorithm, where failure is defined as reaching 100,000 rounds without successfully completing rendezvous.

### Assumptions.

The following assumptions are made in the setup of the simulation: An entire network of radios can be modeled as a single radio because every radio in the network would have identical parameters. Timing can be represented as a discrete round or time slot, equal to the length of a single frequency hop in the network. Radio hardware is abstracted away and would be capable of supporting all of the required operations of the algorithm. All radios have a pre-shared randomly generated seed. One radio has been configured by a user as a master to initialize the network. There is no interference or jamming present. All channels are identical in performance. Each radio has the same number of channels available. Channels do not change, appear, or disappear for the duration of the simulation. All radios are within a bounded clock time of each other, i.e. the difference between any pair of radios is no greater than $X$ milliseconds, which has been represented by a maximum number of rounds of offset. The clock of each radio will not drift for the duration of the simulation.

#### Response Variables.

The measured response variable for this simulation is the number of rounds between the start of the simulation and the completion of the rendezvous of the new radio into an existing network

#### Control Variables.

The control variables for these simulation experiments are: The total number of channels available per radio. The number of channels that are not shared by all of the radios in the simulation, must be less than the total number of channels for rendezvous to occur and should improve performance as the number decreases. The maximum difference in clock timing between each radio, set to a constant factor of 5 rounds, the larger the number the easier input is for the operator but the larger the sliding window becomes.

#### Constant Factors.

The factors being held constant for these simulatin experiments are: The number of radios, fixed at 2 radios because one radio represents the entire existing network and the other radio is attempting to join the network. The number of hops required to synchronize successfully, in real hardware is a function of the amount of data and the data transmission rate. Number of synchronization hops before timeout, set to twice the number of hops required to synchronize successfully, once synchronization begins a timeout is required to allow for failure recovery and re-acquisition of the net-

**Table 1. Response Variables**

| Response Variable | Normal Level | Precision | Relationship to Objective |
|---|---|---|---|
| Time to rendezvous | 0-1000 ms | 1 ms based on simulation clock | Estimate mean and standard deviation |

**Table 2. Control Variables**

| Control Variable | Normal Level | Precision | Proposed Settings | Predicted Effects |
|---|---|---|---|---|
| Channels per radio | 10-50 | 1 channel | [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 250, 500, 750, 1000] | Increasing the number of channels will increase the time taken to rendezvous until it exceeds the sliding window size |
| Non-overlapping channels | Unknown | 1 channel | [1, 5, 10, 500] | Increasing the number of non-overlapping channels will increase the time taken to rendezvous |
| Clock bound difference | 500ms | 1 ms | [50ms, 500ms, 5000ms] | Will increase the time taken for the Multihop algorithm to rendezvous linearly but has no impact on other algorithms |

work. If this value is too low the algorithm may abort during an otherwise successful rendezvous attempt, if too high then bandwidth is wasted on coding for the misses and time wasted on false positives during sliding window searches. Search speed as a percentage of standard hop speed, due to offset timing (not every radio has identical timing down to the millisecond), searching the sliding window is required to identify the timing of the network and the search speed must not be equal to the network hop speed for overlap to occur. Failure timeout at 100,001 rounds, the cutoff point at which an algorithm is stopped and labeled as a complete failure to rendezvous.

**Table 3. Constant Factors**

| Factor | Desired Level | Precision | Anticipated Effects |
|---|---|---|---|
| Number of radios | 2 | 1 radio | None |
| Number of hops required to synchronize successfully | 10 | 1 hop | None |
| Timeout hops | 20 | 1 | Moderate |
| Search speed | 200% | 1% | Moderate |
| Failure timeout | 100,001 | 1 | None |

#### Simulation Environment.

The source code is included in Appendix A and an additional copy of the source and inputs to the simulation along with the collected raw data is available on GitHub at `https://github.com/tinamil/CR-Simulator` It is being executed on the Java 1.8.25 JRE and Windows 10 x64 on a single workstation running a Core i5-3570K CPU and 8GB of RAM.

#### Initial Tests.

First tests were performed to verify whether the MH algorithm worked as expected and to check the code for implementation bugs. Trial and error determined that 50% search speed failed entirely initially with 200% and higher search speeds worked as long as the sliding window was set to the proper 2x+1 value of the maximum offset and the environment was symmetric. Once the initial sliding index was initialized to half the sliding window so that the search could go both forward and backwards then the MH algorithm worked in all cases.

#### Symmetric Test Results.

The charts in figure 2 and figure 3 provide the mean rounds to achieve rendezvous of the Multihop algorithm for comparison against the other rendezvous algorithms from the literature for 10-100 channels and 100-2000 channels. All of the algorithms were implemented into the same simulation software, with the number of overlapping channels set to 100%, making the environment symmetric.

The charts in figures 4 through 9 show the results of a Tukey Honest Significant Difference (HSD) analysis of the Multihop algorithm at each tested window level against each of the other rendezvous algorithms at each channel level. Jump-Stay had the consistently best performance, other than MH, at all channel levels tested.

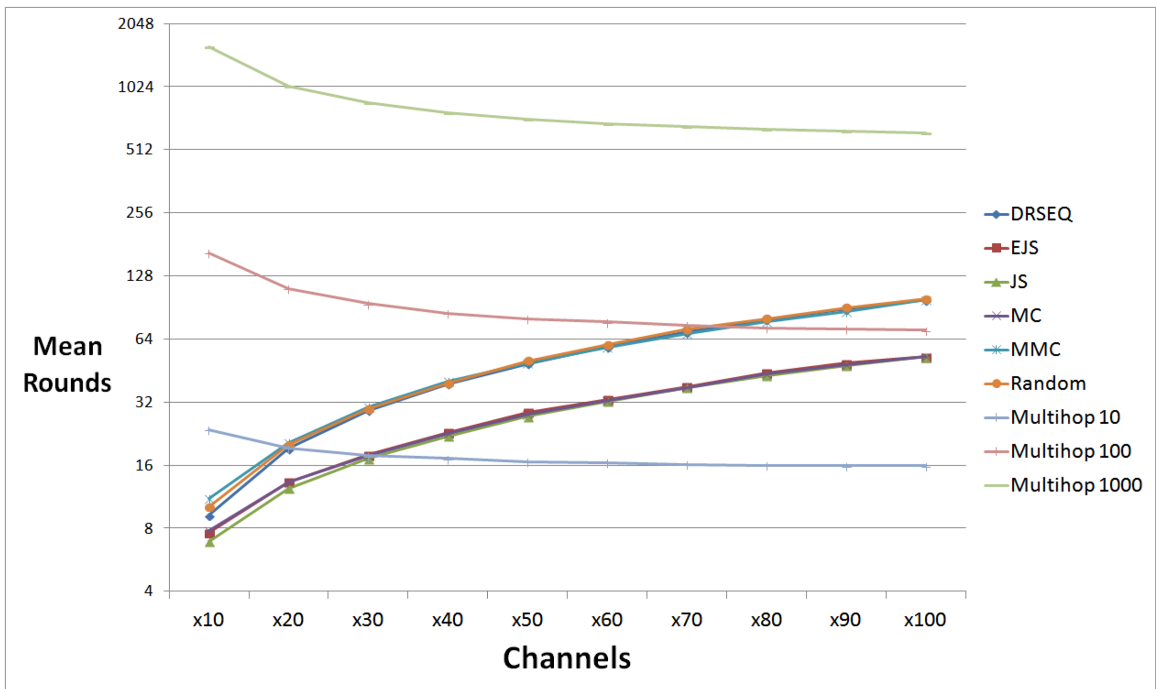



**Figure 2. Logarithmic scale comparison of mean rounds to rendezvous simulation results for 10-100 channels in a symmetric environment (100% channel overlap). The Multihop algorithm is tested with window sizes of 10, 100, and 1000 channels.**

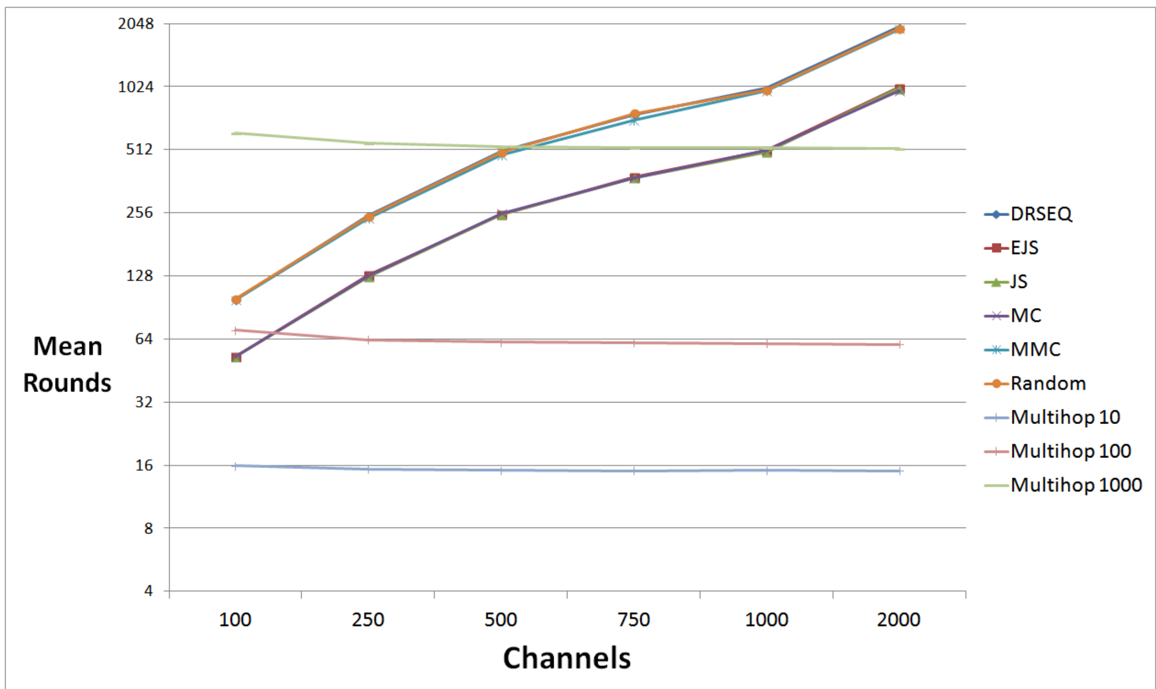



**Figure 3. Logarithmic scale comparison of mean rounds to rendezvous simulation results for 100-2000 channels in a symmetric environment (100% channel overlap). The Multihop algorithm is tested with window sizes of 10, 100, and 1000 channels.**

Specifically JS was always statistically equal or better than other algorithms from the literature that were tested. The MH-10 algorithm performs worse than JS at 10 channels, equal from 20-40 channels, and better from 50-2000 channels. MH-100 is worse from 10-100, and better from 250-2000 channels. MH-1000 is worse at 10-750, equal at 1000, and better at 2000 channels.

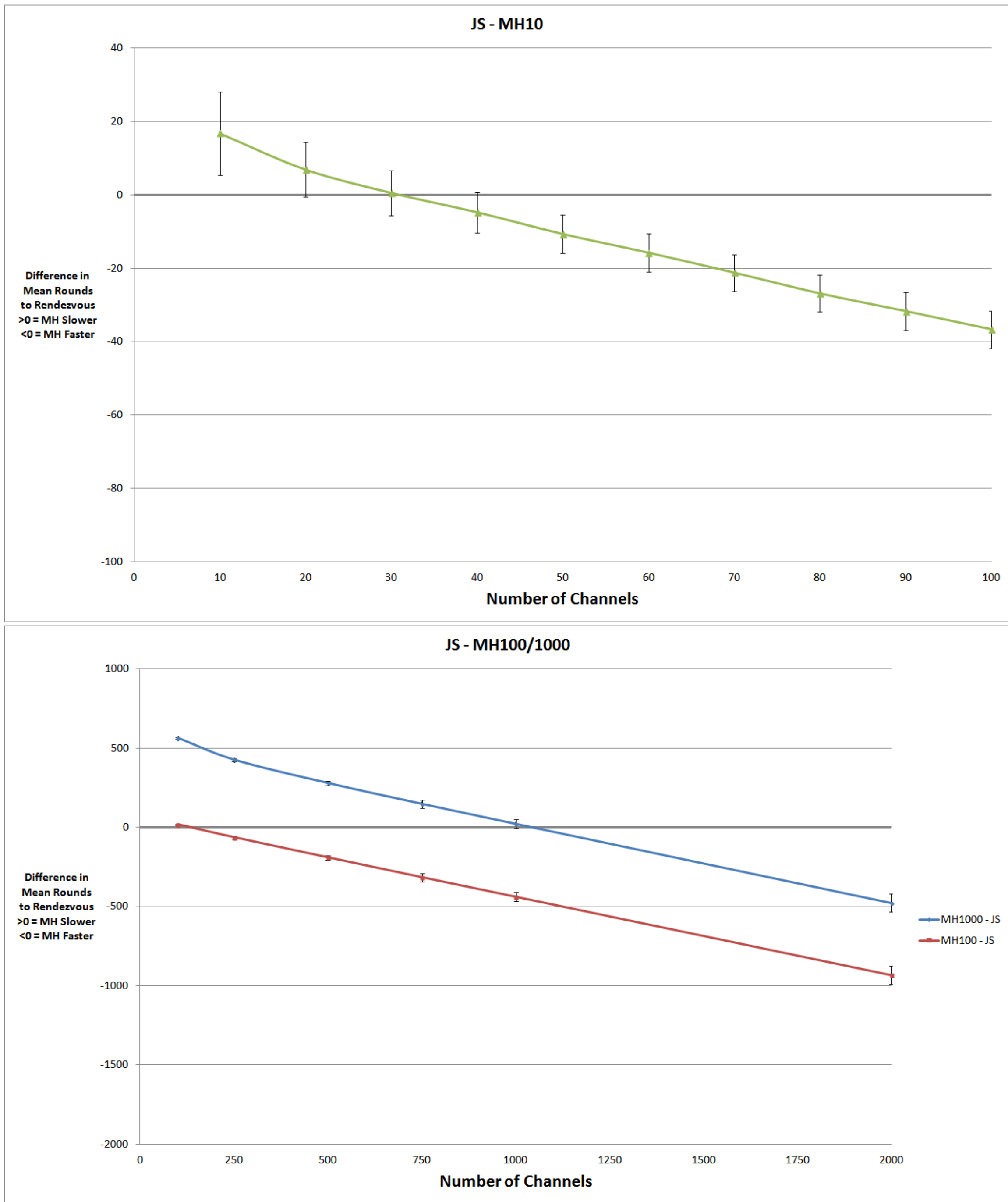



**Figure 4. Tukey HSD comparison of Multihop against JS showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster.**

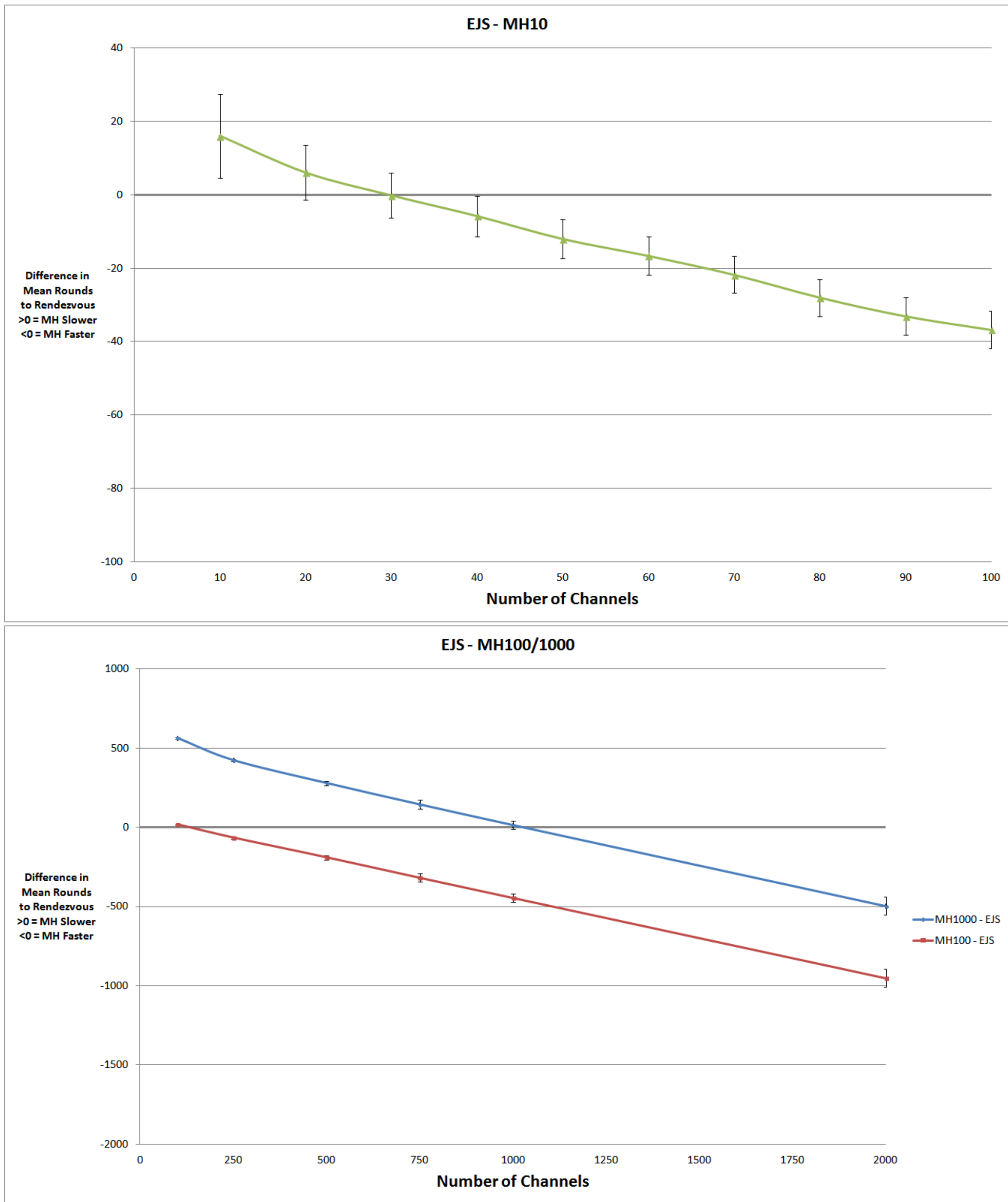



**Figure 5. Tukey HSD comparison of Multihop against EJS showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster.**

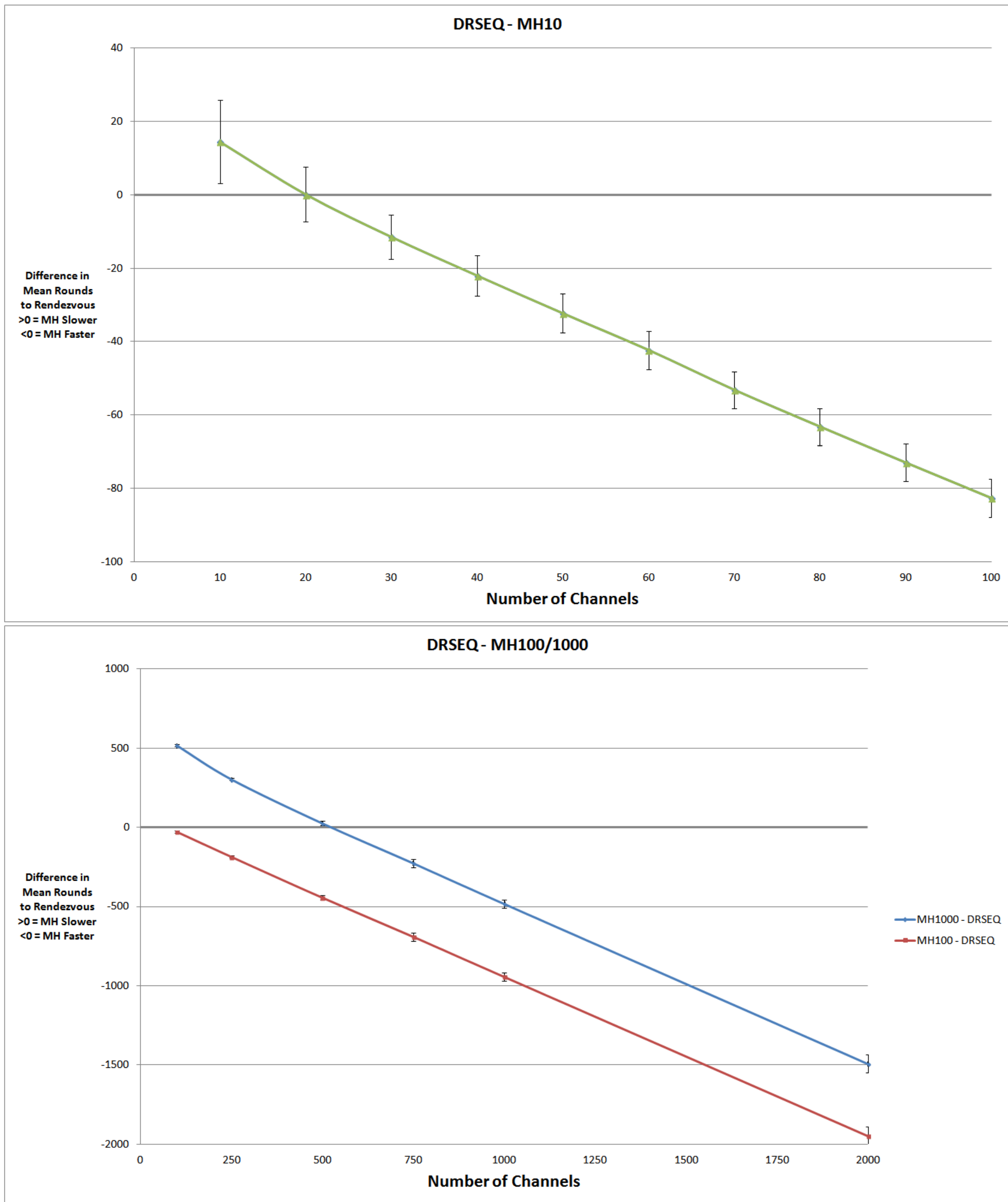



**Figure 6. Tukey HSD comparison of Multihop against DRSEQ showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster.**

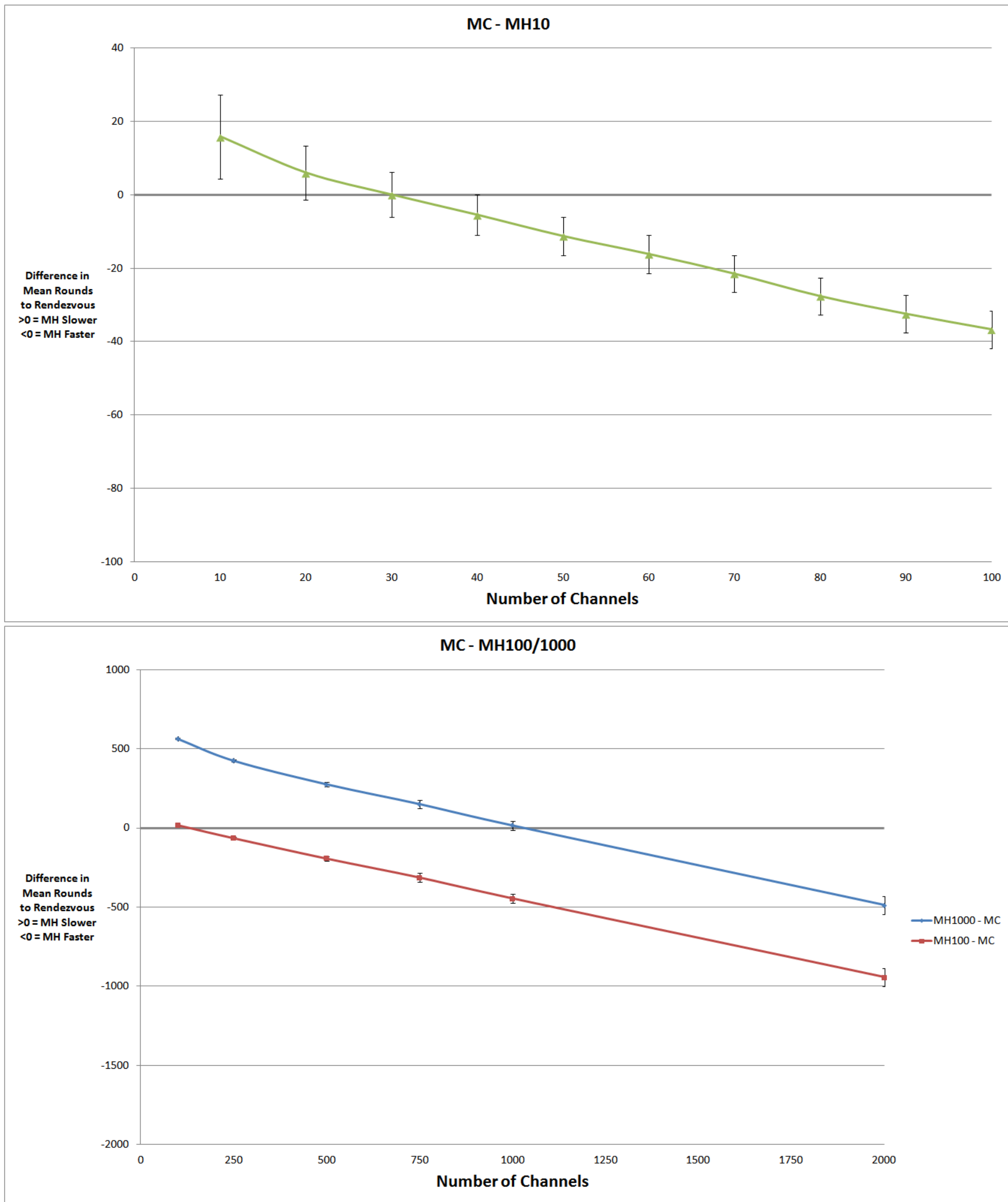



**Figure 7. Tukey HSD comparison of Multihop against MC showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster.**

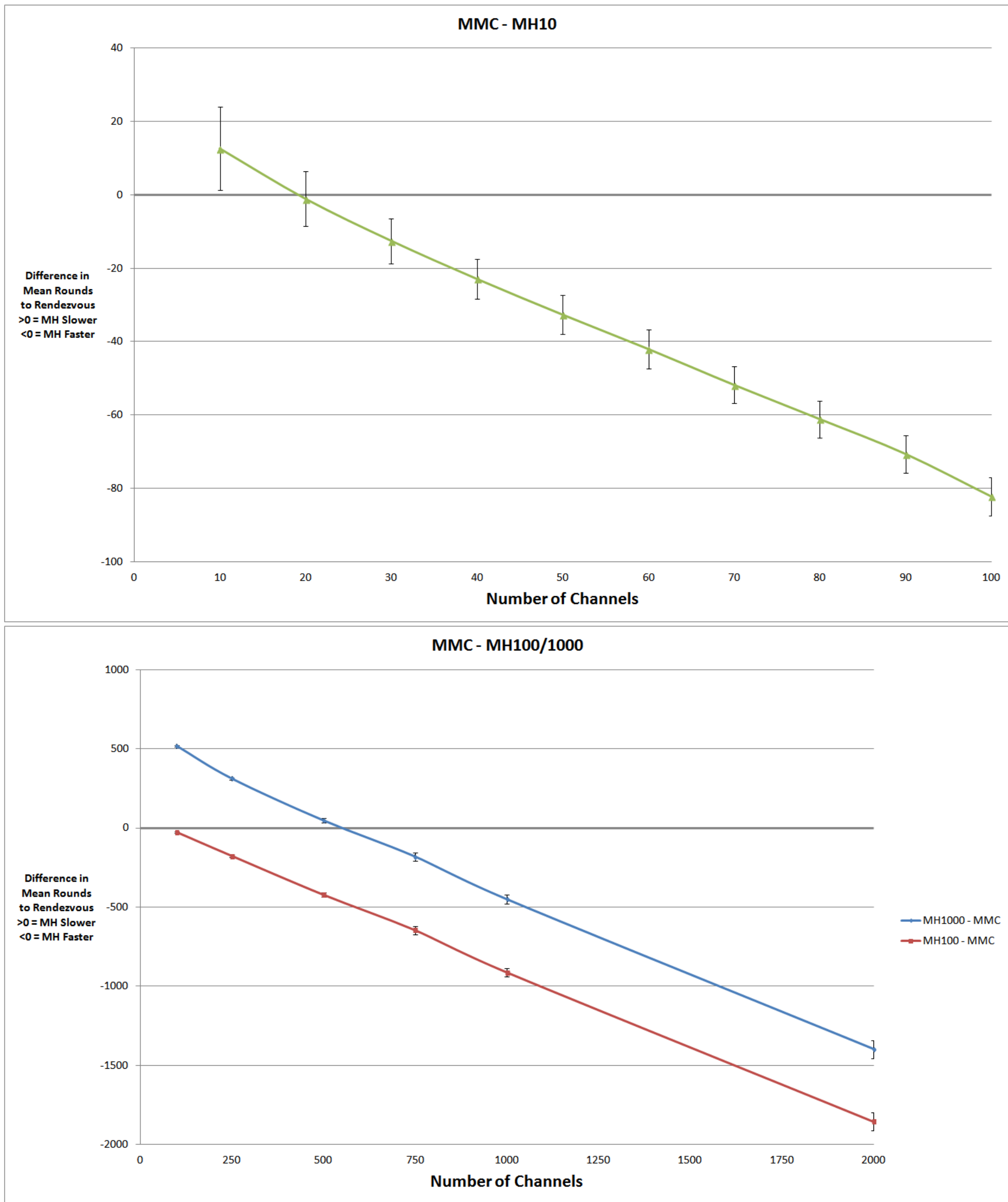



**Figure 8. Tukey HSD comparison of Multihop against MMC showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster.**

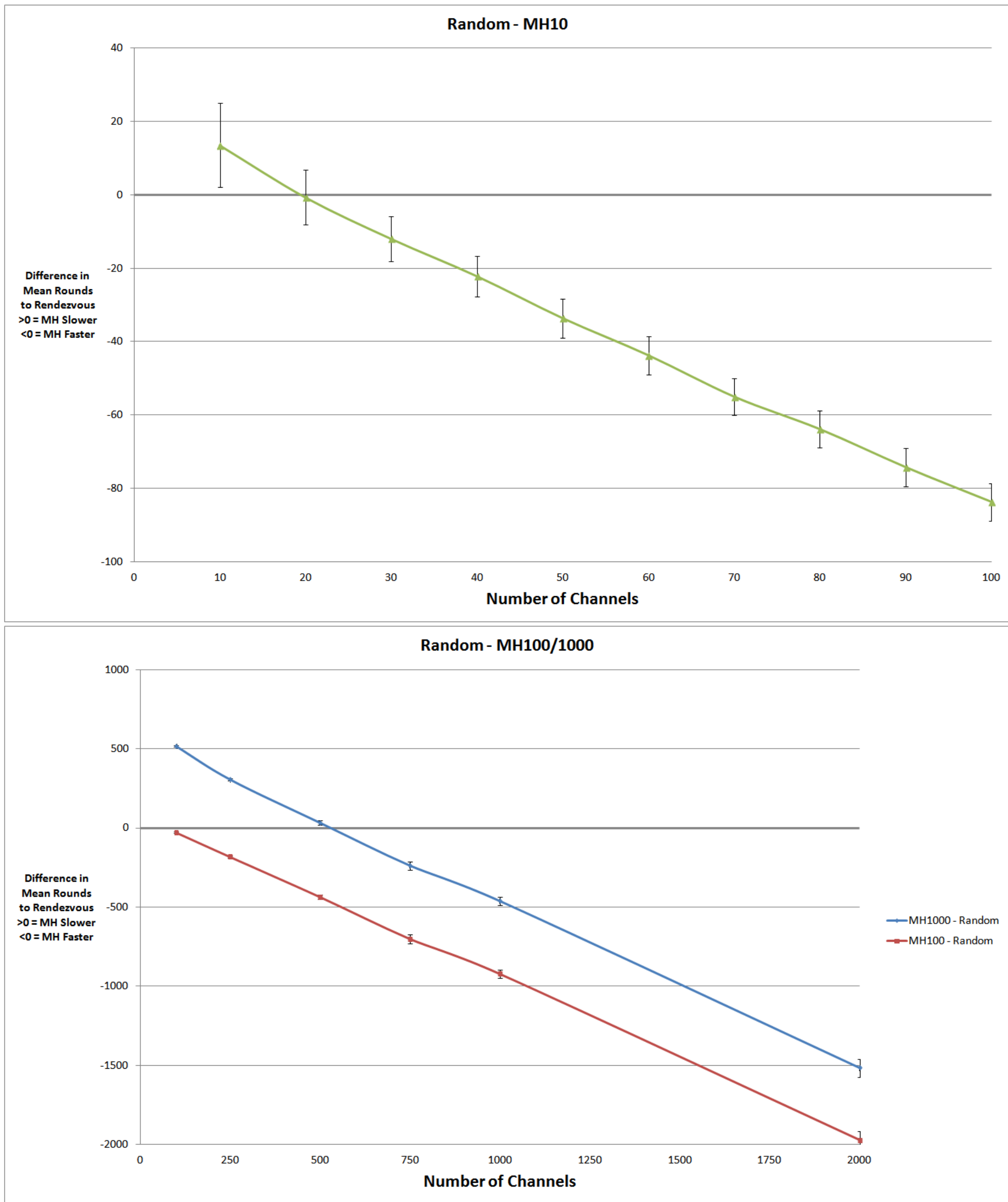



**Figure 9. Tukey HSD comparison of Multihop against Random showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster.**

### Asymmetric Test Results.

Initial tests in an asymmetric environment with with 500 channels overlapping of 1000 channels total, but with each channel assigned a random signal strength and sorted by that strength, showed a $< 1\%$ success rate. Chapter III indicates the algorithm is strongly dependent upon the channel difference rate between each radio, so I performed a series of tests to determine those channel difference rates for different asymmetric environments with randomly assigned noise values to each channel used to sort as opposed to channel IDs. Channel difference rate is measured by performing a comparison of each table to see if the channels across each radio at the same index were equal. Table 4 shows two examples of how difference rate measurements work. The left example is a low difference rate that works because most of the channels are identical in each row, while the right example has a high difference rate because each channel in the second row has been offset from the first.

Figure 10 shows the channel difference rate distributions for an asymmetric environment when sorted by randomly assigned noise. The test was performed by generating a random noise value for each channel in the range of $[0, 1)$, creating two radios that each selected 1000 channels, and setting the channel overlap values. The channel non-overlap values were set to 1, 5, and 10, corresponding to 999, 995, and 990 overlapping channels. Due to the sorting, the larger the difference in noise value between the non-overlapping channels, the smaller the channel correlation will be. Using the previous example, if channels [1, 5] have extremely different noise values then the radios generate [1, 2, 3, 4] and [2, 3, 4, 5], but if [1, 5] have identical noise

**Table 4. Example of two different asymmetric difference rate measurements**

12.5% Difference

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 9 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

100% Difference

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

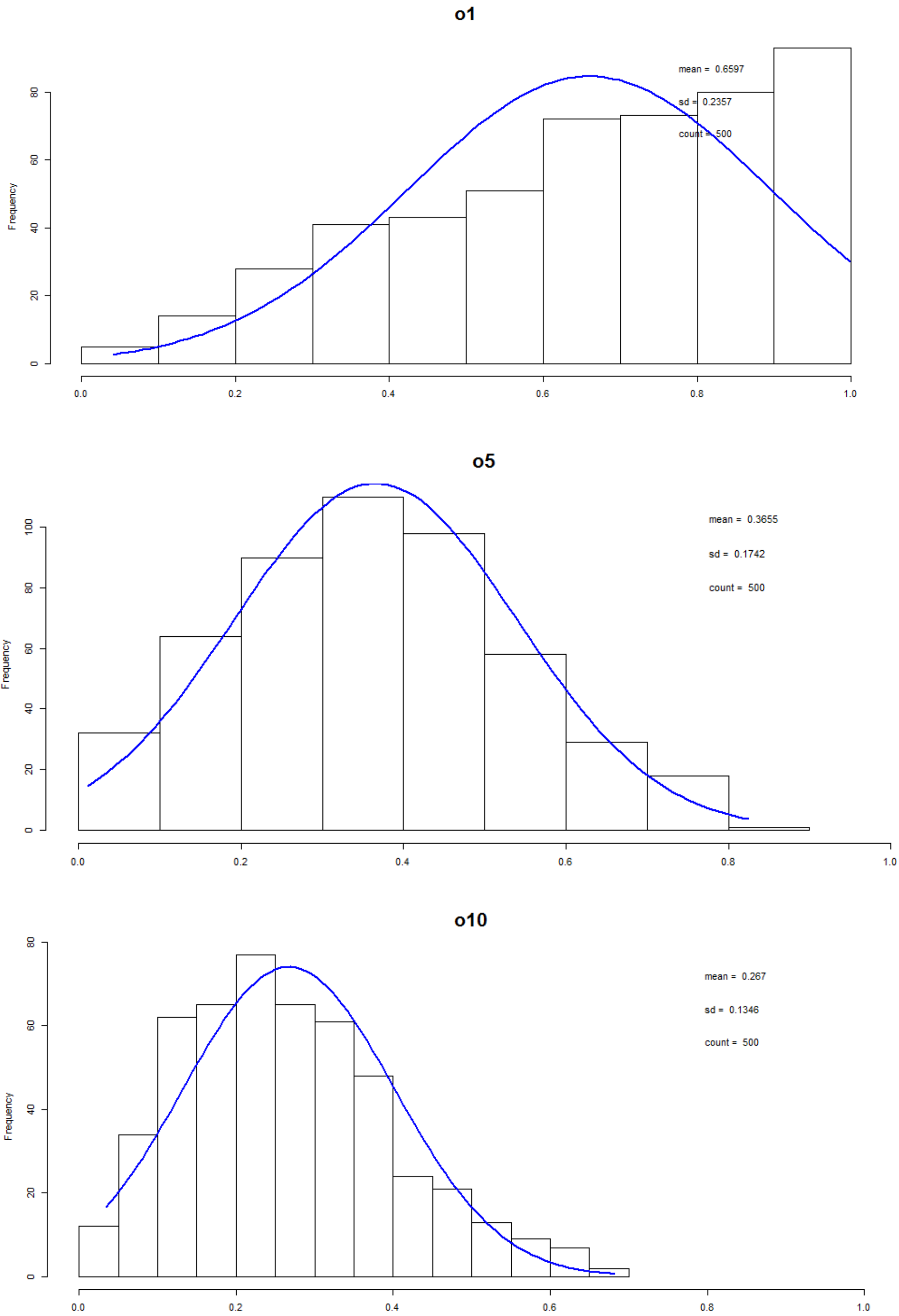



**Figure 10. Channel correlation values of MH algorithm for 1000 channels in an asymmetric environment using randomly generated noise values in the range of $[0, 1)$ with 1, 5, and 10 non-overlapping channels.**

values then they would have been placed into the same index generating [1, 2, 3, 4] and [5, 2, 3, 4].

This suggests that using noise as a sorting method for real world implementation of the algorithm may not work, especially if dramatically different noise values are being registered by different radios due to environmental conditions. If there is one badly misplaced channel then that creates a very large difference between two radios, which would make rendezvous very unlikely. As shown in the o5 graph in figure 10, even just 5 out of 1000 channels being different with randomly assigned noise values creates a mean of a 36% overlap and 64% difference rate. Using the asymmetric analysis from chapter III, that 36% overlap would require $7.2 * n$ rounds on average just to successfully move from the seeking to synchronization phase.

After discovering how badly random noise could perform, this variable was eliminated in future tests by explicitly controlling overlap values. Channels were sorted by channel number instead of noise and assigned channels such that they would have exact overlap. For example, a 50% overlap with 1000 channels would involve radio 1 having channels 1-1000 and radio 2 having channels 1-500 and 1001-1500.

Figure 11 shows the mean rounds to rendezvous of the Multihop algorithm as compared against the Random and MMC algorithms, with figures 12 and 13 showing the statistical differences between the means. The EJS algorithm was the only other implemented asymmetric rendezvous algorithms, but was not included in this test because it gave abnormal results by frequently ($> 10\%$ of tests) failing to rendezvous entirely.

The Multihop algorithm performs best relative to the other tested algorithms when the difference rate was near the $\alpha/\beta$ ratio, approximately 0.5 for these simulations. Below that point the MH algorithm becomes exponentially worse. In the other direction, as the overlap percentage improves the MH algorithm improves its

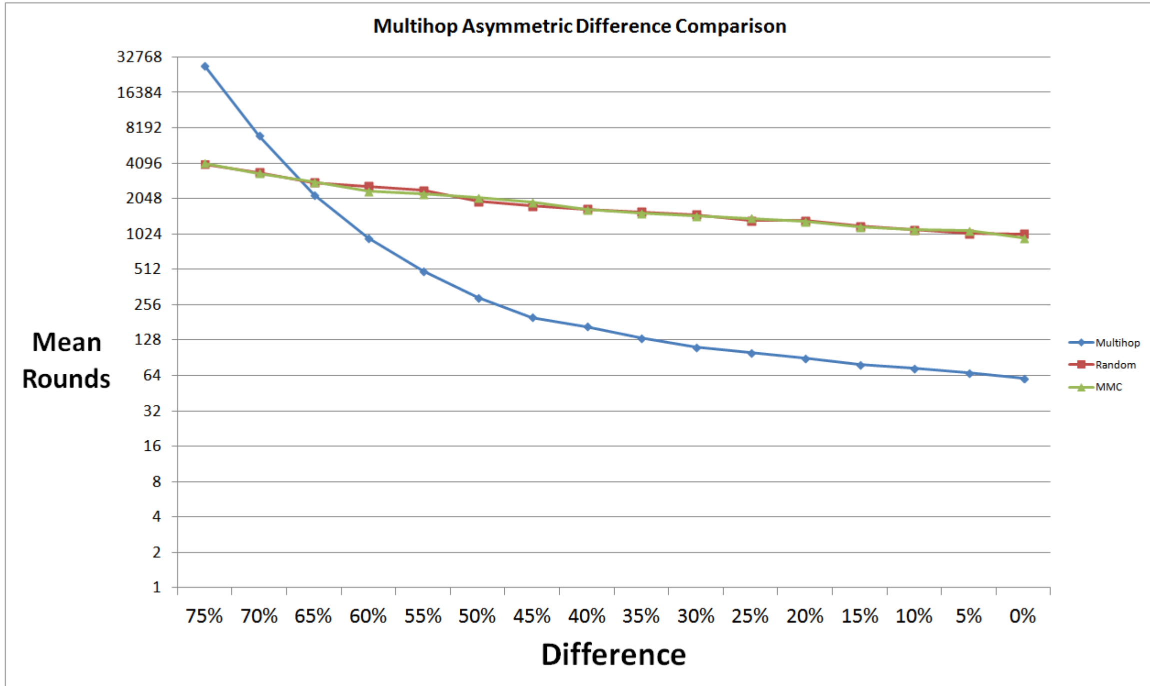



**Figure 11. Logarithmic scale comparison of mean time to rendezvous of the Multihop-100 algorithm with a window size of 100 compared against the Random and MMC algorithms at 1000 channels.**

runtime, but at a slower rate than the other algorithms.

### Timing Validation.

The simulation results of the MH algorithm match the expected results of the symmetric and asymmetric model runtime analyses. The symmetric and asymmetric results are presented in figures 14 and 15, respectively. The simulated results do not always statistically equal the calculated results, with the largest difference being 15.6% between the means at the 65% difference rate, however the results are not consistently above or below. At 65% difference the simulated results were 15.6% faster than calculated while at 40% difference the simulated results were 8.4% lower and the calculated results and would have been within the confidence interval if .995 was used instead of .95. This indicates that the runtime analysis from chapter III is reliable and can be used to predict performance values for known values of the window

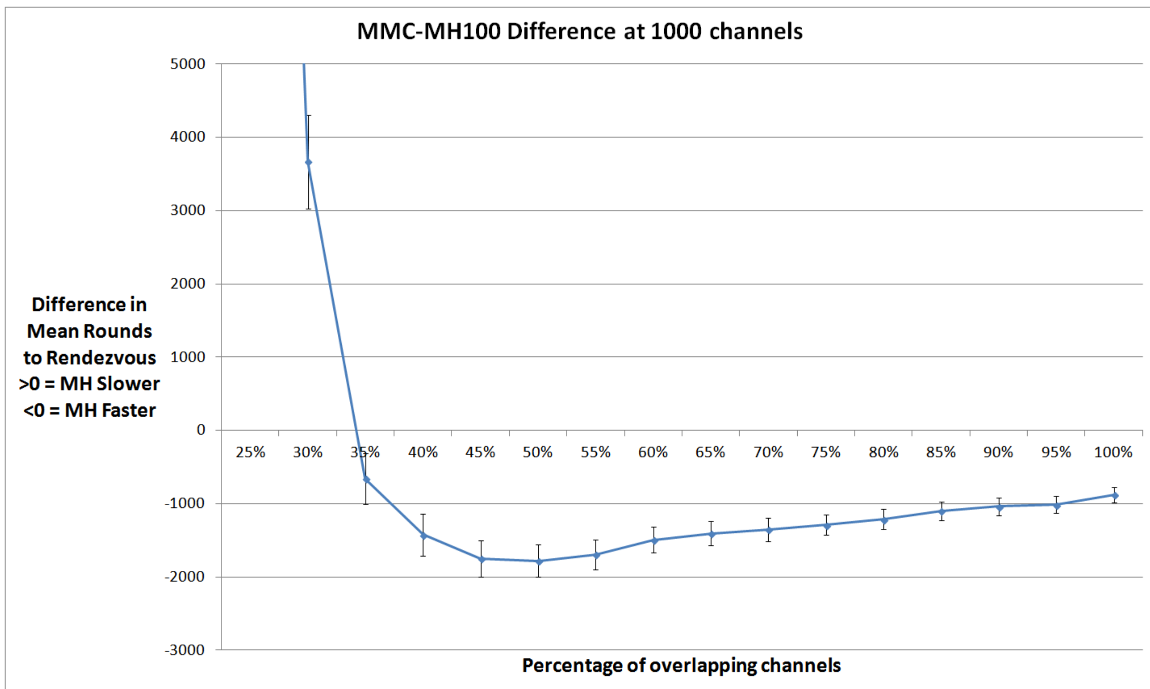



**Figure 12. Tukey HSD comparison of Multihop against MMC showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster.**

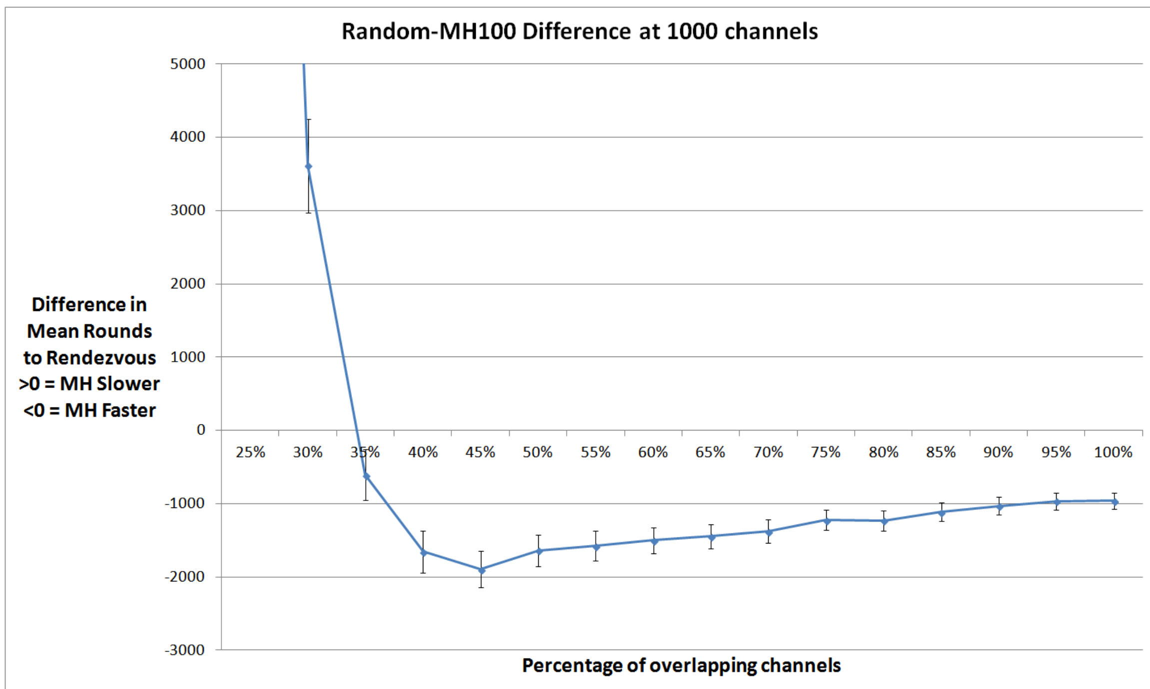
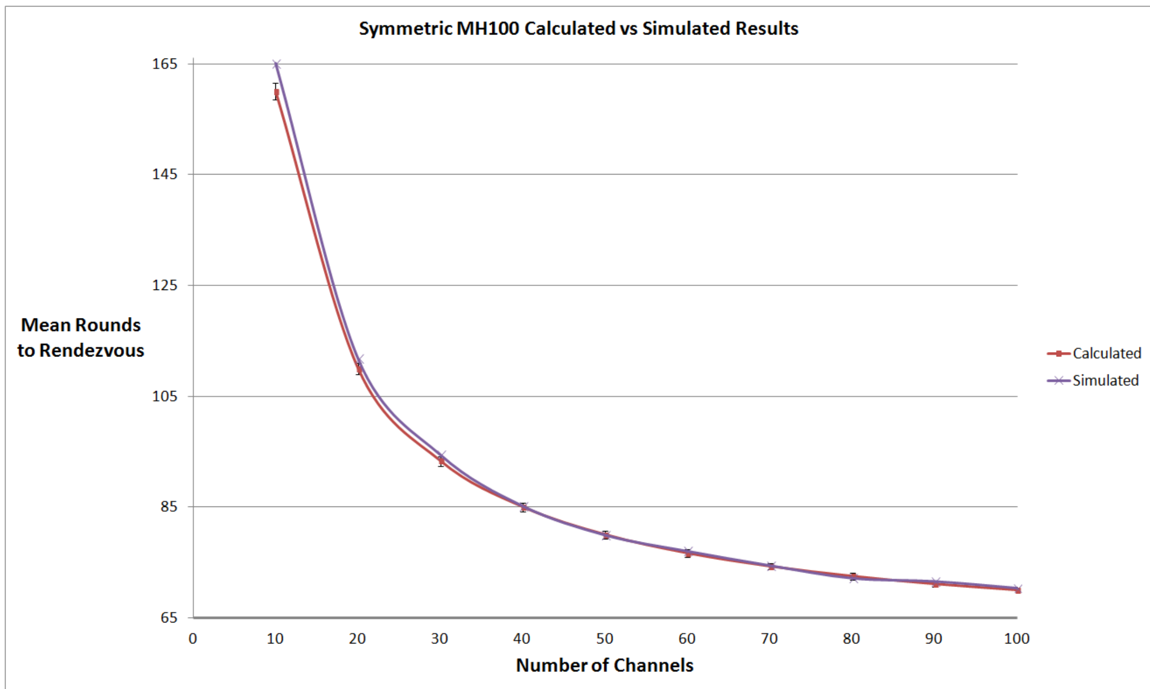



**Figure 13. Tukey HSD comparison of Multihop against Random showing the difference in mean rounds to rendezvous. Error bars show 99% confidence intervals. A value greater than zero indicates the MH algorithm is slower and a value less than zero indicates the MH algorithm is faster.**



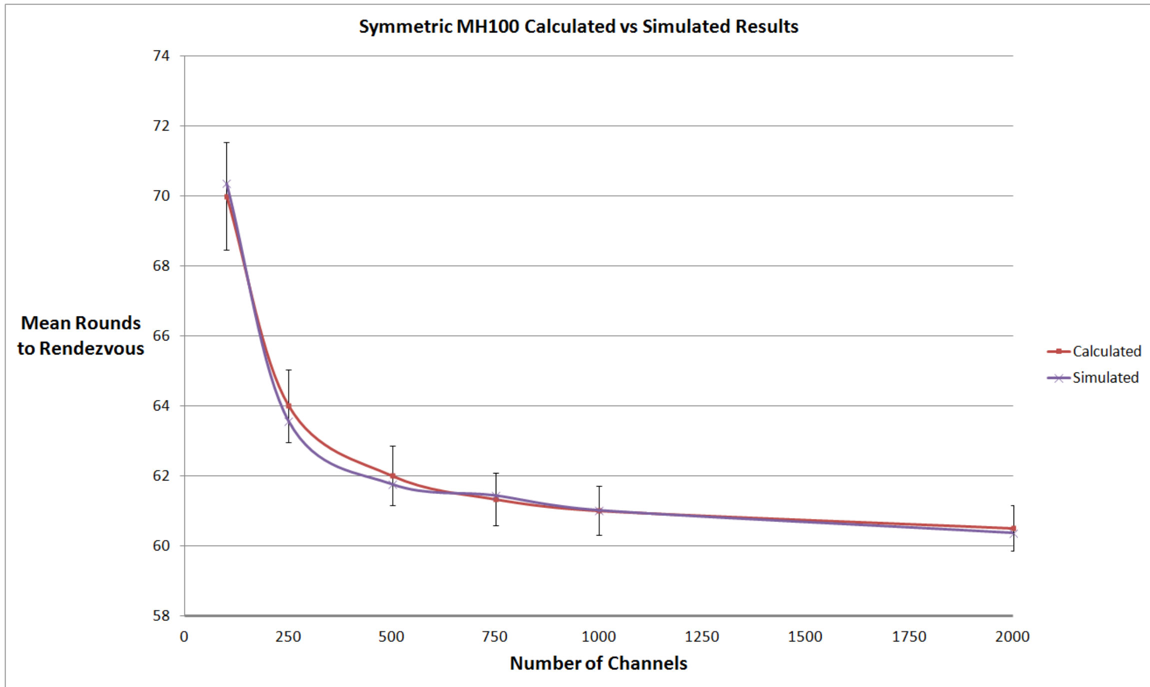




**Figure 14. Comparison of the calculated vs simulated results of the multihop algorithm using a window size of 100 in a fully symmetric environment. Error bars show 95% confidence intervals. The rounds to rendezvous converges towards 60 because the MH100 takes 50 rounds on average ($n/2$) to seek the timing and 10 rounds ($\alpha$) to complete synchronization.**

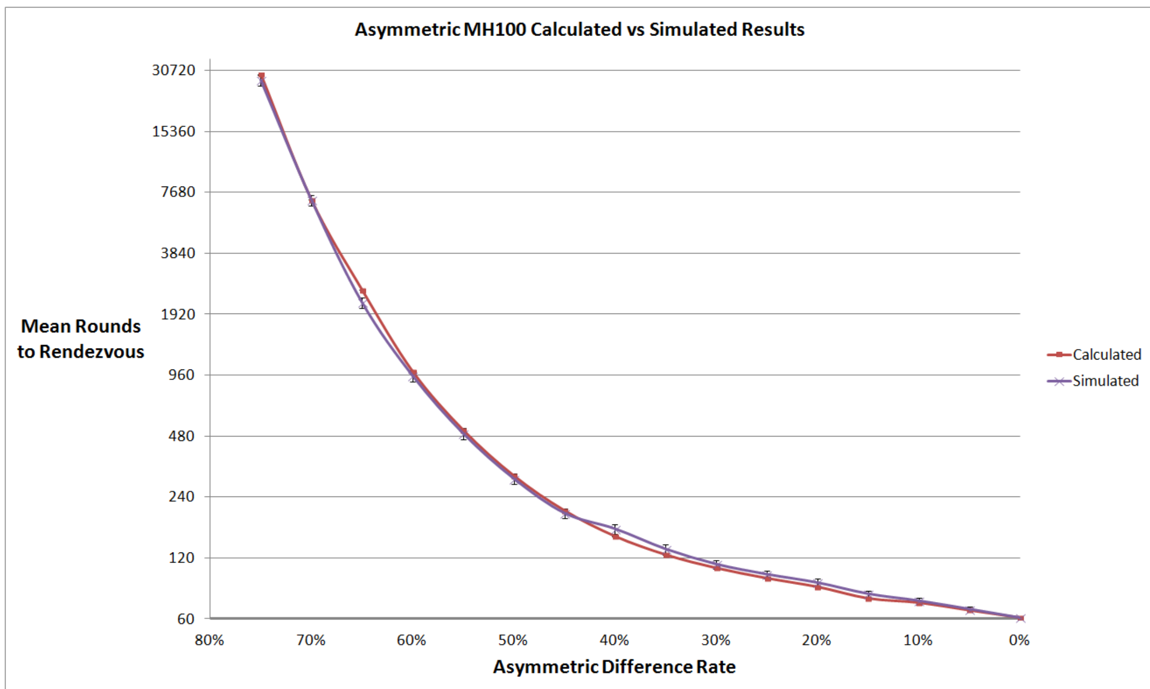



**Figure 15. Logarithmic comparison of the calculated vs simulated results of the multihop algorithm using a window size of 100, fixed at 1000 channels, for an asymmetric overlap ranging from 25%-100%. Error bars show 95% confidence intervals.**

size, channel count, percentage overlap, $\alpha$, and $\beta$ parameters for the MH algorithm.

### Conclusions.

These simulations have shown the potential performance of the Multihop algorithm relative to other well known rendezvous algorithms from the literature. This addresses the second research questions, specifically "Does the proposed algorithm have a faster mean time to rendezvous than existing algorithms?" Under certain conditions, yes, the proposed multihop algorithm does achieve a statistically significant faster mean time to rendezvous than existing algorithms. Those conditions are for small timing bounds and large channel counts, and for asymmetric overlap near the $\alpha/\beta$ ratio. Under other conditions, i.e. large timing bounds and small channel counts, the multihop algorithm performs significantly slower than existing rendezvous algorithms.

# V. GNU Radio Hardware

The goals of implementing the Multihop algorithm in hardware are to demonstrate that the algorithm is viable in real world conditions and to validate the simulation results. The MH algorithm is being implemented on the HackRF One through the GNU Radio software.

### Transmitter.

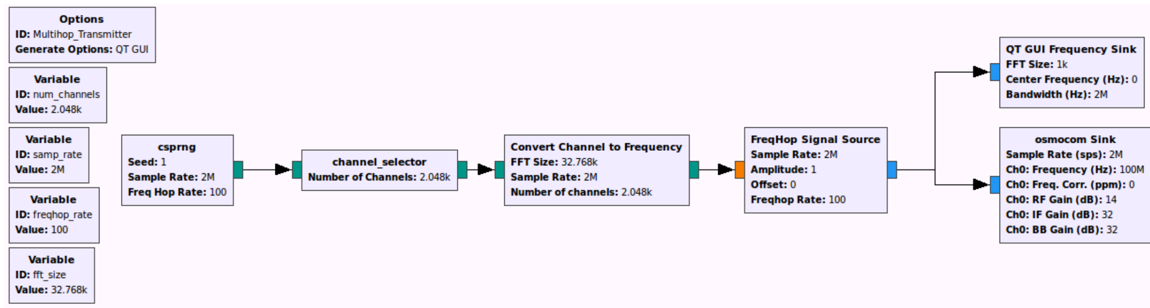



**Figure 16. The setup of the hardware transmitter as implemented in the GNU Radio Companion software. The software selects a channel, converts it to a frequency, and then the signal source generates a sin wave to be transmited to the osmocom Sink block which feeds those values to the HackRF One radio for transmission over the air.**

Figure 16 shows a design implementation of the Multiple Hop algorithm for a radio set to be the network master. For the purposes of testing the algorithm a single radio was utilized to act as the network with one radio attempting to join the network. These specific radios do not easily support actual frequency hopping, so a digital channelizer is utilized to demonstrate the algorithm. This will have no effect on the results of any experiments performed except to make the implementation easier. The transmitting radio constantly sends a carrier wave which represents the channel current and which changes frequency at the freqhop_rate per second.

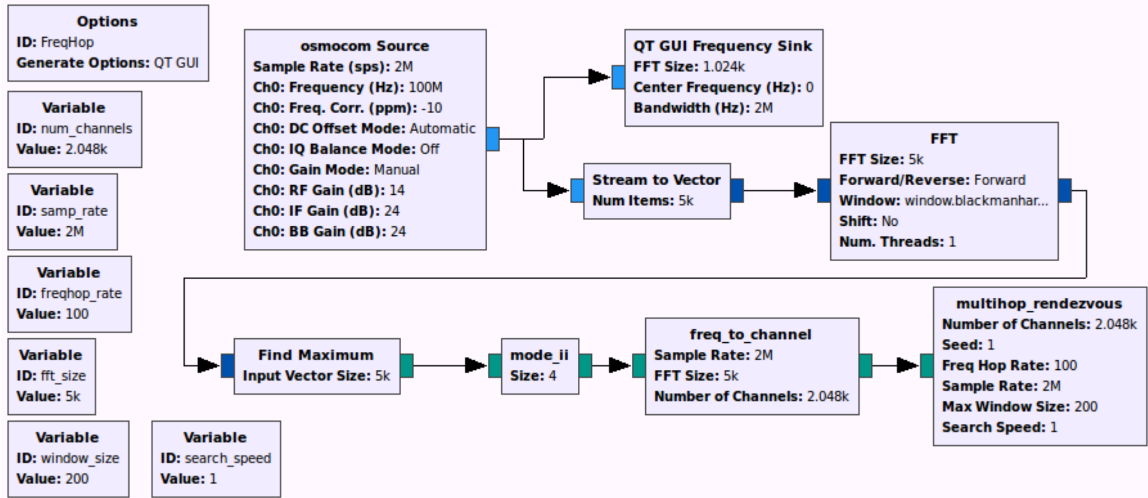



**Figure 17. The setup of the hardware receiver as implemented in the GNU Radio Companion software. The osmocom source reads from the HackRF One radio, then the results are processed through an FFT to find the strongest signal source, converts that value back into a channel index, and feeds that into the rendezvous algorithm for comparison to see if the right values were generated.**

### Receiver.

Figure 17 shows the final implementation of the Multiple Hop algorithm in a joining radio. The receiver reads the carrier wave, converts that into a channel index, and feeds that into the Multihop Rendezvous algorithm. The algorithm then independently generates the channel it is listening to and compares that to the input channel to see if they match. This demonstrates the algorithm works, and converting this to an analog frequency hop system is trivial from a design perspective. The only necessary change is to invert the design so that the Multihop rendezvous block outputs the channel to which the radio listens.

### Hardware.

The full source code for each custom block is included in Appendix B. The FFT, osmocom, and QT GUI blocks were provided by the GNU Radio Companion software and the QT GUI Frequency Sinks are only necessary as instrumentation to allow the

operator to see what is happening in real time. For these experiments the GNU Radio Companion software was run on two laptops. The transmitting laptop was a Lenovo Thinkpad T430 with an Intel Core i5-3210M CPU at 2.5GHz and 12 GB of RAM, running x64 Linux Mint 17.3 inside a VirtualBox VM on Windows 10. The receiving laptop was a Lenovo Thinkpad Z61t with an Intel Core Duo T2300 CPU at 1.66GHz and 2GB of RAM, running x86 Linux Mint 17.3. Both laptops were connected via USB to a HackRF One radio with a 3db antenna attached.

**Problems.**

Clock skew is a significant problem between these radios. Each radio is set to run at exactly 2,000,000 samples per second but they will drift by as much as a 20,000 samples every five seconds. It's not a significant problem to the rendezvous so long as the drift stays within the bound on time, but it's an issue that has to be solved after rendezvous to prevent desynchronization. Additionally, due to the difference in clocking a manual frequency correction of -10 ppm was required on the receiving radio to synchronize the frequencies so that the frequency sent was the same as the frequency received.

**Results.**

The results of the hardware version of the algorithm as compared to the simulation results are located in table 5. Results are significantly worse than in simulation.

**Table 5. T-test comparison of mean rounds to rendezvous for simulation vs hardware results for the Multihop Algorithm in a symmetric enviroment with 2000 channels at 100 hops per second.**

| Window size | Simulation | Hardware | 95% Conf. Interval | P-Value |
|---|---|---|---|---|
| 10 | 15.0659 | 71.91667 | 33.6 - 80.1 | 0.0002213 |
| 100 | 60.356 | 192.8333 | 107.6 - 157.4 | $9.238 * 10^{-12}$ |
| 1000 | 516.209 | 1588.4 | 869.6 - 1274.8 | $1.058 * 10^{-11}$ |

There are two reasons for this. First, timing is manually controlled and both radios are started as close as possible together, as opposed to in simulation where each radio was given a random valid start time. This causes timing to skew towards $n$ rounds as opposed to $n/2$ for random start times. Secondly, the startup time required for the hardware in each radio causes the first potential rendezvous synchronization to fail as the radios generate garbage data until they have completed startup. This adds an $n$ factor to any run which misses a rendezvous that it would not have in simulation due to the imperfect hardware.

The specific size of the sliding window is determined by two factors, the maximum bound on the difference in clocks between the radios and the frequency hop rate. However, the algorithm operates at the speed of the frequency hop rate, so increasing the hop rate does increase the size of the sliding window and therefore the total number of rounds taken to rendezvous but simultaneously decreases the time spent in each round by a proportional amount so that the actual wall clock time taken to rendezvous remains constant. For manual clock timing in which both clocks were started by the same operator, $\pm$500ms was a good bound on the clock limit for being reliable, that corresponds to a window size of 100 at the frequency hop rate of 100 hops / second currently being utilized in the digital channelizer. The 50 ms timing limit of the 10 window size fails to rendezvous about 60% of the time due to the limits of human accuracy and the fact that the computers are running at different speeds and take different amounts of time to initialize the hardware and start the rendezvous sequence.

**Table 6. T-test comparison of mean rounds to rendezvous for simulation vs hardware results for the Modular Clock (MC) Algorithm in a symmetric enviroment with 50 channels at 1 hop per second.**

| Simulation | Hardware | 95% Conf. Interval | P-Value |
|---|---|---|---|
| 27.9264 | 39.0 | .802 - 21.35 | 0.03546 |

For comparison, the Modular Clock (MC) alrgorithm was implemented in hardware as well using the reference implementation published by Theis [33] and the results are presented in table 6. The Modular Clock algorithm hardware implementation is also slower than the simulation implementation, but only by approximately 40% whereas the Multihop algorithm was closer to 300% slower. The largest difference between the two hardware tests, Modular Clock against Multihop, is that the MC algorithm was only tested at 1 hop per second while the MH algorithm was tested at 100 hops per second. The Modular Clock algorithm was tested at only 1 hop per second because that algorithm was too slow at generating hop sequences to be tested at a faster rate using the specific implemenation provided by Theis for generating random numbers. The effect of this difference is that a fixed amount of startup time inherent in each radio means that the Modular Clock algorithm has gone through far fewer hops than the Multihop algorithm.

**Hardware conclusions.**

This demonstrates that the Multihop algorithm can feasibly be implemented into real world radio hardware, and therefore successfully answers the first research question "Does the proposed algorithm successfully allow a cognitive radio to rendezvous and join a frequency hopping network under real world conditions?" with a yes. Further engineering efforts are required to move the system from a prototype to a fully functioning radio, but this implementation has shown that there are no hidden issues to prevent the algorithm from being successfully integrated into full cognitive radios.

# VI. Conclusions

### Objective.

The research objective of this thesis was to provide an algorithm to allow a cognitive radio to join a network of frequency hopping cognitive radios. It does so by presenting the Multihop algorithm, showing how it works, and comparing it to existing rendezvous algorithms.

### Criteria for success.

The specific criteria selected for judging the success of the proposed Multihop algorithm were:

1. Does the proposed algorithm successfully allow a cognitive radio to rendezvous and join a frequency hopping network under real world conditions?
2. Does the proposed algorithm have a faster mean time to rendezvous than existing algorithms?

Question 1 was answered by chapter V, which demonstrated the feasibility of the algorithm by implementing it on a real radio and testing rendezvous through actual radio transmissions. Question 2 was answered by chapter IV, which showed that the Multihop algorithm is time dependent where other algorithms are channel dependent. Therefore, the Multihop algorithm is faster when time differences are small and channel counts large, while other algorithms are faster when time differences are large and channel counts small.

### Significance.

The Multihop algorithm is the only known algorithm which allows for rendezvous to occur over multiple successive hops. All other known algorithms are based on

rendezvous being completed in a single hop. Furthermore, the MH algorithm is one of few rendezvous algorithms which allows for the network to hop in a secure and random order while still allowing other radios with the seed to join the network. The tradeoffs for using the MH algorithm are that 1) a pre-shared seed must be loaded into all radios which will join the network and 2) all radios must have reasonably accurate and precise clocks. The pre-shared seed requirement can be bypassed by setting a default publicly known seed to use, but there is no longer any security inherent in the frequency hopping of the network, which is contrary to the security concerns stated in the objective. The clock requirement could theoretically be bypassed as well but would exponentially slow down the operation of the algorithm as the window size becomes the entire span of time between the generation of the pre-shared seed and the current time.

Furthermore, the simulation testbed source code and my implementation of each algorithm from the literature is also provided in Appendix A order to allow other authors to implement their own rendezvous algorithms and compare them against other known algorithms.

### Possible Future Work.

There are several potential modifications that could be made to the Multihop algorithm to support different use cases. They include the already mentioned removal of the pre-shared seed or clock requirements. Additionally, a bias could be introduced to the CSPRNG in order to favor channels with specific characteristics. This would allow the algorithm to use channels with less noise more often for example, at the cost of removing some randomness from the channel selection which could have some negative impacts on security and is a tradeoff that could be further explored.

# Appendix A. Java Simulation Source Code

## MultiHop.java.

```
package pavlik.net.radio.algorithms.asynchronous;

import java.nio.ByteBuffer;
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Random;
import java.util.logging.Logger;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;

/**
 * Algorithm will be operated by a radio as such:
 *
 * <pre>
 * {@code
 *   while(isSynced() == false){
 *     nextChannel();
 *     broadcastSync();
 *     pauseForHop();
 *   }
 * </pre>
 *
 * @author John
 *
 */
public class MultiHop extends RendezvousAlgorithm {

   private static final Logger log =
      Logger.getLogger(MultiHop.class.getName());

   // Whether to use a 1/X or uniform probability distribution
   private static final boolean USE_BIAS = false;

   // The radios clock will be set to between the current time and the
      current
   // time + MAX_ROUND_OFFSET * FREQHOP_RATE
   private static final int MAX_ROUND_OFFSET = 50;

   // private long timeOffset;
```

```
// private long startTime;
private int currentHopRound;

// Uncomment to override and slow down to 1 second hops for debug
// protected static long HOP_RATE = 1000;

// Number of channels in the sliding window
// private static int WINDOW_CHANNEL_COUNT = ((int) (MAX_TIME_OFFSET /
// HOP_RATE) + 1);
private static final int WINDOW_CHANNEL_COUNT = (2 * MAX_ROUND_OFFSET)
    + 1;

// A sliding time window of indices into the channels[]
int[] slidingWindow = new int[WINDOW_CHANNEL_COUNT];

// Index to the sliding window of the current estimated channel
int currentSlidingIndex;

// The index to the last update of the sliding window
int lastWindowUpdate;

public enum State {
   MasterNetworkRadio, SeekingRendezvous, OperatingNetwork, Syncing;

   // The count of how many hops have been made in the current sync
       attempt
   private long currentHop = 0;

   // Number of successful hop attempts
   private long hitCount = 0;

   // Number of required hops to be correct in order to synchronize
   private static final long REQUIRED_SYNC_HOPS = 10;

   // Maximum number of hop attempts to synchronize before aborting
   private static final long MAX_SYNC_HOPS = 20;

}

State state;
public Channel[] channels;

SecureRandom secureRand;
int[] bias;
```

```
public MultiHop(String id, Channel[] channels, State startingState,
    byte[] seed) {
   super(id);
   secureRand = new SecureRandom(seed);

   currentHopRound = Math.abs(new Random().nextInt()) % MAX_ROUND_OFFSET;
   if (startingState == State.SeekingRendezvous) {
      currentHopRound += MAX_ROUND_OFFSET;
   }
   currentSlidingIndex = currentHopRound;

   log.info("Starting Hop/Sliding Index: " + currentHopRound);
   // Build ranking table with Channels array
   this.channels = channels;
   Arrays.sort(channels, new Comparator<Channel>() {
      public int compare(Channel o1, Channel o2) {
         // return Double.compare(o1.noise, o2.noise);
         return o1.compareTo(o2);
      };
   });
   int biasCount = (channels.length * (channels.length + 1)) / 2;
   bias = new int[biasCount];
   int index = 0;
   for (int i = channels.length; i > 0; --i) {
      for (int j = 0; j < i; ++j) {
         bias[index++] = channels.length - i;
      }
   }
   // Pre-load the sliding window with valid channels
   initializeSlidingWindow(startingState);

   this.state = startingState;
}

private void incrementRound() {
   currentHopRound += 1;
}

@Override
public Channel nextChannel() {
   incrementRound();
   updateSlidingWindow();
   switch (state) {
   case MasterNetworkRadio:
   case OperatingNetwork:
   case Syncing:
```

```
      currentSlidingIndex = (currentSlidingIndex + 1) %
         WINDOW_CHANNEL_COUNT;
      log.info(id + " Sliding Window: " +
         Arrays.toString(slidingWindow));
      log.info(id + " Sliding index = " + currentSlidingIndex);
      break;
   case SeekingRendezvous:
      // currentSlidingIndex = (currentSlidingIndex + 1) %
      // (WINDOW_CHANNEL_COUNT - 1);
      log.info(id + " Last window update: " + lastWindowUpdate);
      log.info(id + " Sliding Window: " +
         Arrays.toString(slidingWindow));
      log.info(id + " Sliding index = " + currentSlidingIndex);
      break;
   default:
      throw new RuntimeException("Undefined state: " + state);
   }
   return channels[slidingWindow[currentSlidingIndex]];
}

private int generateSecureRandomInt() {
   byte[] bytes = new byte[4];
   secureRand.nextBytes(bytes);
   int nextVal = Math.abs(ByteBuffer.wrap(bytes).getInt());
   if (USE_BIAS) {
      return bias[nextVal % bias.length];
   } else {
      return nextVal;
   }
}

private void initializeSlidingWindow(State startingState) {
   int targetIndex;
   switch (startingState) {
   case MasterNetworkRadio:
      targetIndex = currentHopRound;
      break;
   case OperatingNetwork:
   case SeekingRendezvous:
   case Syncing:
   default:
      targetIndex = WINDOW_CHANNEL_COUNT;
   }
   for (int i = 0; i < targetIndex; ++i) {
      slidingWindow[i] = generateSecureRandomInt() % channels.length;
   }
```

```
   lastWindowUpdate = targetIndex;
}

private void updateSlidingWindow() {
   while (lastWindowUpdate <= currentHopRound) {
      slidingWindow[lastWindowUpdate % WINDOW_CHANNEL_COUNT] =
          generateSecureRandomInt() % channels.length;
      lastWindowUpdate++;
   }
}

@Override
public void receiveBroadcast(Channel currentChannel, String message) {
   // Ignore any messages sent from this radio (every radio always hears
   // its own broadcast)
   if (message.startsWith(id))
      return;

   log.info("Message received: " + message);

   switch (state) {

   case MasterNetworkRadio:
      break;

   case OperatingNetwork:
      break;

   case SeekingRendezvous:
      if (message.contains("OHELLO")) {
         log.info("Radio " + id + " switching SYNCING state");
         // currentHopRound = lastWindowUpdate;
         state = State.Syncing;
         state.currentHop = 0;
         state.hitCount = 0;
      }
      break;

   case Syncing:
      if (message.contains("OHELLO")) {
         log.info("Syncing success, up to " + state.hitCount);
         state.hitCount += 1;
      }
      if (state.hitCount >= State.REQUIRED_SYNC_HOPS) {
         log.info("Radio " + id + " switching DONE state");
         state = State.OperatingNetwork;
```

```
      }
      break;

   default:
      throw new RuntimeException("Invalid state defined: " + state);
   }
}

@Override
public void broadcastSync(Channel currentChannel) {
   switch (state) {
   case MasterNetworkRadio:
      currentChannel.broadcastMessage(id + " 0" + "HELLO on channel: " +
          currentChannel.toString());
      break;

   case OperatingNetwork:
      break;

   case SeekingRendezvous:
      // Don't broadcast anything, just listen for messages from the
      // master network
      break;

   case Syncing:
      state.currentHop += 1;
      if (state.currentHop > State.MAX_SYNC_HOPS) {
         log.info("Radio " + id + " switching back to SEEKING state");
         state = State.SeekingRendezvous;
      }
      break;

   default:
      throw new RuntimeException("Undefined state: " + state + "
          received in radio: " + id);
   }
}

@Override
public boolean isSynced() {
   switch (state) {
   case MasterNetworkRadio:
   case OperatingNetwork:
      return true;

   case SeekingRendezvous:
```

```
      case Syncing:
         return false;
      default:
         throw new RuntimeException("Undefined state: " + state);
      }
   }

}
```

**TextInterface.java.**

```
package pavlik.net;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileFilter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.logging.Logger;

import pavlik.net.Simulation.SimListener;

public class TextInterface {
   private static final Logger log =
       Logger.getLogger(TextInterface.class.getName());
   private static final int totalRunCount = 1000;

   private static final String channels = null;
   // private static final String timing = null;
   private static final String configDirectory = "config/asyncRandTest";

   public static void main(String[] args) throws IOException {
      log.fine("Begin Main");
      File dir = new File(configDirectory);
      File[] files = loadConfigFiles(dir);
      for (File file : files) {
         executeSim(file, 1);
      }
   }

   public static File[] loadConfigFiles(File directory) {
      return loadConfigFiles(directory, ".xml");
```

```
}

public static File[] loadConfigFiles(File directory, String
    acceptString) {
   if (!directory.isDirectory()) {
      return new File[] { directory };
   }
   File[] configFiles = directory.listFiles(new FileFilter() {
      @Override
      public boolean accept(File pathname) {
         return (!pathname.isDirectory() &&
             pathname.getName().endsWith(acceptString));
      }
   });
   List<File> allConfigFiles = new
       ArrayList<>(Arrays.asList(configFiles));
   File[] subFolders = directory.listFiles(new FileFilter() {
      @Override
      public boolean accept(File pathname) {
         return pathname.isDirectory();
      }
   });
   for (File subdir : subFolders) {
      File[] subConfigs = loadConfigFiles(subdir, acceptString);
      allConfigFiles.addAll(Arrays.asList(subConfigs));
   }
   return allConfigFiles.toArray(new File[0]);
}

private static void executeSim(File configFile, int runs) {
   for (int i = 0; i < totalRunCount; ++i) {
      Simulation sim = ConfigurationLoader.loadConfiguration(configFile,
          channels);
      sim.addListener(new TextInterface().new TextListener(sim,
          configFile, runs));
      sim.run();
   }
}

private class TextListener implements SimListener {
   Simulation sim;
   //final int runs;
   final File configFile;

   public TextListener(Simulation sim, File configFile, int runs) {
      this.sim = sim;
```

```
      this.configFile = configFile;
      //this.runs = runs;
   }

   @Override
   public void complete(long timeSpent) {
      String filename = "output/" + configFile.getPath() + "/" +
          sim.getRendezvousString() + ".txt";
      File check = new File(filename);
      if (!check.exists())
         try {
            check.getParentFile().mkdirs();
            check.createNewFile();
         } catch (IOException e1) {
            log.severe(e1.getMessage());
            e1.printStackTrace();
         }
      try (final BufferedWriter writer = new BufferedWriter(new
          FileWriter(filename, true))) {
         writer.write(Long.toString(sim.getRounds()));
         writer.newLine();
         writer.flush();
      } catch (IOException e) {
         log.severe(e.getMessage());
         e.printStackTrace();
      }
   }
  }
}
```

**ConfigurationLoader.java.**

```
package pavlik.net;

import java.io.File;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import java.util.logging.Logger;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
```

```java
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.AlgorithmFactory;
import pavlik.net.radio.Radio;
import pavlik.net.radio.RendezvousAlgorithm;

/**
 * XML DOM Parser class that converts XML configuration files into Java
    objects.
 *
 * Reference code used:
 *
    http://docs.oracle.com/cd/B28359_01/appdev.111/b28394/adx_j_parser.htm#ADXDK3000
 * http://www.mkyong.com/java/how-to-read-xml-file-in-java-dom-parser/
 *
 * @author John
 *
 */
public class ConfigurationLoader {
  private static final Logger log          =
      Logger.getLogger(ConfigurationLoader.class
                                          .getName());
  public static String  defaultConfig = "DefaultConfiguration.xml";

  public static Simulation loadConfiguration(File configFile, String
      channelsOverride) {
     log.fine("Loading configuration");
     File config;
     if (configFile == null) config = new File(defaultConfig);
     else config = configFile;
     Document document = readXML(config);
     if (document == null) return null;

     Simulation simulation = loadNetworkConfiguration(document);
     // if (timingOverride != null) {
     // simulation.setTiming(timingOverride);
     // }

     loadRadiosConfiguration(document, simulation, channelsOverride);
     return simulation;
  }
```

```java
private static Document readXML(File xmlFile) {
   try { // DOM setup
      DocumentBuilderFactory dbFactory =
          DocumentBuilderFactory.newInstance();
      DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

      Document document = dBuilder.parse(xmlFile);

      document.getDocumentElement().normalize();
      return document;
   } catch (ParserConfigurationException | SAXException | IOException e)
       {
      e.printStackTrace();
      log.severe("Exception! " + e.toString());
      return null;
   }
}

private static Simulation loadNetworkConfiguration(Document doc) {
   Simulation sim = new Simulation();
   NodeList networkList = doc.getElementsByTagName("network");
   Node root = networkList.item(0);
   if (root.getNodeType() == Node.ELEMENT_NODE) {
      Element rootElement = (Element) root;
      String rendezvousString = rootElement.getAttribute("algorithm");
      sim.setRendezvousString(rendezvousString);
   } else {
      throw new RuntimeException("Root was not an element");
   }
   return sim;
}

private static void loadRadiosConfiguration(Document doc, Simulation
    simulation,
      String channelOverride) {
   log.fine("Loading radios");
   Set<Radio> radioSet = new HashSet<>();
   NodeList nodeList = doc.getElementsByTagName("radio");
   boolean first = true;
   AlgorithmFactory factory = new AlgorithmFactory();
   for (int nodeIndex = 0; nodeIndex < nodeList.getLength();
       nodeIndex++) {
      Node node = nodeList.item(nodeIndex);
      if (node.getNodeType() == Node.ELEMENT_NODE) {
         Element eElement = (Element) node;
         String name = eElement.getAttribute("name");
```

```
            String channelString = eElement.getAttribute("channels");
            if (channelOverride != null) channelString = channelOverride;
            Channel[] channels =
               simulation.getSpectrum().buildChannels(channelString);
            RendezvousAlgorithm algorithm = factory.getAlgorithm(simulation
                  .getRendezvousString(), name, channels, first);
            first = false;
            Radio radio = new Radio(name, algorithm);
            radioSet.add(radio);
         } else {
            throw new RuntimeException("Identified non-element node: " +
               node.getNodeName()
                 + " type: " + node.getNodeType());
         }
      }
      simulation.addRadios(radioSet);
   }
}
```

**Simulation.java.**

```
package pavlik.net;

import java.util.HashSet;
import java.util.Set;
import java.util.logging.Logger;

import pavlik.net.Channel.Spectrum;
import pavlik.net.radio.Radio;

public class Simulation {
   private static final Logger log            =
      Logger.getLogger(Simulation.class.getName());

   // public int timingType;
   public Set<Radio>     allRadios       = new HashSet<>();
   private volatile boolean running         = true;
   private Set<SimListener> simList         = new HashSet<>();
   private String        rendezvousString = "";
   private long          clock          = 0;
   private long          rounds           = 0;
   private Spectrum      spectrum      = new Spectrum();

   public void addRadios(Set<Radio> radios) {
      allRadios.addAll(radios);
```

```java
}

public void run() {
   log.info("Begin simulation");
   long start = System.nanoTime();
   while (running) {
      rounds += 1;
      for (Radio radio : allRadios) {
         radio.nextChannel();
      }
      for (Radio radio : allRadios) {
         radio.sync();
      }
      running = false;
      for (Radio radio : allRadios) {
         if (!radio.isSyncComplete()) running = true;
      }
      if (rounds > 100000) {
         running = false;
      }
   }
   clock = System.nanoTime() - start;
   log.info("End simulation");
   log.info("Time spent: " + clock);
   complete(clock);
}

public void stopSimulation() {
   running = false;
   for (Radio radio : allRadios) {
      radio.stopSimulation();
   }
}

private void complete(long timeSpent) {
   for (SimListener simListener : simList) {
      simListener.complete(timeSpent);
   }
}

public boolean addListener(SimListener listener) {
   return simList.add(listener);
}

public boolean removeListener(SimListener listener) {
   return simList.remove(listener);
```

```
    }

    interface SimListener {
        public void complete(long timeSpent);
    }

    public String getRendezvousString() {
        return rendezvousString;
    }

    public long getTimeSpent() {
        return clock;
    }

    public long getRounds() {
        return rounds;
    }

    public Spectrum getSpectrum() {
        return spectrum;
    }

    public void setRendezvousString(String rendezvousString) {
        this.rendezvousString = rendezvousString;
    }
}
```

**Channel.java.**

```
package pavlik.net.Channel;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import java.util.logging.Logger;

public class Channel implements Comparable<Channel> {
    private static final Logger log =
        Logger.getLogger(Channel.class.getName());

    final int       id;
    public final double noise = new Random().nextDouble();

    Set<ChannelListener> listeners;
```

```java
    /**
     * Set the constructor protected in order to force use of the
         Spectrum.buildChannels factory
     * that allows string parsing
     */
    Channel(int channel) {
        this.id = channel;
        listeners = new HashSet<>();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Channel) {
            return id == ((Channel) obj).id;
        }
        return super.equals(obj);
    }

    public boolean addListener(ChannelListener listener) {
        return listeners.add(listener);
    }

    public boolean removeListener(ChannelListener listener) {
        return listeners.remove(listener);
    }

    public void broadcastMessage(String string) {
        for (ChannelListener listener : listeners) {
            log.fine("Broadcasting string: " + string + " on channel: " + id);
            listener.receiveBroadcast(this, string);
        }
    }

    @Override
    public String toString() {
        return Integer.toString(id);
    }

    @Override
    public int compareTo(Channel o) {
        return Integer.compare(id, o.id);
    }
}
```

**ChannelListener.java.**

```
package pavlik.net.Channel;

public interface ChannelListener {
   void receiveBroadcast(Channel channel, String message);
}
```

**Spectrum.java.**

```
package pavlik.net.Channel;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
import java.util.logging.Logger;

public class Spectrum {

   Map<Integer, Channel>    channelSet = new HashMap<>();
   private static final Logger log       =
      Logger.getLogger(Spectrum.class.getName());

   /**
    * Build a set of channels and add them to the global list and return a
       set for local use
    *
    * @param channelString
    *           A string that must contain integers that are comma
       separated and dash separated,
    *           e.g. "1-3,5".
    */
   public Channel[] buildChannels(String channelString) {
      log.fine("Building channels");
      Set<Channel> channelSet = new TreeSet<>();
      String[] channelCommaSplits = channelString.split(",");
      for (String commaChannel : channelCommaSplits) {
         String[] dashSplit = commaChannel.split("-");
         try {
            if (dashSplit.length == 1) {
               int num1 = Integer.parseInt(dashSplit[0].trim());
               channelSet.add(buildChannel(num1));
            } else if (dashSplit.length == 2) {
               int num1 = Integer.parseInt(dashSplit[0].trim());
```

```java
            int num2 = Integer.parseInt(dashSplit[1].trim());
            /**
             * Swap if necessary so that num1 <= num2 after this
             */
            if (num1 > num2) {
               int tmpNum = num2;
               num2 = num1;
               num1 = tmpNum;
            }
            for (int channel = num1; channel <= num2; ++channel) {
               channelSet.add(buildChannel(channel));
            }
         } else {
            System.err.println("Unable to parse a dashSplit: ");
            for (String s : dashSplit) {
               System.err.println(s);
            }
         }
      } catch (NumberFormatException ex) {
         ex.printStackTrace();
      }
   }
   return channelSet.toArray(new Channel[0]);
}

private Channel buildChannel(int channelNum) {
   if (channelSet.containsKey(channelNum)) {
      return channelSet.get(channelNum);
   } else {
      Channel channel = new Channel(channelNum);
      channelSet.put(channelNum, channel);
      return channel;
   }
}

// /**
// * Get a set of all channels in the spectrum
// *
// * @return a Set of Channel
// */
// public Set<Channel> getChannels() {
// return channelSet;
// }
}
```

**AlgorithmFactory.java.**

```
package pavlik.net.radio;

import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.algorithms.asynchronous.EnhancedJumpStay;
import pavlik.net.radio.algorithms.asynchronous.JumpStay;
import pavlik.net.radio.algorithms.asynchronous.ModifiedModularClock;
import pavlik.net.radio.algorithms.asynchronous.MultiHop;
import pavlik.net.radio.algorithms.asynchronous.RandomAlgorithm;
import pavlik.net.radio.algorithms.asynchronous.ShortSequenceBased;
import pavlik.net.radio.algorithms.synchronous.DRSEQ;
import pavlik.net.radio.algorithms.synchronous.GeneratedOrthogonalSequence;
import pavlik.net.radio.algorithms.synchronous.ModularClock;

public class AlgorithmFactory {
   private static byte[] seed;
   private static final int SEED_SIZE = 512;

   public AlgorithmFactory() {
      try {
         seed = SecureRandom.getInstanceStrong().generateSeed(SEED_SIZE);
      } catch (NoSuchAlgorithmException e) {
         e.printStackTrace();
      }
   }

   public RendezvousAlgorithm getAlgorithm(String rendezvousString, String
       id, Channel[] channels,
         boolean firstRadio) {
      switch (rendezvousString) {
      case "random":
         return new RandomAlgorithm(id, channels);
      case "orthogonal":
         return new GeneratedOrthogonalSequence(id, channels);
      case "mc":
         return new ModularClock(id, channels);
      case "mmc":
         return new ModifiedModularClock(id, channels);
      case "jumpstay":
      case "js":
         return new JumpStay(id, channels);
      case "enhancedjumpstay":
```

```
      case "ejs":
        return new EnhancedJumpStay(id, channels);
      case "drseq":
        return new DRSEQ(id, channels);
      case "ssb":
        return new ShortSequenceBased(id, channels);
      case "fh":
        return new MultiHop(id, channels,
            firstRadio ? MultiHop.State.MasterNetworkRadio :
              MultiHop.State.SeekingRendezvous, seed);
      default:
        return null;
    }
  }
}
```

**Radio.java.**

```
package pavlik.net.radio;

import java.util.logging.Logger;

import pavlik.net.Channel.Channel;

public class Radio {

  private static final Logger log    =
     Logger.getLogger(Radio.class.getName());

  String                 id;
  volatile boolean       running = true;
  Channel                currentChannel;
  public RendezvousAlgorithm algorithm;

  public Radio(String name, RendezvousAlgorithm algorithm) {
    log.info("Radio created: " + name);
    this.id = name;
    this.algorithm = algorithm;
  }

  public void stopSimulation() {
    running = false;
  }

  // public void nextStep() {
```

```java
    // if (running) {
    // nextChannel();
    // algorithm.broadcastSync(currentChannel);
    // algorithm.pauseForHop();
    // }
    // }

    public void sync() {
        algorithm.broadcastSync(currentChannel);
    }

    public void nextChannel() {
        if (currentChannel != null) {
            currentChannel.removeListener(algorithm);
        }
        currentChannel = algorithm.nextChannel();
        log.info(id + " now on channel :" + currentChannel);
        currentChannel.addListener(algorithm);
    }

    public boolean isSyncComplete() {
        return algorithm.isSynced();
    }

}
```

**RendezvousAlgorithm.java.**

```java
package pavlik.net.radio;

import java.util.logging.Logger;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.protocol.RadioProtocol;

public abstract class RendezvousAlgorithm implements RadioProtocol {

    private static final Logger log =
        Logger.getLogger(RendezvousAlgorithm.class.getName());
    boolean synced = false;
    protected String id;

    public abstract Channel nextChannel();

    public RendezvousAlgorithm(String id) {
```

```
    this.id = id;
  }

  @Override
  public void receiveBroadcast(Channel currentChannel, String message) {
    if (message.startsWith(id))
      return;
    log.info("Message received: " + message);
    if (message.contains("0HELLO")) {
      currentChannel.broadcastMessage(id + " 1" + "ACKHELLO on channel:
          " + currentChannel.toString());
    }
    if (message.contains("1ACKHELLO")) {
      currentChannel.broadcastMessage(id + " 2ACK");
      synced = true;
    }
    if (message.contains("2ACK")) {
      synced = true;
    }
  }

  @Override
  public void broadcastSync(Channel currentChannel) {
    currentChannel.broadcastMessage(id + " 0" + "HELLO on channel: " +
         currentChannel.toString());
  }

  public boolean isSynced() {
    return synced;
  }
}
```

**EnhancedJumpStay.java.**

```
package pavlik.net.radio.algorithms.asynchronous;

import java.math.BigInteger;
import java.util.Random;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;

public class EnhancedJumpStay extends RendezvousAlgorithm {

  Channel[]  channels;
```

```
    int         prime;
    int         r0, i0, t;
    Random      rand = new Random();

    public EnhancedJumpStay(String id, Channel[] channels) {
        super(id);
        this.channels = channels;
        this.prime =
            BigInteger.valueOf(channels.length).nextProbablePrime().intValue();
        this.r0 = rand.nextInt(channels.length) + 1;
        this.i0 = rand.nextInt(prime) + 1;
        this.t = 0;
    }

    @Override
    public Channel nextChannel() {
        int n = Math.floorDiv(t, (4 * prime));
        int i = ((i0 + n - 1) % prime) + 1;
        Channel channel = EJSHopping(i, t);
        t += 1;
        return channel;
    }

    private Channel EJSHopping(int i, int t) {
        t %= 4 * prime;
        int j;
        if (t < 3 * prime) {
            j = ((i + t * r0 - 1) % prime) + 1;
        } else {
            j = r0;
        }
        if (j >= channels.length) j %= channels.length;
        return channels[j];
    }
}
```

**JumpStay.java.**

```
package pavlik.net.radio.algorithms.asynchronous;

import java.math.BigInteger;
import java.util.Random;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;
```

```java
public class JumpStay extends RendezvousAlgorithm {

   Channel[]  channels;
   int        prime;
   int        r0, i0, t;
   Random     rand = new Random();

   public JumpStay(String id, Channel[] channels) {
      super(id);
      this.channels = channels;
      this.prime =
         BigInteger.valueOf(channels.length).nextProbablePrime().intValue();
      this.r0 = rand.nextInt(channels.length) + 1;
      this.i0 = rand.nextInt(prime) + 1;
      this.t = 0;
   }

   @Override
   public Channel nextChannel() {
      int n = Math.floorDiv(t, (3 * prime));
      int r = ((r0 + n - 1) % channels.length) + 1;
      int m = Math.floorDiv(t, 3 * channels.length * prime);
      int i = ((i0 + m - 1) % prime) + 1;
      Channel channel = JSHopping(r, i, t);
      t += 1;
      return channel;
   }

   private Channel JSHopping(int r, int i, int t) {
      t %= 3 * prime;
      int j;
      if (t < 2 * prime) {
         j = ((i + t * r - 1) % prime) + 1;
      } else {
         j = r;
      }
      if (j > channels.length) j = ((j - 1) % channels.length) + 1;
      return channels[j - 1];
   }
}
```

**ModifiedModularClock.java.**

```java
package pavlik.net.radio.algorithms.asynchronous;
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;

public class ModifiedModularClock extends RendezvousAlgorithm {

   Channel[]     channels;
   int           index;
   int           prime;
   int           rate;
   int           timeCount  = 0;
   static Random rand    = new Random();

   public ModifiedModularClock(String id, Channel[] channels) {
      super(id);
      this.channels = channels;
      this.index = rand.nextInt(channels.length);
      this.prime = randomPrime(channels.length, channels.length * 2);
      this.rate = rand.nextInt(channels.length-2) + 2;
   }

   @Override
   public Channel nextChannel() {
      timeCount += 1;
      if (timeCount > 2 * (prime * prime)) {
         this.prime = randomPrime(channels.length, channels.length * 2);
         this.rate = rand.nextInt(channels.length - 2) + 2;
         timeCount = 0;
      }
      index += rate;
      index %= prime;
      if (index < channels.length) {
         return channels[index];
      } else {
         return channels[rand.nextInt(channels.length)];
      }
   }

   public static int randomPrime(int start, int end) {
      List<Integer> primes = new ArrayList<>();
      for (int current = start; current <= end; current++) {
         long sqr_root = (long) Math.sqrt(current);
```

```
         boolean is_prime = true;
         for (long i = 2; i <= sqr_root; i++) {
            if (current % i == 0) {
               is_prime = false; // Current is not prime.
               break;
            }
         }
         if (is_prime) {
            primes.add(current);
         }
      }
      return primes.get(rand.nextInt(primes.size()));
   }
}
```

**RandomAlgorithm.java.**

```
package pavlik.net.radio.algorithms.asynchronous;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;

public class RandomAlgorithm extends RendezvousAlgorithm {

   Channel[]        channels;
   java.util.Random rand = new java.util.Random();

   public RandomAlgorithm(String id, Channel[] channels) {
      super(id);
      this.channels = channels;
   }

   @Override
   public Channel nextChannel() {
      return channels[rand.nextInt(channels.length)];
   }

}
```

**ShortSequenceBased.java.**

```
package pavlik.net.radio.algorithms.asynchronous;
```

```
import java.util.Random;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;

public class ShortSequenceBased extends RendezvousAlgorithm {

   private Channel[] channels;
   private int       index;
   private Random    rand  = new Random();

   public ShortSequenceBased(String id, Channel[] channels) {
      super(id);
      this.channels = channels;
      this.index = rand.nextInt(channels.length);
   }

   @Override
   public Channel nextChannel() {
      index += 1;
      if (index >= channels.length * 2) {
         index = 0;
      }
      if (0 <= index && index < channels.length) {
         return channels[index];
      } else {
         return channels[channels.length - 1 - index % channels.length];
      }
   }
}
```

**DRSEQ.java.**

```
package pavlik.net.radio.algorithms.synchronous;

import java.util.Random;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;

public class DRSEQ extends RendezvousAlgorithm {

   Channel[]  channels;
   int        index;
   Random     rand  = new Random();
```

```
    public DRSEQ(String id, Channel[] channels) {
        super(id);
        this.channels = channels;
        this.index = rand.nextInt(channels.length);
    }

    @Override
    public Channel nextChannel() {
        index += 1;
        index %= 2*channels.length + 1;
        if (index < channels.length) {
            return channels[index];
        } else if (index == channels.length) return
             channels[rand.nextInt(channels.length)];
        else return channels[2 * channels.length - index];
    }

}
```

**GeneratedOrthogonalSequence.java.**

```
package pavlik.net.radio.algorithms.synchronous;

import java.util.Arrays;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;

/**
 * Only guaranteed under synchronous model
 * @author jpavlik
 *
 */
public class GeneratedOrthogonalSequence extends RendezvousAlgorithm {

    Channel[] channels;
    java.util.Random rnd = new java.util.Random();
    int index;

    public GeneratedOrthogonalSequence(String id, Channel[] channels) {
        super(id);
        Arrays.sort(channels);
        this.channels = buildSequence(channels);
        this.index = rnd.nextInt(channels.length);
```

```
    }

    @Override
    public Channel nextChannel() {
        return channels[index++ % channels.length];
    }

    public Channel[] buildSequence(Channel[] observedChannels) {
        int length = observedChannels.length;
        Channel[] channelSequence = new Channel[length * length + length];
        int seqIndex = 0;
        for (int i = 0; i < length; ++i) {
            channelSequence[seqIndex++] = observedChannels[i];
            for (int j = 0; j < length; ++j) {
                channelSequence[seqIndex++] = observedChannels[j];
            }
        }
        return channelSequence;
    }

}
```

**ModularClock.java.**

```
package pavlik.net.radio.algorithms.synchronous;

import java.math.BigInteger;
import java.util.Random;

import pavlik.net.Channel.Channel;
import pavlik.net.radio.RendezvousAlgorithm;

public class ModularClock extends RendezvousAlgorithm {

    Channel[] channels;
    int index;
    int prime;
    int rate;
    int timeCount = 0;
    Random rand = new Random();

    public ModularClock(String id, Channel[] channels) {
        super(id);
        this.channels = channels;
        this.prime = BigInteger.valueOf(channels.length).nextProbablePrime()
```

```
            .intValue();
        this.index = rand.nextInt(channels.length);
        this.rate = rand.nextInt(channels.length - 2) +2;
    }

    @Override
    public Channel nextChannel() {
        timeCount += 1;
        if (timeCount > 2 * prime) {
            rate = rand.nextInt(prime);
            timeCount = 0;
        }
        index += rate;
        index %= prime;
        if (index < channels.length) {
            return channels[index];
        } else {
            return channels[index % channels.length];
        }
    }

}
```

**RadioProtocol.java.**

```
package pavlik.net.radio.protocol;

import pavlik.net.Channel.ChannelListener;
import pavlik.net.Channel.Channel;

public interface RadioProtocol extends ChannelListener {

    public Channel nextChannel();

    public void receiveBroadcast(Channel channel, String message);

    public void broadcastSync(Channel channel);

    public boolean isSynced();
}
```

# Appendix B. GNU Radio Source Code

**channel_selector.h.**

```
#ifndef INCLUDED_MULTIHOP_CHANNEL_SELECTOR_IMPL_H
#define INCLUDED_MULTIHOP_CHANNEL_SELECTOR_IMPL_H

#include <multihop/channel_selector.h>

namespace gr {
  namespace multihop {

    class channel_selector_impl : public channel_selector
    {
     private:
      std::vector<int> channels;

     public:
      channel_selector_impl(int num_channels);
      ~channel_selector_impl();

      // Where all the action really happens
      int work(int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items);
    };

  } // namespace multihop
} // namespace gr

#endif /* INCLUDED_MULTIHOP_CHANNEL_SELECTOR_IMPL_H */
```

**channel_selector.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "channel_selector_impl.h"

namespace gr {
  namespace multihop {

    channel_selector::sptr
```

```
channel_selector::make(int fft)
{
  return gnuradio::get_initial_sptr
    (new channel_selector_impl(fft));
}

/*
 * The private constructor
 */
channel_selector_impl::channel_selector_impl(int num_channels) :
   gr::sync_block("channel_selector",
   gr::io_signature::make(1, 1, sizeof(int)),
   gr::io_signature::make(1, 1, sizeof(int)))
 {
   for(int i = 0; i < num_channels; ++i){
    channels.push_back(i);
  }
}

/*
 * Our virtual destructor.
 */
channel_selector_impl::~channel_selector_impl(){

}

int
channel_selector_impl::work(int noutput_items,
       gr_vector_const_void_star &input_items,
       gr_vector_void_star &output_items)
{
    const int *in = (const int *) input_items[0];
    int *out = (int *) output_items[0];

    for(int i = 0; i < noutput_items; ++i){
      out[i] = channels[in[i] % channels.size()];
    }

    // Tell runtime system how many input items we consumed on
    // each input stream.
    consume_each (noutput_items);

    // Tell runtime system how many output items we produced.
    return noutput_items;
}
```

```
  } /* namespace multihop */
} /* namespace gr */
```

**channel_to_freq.h.**

```
#ifndef INCLUDED_MULTIHOP_CHANNEL_TO_FREQ_IMPL_H
#define INCLUDED_MULTIHOP_CHANNEL_TO_FREQ_IMPL_H

#include <multihop/channel_to_freq.h>

namespace gr {
  namespace multihop {

    class channel_to_freq_impl : public channel_to_freq
    {
     private:
      double fft_bin_size;
      double fft_channel_ratio;
      int round;

     public:
      channel_to_freq_impl(int fft_size, double sample_rate, int
          num_channels);
      ~channel_to_freq_impl();

      // Where all the action really happens
      int work(int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items);
    };

  } // namespace multihop
} // namespace gr

#endif /* INCLUDED_MULTIHOP_CHANNEL_TO_FREQ_IMPL_H */
```

**channel_to_freq.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
```

```
#include "channel_to_freq_impl.h"

namespace gr {
  namespace multihop {

    channel_to_freq::sptr
    channel_to_freq::make(int fft_size, double sample_rate, int
        num_channels)
    {
      return gnuradio::get_initial_sptr
      (new channel_to_freq_impl(fft_size, sample_rate, num_channels));
    }

    /*
     * The private constructor
     */
    channel_to_freq_impl::channel_to_freq_impl(int fft_size, double
        sample_rate, int num_channels)
      : gr::sync_block("channel_to_freq",
              gr::io_signature::make(1, 1, sizeof(int)),
              gr::io_signature::make(1, 1, sizeof(float)))
    {
      round = 0;
      fft_bin_size = sample_rate / fft_size;
      fft_channel_ratio = (fft_size / 2.0) / (double)num_channels;
      std::cout << "fft_channel_ratio = " << fft_channel_ratio << std::endl;
    }

    /*
     * Our virtual destructor.
     */
    channel_to_freq_impl::~channel_to_freq_impl()
    {
    }

    int
    channel_to_freq_impl::work(int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items)
    {
        const int *in = (const int *) input_items[0];
        float *out = (float *) output_items[0];

        for(int i = 0; i < noutput_items; ++i){
          int fft_bin = (in[i] * fft_channel_ratio + fft_channel_ratio *
              0.5);
```

```
        out[i] = fft_bin * fft_bin_size; //Convert to frequency
        std::cout << "Round " << round++ << " ";
        std::cout << "Channel: " << in[i] << " & Bin: " << fft_bin << " &
            Freq: " << out[i] << std::endl;
      }

      return noutput_items;
    }

  } /* namespace multihop */
} /* namespace gr */
```

**csprng.h.**

```
#ifndef INCLUDED_MULTIHOP_CSPRNG_IMPL_H
#define INCLUDED_MULTIHOP_CSPRNG_IMPL_H

#include <multihop/csprng.h>
#include <limits.h>
#include <ctime>

#define BILLION 1000000000L

namespace gr {
  namespace multihop {

    class csprng_impl : public csprng
    {
     private:
      uint64_t counter;
      const static uint64_t BASE_TIME = 12476075133901;
      uint64_t last_update;
      uint64_t ns_per_hop;
      uint64_t samps_per_hop;
      uint64_t samples;

      uint64_t time_sync();

     public:
      csprng_impl(int seed, double freq_rate, double samp_rate);
      ~csprng_impl();
      uint64_t next();

      // Where all the action really happens
      int work(int noutput_items,
```

```
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items);
  };

 } // namespace multihop
} // namespace gr
```

**csprng.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "csprng_impl.h"

namespace gr {
  namespace multihop {

    csprng::sptr
    csprng::make(int seed, double freq_rate, double samp_rate)
    {
      return gnuradio::get_initial_sptr
        (new csprng_impl(seed, freq_rate, samp_rate));
    }

    /*
     * The private constructor
     */
    csprng_impl::csprng_impl(int seed, double freq_rate, double samp_rate)
      : gr::sync_block("csprng",
              gr::io_signature::make(0, 0, 0),
              gr::io_signature::make(1, 1, sizeof(int)))
    {
      srand(seed);
      ns_per_hop = BILLION/freq_rate;
      counter = 0;
      //last_update = BASE_TIME;
      last_update = 0;
      time_sync();
    }

    /*
     * Our virtual destructor.
     */
```

```
  csprng_impl::~csprng_impl()
  {
  }

  uint64_t csprng_impl::time_sync(){
    //struct timespec now;
    //clock_gettime(CLOCK_MONOTONIC, &now);
    //uint64_t now_ns = (BILLION * (now.tv_sec) + now.tv_nsec);
    //uint64_t elapsed_nanoseconds = now_ns - last_update;
    //uint64_t elapsed_hops = elapsed_nanoseconds / ns_per_hop;
    //if(elapsed_hops > 0){
    //  last_update = now_ns;
    //  for(uint64_t i = 0; i < elapsed_hops; ++i){
    //    counter += 1;
    //  }
    //}
    std::cout<< "CSPRNG: " << counter << std::endl;
    return counter;
  }

  uint64_t csprng_impl::next(){
    //counter += 1;
    //return counter;
    return rand();
  }

  int
  csprng_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
  {
      int *out = (int *) output_items[0];
      for(int i = 0; i < noutput_items; ++i){
        out[i] = (int)next();
      }

      // Tell runtime system how many output items we produced.
      return noutput_items;
  }

 } /* namespace multihop */
} /* namespace gr */
```

**freq_to_channel.h.**

```
#ifndef INCLUDED_MULTIHOP_FREQ_TO_CHANNEL_IMPL_H
#define INCLUDED_MULTIHOP_FREQ_TO_CHANNEL_IMPL_H

#include <multihop/freq_to_channel.h>
#include <cmath>

namespace gr {
  namespace multihop {

    class freq_to_channel_impl : public freq_to_channel
    {
     private:
      double channel_fft_ratio;

     public:
      freq_to_channel_impl(int fft_size, double sample_rate, int
          num_channels);
      ~freq_to_channel_impl();

      // Where all the action really happens
      int work(int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items);
    };

  } // namespace multihop
} // namespace gr

#endif /* INCLUDED_MULTIHOP_FREQ_TO_CHANNEL_IMPL_H */
```

**freq_to_channel.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "freq_to_channel_impl.h"

namespace gr {
  namespace multihop {

    freq_to_channel::sptr
```

```
  freq_to_channel::make(int fft_size, double sample_rate, int
      num_channels)
  {
    return gnuradio::get_initial_sptr
      (new freq_to_channel_impl(fft_size, sample_rate, num_channels));
  }

  /*
   * The private constructor
   */
  freq_to_channel_impl::freq_to_channel_impl(int fft_size, double
      sample_rate, int num_channels)
    : gr::sync_block("freq_to_channel",
            gr::io_signature::make(1, 1, sizeof(int)),
            gr::io_signature::make(1, 1, sizeof(int)))
  {
    channel_fft_ratio = num_channels / ((double)fft_size / 2);
    std::cout << "Channel_fft_ratio = " << channel_fft_ratio << std::endl;
  }

  /*
   * Our virtual destructor.
   */
  freq_to_channel_impl::~freq_to_channel_impl()
  {
  }

  int
  freq_to_channel_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
  {
      const int *in = (const int *) input_items[0];
      int *out = (int *) output_items[0];


      for(int i = 0; i < noutput_items; ++i){
        out[i] = round(in[i] * channel_fft_ratio);
        //std::cout << "Input freq to channel: " << in[i] << std::endl;
      }

      // Tell runtime system how many output items we produced.
      return noutput_items;
  }

} /* namespace multihop */
```

```
} /* namespace gr */
```

freqhop_signal_source.cc.

```
#ifndef INCLUDED_MULTIHOP_FREQHOP_SIGNAL_SOURCE_IMPL_H
#define INCLUDED_MULTIHOP_FREQHOP_SIGNAL_SOURCE_IMPL_H

#include <multihop/freqhop_signal_source.h>
#include <cmath>

namespace gr {
  namespace multihop {

    class freqhop_signal_source_impl : public freqhop_signal_source
    {
     private:
      double     d_sampling_freq;
      double     d_ampl;
      double     d_frequency;
      double   d_freqhop;
      gr_complex    d_offset;
      unsigned long    time;
      unsigned int interp_rate;
      const static double D_PI =
          3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

     public:
      freqhop_signal_source_impl(double sampling_freq, double ampl,
          gr_complex offset, double freqhop_rate);
      ~freqhop_signal_source_impl();


      // Where all the action really happens
      int work(int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items);
    };

  } // namespace multihop
} // namespace gr

#endif /* INCLUDED_MULTIHOP_FREQHOP_SIGNAL_SOURCE_IMPL_H */
```

**freqhop_signal_source.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "freqhop_signal_source_impl.h"

namespace gr {
  namespace multihop {

    freqhop_signal_source::sptr
    freqhop_signal_source::make(double sampling_freq, double ampl,
        gr_complex offset, double freqhop_rate)
    {
      return gnuradio::get_initial_sptr
        (new freqhop_signal_source_impl(sampling_freq, ampl, offset,
            freqhop_rate));
    }

    /*
     * The private constructor
     */
    freqhop_signal_source_impl::freqhop_signal_source_impl(double
        sampling_freq, double ampl, gr_complex offset, double freqhop_rate)
      : gr::sync_interpolator("freqhop_signal_source",
              gr::io_signature::make(1, 1, sizeof(float)),
              gr::io_signature::make(1, 1, sizeof(gr_complex)),
              round(sampling_freq / freqhop_rate)),
      d_sampling_freq(sampling_freq), d_ampl(ampl), d_offset(offset)
    {
      time = 0;
      interp_rate = round(sampling_freq / freqhop_rate);
    }

    /*
     * Our virtual destructor.
     */
    freqhop_signal_source_impl::~freqhop_signal_source_impl()
    {
    }

    int
    freqhop_signal_source_impl::work(int num_output_items,
          gr_vector_const_void_star &input_items,
```

```
            gr_vector_void_star &output_items)
    {
      const float *in = (const float *) input_items[0];
      gr_complex *optr = (gr_complex*) output_items[0];
      int ninput_items = num_output_items / interp_rate;
      for(int i = 0; i < ninput_items; ++i){
        float freq = in[i];
       // std::cout << "Freq: " << freq << std::endl;
        for(int j = i; j < (i+1) * interp_rate; ++j){
          double val = 2 * D_PI * freq * time / d_sampling_freq;
          time += 1;
          optr[j] = gr_complex(d_ampl*cos(val), d_ampl*sin(val));
           if(d_offset != gr_complex(0,0)){
             optr[j] += d_offset;
           }
        }
      }
      return num_output_items;
    }



  } /* namespace multihop */
} /* namespace gr */
```

**max_cf.h.**

```
#ifndef INCLUDED_MULTIHOP_MAX_CF_IMPL_H
#define INCLUDED_MULTIHOP_MAX_CF_IMPL_H

#include <multihop/max_cf.h>

namespace gr {
  namespace multihop {

    class max_cf_impl : public max_cf
    {
     private:
      int vec_size;

     public:
      max_cf_impl(int vec_size);
      ~max_cf_impl();

      //Find the magnitude squared (real^2 + imag^2)) of a complex number
```

```
      float magnitude2(gr_complex val);

      // Where all the action really happens
      int work(int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items);
    };

  } // namespace multihop
} // namespace gr

#endif /* INCLUDED_MULTIHOP_MAX_CF_IMPL_H */
```

**max_cf.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "max_cf_impl.h"

namespace gr {
  namespace multihop {

    max_cf::sptr
    max_cf::make(int vec_size)
    {
      return gnuradio::get_initial_sptr
        (new max_cf_impl(vec_size));
    }

    /*
     * The private constructor
     */
    max_cf_impl::max_cf_impl(int vsize)
      : gr::sync_block("max_cf",
              gr::io_signature::make(1, 1, vsize*sizeof(gr_complex)),
              gr::io_signature::make(1, 1, sizeof(int)))
    {
        vec_size = vsize;
    }

    /*
     * Our virtual destructor.
```

```
     */
    max_cf_impl::~max_cf_impl()
    {
    }

    int
    max_cf_impl::work(int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items)
    {
        const gr_complex *in = (const gr_complex *) input_items[0];
        int *out = (int *) output_items[0];

        for(int i = 0; i < noutput_items; ++i){
          int start = i * vec_size;
          float max_val = magnitude2(in[start + 1]);
          int max_loc = start;
          for(int j = start; j < start+vec_size; ++j){
            float new_max = magnitude2(in[j]);
            int new_loc = j - (i*vec_size);
            if(new_max > max_val){
              max_val = new_max;
              max_loc = new_loc;
            }
          }
          //std::cout << "Max Loc = " << max_loc << std::endl;
          out[i] = max_loc;
        }

        // Tell runtime system how many output items we produced.
        return noutput_items;
    }

    float max_cf_impl::magnitude2(gr_complex val){
      return val.real()*val.real() + val.imag()*val.imag();
    }

  } /* namespace multihop */
} /* namespace gr */
```

**mode_ii.h.**

```
#ifndef INCLUDED_MULTIHOP_MODE_II_IMPL_H
#define INCLUDED_MULTIHOP_MODE_II_IMPL_H
```

```
#include <multihop/mode_ii.h>
#include <map>

namespace gr {
  namespace multihop {

    class mode_ii_impl : public mode_ii
    {
     private:
      int i_size;

     public:
      mode_ii_impl(int size);
      ~mode_ii_impl();

      // Where all the action really happens
      int work(int noutput_items,
          gr_vector_const_void_star &input_items,
          gr_vector_void_star &output_items);
    };

  } // namespace multihop
} // namespace gr

#endif /* INCLUDED_MULTIHOP_MODE_II_IMPL_H */
```

**mode_ii.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "mode_ii_impl.h"

namespace gr {
  namespace multihop {

    mode_ii::sptr
    mode_ii::make(int size)
    {
      return gnuradio::get_initial_sptr
        (new mode_ii_impl(size));
    }
```

```
/*
 * The private constructor
 */
mode_ii_impl::mode_ii_impl(int size)
  : gr::sync_decimator("mode_ii",
          gr::io_signature::make(1, 1, sizeof(int)),
          gr::io_signature::make(1, 1, sizeof(int)), size),
          i_size(size)
{}

/*
 * Our virtual destructor.
 */
mode_ii_impl::~mode_ii_impl()
{
}

int
mode_ii_impl::work(int noutput_items,
       gr_vector_const_void_star &input_items,
       gr_vector_void_star &output_items)
{
    const int *in = (const int *) input_items[0];
    int *out = (int *) output_items[0];

    for(int i = 0; i < noutput_items; ++i){
      int start = i * i_size;
      std::map<int, int> map;

      for(int j = start; j < start + i_size; ++j){
        if(map.count(in[j]) == 0) map[in[j]] = 1;
        else map[in[j]] = map[in[j]] + 1;
      }

      int max_item = -1;
      int max_mode = -1;
      for (std::map<int,int>::iterator it = map.begin(); it !=
          map.end(); ++it) {
        //std::cout << "Max_mode values: " << it->first << " = " <<
            it->second << std::endl;
        if(it->second > max_mode){
          max_mode = it->second;
          max_item = it->first;
        }
      }
      out[i] = max_item;
```

```
        //std::cout << "Max_mode: " << max_item << " = " << max_mode <<
            std::endl;
      }
      // Tell runtime system how many output items we produced.
      return noutput_items;
    }

  } /* namespace multihop */
} /* namespace gr */
```

**multihop_rendezvous.h.**

```
#ifndef INCLUDED_MULTIHOP_MULTIHOP_RENDEZVOUS_IMPL_H
#define INCLUDED_MULTIHOP_MULTIHOP_RENDEZVOUS_IMPL_H

#include <multihop/multihop_rendezvous.h>
#include "csprng_impl.h"
#include <vector>
#include <stdexcept>

namespace gr {
  namespace multihop {
    enum State { SEEKING, SYNCING, DONE };
    class multihop_rendezvous_impl : public multihop_rendezvous
    {
     private:
      int max_window_size;
      int num_channels;
      std::vector<int> sliding_window;
      csprng_impl* rng;
      int last_window_update;
      int hop_round;
      int sliding_index;
      int search_rate;
      int finished;

      State state;
      int syncing_count;
      int syncing_success;

        // Number of required hops to be correct in order to synchronize
        const static long REQUIRED_SYNC_HOPS = 10;

        // Maximum number of hop attempts to synchronize before aborting
        const static long MAX_SYNC_HOPS = 20;
```

```
    void update_sliding_window();

   public:
    multihop_rendezvous_impl(int seed, double freq_rate, double
        samp_rate, int num_channels, int window_size, int search_speed);
    ~multihop_rendezvous_impl();
    int next_channel();

    // Where all the action really happens
    int work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items);
  };

 } // namespace multihop
} // namespace gr

#endif /* INCLUDED_MULTIHOP_MULTIHOP_RENDEZVOUS_IMPL_H */
```

**multihop_rendezvous.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "multihop_rendezvous_impl.h"

namespace gr {
  namespace multihop {

    multihop_rendezvous::sptr
    multihop_rendezvous::make(int seed, double freq_rate, double
        samp_rate, int num_channels, int window_size, int search_speed)
    {
      return gnuradio::get_initial_sptr
        (new multihop_rendezvous_impl(seed, freq_rate, samp_rate,
            num_channels, window_size, search_speed));
    }

    /*
     * The private constructor
     */
```

```
multihop_rendezvous_impl::multihop_rendezvous_impl(int seed, double
    freq_rate, double samp_rate, int nchannels, int window_size, int
    search_speed)
  : gr::sync_block("multihop_rendezvous",
          gr::io_signature::make(1, 1, sizeof(int)),
          gr::io_signature::make(0, 0, 0))
{
  max_window_size = window_size;
  sliding_window = std::vector<int>(max_window_size);
  rng = new csprng_impl(seed, freq_rate, samp_rate);
  num_channels = nchannels;
  search_rate = search_speed;

  sliding_index = 0;
  last_window_update = 0;
  hop_round = 0;

  state = SEEKING;
  syncing_count = 0;
  syncing_success = 0;

  for(int i = 0; i < window_size / 2; ++i){
    next_channel();
  }
}

/*
 * Our virtual destructor.
 */
multihop_rendezvous_impl::~multihop_rendezvous_impl()
{
  delete rng;
}

int
multihop_rendezvous_impl::work(int noutput_items,
       gr_vector_const_void_star &input_items,
       gr_vector_void_star &output_items)
{
    const int *in = (const int *) input_items[0];

    for(int i = 0; i < noutput_items; ++i){
      int received_channel = in[i];
      //if(received_channel >= num_channels){
        //std::cout << "Decrementing " << received_channel << std::endl;
      //  received_channel = received_channel - num_channels;
```

```
      //}
      int expected_channel = next_channel();

      if(received_channel != expected_channel){
              std::cout << "Expected " << expected_channel << " but got
                  " << received_channel << std::endl;
              //if(state == DONE) state = SEEKING;
          }

     if(state == SYNCING){
       syncing_count++;
       if(received_channel == expected_channel){
         syncing_success += 1;
         std::cout << "SYNCING SUCCESS: " << syncing_success << " / "
             << syncing_count << std::endl;
       }
       if(syncing_success >= REQUIRED_SYNC_HOPS){
         state = DONE;
         finished = hop_round;
       }
       if(syncing_count > MAX_SYNC_HOPS){
         state = SEEKING;
         std::cout << "ABORT SYNCING" << std::endl;
       }
     }
     else if(state == SEEKING){
       if(received_channel == expected_channel){
          std::cout << "BEGIN SYNCING" << std::endl;
          state = SYNCING;
              syncing_count = 0;
              syncing_success = 0;
            }
      }
      else if(state == DONE){
        std::cout << "SYNC COMPLETE AT " << finished << std::endl;
      }
   }

   //pauseForHop();

    return noutput_items;
}

void multihop_rendezvous_impl::update_sliding_window() {
    while (last_window_update <= hop_round) {
```

```
        sliding_window[last_window_update % max_window_size] =
           rng->next() % num_channels;
        last_window_update++;
      }
    }

   int multihop_rendezvous_impl::next_channel(){
     hop_round += 1;
     std::cout << "Round " << hop_round << " ";
     update_sliding_window();
       switch (state) {
       case DONE:
       case SYNCING:
            sliding_index = (sliding_index + 1) % max_window_size;
         break;
       case SEEKING:
           if(hop_round % search_rate != 0)
               sliding_index = (sliding_index + 1) % max_window_size;
          break;
       default:
          throw std::invalid_argument("Invalid state received");
       }
       /*std::cout << "Sliding Window: ";
       for(int i = 0; i < max_window_size; ++i){
         std::cout << sliding_window[i];
         if(i == sliding_index) std::cout << "*";
         std::cout << ", ";
       }
       std::cout << std::endl;*/
       return sliding_window[sliding_index];
   }

  } /* namespace multihop */
} /* namespace gr */
```

**modular_clock_rendezvous.h.**

```
#ifndef INCLUDED_MULTIHOP_MODULAR_CLOCK_RENDEZVOUS_IMPL_H
#define INCLUDED_MULTIHOP_MODULAR_CLOCK_RENDEZVOUS_IMPL_H

#include <multihop/modular_clock_rendezvous.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
```

```
namespace gr {
  namespace multihop {

    class modular_clock_rendezvous_impl : public modular_clock_rendezvous
    {
     private:
      int num_channels;
      bool transmitter;

      int index;
      int prime;
      int rate;
      int round;

      int count;

     public:
      modular_clock_rendezvous_impl(bool transmitter, int num_channels);
      ~modular_clock_rendezvous_impl();

      // Where all the action really happens
      void forecast (int noutput_items, gr_vector_int
          &ninput_items_required);

      int general_work(int noutput_items,
             gr_vector_int &ninput_items,
             gr_vector_const_void_star &input_items,
             gr_vector_void_star &output_items);


      unsigned int getSeed();
      int gcd(int a, int b);
      bool primeCheck(int x);
      int primeFind(int x, int bound);
      int next_channel();
    };



  } // namespace multihop
} // namespace gr


#endif /* INCLUDED_MULTIHOP_MODULAR_CLOCK_RENDEZVOUS_IMPL_H */
```

**modular_clock_rendezvous.cc.**

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "modular_clock_rendezvous_impl.h"

namespace gr {
  namespace multihop {

    modular_clock_rendezvous::sptr
    modular_clock_rendezvous::make(bool transmitter, int num_channels)
    {
      return gnuradio::get_initial_sptr
        (new modular_clock_rendezvous_impl(transmitter, num_channels));
    }

    /*
     * The private constructor
     */
    modular_clock_rendezvous_impl::modular_clock_rendezvous_impl(bool
        p_transmitter, int p_nchannels)
      : gr::block("modular_clock_rendezvous",
              gr::io_signature::make(0, 1, sizeof(int)),
              gr::io_signature::make(0, 1, sizeof(int)))
    {
      srand(getSeed());
      transmitter = p_transmitter;
      num_channels = p_nchannels;
      round = 0;
      prime = 0;
      while(prime == 0){
        prime = primeFind(num_channels, num_channels * 2);
      }
      index = rand() % num_channels;
      rate = (rand() % (num_channels - 2)) + 2;

      count = 0;
    }

    /*
     * Our virtual destructor.
     */
    modular_clock_rendezvous_impl::~modular_clock_rendezvous_impl()
```

```
{
}

void
modular_clock_rendezvous_impl::forecast (int noutput_items,
    gr_vector_int &ninput_items_required)
{
  if(transmitter){
    ninput_items_required[0] = 0;
  }
  else{
    ninput_items_required[0] = noutput_items;
  }
}

unsigned int
modular_clock_rendezvous_impl::getSeed(){
  FILE *file = fopen("/dev/random", "r");
  unsigned int temp;
  int size = fread(&temp, 4, 1, file);
  fclose(file);
  return temp;
}

int modular_clock_rendezvous_impl::gcd(int a, int b){
  int c;
  while(true){
    c = a%b;
    if(c==0) return b;
    a = b;
    b = c;
  }
}

bool modular_clock_rendezvous_impl::primeCheck(int x){
  for(int i = 2; i <= int(sqrt(x)); ++i){
    if(gcd(i,x) > 1) return false;
  }
  return true;
}

int modular_clock_rendezvous_impl::primeFind(int x, int bound){
  while(x < bound){
    if(primeCheck(x)) return x;
    x++;
  }
```

```
  return 0;
}

int
modular_clock_rendezvous_impl::general_work (int noutput_items,
                   gr_vector_int &ninput_items,
                   gr_vector_const_void_star &input_items,
                   gr_vector_void_star &output_items)
{

    if(transmitter){
      int *out = (int *) output_items[0];
      out[0] = next_channel();
      return 1;
    }else{
       const int *in = (const int *) input_items[0];
       for(int i = 0; i < ninput_items[0]; ++i){
         int received_channel = in[i];
         int guessed_channel = next_channel();
         if(received_channel == guessed_channel){
            std::cout << "DONE AT " << count << std::endl;
         }
      }
      consume_each (ninput_items[0]);
      return 0;
    }

}

int modular_clock_rendezvous_impl::next_channel(){
  round += 1;
  count += 1;
  if(round > 2 * prime){
    round = 0;
    prime = 0;
    while(prime == 0){
      prime = primeFind((rand() % num_channels) + num_channels,
          num_channels * 2);
    }
    rate = (rand() % (num_channels - 2)) + 2;
  }
  index = fmod((index + rate), prime);
  if(index >= num_channels){
    srand(getSeed());
    index = rand() % num_channels;
  }
```

```
      return index;
    }

  } /* namespace multihop */
} /* namespace gr */
```

# Bibliography


1. ISO/IEC/IEEE International Standard – Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks– Specific requirements – Part 22: Cognitive Wireless RAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Policies and procedures for operation in the TV Band. *ISO/IEC/IEEE 8802-22:2015*, pages 1–678, May 2015.

2. Paramvir Bahl, Ranveer Chandra, and John Dunagan. Ssch: Slotted seeded channel hopping for capacity improvement in ieee 802.11 ad-hoc wireless networks. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, MobiCom '04, pages 216–230, New York, NY, USA, 2004. ACM.

3. Elaine B Barker and John M Kelsey. Sp 800-90a. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators, National Institute of Standards & Technology, Gaithersburg, MD*, 2012.

4. Kaigui Bian, Jung-Min Park, and Ruiliang Chen. A quorum-based framework for establishing control channels in dynamic spectrum access networks. In *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, pages 25–36, New York, NY, USA, 2009. ACM.

5. Guey-Yun Chang, Jen-Feng Huang, and Yao-Shian Wang. Matrix-based channel hopping algorithms for cognitive radio networks. *Wireless Communications, IEEE Transactions on*, PP(99):1–1, 2015.

6. C. Cormio and K.R. Chowdhury. An adaptive multiple rendezvous control channel for cognitive radio wireless ad hoc networks. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 346–351, March 2010.

7. Bruce A Fette. *Cognitive radio technology.* Academic Press, 2009.

8. S. Geirhofer, Lang Tong, and B.M. Sadler. Cognitive radios for dynamic spectrum access - dynamic spectrum access in the time domain: Modeling and exploiting white space. *Communications Magazine, IEEE*, 45(5):66–72, May 2007.

9. Zhaoquan Gu, Qiang-Sheng Hua, Yuexuan Wang, and F.C.M. Lau. Nearly optimal asynchronous blind rendezvous algorithm for cognitive radio networks. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2013 10th Annual IEEE Communications Society Conference on*, pages 371–379, June 2013.

10. Headquarters, Department of the Army. *FM 6-02 Signal Support to Operations*, January 2014.

11. Shoichi Hirose. Security analysis of drbg using hmac in nist sp 800-90. In *Information Security Applications*, pages 278–291. Springer, 2009.

12. Juncheng Jia, Qian Zhang, and Xuemin Shen. Hc-mac: A hardware-constrained cognitive mac for efficient spectrum management. *Selected Areas in Communications, IEEE Journal on*, 26(1):106–117, Jan 2008.

13. Donald E Knuth. The art of computer programming (volume 2), 1981.

14. Hugo Krawczyk. How to predict congruential generators. *Journal of Algorithms*, 13(4):527–545, 1992.

15. J. Landt. The history of rfid. *Potentials, IEEE*, 24(4):8–11, Oct 2005.

16. L. Lazos, Sisi Liu, and M. Krunz. Spectrum opportunity-based control channel assignment in cognitive radio networks. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2009. SECON '09. 6th Annual IEEE Communications Society Conference on*, pages 1–9, June 2009.

17. Zhiyong Lin, Hai Liu, Xiaowen Chu, and Yiu-Wing Leung. Enhanced jump-stay rendezvous algorithm for cognitive radio networks. *Communications Letters, IEEE*, 17(9):1742–1745, September 2013.

18. Hai Liu, Zhiyong Lin, Xiaowen Chu, and Y.-W. Leung. Jump-stay rendezvous algorithm for cognitive radio networks. *Parallel and Distributed Systems, IEEE Transactions on*, 23(10):1867–1881, Oct 2012.

19. Hai Liu, Zhiyong Lin, Xiaowen Chu, and Y.-W. Leung. Taxonomy and challenges of rendezvous algorithms in cognitive radio networks. In *Computing, Networking and Communications (ICNC), 2012 International Conference on*, pages 645–649, Jan 2012.

20. Hai Liu, Zhiyong Lin, Xiaowen Chu, and Yiu-Wing Leung. Ring-walk based channel-hopping algorithms with guaranteed rendezvous for cognitive radio networks. In *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 755–760, Washington, DC, USA, 2010. IEEE Computer Society.

21. Liangping Ma, Xiaofeng Han, and Chien-Chung Shen. Dynamic open spectrum sharing mac protocol for wireless ad hoc networks. In *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 203–213, Nov 2005.

22. R.K. McLean, M.D. Silvius, K.M. Hopkinson, B.N. Flatley, E.S. Hennessey, C.C. Medve, J.J. Thompson, M.R. Tolson, and C.V. Dalton. An architecture for coexistence with multiple users in frequency hopping cognitive radio networks. *Selected Areas in Communications, IEEE Journal on*, 32(3):563–571, March 2014.

23. J. Misic and V.B. Misic. Probabilistic vs. sequence-based rendezvous in channel-hopping cognitive networks. *Parallel and Distributed Systems, IEEE Transactions on*, 25(9):2418–2427, Sept 2014.

24. V.B. Misic and J. Misic. Cognitive mac protocol with transmission tax: Dynamically adjusting sensing and data performance. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5, Dec 2010.

25. J. Mitola and Jr. Maguire, G.Q. Cognitive radio: making software radios more personal. *Personal Communications, IEEE*, 6(4):13–18, Aug 1999.

26. Joseph Mitola III. *Cognitive radio*. PhD thesis, Royal Institute of Technology, 2000.

27. Young-Hyun Oh and D.J. Thuente. Channel detecting jamming attacks against jump-stay based channel hopping rendezvous algorithms for cognitive radio networks. In *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, pages 1–9, July 2013.

28. Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

29. V.A. Reguera, E. Ortiz Guerra, R. Demo Souza, E.M.G. Fernandez, and G. Brante. Short channel hopping sequence approach to rendezvous for cognitive networks. *Communications Letters, IEEE*, 18(2):289–292, February 2014.

30. Berry Schoenmakers and Andrey Sidorenko. Cryptanalysis of the dual elliptic curve pseudorandom generator. *IACR Cryptology ePrint Archive*, 2006:190, 2006.

31. Jongmin Shin, Dongmin Yang, and Cheeha Kim. A channel rendezvous scheme for cognitive radio networks. *Communications Letters, IEEE*, 14(10):954–956, October 2010.

32. N.C. Theis, R.W. Thomas, and L.A. DaSilva. Rendezvous for cognitive radios. *Mobile Computing, IEEE Transactions on*, 10(2):216–227, Feb 2011.

33. Nicholas C Theis. The modular clock algorithm for blind rendezvous. Master's thesis, Air Force Institute of Technology, 2009.

34. National Telecommunications U.S. Department of Commerce and Office of Spectrum Management Information Administration. United states frequency allocations: The radio spectrum, August 2011. `https://www.ntia.doc.gov/files/ntia/publications/spectrum_wall_chart_aug2011.pdf`.

35. D. Yang, J. Shin, and C. Kim. Deterministic rendezvous scheme in multichannel access networks. *Electronics Letters*, 46(20):1402–1404, September 2010.

36. Jun Zhao, Haitao Zheng, and Guang-Hua Yang. Distributed coordination in dynamic spectrum allocation networks. In *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 259–268, Nov 2005.

37. Qing Zhao and B.M. Sadler. A survey of dynamic spectrum access. *Signal Processing Magazine, IEEE*, 24(3):79–89, May 2007.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704–0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

**1. REPORT DATE** *(DD–MM–YYYY)*
26–03–2016

**2. REPORT TYPE**
Master's Thesis

**3. DATES COVERED** *(From — To)*
Sept 2014 — Mar 2016

**4. TITLE AND SUBTITLE**
Multihop Rendezvous Algorithm for Frequency Hopping Cognitive Radio Networks

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**5d. PROJECT NUMBER**
16G109

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**6. AUTHOR(S)**
Pavlik, John A., Captain, USA

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

**8. PERFORMING ORGANIZATION REPORT NUMBER**
AFIT-ENG-MS-16-M-039

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Michael Gudaitis
525 Brooks Rd
Rome Labs AFB, NY 13441
DSN 587-4478
michael.gudaitis@us.af.mil

**10. SPONSOR/MONITOR'S ACRONYM(S)**
AFRL/RITE

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION / AVAILABILITY STATEMENT**
DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**13. SUPPLEMENTARY NOTES**
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

**14. ABSTRACT**
Cognitive radios allow the possibility of increasing utilization of the wireless spectrum, but because of their dynamic access nature require new techniques for establishing and joining networks, these are known as rendezvous. Cognitive radio networks utilizing frequency hopping that is too fast for synchronization packets to be exchanged in a single hop require a rendezvous algorithm that supports multiple hop rendezvous. We propose the Multiple Hop (MH) rendezvous algorithm based on a pre-shared sequence of random numbers, bounded timing differences, and similar channel lists to successfully match a percentage of hops. It is tested in simulation against other well known rendezvous algorithms and implemented in GNU Radio for the HackRF One. We recommend the Multihop algorithm for use cases with a fast frequency hop rate and a slow data transmission rate requiring multiple hops to rendezvous or use cases where the channel count equals or exceeds 250 channels, as long as timing data is available and all of the radios to be connected to the network can be pre-loaded with a shared seed.

**15. SUBJECT TERMS**
Cognitive,Radio,Rendezvous,Multihop,Thesis

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Dr. K. M. Hopkinson, AFIT/ENG |
| U | U | U | UU | 123 | 19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, x4579; Kenneth.Hopkinson@afit.edu |