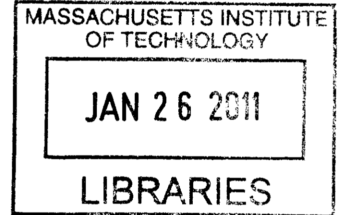
Collaborative Internet and Voice Data Transfer using Bluetooth Mesh Networking


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JAN 26 2011
LIBRARIES
ARCHIVES

by

Peter S. Kruskall

S.B. MIT, 2008

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 2010

Copyright 2010 Peter S. Kruskall. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author ____

Department of Electrical Engineering and Computer Science

February 1, 2010

Certified by ____

Larry Rudolph, Research Affiliate

Thesis Supervisor

Accepted by ____

Dr. Christopher J. Terman

Chairman, Department Committee on Graduate Theses

Thesis Supervisor

ABSTRACT

We present a new networking protocol, AirRAID, intended for wireless devices that, using the collective power of multiple devices within short-range communication sight, extends the availability of a secondary medium over an ad-hoc mesh network, resilient to the erratic movements of the mobile nodes from which it is comprised. We suggest improvements to the Bluetooth discovery algorithm, making use of a quantized hop velocity space to lower the probability of two devices missing each other completely during discovery, and introduce the concept of redundant backup paths to the wireless mesh, allowing for improved reliability in dynamic mesh network situations

Acknowledgments

This work would not have come to its completion were it not for an innumerable number of friends, family members, and others who offered their support and encouragement throughout the process. This list is by no means complete:

I must first thank my family for putting up with me throughout the entire process: Steve, Lauren and Gillian Kruskall – for putting up with an engineer over the many years. I owe them my infinite gratitude for all of their love and encouragement over the past year and a half. To my Mom, Margot Kruskall, whose voice still rings clear in me as both a comfort and guide. In addition, the four aunts: Carol Moffitt, Jan Bennett, Paula Gowdy and Denni O'Day, for providing counsel at any time of the day or night. And certainly, the extended family: Gary, Kristin, Erin and Brian Gowdy – all of whom are in and of themselves inspiring in their day to day lives.

Many thanks to Paul English, Giorgos Zacharias, Carl Livadas and all others of Kayak.com who put up with an erratic schedule throughout the end-game. To Bill O'Donnell, especially, for teaching me about the delicate art of high-octane coffee.

The word 'thanks' cannot convey my gratitude to Fen Tung and David Lee, who were the first to hear of the idea for this thesis, who encouraged me to pursue the concept further, and who lent me a hand more times than I can count.

I owe a lot to Matthew Goral (SB '12) and Iolanthe Chronis (SB '08, MEng '10), whose friendship and help throughout the process means more to me than they may realize. To the brothers of Sigma Nu, and to my friends Albert Kwon (SB '08) and Olivia Kim (SB '08) – all of whom kept tabs on me throughout the process.

To Professor Dina Katabi, whose counsel I sought on many a wireless communication protocol, and who provided fantastic insight into the world of wireless networking.

To Manolis Kellis, whose thoughtful and kind advising throughout MIT inspired and challenged me (in a good way). To Jud Harward, Chris Cassa, George Kocur and Tony Eng, who graciously allowed me the opportunity to teach here at MIT, and to my students, for being amazing and hilarious throughout the semesters for which I was blessed to have them.

To Anne Hunter, with whom meetings were nothing but side-splitting joys. To Kim Benard for putting up with me and acting as an advisor throughout my last semesters here at MIT. To all of my professors, to whom I am eternally grateful.

To all those I do not mention, for having an extremely meaningful impact on my life, no matter how small you may think it is. You're probably underestimating its importance to me.

And of course, to Larry Rudolph, for his wisdom, guidance and tolerance throughout times that were stressful and those that were not, in addition to his providing me with numerous amounts of test equipment, including phones, bluetooth dongles, cables, and last but certainly not least: food.

Chapter 1

Overview

We introduce the AirRAID protocol, a novel protocol for the sharing of a secondary medium (such as 3G or 802.11 WiFi) over an extremely dynamic wireless mesh network supported by Bluetooth-enabled handheld cellular devices. Making use of the Symbian and iPhone operating systems, we explore the possibility of implementing, over hand-held devices, a reliable mesh network designed to reduce load on existing cellular networks. We envision the possibility of having to rely less on existing (expensive) network infrastructure as we take most of the cellular load off of towers themselves, and place it into the hands of the consumers, saving power, cost and construction of additional cellular towers.

Perhaps due to insufficient technology or algorithmic investigation, we believe that the field of hand-held wireless mesh networking has yet to be explored to its full potential, yet as technology races forward with respect to innovation, so does the potential for the Bluetooth chipset on most mobile phones to be able to handle networks such as an AirRAID mesh. With this in mind, we also introduce an improvement over existing Bluetooth discovery algorithms to reduce the time needed for network creation.

1.1 Introduction and Motivation

As the number of hand-held devices increase in the world, so does the need for constant connectivity to that amorphous network of data that we simply call, 'the cloud'. The introduction of new data network overlays on top of the existing cellular phone infrastructure brought internet connectivity to areas where cell signals may penetrate; however, the base speed, throughput and reliability of such connections are often suspect, dependent on load, in particular, the number of users looking to send transmissions through the same network at the same time.

In addition to having capabilities for high-speed data transfer through cellular transmissions, many of the new handheld devices have support for ad-hoc communication between one another, typically in the form of either 802.11 WiFi ad-hoc connections, or Bluetooth radio communication, and yet, oddly enough, these forms of inter-device communication are often forgone for the alternative of sending data through the 'cloud' over cellular connections, even when devices are within close proximity.

With such infrastructure in place, it seems that only for lack of correct routing algorithms and protocols, mobile devices are stuck using suboptimal paths to communicate not only between devices, but also to the internet as a whole.

Our case example is the Apple iPhone, a handheld 'smart phone' which is able to connect to the internet in two different ways, either by making use of the AT&T cellular data network (Edge or 3G), or by connecting to a wireless router that is within range. The iPhone prefers, when given the option, to connect to the internet using 802.11, and will switch active connections from using the Edge/3G networks to a WiFi connection the minute it is able. This switch produces transition delays, where while the iPhone attempts to secure a new IP address, it is temporarily rendered blind to the internet and all open connections. It is readily apparent that Apple has provided no infrastructure

for the device to temporarily be dual-homed, hence requiring that this switch occur.

As an example of an inefficiency, there exist many apps for the iPhone that share data across two mobile phones, and yet choose to send data through the internet rather than through bluetooth when the two phones are in close proximity. The popular vcard-sharing application 'Bump' (an application which, when two phones are bumped together, allows the two phones to store each other's contact information without manual entry) as of the publication date of this thesis, was still exchanging contact information by beaming the location of the two devices to a remote server through the 3G/Edge network, requiring the server to make the determination as to proximity, despite the availability of bluetooth on the two devices.

The complex communication schema currently employed by the iPhone would make it seem, then, antithetical to propose making use of yet another communication schema for accessing the internet, however, with this project, we aim to show that by using a unique routing protocol, as well as a robust data packet structure, we can achieve multiple goals with respect to improved functionality, including minimizing transition times between communication schemas, increasing overall throughput, and reducing load on the cellular data networks, all while correcting the inefficiencies seen in the current routing protocols and providing a protocol that is meant to work seamlessly with existing APIs.

1.2 Mesh Network Motivation

In densely populated environments, where one can make an assumption that very few handheld devices are ever completely isolated from another device, it is conceivable that one could extend the concept of the now common-place Internet Protocol, Border Gateway and Address Resolution protocols, allowing each handheld device to act as a router to which other handheld devices could connect, sharing data in a manner much akin to that of wireless router and lap-

top computer. However, handheld devices are unlike the static wireless routers of previous projects, but rather are extremely non-static nature, altering the topology of the potential mesh at a relatively high rate.

If we are able to solve the issue of the possible ever-changing network topology, devices with idle bluetooth connections and active connections to the internet, whether it be through 802.11 or a cellular data connection, are perfect candidates for allowing peers, who may or may not have an active internet connection, to connect and bounce data to the internet through the idle device.

In addition, it is conceivable that one could, much like a RAID controller given multiple hard drives, split the transfer of data across two or more connections to the internet (much akin to the idea of 'striping' seen in RAID devices), theoretically resetting the maximum wireless throughput to be that of bluetooth chipset's data transfer limitations (further throttled, of course, by the available bandwidth of the internet connection at the mesh network's exit point).

1.3 Proposed Protocol

With this idea in mind, we propose a novel mesh network system, the Air-RAID (or an over-the-Air Redundant Array of Independent Devices), making use of the mostly unused bluetooth transmitters in hand held devices, and employing a revised version of a robust path routing algorithm to allow for a secondary mesh network infrastructure to arise from the bluetooth transmitters commonly found in handheld devices today. In addition to providing an additional method of access to the internet from these handheld devices, we also envision said network acting as an extension of the actual cellular network, eventually allowing for fewer cell towers to be placed in densely populated environments, allowing cell phones themselves to act as replacement towers.

Chapter 2

Technology

2.1 Cellular Data Network Replication

2.1.1 Architecture

Cellular networks function by partitioning a geographic space into (typically) hexagonal areas simply known as 'cells', in which a single frequency is used to communicate between towers and handheld devices. Towers are placed on the corners of these 'cells', and are outfitted with directional antennas, allowing for minimal interference or overlap of frequency use.

A device may operate within one cell, but as it moves towards another, we wish for communication to continue uninterrupted. What occurs in a typical cellular network is known as a 'soft handoff', a way for the device to switch over to a new frequency in the new cell, without losing any data packets along the way. In order to complete this switch losslessly, the handheld device temporarily maintains two separate connections across two frequencies (at the border of the two cells), and, after checking that the new frequency is indeed safe to communicate on, cuts the former connection, leaving the new connection intact and functional for continuing the data streams involved.

All this is fine if the device is currently involved in a conversation, as the

towers and core network of the cellular system are in constant communication and able to track the device, knowing where to send data packets that are meant for that particular device; however, what if there is no active connection to the device, and the core network needs to make contact? To track the multiple devices across the numerous cells, the core network keeps a database that tracks in which cell(s) the device was least seen. When wanting to contact the device, the towers go through a process known as paging, sending out a message on multiple control frequencies on multiple towers to attempt to locate the phone. Rather than sending a broadcast message across the entire network of cells, using the database that tracks the devices, the core network sends the message to a small subset of towers (known as the "Location Area") in which the phone is most likely to be given its last location and the time since its last contact [2].

2.2 Replicating a Cell Network

In order to replicate a cell network, one must begin to apply a small amount of what we like to call 'fractal logic' to the challenge. Let handheld devices, those that are in sight of real towers, take the place of what we would typically call towers, and let handheld devices which are unable to see a cellular connection take the place of what we would typically call handheld devices. For simplicity, let's call handheld devices that are within sight of real towers "repeater nodes", and those that are not "blind nodes".

There are behaviors that we must now replicate given the new identities of these devices; namely, paging and handing off connections.

2.2.1 Paging

To accomplish paging, and to do so with scalability in mind, we'd like to replicate the idea of the "Location Area" from section 2.1.1. The core network needs to modify the way in which it stores the location of its handheld devices. Rather than being able to calculate a list of towers in range of which the device

may likely lie, in an ideal implementation the core network would also maintain a "Location Area" of repeater nodes near which a potentially blind node is likely to be. In a simplistic implementation, the core network may send a paging message to the location area, and have each repeater node check their blind neighbors (if any) to see if one matches for whom the paging message was sent. If so, a connection can be set up to be forwarded through the repeater node to the tower.

2.2.2 Handing Off Connections

Handing off connections adds a whole new level of complexity to the system, now that there are two levels of communication at work. In addition, the amount of complexity is dependent on the motivations of the respective parties, both users and cellular providers, for it may be in the best interest of the cellular provider to keep as many connections *off* of its towers as possible (a behavior we'll refer to as provider-centric), while users may prefer connections through towers (user-centric), as they may be more reliable than connections through repeater nodes.[1] We leave the exact tuning of the system up to the cellular providers, encouraging the detailed analysis of performance versus cost before defaulting to the expected provider-centric behavior.

We detail four scenarios in which a handoff must take place, and describe the behavior that we would prefer to see in each situation. In each scenario, we assume that providers have selected to be user-centric, namely that the cellular providers are not looking to minimize load on their networks, and would prefer that data traffic be kept directly between tower and node.

Tower to Tower handoff

In this new system , we expect to keep tower to tower handoffs unchanged. Their behavior would remain as it is outlined in section 2.1.1

[1]Despite the labels of the two behaviors, we make no assumptions as to the preferred behavior on either party's behalf. It is very possible that providers would prefer to have traffic over their towers for profit-based reasons

Tower to Repeater Node Handoff

In an ideal world, being able to monitor the device to tower signal strength, we may have an idea of when a handoff would be necessary. As signal drops, the AirRAID system begins to scan on bluetooth for local repeater nodes within as few bluetooth hops as possible.

Once it has located a suitable repeater node within range, the device may start replicating data/voice packets sent to the tower to avoid loss, and finally, once signal is completely lost, the device will rely solely on the bluetooth mesh network to communicate packets through to the cellular network, dropping the cellular connection, and completing the soft handoff described in section 2.1.1.

Repeater Node to Tower Handoff

If a node remains on the bluetooth mesh, but comes to be in sight of a tower, the reverse of the above handoff should occur. Once again, data is replicated until the connection between node and tower is deemed reliable (assuming a preference for tower communication). Only then will the device cease sending packets through the bluetooth mesh, and begin solely sending packets directly to the tower.

Repeater Node to Repeater Node Handoff

In the event that a tower is not visible, and the current repeater node looks as though it is unreliable (which, given the dynamic topology of a handheld-device-based network, may occur quite often), it is in the best interest of the blind node to seek a backup repeater node such that it may perform a soft handoff to the other repeater.

During anytime that the node is active on the bluetooth mesh, meaning that it has an active data/voice connection that traverses the mesh), the node will poll for other 'backup paths' to the tower. For maximal reliability, the node should look for paths that are as disjoint from possible from its current primary path. Keeping this backup path in memory, the node only performs

a soft handoff when either packet loss percentage is deemed to be too high to maintain a reliable stream of data, or when the backup path is deemed as more reliable (i.e. fewer hops) than the primary path. If the latter occurs, then the old primary path is swapped with the new backup path, such that a backup (albeit, a potentially lossy one) is still available in case of topology changes.

At no point during its membership on the mesh network should a node be without a backup path unless it is in the process of actively searching for one.

Chapter 3

Bluetooth Communication Technology

3.1 Bluetooth Overview

Bluetooth, used to allow devices to communicate over a short range (typically, 100 meters or less), operates on the ISM (2.4 GHz) band, and uses channel hopping over 79 distinct channels to minimize interference from other devices operating on the same band.

3.1.1 Bluetooth Discovery Algorithm

The true Bluetooth discovery algorithm splits the wireless spectrum in half, creating two subsets of channels. After pseudo-randomly sorting the two subsets, the device hops through the first subset for approximately 2.5 seconds, after which point it jumps to the other subset, and again hops through that set. We simplify the logic below to assume that the device does not split the spectrum in half, but instead randomly jumps through the 79 channels.

When one bluetooth device would like to communicate with another, they first must "discover" each other on the bluetooth spectrum. To accomplish

this, each device randomly scans a subset of the available frequencies, listening for evidence of other active bluetooth devices. Assuming two devices are both attempting to discover each other, the average discovery time for the two is relatively long, and, in some earlier implementations of bluetooth, had an upper limit of 10.24 seconds.

In the worst case, if both devices are in "discovery" mode, it is probabilistically possible that the two devices will never even encounter each other on their random traversals through channel space, each presuming that the other doesn't even exist (despite perhaps extremely close proximity).

Bluetooth has often been written off as a meaningful communication schema simply because of its relatively long acquisition time, and as such, many attempts have been made to improve the timing of the primary discovery method.

Woodings et al. worked on improving the close-proximity device situation, where two devices that were practically next to each other may not discover each other on the bluetooth spectrum in the 10.24 seconds that were allotted to the discovery method. To speed up the discovery process, et al. used the added medium of infrared to communicate and mutually agree upon a bluetooth channel on which the two devices would be able to communicate.

While this method works extremely well for situations that they devised (for example, checking out at a store using your phone at a cash register), when devices are not in line of sight, their method returns to the failure mode of typical bluetooth discovery, and even adds additional timing onto the discovery scheme as the devices must first ensure that they are not in infrared range [12].

3.1.2 A theoretical improvement

Introducing randomness to the bluetooth discovery algorithm also introduces the very real possibility that the two devices will never find each other on the bluetooth spectrum, and perhaps, that if they do, the timing of the ordeal is simply too long for a user to sit through patiently.

The problem lies with the random jumping. With randomness, there always

exists the possibility that the two devices will simply miss each other at each jump. That is to say:

The probability that a device lands on any one channel during a jump is $\frac{1}{n}$ where n is the number of channels available on the bluetooth spectrum. The probability, then, simply assuming complete randomness and perfectly-timed jumps (i.e. each device jumps at the same time), of two devices finding each other on the same channel is $\frac{1}{n^2}$.

When n is 79, this probability of mutual discovery is less than 1% per jump. To be more specific, assuming independence of all stochastic processes:

The probability that, at each hop, two nodes see each other in channel is:
$\frac{1}{79^2}$ (the probability they each pick the same channel)
The probability that, then, the two nodes miss each other is simply: $(1 - \frac{1}{79^2})$
Given that the nodes hop channels at a speed of approximately $1600\frac{channels}{second}$, and have an upper limit search time of 10.24 seconds, the likelihood that, in 10.24 seconds, the two nodes miss each other completely is:

$(1 - \frac{1}{79^2})^(10.24 * 1600) = 0.072 = 7.2\%$

With each discovery, there is approximately a 1 in 14 chance that the two devices miss each other completely. In a mesh network that relies on mutual discovery, it is likely that many acquisition failures would occur, especially as the network size scales; however, our overall goal of a relaible mesh network certainly would gain from the construction of a network that is accurate in terms of determining its own edge connectivity (which nodes are able to see each other). With the current Bluetooth acquisition protocol, the network is likely to miss a significant portion of edge information, degrading overall performance.

An improvement may be found in eliminating a portion of the randomness. Imagine, for the sake of comprehension, a racetrack along which two blind runners, staggered, are running in the same direction. The goal of the two runners is to pass off a baton between the two of them, and hence, to do so, they must be at the same part of the track at the same time.

Bluetooth's current discovery method works as though these two blind run-

ners had the miraculous ability of teleportation, and as though the two had imagined that the best way to encounter each other on the track was to randomly appear in different places along its length, hoping to bump into each other at some point if they continue teleporting randomly.

Were this teleportation method the norm in relay races, the two runners would be a comical, albeit confusing, sight to observe. Luckily, in normal relay races, a forerunner runs somewhat slower than his/her counterpart with the baton, allowing him/her to catch up and make the hand-off.

To find an analog in bluetooth radio discovery schemas, let's let one device begin 'running' the channels, numbered $1 - 79$, sequentially at a given speed v. Let another device also begin running the channels, in the exact same order, but not necessarily starting in the same location, *and* at a different speed. Given enough time, the faster device will eventually catch up to the slower device, and be able to communicate on whichever channel they both happen to have landed at the time. This situation eliminates the possibility that, given enough time, the two runners would never find each other.

In the worst case scenario, a faster device 'A' would start one channel above device 'B'. In this situation, the amount of time it would time for the two to meet, assuming 'velocity' is in channels per second, is simply:

$\frac{1}{V_a - V_b} * 79 channels$

Naturally, the larger the difference, the faster the acquisition time.[1] In absolute worst case, V_a approaches V_b, asymptotically bringing the acquisition time to approach infinity. In order to improve upon the current acquisition method, it is in our best interest to select speeds which would produce a guaranteed acquisition time of less than 10.24s (assuming $V_a! = V_b$). We use this bound to find the minimum size of quantized channel-hopping speeds:

$\frac{1}{V_a - V_b} * 79 < 10.24$

$\frac{79}{10.24} < V_a - V_b$

[1]Note that the best case, of course, would be for one node to sit completely still. However, deciding which node should sit still is yet another relatively similar problem to the original case at hand

$7.71 < V_a - V_b$

$V_b + 7.71 < V_a$

Therefore, in order for this new technique to not only work, but also perform better than the previous algorithm, distinct nodes must be assigned unique 'hop velocities' which are at least $7.71 \frac{channels}{second}$ apart from each other.

Normal bluetooth pseudo-random channel hopping occurs at approximately 1600 channels per second. Presuming both a maximum speed of $1600 \frac{channels}{second}$ as well as wanting to always have each device's 'hop velocity' to be at least $7.71 \frac{channels}{second}$ away from others, we should quantize the space in such a manner as to allow 207 different speeds (numbered 1 through 207, and multiplied by 7.71 to get the true channel-hopping speed).

Again, we have a similar problem as before. With only 207 speeds to choose from, it's possible that two devices may pick the same speed, yet, unlike before, they would be **guaranteed** to not find each other; however, if, in a simple implementation, we choose to pick a pseudorandom speed for each discovery process, the probability that two nodes pick the same speed (assuming, again, independence), is just $\frac{1}{207^2} = 2.33 * 10^{-5}$, which is significantly less than 1%.

To analyze further, we note that, in the second-to-worst case (where two non-equal velocities are chosen, but they are proximal to one another), the acquisition time is approximately 10.24 seconds.

With our algorithm, we are able to quantize the space differently based on the minimization of two (unfortunately, conflicting) parameters: The probability of a complete miss, or the maximum time spent before successful discovery. We have already minimized our probability of a complete miss given a maximum time of $10.24s$ and have shown that the optimal quantization implies speed differences of at least $7.71 \frac{channels}{sec}$, but similarly we can retweak the quantization to minimize the discovery time. The larger the channel-hopping quanta, the faster the acquisition time, but the more likely a complete miss becomes. Holding the probability of a complete miss to be at the original 7%, we will solve for

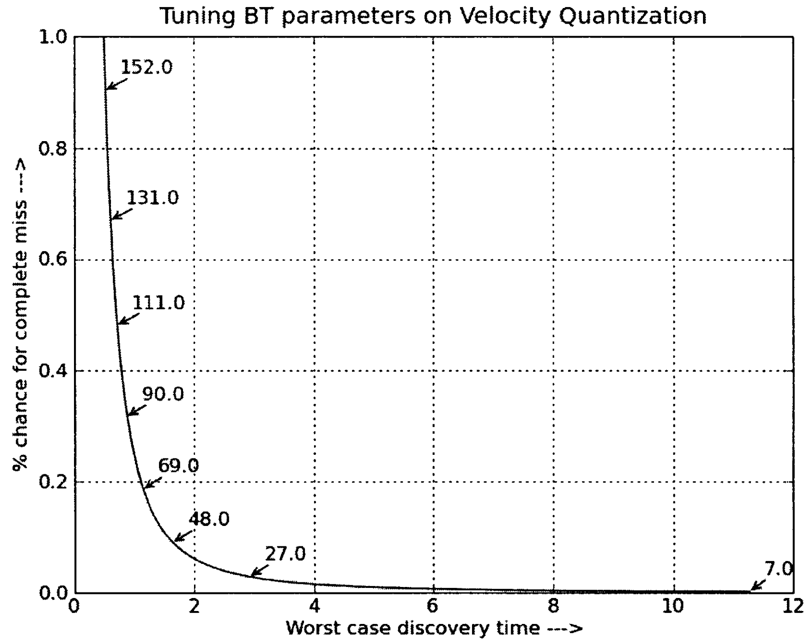


Figure 3.1: Tuning Bluetooth Parameters ; Those values pointed to on the graph are channel-hopping speeds

the largest size of the quanta possible:

The number of quanta, given a quanta size Q is assumed to be: $\frac{1600}{Q}$

Hence, the probability of two nodes picking the same speed is:

$(\frac{1}{\frac{1600}{Q}})^2$

We would like to bound this probability at 7%:

$(\frac{Q}{1600})^2 < 0.07$ $Q < 423.3 channels$

Meaning, if we were to quantize our velocity space with maximal differences of $423.3\frac{channels}{second}$, we would expect a 7% chance of a complete miss during discovery.

To further understand the trade-offs, the graph below shows the tradeoffs associated with selecting different quantizations of the channel velocity space:

The actual implementation of this protocol is outside of the scope of this thesis, but its description is included as an additional hurdle to jump before a bluetooth mesh network becomes a strong viable option for inter-device connec-

tivity.

Chapter 4

Previous Mesh Networking Examples

4.1 Roofnet Approach

Growing out of an MIT research group, the company Meraki presents a unique solution that creates a mesh network out of static signal repeaters for WiFi. Intended for use in cities and towns as a way to provide wireless access to the public, there is little indication that Meraki has worked with incorporating mobile devices (laptops, phones, etc) as additional repeater nodes [9].

4.2 Cartalk Approach

Cartalk, also known as the WAVE standard, has emerged as a possible method to connect motor vehicles to the internet. As one of the faster-moving nodes that would require connectivity, a car proves to be a difficult challenge in terms of maintaining said connectivity, especially as network conditions change (presumably at an extremely fast rate). The WAVE solution employs the use of stationary access points for providing internet connectivity, as well as creating secondary structure for car-to-car communication and routing.

Through the use of advanced routing algorithms meant to adapt to ever-changing topology, Cartalk approaches closely what we wish to implement for the mobile pedestrian, albeit requires additional infrastructure and hence increased cost to implement on national highways, at the very least [11].

4.3 Dynamic Source Routing

A routing algorithm by the name of Dynamic Source Routing works by creating a mesh network that is extremely flexible in the case that nodes leave or join the network. But the DSR algorithm has not yet been publically applied to hardware such as cell phones or any other wireless-enabled device where a media-to-media adapter interface (e.g. Bluetooth to CDMA, Bluetooth to WiFi, etc.) would serve a constructive purpose [7].

DSR is an extremely promising routing algorithm for situations in which the arrangement of the nodes in the network is dynamic or even unknown. Paths are generated by flooding the network with a request, having each node forward the request until it hits the target node. Upon reception, the target node replies back to the source with a traversal list of the original path request packet, which has now become a valid path between the two nodes in question. Requesting paths through the mesh as they are needed, DSR can be relatively flaky, dropping connections as often as nodes exit the mesh, but is easily recoverable by simply requesting a new path. In addition, the DSR specification implies optional optimizations including path-fusion, wherein if a node A receives a request to another node B, and that node A already has a known path to node B, rather than continuing to forward the path request, node A will simply reply back with a fused path of the requests's traversal list and its known path.

4.4 The AirRAID Approach

The AirRAID protocol is based heavily off of DSR, yet adds additional optimizations to minimize the number of nodes involved in communication at any given time. AirRAID puts an inherent value on knowing and keeping paths between nodes. Rather than throwing them away as they become invalid, each node in an AirRAID mesh is responsible for maintaining each path for which it is an active node. In doing so, each node maintains a backup node for which it is able to forward traffic should its original forwarding node exit the mesh for any reason.

In addition, to minimize the number of hops traversed in any transmission, each AirRAID node continuously searches for ways to minimize the number of hops that any path contains. Just as moving to using a backup node may potentially lengthen a path, each AirRAID node can 'tighten' a path by finding a way to skip nodes on its traversal list.

Chapter 5

Initial Work on the Symbian Operating System

Originally, with the idea of using the Symbian operating system, along with its port of the python language for the initial implementation of AirRAID, we considered the boon of having each node be able to conceive a map of the mesh, for if each node had an idea of where its data's destination were located, each could employ intelligent geo-informed routing techniques to minimize what could otherwise be a large amount of flooding across the mesh.

5.1 Forming a map from minimal information

Forming a map of a mesh network when one node is only able to directly see its neighbors is akin to pinpointing the location of lost spelunkers in a cave while being anchored to the floor, equipped with nothing but a weak flashlight. In other words, to form the map, there is going to be a good deal of shouting. We considered, then, the possibility of taking a page from the link state routing protocol, namely, having each node inform all other nodes in the mesh of its neighbors, but with an added bit of information. In addition to sending its list of neighbors to all neighbor nodes, each node would send what it considered to

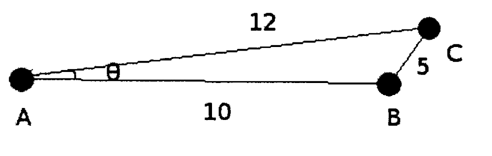


Figure 5.1: Example Triangulation of 3 Handsets

be the approximate distance between itself and its neighbors to anyone in range.

Approximating distance, however, with a weak flashlight, is not an easy task. Straight line distance can be approximated using low level bluetooth signal strength measurements, such as the RSSI[4]. The biggest limitation, however, would be the lack of directionality with any measurement obtained. The good news is that this too can be overcome in dense networks.

Imagine that we have a very simple network of three handsets, labeled A, B, and C. Presuming that all three handsets were in range of each other, without necessarily being able to absolutely locate each handset in an absolute positioning manner, each handset would be able to know at least the relative position of the others. Each handset would not only report their perceived distance to each other, but would also report to each handset the complete set of distances between itself and all other in range handsets, allowing each handset to construct its own map.

As an example, let handset A be 10 units away from handset B. Handset B is 5 units away from handset C. Finally, node A reports a distance of 12 units to handset C. Constructing the triangle between the three of them would allow, for example, handset A to conclude the angle between handsets B and C using the law of cosines:

$\theta = cos^{-1}(\frac{a^2+b^2-c^2}{2ab})$

If every handset was in site of at least two other handsets, this method could be done repeatedly in order to obtain a complete map of all the handsets in an AirRAID network.

Even phones that were out of range of a given handset could be approximately placed relative to known handsets by assuming the worst case scenario

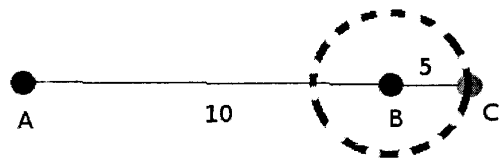


Figure 5.2: Example Triangulation of 3 Handsets

(i.e. the handset that is out of range is as far as possible away from the handset generating the map). If in the above example, we knew as we did above that the distance between A and B was 10, and that between B and C was 5, we wouldn't be able to relatively position all three handsets. However, knowing that C is not in visible range of A, we can assume the worst, and ascribe a straight line relationship between all three handsets.

We need to remember that a map is not necessarily what we are trying to get out of these calculations, but rather we are trying to determine whether a given phone is worth routing packets through. With this in mind, we are perfectly happy working with the worst case scenario in deciding which connections to maintain in the mesh network.

5.2 Initial Experiments with Symbian Bluetooth

Work in the 2008 spring semester focused on testing the feature set of a Symbian phone using Bluetooth to communicate to a fake handset simulated by a computer. Using python, the following tests were carried out to evaluate the feasibility of the entire project.

5.2.1 Water Interference Test

Concern had been expressed that, as water absorbs electromagnetic radiation, that the more humans (read: handsets) introduced into a small area, the larger the number of packets lost. This wouldn't bode well for the system as voice quality is dependent on the number of lost packets in a given connection. However, the denser the network, the more connections that can be made,

hence the more reliable the overall end-to-end connections. This test served to show that there is little to worry about in most normal situations involving a relatively densely packed area.

Implementation

We installed onto a Symbian handset a python script which allowed the phone to act as a simple echo service. Once another bluetooth device connected to the handset and identified itself, anything sent from the computer to the handset would be immediately reflected back to the computer.

On the PC we ran a python script which, after finding and connecting to the phone, would send timestamps repeatedly to the phone over an RFCOMM connection. Every time a packet returned, the PC would mark the time that it came back. After sending approximately 2000 packets, the computer would calculate average transit time for each packet, as well as sum up the number of lost packets. A packet was labeled as lost if it took longer than 0.5 seconds to return to its source. We chose to impose this definition of lost packets as any voice data that returns after a significant delay might as well be considered useless, as the playback time for the data in question may have already passed.

This test was to be run multiple times in varying environments. Most importantly, at least two runs of the test were to take place in each location, one with a large number of people between transmitters, and another with a relatively few number of people between transmitters.

Results

Various runs of the test in a 10x15 foot room produced the following results:

Discussion

The above data shows no visible correlation between the amount of people in the nearby area and the number of dropped packets and/or packet transmission time. This may have been for any one of a number of reasons.

Test Run	Missed	Turnaround Time (avg in seconds)
1	0	0.074
2	0	0.071
3	0	0.071
4	0	0.071
5	0	0.074
6	0	0.071
7	0	0.071
8	0	0.071
9	0	0.072
10	0	0.071

Figure 5.3: Water Interference Tests run with Dense Population

Test Run	Missed	Turnaround Time (avg in seconds)
1	0	0.070
2	0	0.070
3	0	0.072
4	0	0.071
5	0	0.071
6	0	0.072
7	2	0.073
8	0	0.071
9	0	0.072
10	0	0.072

Figure 5.4: Water Interference Tests run with Sparse Population

Because the computer implementation was done on a Windows machine using Python, the typical bluetooth voice protocol, SCO, was unavailable as an option to test. SCO is an asynchronous protocol optimized for voice communication over a bluetooth link, and as such would be the natural choice for testing a network meant to carry voice data.

With RFCOMM, packets are expected to arrive in order. If there is a delay in one packet, all the following packets are delayed until the packet in question can be reliably transmitted.

It's also possible that the layout of the testing location shown above was sufficient enough to guarantee packet delivery in a timely manner, despite the number of nearby people, simply by bouncing the signal off of nearby architectural sounding boards.

5.2.2 Distance Curve Test

As Python on the Symbian OS is a relatively comfortable language to program in, we had originally hoped that we would not need low level operations in implementing the overall system. As a substitute, we had hoped that we could use the transit time for a ping between handsets, combined with the number of lost packets, to estimate the distance between two handsets.

Implementation

Much like the implementation outlined above for 5.2.1, we placed on a Symbian phone a script that turned it into an echo system, as above, but with extra features in that keypresses on the keypad of the phone would be sent back to the laptop to be used in data collection.

On the laptop, a continuous stream of packets were sent from computer to phone once identification was complete. As the handset holder walked away from the computer, he would press a button on the phone to notify the computer that the physical distance between the phone and the computer was increasing. He would similarly press a separate button to notify the computer that distance

Distance (paces)	Average Transmission Time	Standard Deviation
0	0.033	0.020
1	0.033	0.017
2	0.032	0.010
3	0.030	0.017
4	0.030	0.015
5	0.030	0.017
6	0.033	0.023
7	0.035	0.020
8	0.048	0.033
9	0.033	0.019
10	0.035	0.022
11	NO DATA	NO DATA
12	0.045	0.036

Figure 5.5: Test Results for Distance vs Packet Turnaround Time

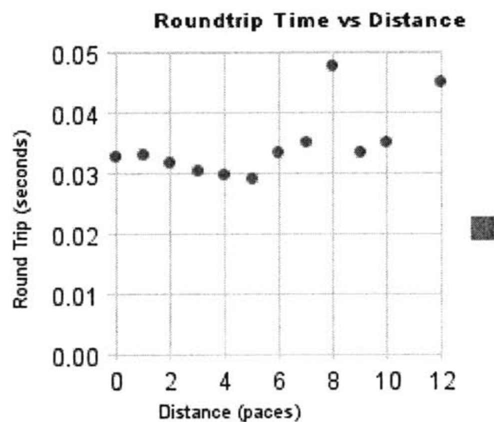


Figure 5.6: Round Trip time Versus Distance

had decreased.

When the test was over, the handset user could press another key on the phone to signal the computer to tally up the statistics, including average transmission time as well as the number of missed packets at each distance level.

Results

Discussion

The data above shows little to no correlation of distance to packet turnaround time using a bluetooth RFCOMM connection. The data point with distance 8

appears interesting, but due to a high standard deviation, is likely not significant to the original goal of the test. As we are trying to obtain a heuristic for distance, such similar values and high standard deviations imply that this method will not work for identifying distances between handsets.

As the handset holder walked out of range of the computer, it became quickly evident that issuing further commands to the computer was not possible until the user walked back into range. The packets that were then marked at an incorrect distance were only a small part of the dataset, and as such were not worrisome in consideration of the results.

Chapter 6

Implementation of the AirRAID Communication Layer

In this chapter we explain the overall algorithm that, learning what was and was not possible from Symbian experiments, we imagined would be feasible on existing hardware. The AirRAID communication layer, when implemented correctly, should expose nothing more than a method for sending data to a certain peer (given an IP address). Ideally, AirRAID should act as an additional layer in between the application and physical medium layers, and do so with minimal changes necessary for an application to implement it as a communication framework.

6.1 Mesh Creation

Each node periodically sends out service advertisements to the primary radio network (in our case, Bluetooth), notifying local nodes that a specific

application-specific AirRAID network is available.[1]

Upon hearing that a new neighbor has become available, other nodes in range will attempt to connect. Once the connection is established, we begin to exchange AirRAID packets to establish paths as well as communicate basic information between participants in the mesh network.

6.1.1 AirRAID Packets

AirRAID packets are, in essence, associative dictionaries that are encoded upon transmission into a sendable format, such that they may traverse a lower-level communication layer. Upon reception at a node, a PacketDecoder looks at the data received, and determines what type of Packet it has received, propagating the packet up to the AirRAID communication layer. There are many different types of Packets, but each has many data fields in common. Each packet contains a source id (the ID of the originating node), as well as the id of its destination. In addition, it maintains the address of where the packet should be forwarded to next (it's "Next Hop"), as well as a list of the nodes that it has traversed thus far. Finally, each packet maintains a separate dictionary of data, used to contain the payload of each Packet subtype.

Each time that a packet leaves a node, the node updates the packet's list of traversed nodes with the node's ID. Otherwise, most of the processing for packets is type-specific, and will be explained in the sections that follow.

In the following sections, we will introduce the packet subtypes in order of relevance.

6.1.2 Pleasant Introductions

Upon connection, and periodically thereafter, each node produces what is known as a HELLO_PACKET, which contains information pertinent to its existence in the mesh, including its bluetooth id, its connectivity to the internet and

[1] Note that we chose to separate applications for anticipated performance issues. In separating applications, we are able to avoid the challenge of fair load balancing. Future versions of AirRAID will attack this challenge.

cellular networks, its approximate location, its neighbors, and even its battery life. HELLO_PACKETs are only allowed to travel one hop, and are actively dropped by each receiving node, that is, once it has finished storing the relevant information about the sender.

6.1.3 Path Requisition

In our system, requesting a path to an internet destination is analogous to requesting a path to a wifi-connected node, and as such, the two are treated as one process in that the bluetooth id 'WIFI' is used to represent a hop to WiFi, 'CELL' to hop to the cellular data network, and 'INTERNET' to represent either.

Based off of the path routing algorithm used in DSR, AirRAID allocates new paths based on user demand, and by progressively populating a traversal list as path requests propagate through the network.

To request a path, a node fires, to its neighbor nodes, a specific kind of packet known as a PATH_REQUEST packet, containing a destination ('WIFI', 'CELL' or 'INTERNET' being acceptable options), an empty traversal list, and a unique 'Path ID'. At the same time, we store a table of path statuses in the communcation layer, marking down any outgoing request along with a timestamp of the request.

When a node receives a PATH_REQUEST packet, it performs one of two actions, depending on where the request is headed:

If the destination of the PATH_REQUEST packet is not the current node, we append the bluetooth identifier of the current node onto the traversal list, and forward the packet on to each of our neighbor nodes. Note that to avoid duplicate PATH_REQUEST packets, at each node, when forwarding a path request, we act as though the PATH_REQUEST were originating at the forwarding node, up to and including marking it down on a table of path statuses (labeling the current path as a new request). If a duplicate PATH_REQUEST does come in later from another node, we can compare its path id to those stored in our table,

and ignore those that are already known.

If on the other hand, the PATH_REQUEST packet's destination is the current node, then we know that the traversal list inside the PATH_REQUEST packet is a valid path to transmit data between the two nodes. The current node constructs a response packet known as a PATH packet, containing the path id, as well as the full traversal list of the original request (appending itself as the terminus), and sends the packet back along the path from which it came.

When a node receives a PATH packet, it first ensures that its own bluetooth identifier is on the traversal list, and, if so, forwards the packet to the previous packet on the list (hence, propagating the PATH back to the origin). At the same time, the node updates its Path Status Table (path_id -¿ *alive*).

In the scenario in which there exists no path between the two nodes (i.e., the destination node is not on our mesh subnet), the PATH_REQUEST packets are naturally dropped by attrition, and a timeout on the originating node notifies the system that the request has failed. In our implementation, we have set the timeout to be around 3 seconds.

All of this assumes an ideal situation with no errors, no drops, and no changes in network topology. However, it would be foolish to assume that none of these occur, especially given our network constituents, namely, nodes that can potentially move at speeds of 90 mph. We will address this further in section 6.1.4.

6.1.4 Path Maintenance

Retaining resilience in the face of a changing topology is critical. To accomplish this task, the AirRAID communication layer not only listens for when its neighbor nodes disappear, but also periodically checks each of its allocated paths to see if it is able to find a backup node through which it could forward traffic if the normal node were to suddenly fail.

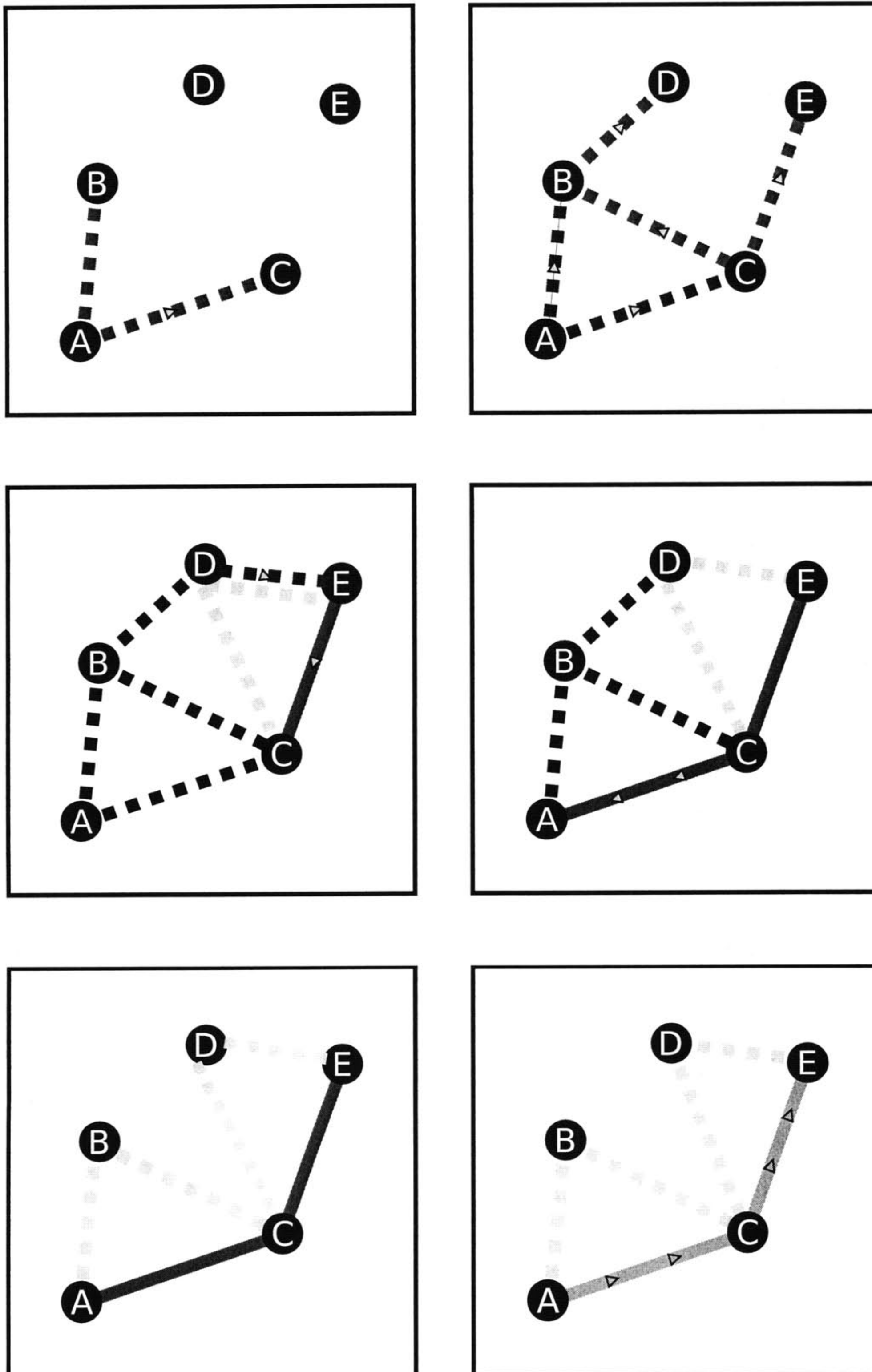


Figure 6.1: The path request process. Dashed blue lines represent active path requests. Dashed black lines represent past path requests. Solid blue lines represent confirmed paths propagating backwards. Solid green represents a completed path, while it's yellow neighbor edges represent backup paths, which are allocated at each backwards propagation phase.

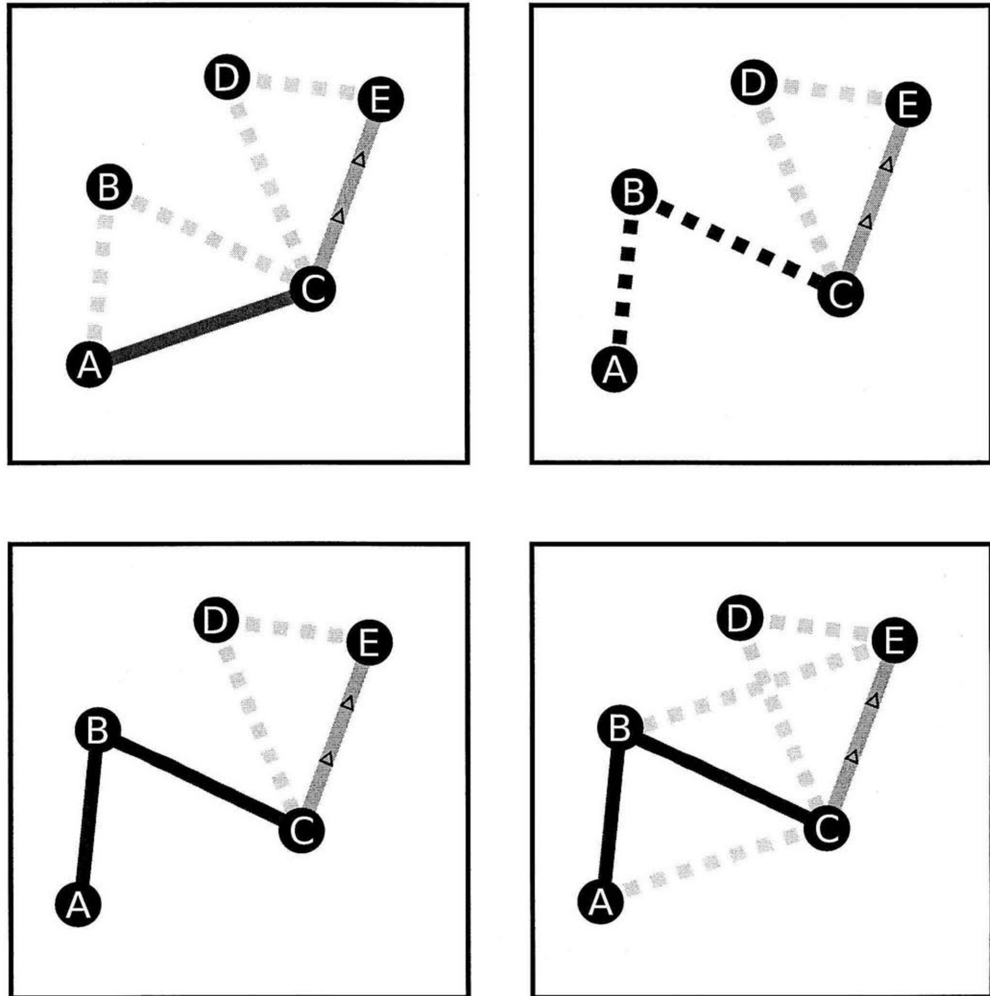


Figure 6.2: The path maintenace process. In this situation, as before, green lines represent active paths. The red line between A and B represents a failed edge. A requests its backup path to activate (asking B, who in turn alerts C). Once the path is re-established, new backups are acquired for all nodes.

Finding a Backup from Scratch

After the allocation of a path, every 20 seconds, at each node the AirRAID communication layer enumerates through its paths, and evaluates the backup situation for each path. In the situation where a path has no backup node, the current node looks at its neighbors for two different types of nodes: those who have the path's next hop in their neighbor list, and those who have the node 2 hops down in their neighbor list. It is clearer with an example:

Let there be an allocated path with nodes N_1 to N_4. It turns out that an additional node $N_2.5$ arrives in the mesh, connecting to nodes N_2 and N_3. N_2, during its normal backup search, sees that it can forward packets to $N_2.5$ and have it in turn forward packets to N_3. This effectively would increases the path length by 1 node, but allows for reliability in the face of a changing topology.

To initialize this arrangement, N_2 sends a BACKUP_INIT packet to $N_2.5$. Inside this packet are wrapped two data fields: the path id of the path that $N_2.5$ is acting as backup for, as well as a traversal list that it is expected to use should its need as backup arise. Upon receiving this packet, $N_2.5$ enters a new path into its records, and records the status as "BACKUP".

In another example, perhaps $N_2.5$, instead of being connected to N_3, connects to N_4. Using it as a backup in this situation keeps the path length the same, but reliability, again, increases.

During this periodic maintenance check, if a backup already exists for a given path at a certain node, the node with the backup sends a BACKUP_KEEPALIVE packet to its backup, signifying that, despite having not had to forward data as a backup path, the backup node should continue to keep its path available. Upon receiving a BACKUP_KEEPALIVE packet, a node looks at its path status table and determines if it really is a backup for that path. If so, if updates the path status with a new expiration time, effectively keeping it alive, and then forwards the BACKUP_KEEPALIVE packet to the next node on its traversal list for that path id.

Path Tightening

In addition, every 20 seconds at each node, the communication layer enumerates through its paths once more, and begins what we call 'path tightening'. Examining the traversal list for the path, as well as the current node's information on all of its neighbors (thanks to periodic HELLO_PACKETs), the node determines whether it is able to shorten the allocated path by skipping over any number of nodes in order to reach its destination. As an example, let us assume a path has been allocated between nodes N_1 and N_10 (with of course, numbers 2 through 9 connected in a straight line between the two endpoints). At its path maintenance stage, N_3 may realize that it N_8 is actually within range, and hence it can forward packets directly to it. As our goal is to keep paths short, N_3 then makes the decision to modify the path permanently. In doing so, it first sends a PATH_MODIFIED_PACKET to N_8; This PATH_MODIFIED_PACKET contains within it the path id in question, as well as a new traversal list for the path, such that each subsequent node can be informed as to what has happened upstream. Then, no matter what the current backup situation is, N_3 will replace its current backup with N_4 (who we know is a reliable backup as we've been sending data through it previously). [2]

Activating Backup Paths

In the event that an allocated path has an interruption at a given node, and that node has a backup ready to go, to activate the backup path, that node only need send a PATH_MODIFIED_PACKET to its backup, which in turn notifies the backup that it can expect traffic.

Backup Failure

Backup nodes do not actively perform maintenance on the paths for which they act as backup. Instead, when a BACKUP_KEEPALIVE packet comes in,

[2] Note that any previous backup will automatically deallocate its path upon not hearing a continued supply of keep alive packets

the backup node attempts to forward it to the next node on the path traversal list. If it transfer fails, the backup node assumes that it is no longer safe for it to act as a backup for the path in question, and in turn, it sends a BACKUP_FAILED packet backwards through its stored traversal list.

Once the BACKUP_FAILED packet arrives at an active node (or, in other words, a node for which the path status is not listed as "BACKUP"), that node removes the backup node from its memory, and temporarily, until it can find another backup, becomes a point of instability.

Overall Behavior of Tightening and Backup Recovery

In the optimal case, as nodes move physically throughout the mesh, paths should contract and expand to accomodate the changing conditions. Ideally, the paths contract as best they can to minimize the number of failure points, and yet, as an additional complication, the fewer nodes to traverse in a path, the fewer backups there are in case of failure; however, one can always simply maintain additional backups at each node. The discussion on keeping additional backups is left until section 7.2.3.

6.1.5 Data Packets

When a node wishes to communicate over an allocated path, it creates and broadcasts a DATA_PACKET. For a DATA_PACKET to traverse the mesh, it must be tagged with a path id, telling each node that it encounters implicitly where it is headed and how it got to where it is now. Besides that, it need only have a block of data.

Upon receiving a DATA_PACKET, a node looks at the packet's destination field, and decides whether or not the packet has encountered its final destination. If so, the data is recovered and propagated up to a delegate of the AirRAID communication layer.

If, on the other hand, the DATA_PACKET's destination is not the current node, the path id is examined and used to look up the traversal route for the

packet. We assume that if a DATA_PACKET arrives at a node, that it must be on the path's traversal route, and hence, using the path information stored at the node (namely, the entire traversal list), the node finds itself on the list, and simply forwards the packet to the next node on the list.

There is always the possibility that the transfer of a DATA_PACKET to its next destination will not proceed without a failure. In that event, if the node transferring the DATA_PACKET has a backup, it simply activates the backup path, and continues the transfer. If on the other hand, it has no backup node to turn to, it must send a PATH_FAILED packet, containing the path id as well as the reason for failure, back upstream along the path.

Upon receiving a PATH_FAILED packet, if a node has a backup, it will activate it, but if not, further back propagation of the PATH_FAILED packet continues until it reaches the source, at which point the device is notified of the failure, leaving further action in the hands of the developer.

6.1.6 Path Expiration

Once a path has been allocated, it would normally be in our best interest to keep the path alive, even if there is no current active data transfer over the path; however, doing so is expensive in that it has the potential to hog connection resources, and so, it is best if we consider no path to be permanent.... After 60 seconds of non-activity, any path on a given node is deemed expired and invalid for further forwarding.

Activity is considered to be either a DATA_PACKET or a BACKUP_KEEPALIVE packet (if the node is indeed a backup for the given path).

When this happens, if a DATA_PACKET comes through after its path's expiration, a PATH_FAILED packet is sent back explaining that, rather than a topology change, simply a lack of activity caused the path to close.

Dealing with Path Request errors

There are multiple points of failure during the path request phase of the AirRAID DSR implementation. As our nodes are potentially constantly moving, it is certainly conceivable that a PATH_REQUEST packet will never be able to reach its destination. Timeouts at each node allow feedback to the software to notify it that a path was simply not available at the time of request. We leave it up to our users as to what behavior they wish to implement in such an event.

6.2 AirRAID Optional Flags

6.2.1 Non-Bidirectional Paths

Above, in section 6.1.3, we assumed that, once a PATH_REQUEST packet has made its way to its destination, that the traversal list should contain a valid path for the two nodes to communicate. This is, for the most part true, as long as the topology of the mesh is relatively stable, for instance when dealing with something as simple as, say, a small and or quick file transfer between nodes A and B. For the most part, A is doing most of the communication; perhaps B sends back ACK packets periodically in response to data packets, but for the most part, little data need travel spontaneously on the reverse path. Potentially, the path from A to B may become invalid, and A may simply request another path, resubmit any data for which an ACK was not received, and continue the transfer.

In another situation, however, node B may have information for A that is not necessarily in response to a packet that A sent, or in other words, it is possible that A, rather than B, after a period of silence, would be expecting data from the other and, due to a path invalidation or even modification in said period of silence, be unable to receive it. For instance, should the path from A to B suddenly have a tightening, B would not be informed, leaving it to assume the incorrect path as a viable data channel. In our above implementation, B would

likely wait for A to reinitialize a path (which could take some time, perhaps sped up if A sends periodic keep-alive packets) until it is able to submit its data.

As an optimization, we have coded into the first testing suite of AirRAID the option of asking for a non-bidirectional paths. That is to say, upon hearing a path request, B (as in our above scenario) initializes its own **unique** path request back to A, performing similar maintenance on this reverse path. By maintaining this extra path, B is able to spontaneously send any information it needs back to A, even if the topology of the network changes.

The advantage to this method lies in its resilient nature. For mutual data transfer to truly fail, both paths must be severed. The disadvantage, unfortunately, lies in the amount of resources that the additional path consumes. Each path, remember, must have periodic checks and maintenance performed to ensure its reliability, and in doing so a number of cycles and data packets are sent between nodes, whereas with a bidirectional path, only one path need be maintained, albeit should the path go down, node B (the destination) would be unable to communicate additional data to A without explicitly opening another path.

Non-Bidirectional paths become extremely useful in high-latency communication, such as when asking another node to contact the secondary medium, say, for an HTTP request. In the time it takes for the request to be completed on the internet, node A should not need to constantly monitor and react to path failures on a path through which no data is necessarily traveling. Node B, rather, in maintaining its backwards path, has the responsibility of finding node A when data begins to stream back from the remote destination.

6.3 AirRAID Implementation on the Apple iPhone

We decided to implement the first draft of AirRAID on the Apple iPhone, as it has come to garner a large percentage of the market, including such a particularly large number of phones in New York City, that AT&T has had

to expand its operational radio range to account for air time demand. With a network stretched to its breaking point, and a phone on which it is fairly easy to develop, there seems no better fit for the AirRAID protocol to make its debut. In addition, with the release of the 3.0 SDK, Apple opened up the possibility for phone-to-phone Bluetooth communication, allowing us to demonstrate a functional, albeit rudimentary, implementation of AirRAID on what seems to be a fairly ubiquitous phone.

Due to the lack of access to actual call information, the current implementation of AirRAID focuses on the forwarding and maintenance of active internet connections between handheld devices; however, expanding the implementation to include voice calls would require nothing more than the cooperation of the phone manufacturers (Apple) to obtain access to the radio API, and the cellular network (AT&T) to slightly modify their existing backend technology to account for the new mesh network option for connectivity.

In the following sections, we assume that the reader is familiar with basic Objective-C and at least somewhat familiar with Apple Cocoa and Foundation classes. For those who are not, Stanford's archives of the class CS193P prove extremely useful in bootstrapping one's way to proficiency[10].

The goal of the iPhone implementation of AirRAID is to provide a seamless integration with current networking APIs available on the iPhone OS. To begin with, however, we focus on creating a communication layer that takes existing NSUrlRequests (a typical way of accessing the internet through the iPhone OS) and adds additional functionality as to facilitate the potential forwarding of requests to other phones in range, should the phone issuing the request have no internet connection.

The current Apple iPhone implementation of the AirRAID protocol exposes more than is necessary: a requestPathTo method, as well as a sendData method. Ideally, the two would be combined such that any application making use of the protocol would require little more than the inclusion of our libraries to facilitate the added benefit of using an AirRAID mesh.

6.3.1 Utilizing the Bonjour over Bluetooth API

Apple's iPhone SDK is available at no cost (however, deploying applications to the Bluetooth enabled phones requires a one-time fee), and allows access to many of the phone's APIs. In its most recent release, Apple made available a slightly irregular, albeit functional, version of bluetooth available to developers; however, unlike traditional bluetooth, the task of polling, hopping, and most other low level actions are abstracted behind another communication layer (Apple's traditional network protocol: Bonjour).

In its strange implementation, rather than actively polling for other bluetooth enabled devices, the iPhone Bluetooth API allows for one to instantiate what's called a 'Session' (GKSession), which, once given a particular name and set to any of Server, Client or Peer mode, begins looking for other phones which have a similarly named session that is active. When the phone asynchronously (and may we add, clandestinely) discovers another node, it is able to notify a delegate of its discovery. At that point, one can decide to tell the session to connect to the device. All of the intricacies of polling, discovery, and channel hopping are purposefully abstracted underneath the API layer, leaving little for the iPhone programmer to do but assume that the phone is able to accurately and with appropriate frequency poll the airwaves for other devices. Sending data between the phones, once connected, is as simple as specifiying to which phones (specified by a unique bluetooth identifier) one wishes to send an NSData object.

Sessions can be thought of as blobs into which nodes(phones) conglomerate. Once a node has joined a session, it is technically connected to all other phones within that session. Much like a switched network, data can be targeted towards a single node, but no one node may choose to disconnect from any other node individually. Rather, each node, should it wish to disconnect, must disconnect from the conglomerate. This model is disconcerting to the mesh-network programmer, as it would seem as though Apple has already solved the mesh-networking problem. We discuss this further in section 6.3.2

6.3.2 The Implications of the Session Model

In terms of mesh networking, the required behavior of using 'sessions' is both useful and worrisome. Its usefulness becomes clear when one reconsiders our initial goal of effectively extending the range of another radio communication medium. One can imagine the cellular network (or the internet, as we implement it) to be the metaphorical siphons into an ever-expanding pool of nodes. The idea of jumping from one subnet to another is a mute point, as the iPhone SDK seemingly handles the behavior seamlessly; however, the conglomeration behavior has an extreme negative: No phone can actively decide to which other phones it maintains active connections to at any given time. Instead, while it is able to choose who it connects to initially, it must disconnect from a **session**, as opposed to an individual node, leaving little options for dynamic rearrangement of the mesh.

An additional complication exists in that Apple's Session model implements its own form of multi-hop mesh networking, albeit a somewhat riskier version than what AirRAID is able to provide. Whenever two nodes connect to each other (or rather, when they connect to a session through each other), each shares the list of nodes that they are aware of, much like the networking methods of a simple link-state system.

The good news is, assuming that there are either very few, or a large number, of phones within a given area, and knowing that bluetooth has connection limits, the iPhone Bluetooth API seems to be intelligent enough to not allow connections over its limit, meaning that the network somewhat self-organizes. If one phone cannot connect to another due to the connection limit, it will simply seek out other phones that have not yet reached their limit.

The bad news is that there seemingly is no redundancy implemented into the Bonjour over Bluetooth system. If a node A is connected to another node C through yet another node B, and node B disconnects, whether A and C are within range or not, C loses its connection to the session.

One additional issue that arises by using Apple's Bluetooth API is the ran-

dom creation of separate disjoint subnets, when what we really prefer is a relatively wide-spread connected mesh (for optimal reachability without having to resort to forwarding packets over the secondary medium, namely WiFi or the cellular data network).

In later versions of AirRAID, we expect to implement a controller that, reachable by the internet, is able to forward traffic intelligently from one subnet to another (see section 7.2.1)

6.3.3 Overriding Apple Bluetooth Mesh Networking - ARGK Overlay

In order to overcome the automatic mesh networking that Apple attempts in its implementation of Bluetooth, we needed to find a way to prevent multiple hops at the Bonjour layer. To do so, rather than using the provided GKSession object to maintain connections, we have extended it and created a new kind of session object (named for the moment a ARGKSession) which presents a new model for handling connections.

Rather than having a Bluetooth ID, each node is identified on the AirRAID network by a ARGK_ID, a lengthy hexademical string unique to each device. In addition, we initialize multiple GKSessions, each with a different service name (i.e. AirRAID-1, AirRAID-2...), equating each session to a 'spoke' on a metaphorical bicycle wheel with the node at its center.

Each spoke is programmed to allow only 1 connection at a time, preventing nodes from sharing neighbor lists with each other (and hence preventing Apple from utilizing its multihop algorithm).

When a data packet for a node (addressed at the packet level by ARGK_ID, and at the bluetooth level by a Bluetooth id) comes in to any other node, the ARGK overlay notifies the AirRAID Communication Layer that a packet for the given ARGK_ID has arrived.

The communication layer then does its work to determine to where the packet should be forwarded next, and notifies the ARGK layer that it wishes

```
- (void)session:(GKSession *)session peer:(NSString *)btID
        didChangeState:(GKPeerConnectionState)state {
  // ...

  switch (state) {
    // ...

    case GKPeerStateConnected:
      // ...
      [session setAvailable:NO];
    case GKPeerStateDisconnected:
      // ...
      [session setAvailable:YES];
  // ...
}
```

Figure 6.3: Sample code for intentionally limiting the iPhone Bluetooth API. Assuming correct mutex handling, this code limits sessions (spokes) to one connection. Intentionally left out is further bookkeeping code from the ARGKSession module

to forward the packet onward to another node (identified, again, by ARGK id). Here, the abstraction becomes slightly taxing in processor cycles. In order to forward a packet on to the next node, the ARGK layer must decide to which bluetooth id and on which spoke it will send the packet.

During initial connection negotiation, the ARGK layer creates and maintains a bidirectional list between Bluetooth IDs and ARGK_IDs, but remember that while, when given a Bluetooth ID, there exists but one ARGK_ID, when given an ARGK_ID, it is possible that there are multiple Bluetooth IDs for a single ARGK_ID (one Bluetooth ID per open GKSession 'spoke' at each node). As such, when a packet is to be forwarded to a node D, the ARGK layer must search for each of D's Bluetooth IDs on each of its spokes, and once it finds it, may forward the packet onward.

6.4 Experimental Run

While the AirRAID module has no visual user interface associated with it, we coded up a simple debug console and map for use with AirRAID tests. The user interface consists of 6 labeled (albeit cryptically) buttons on the limited space at bottom of the screen:

1. "RP" - Request Path - Allows for a user to specify a specific node to request a path to
2. "HB" - Heartbeat - Performs maintenance on all paths (typically run periodically, but also on demand here)
3. "WIFI" - Request Path to WiFi - Sends out a Path Request to all nodes looking for a WiFi connection
4. "T1" - Request to download a remote web-hosted file of approximately 50 kilobytes through the AirRAID layer
5. "T2" - Request to download a remote web-hosted file of approximately 1.2 MB through the AirRAID layer
6. "PA" - Show Paths - Shows general status of the AirRAID and ARGK Layers

In general, our tests were set up such that a user could request to download a remote file off of a webserver by accessing WiFi through the AirRAID layer. To do so, the user of a local node first presses "WIFI", allocating a wireless path. Once the AirRAID layer notifies the UI that a path has been found, the user may click on the "T1" button, which issues a request for the remote file over the path on AirRAID. On a remote node with WiFi, hearing the request, the node allocates a reverse path back to the requesting node, and then issues the web request on WiFi. As return packets from WiFi come in, they are forwarded over AirRAID to the original requesting node. When the transfer finishes, the

remote node sends a simple signal "DONE", telling the local node that it was finished sending packets for the moment.

Meanwhile, as a metric, the local node starts a timer when the user presses "T1", and stops when the "DONE" packet comes in. From this time, approximate transfer rate is calculated and shown in the debug console. Tests were run in only two arrangements, due to budgetary constraints. Results are discussed below.

6.4.1 Complications during Testing

Apple's Bluetooth API, not being perfect, suffers from numerous disconnects, as well as a seemingly extremely finicky Bluetooth chip. At times, despite allocating a number of sessions on a given node, Bluetooth would not activate unless one restarted the node. At times, these sorts of failures occurred during testing itself, causing disconnects and irrecoverable connection/path failures.

The tests described below are the successful tests that were recorded. Ideally, for a deployable release of AirRAID, we would find a way to circumvent the Apple Bonjour over Bluetooth API and hopefully have tighter control over the Bluetooth chipset.

6.4.2 Single-hop Tests

Single hop tests typically consisted of two iPod touches, one with wireless networking and Bluetooth enabled, and the other with only Bluetooth enabled. We placed a single file of approximately 50 kilobytes on a remote webserver (hosted at MIT), and launched AirRAID on both devices.

Once connections and HELLOs were exchanged, we ran the same test but from the two separate nodes. Acting as a control experiment, we do the above steps on the node that already has WiFi access, recording the bit rate without using AirRAID. From there, we run the test on the Bluetooth-only node, and record the bit rate. We'd imagine a significant hit on bit rate, due to the latency behind allocating paths, as well as forwarding the traffic through the Bluetooth

layer.

However, for the most part, with only two nodes in the AirRAID constellation, we observed little to no performance hit on bitrate on the Bluetooth-only node versus the WiFi-enabled node. Working with a slow connection in the first place, accessing the 50 kB file from the WiFi-enabled node garnered for one test a 12 kB/sec rate, while the Bluetooth-only node maintained an 11 kB/sec rate.

6.4.3 Multi-hop Tests

Multi-hop tests of the AirRAID system involved a small mesh of 3 nodes: two iPod touches and one iPhone. For initial tests of multi-hop wireless access, each node was set to allow for no more than 2 spokes to be instantiated at the ARGK layer, meaning that, typically, the network would arrange itself into a line.

Much like above, we ran two tests, one from the WiFi-enabled node, and the other from furthest Bluetooth-only node. On one particular test, we observed 12 kB / sec for the WiFi-enabled node, and 6 kB/sec on the Bluetooth-only node, a 50% hit on bandwidth.

6.4.4 Discussion

A 50% hit on bandwidth is significant, but remember that at this juncture, our goal is not to maximize throughput, but to extend coverage of a wireless network further using a secondary radio medium. Any reasonable bit rate is a successful test in our eyes.

6.4.5 Further tests

Ideally, with a larger number of nodes, it would be interesting to test the path backup and tightening methods to be sure that little to no packet loss occurred.

6.5 Simulation

For the purposes of evaluation, rather than simply trust results from a live simulation, we decided to implement a test simulation, using a simulated bluetooth net, including simulated noise: dropped packets, and random disconnects – all with the hope of determining the efficacy of the redundancy of the algorithm. We choose to measure throughput, as well as the dropped packet percentage, and comparing the results with existing protocols (including Apple's Bonjour + Bluetooth protocol, native to the iPhone platform).

6.5.1 Simulation Code Structure

With a nod to the modularity of the implementation described above, implementing simulation code simply meant replacing all existing communication modules below the Communication Layer level.

Replacing the ARGKSession object with a simulation version (simply disconnecting the module from any reliance on Bluetooth hardware), as well as defining a Bluetooth simulation network on which the new ARGKSimulationSessions can communicate, for the most part, a complete simulation could use all of the original code used in the live demonstration.

We introduce the concept of a 'SimulationNode', or a simulated user device, positioned somewhere in a cartesian plane. Phones are considered in range if there distance is below a certain threshold. Packets are transmitted only to those phones that are within range, and are transmitted successfully with a predefined drop-rate of α.

In addition, small modifications were made to the Communication Layer, preventing it, again, from using actual bluetooth hardware.

6.5.2 Testing Method

While we were unable to gather data from simulation runs, we provide a description of the testing method for the reader to duplicate should s/he wish

to expand upon our implementation: Running on the iPhone simulator, the AirRAID simulation constructs 10 artificial phones (SimulationNodes), and drops them onto a 2D plane. Each running in its own thread, the phones begin to look for each other, sending packets as they see fit for AirRAID infrastructure, as well as moving randomly but not necessarily discontinuously.

After a brief delay, the first node created is told to send approximately 1 MB of data to the furthest node it can find.

Calculating throughput would be relatively trivial, whereas on the other hand, calculating the number of dropped packets is not. Rather than explicitly count each dropped packet, we recommend counting the number of path status packets transmitted, including backup control messages, as well as any packets that imply that a path has suffered a failure (PATH_FAILED packets).

Chapter 7

Future Implementations of AirRAID

Future implementations of AirRAID will focus on implementing additional functionality in terms of connectability, especially across disparate subnets. Much like the existing cell infrastructure, it may be possible that there exists a need for a node on one AirRAID subnet to contact another directly. In addition, we expect to implement even more reliable modes of transfer over the existing routing infrastructure.

7.1 Other Platforms

The iPhone has proved to be a relatively closed medium for presenting the first iteration of AirRAID. Potentially, as the algorithm exists for extending one radio communication schema over another, we have limitless venues for implementation. Sticking to Bluetooth and WiFi, moving an implementation to laptops could prove useful; however, most laptop users tend to be rather statically placed, thus the mesh topology is not dynamic enough to warrant DSR or any algorithm that would be robust against changes in network topology.

AirRAID may work well in other dual-band situations, such as in inter-

satellite communication, where potentially a signal from Earth to a single geosynchronous satellite may be propagated to another non-geosynchronous satellite through an AirRAID mesh.

7.2 Improvements on AirRAID itself

7.2.1 Implementing an Internet-based Traffic Controller

In order for subnet-to-subnet communication to function, much like the existing cell infrastructure, there must be a third party that is reachable by all nodes, able to locate a certain node amidst a number of mesh subnets. Future versions of AirRAID could make use of a traffic controller server (TCS) that sits on the internet, which keeps track of which subnet each node is on. When a node joins a given subnet, and sends its HELLO_PACKET to its neighbor, assuming internet access somewhere in the subnet, each neighbor could report its sighting of the node on its subnet to the TCS. When a node from another mesh subnet wishes to find this node, it can contact the TCS, asking for the IP of a WiFi-enabled node on the AirRAID subnet. PATH_REQUESTS and other packets would then be allowed to transfer from the AirRAID network over to IP, using WiFi-enabled nodes as gateways.

7.2.2 Implementing TCP/IP over AirRAID

Despite our backup and path tightening methods implemented in AirRAID, there is still a certain amount of packet loss that can occur. Future versions of AirRAID could implement TCP/IP streams over the existing AirRAID infrastructure, ACK packets, sequence numbers and transmission windows included. Rather than deal with AirRAID directly, users could open up TCP/IP streams between themselves and their targets, establishing additional reliability.

7.2.3 Multiple Backups

Future versions of AirRAID could potentially and easily maintain a larger list of backup nodes in case of path failure. As nodes fail, the communication layer could remove and use backup nodes from a priority queue keyed on any heuristic of reliability (potentially, round trip times, if that becomes available for use in later versions of the iPhone SDK).

7.2.4 Data Striping

Original goals of the AirRAID protocol involved using the mesh for more than just data transfer reliability. Included on the idealist list of features was the option to stripe data, or to split data transfers across multiple paths for faster overall transfers. Once an implementation of AirRAID has become secure in providing reliable communication over a mesh, we intend to focus on implementing an option to stripe data across disjoint paths in the mesh. Potential applications include video conferencing, which by itself can require not only an amount of bandwidth large enough to warrant striping, but also timely delivery of unreliable packets, which can potentially be better guaranteed by multiple streams on disjoint paths.

7.3 Conclusion

As the world becomes increasingly mobile, the need to extend communication connectivity scales accordingly. More and more focus is being placed on mobile computing, and yet the idea of a constant mobile connection to the internet and/or other devices has yet to become part of our vernacular. Devices such as the Kindle, smart phones, and other mobile-enabled devices are off to a good start with respect to providing connectivity to users who are, themselves, becoming increasingly mobile.

The AirRAID protocol serves as a method to strengthen the mobile communication infrastructure, acting as an adapter between two media. By using

a short range communication protocol with high reliability, connected via the AirRAID protocol to a (potentially) low reliability long range communication protocol, we can take advantage of the additive power of multiple mobile devices to provide a highly reliable and far-reaching mobile communication solution.

While the iPhone seemed an ideal platform for AirRAID's debut, unfortunately, the amount of developer control over its functionality is simply not enough to provide an ideal implementation of the protocol. However, as an initial testbed, the implementation on the iPhone serves its purpose in showing that the protocol would function reliably and provide a certain level of added mesh-network functionality fueled by an already robust 3G or WiFi data connection.

Whether the AirRAID protocol is adoptable into a communications standard of mobile devices remains to be seen, but its theoretical as well as limited measured efficacy makes it a good potential candidate for future cellular communication schema.

Bibliography

[1] Benjamin A. Chambers. The grid roofnet: a rooftop ad hoc wireless network. Master's thesis, Massachusetts Institute of Technology, 2002.

[2] Campbell et al. Design, implementation and evaluation of cellular ip. *IEEE Personal Communications, August 2000*, 2000.

[3] Josh Broch et al. A performance comparison of multi-hop wireless ad hoc network routing protocols. Computer Science Department, Carnegie Mellon University.

[4] Silke Feldmann et al. Bluetooth-based positioning system: concept, implementation and experimental evaluation.

[5] Albert S. Huang and Larry Rudolph. *Bluetooth Essentials for Programmers.* Cambridge University Press, 2007.

[6] Apple Inc. iPhone Dev Center. http://developer.apple.com/iphone/.

[7] David B Johnson. Routing in ad hoc networks of mobile hosts. In *Proceedings of the Workshop on Mobile Computing Systems and Applications.*

[8] David A. Maltz. *On-Demand Routing in Multi-hop Wireless Mobile Ad Hoc Networks.* PhD thesis, Carnegie Mellon University, 2001.

[9] Meraki. http://www.meraki.com.

[10] Stanford University. CS193p - iPhone Application Development. http://www.stanford.edu/class/cs193p/.

[11] Xiang, W., Richardson, P., and Guo, J. An overview of wireless access for vehicular environments technology. *IEEE Transactions on Vehicle Technology.*

[12] Ryan Woodings, Derek Joos, Trevor Clifton, and Charles D. Knutson. Rapid heterogeneous connection establishment: Accelerating bluetooth inquiry using IrDA.

[13] Hongyi Wu and Chunming Qiao. Modeling icar via multi-dimensional markov chains. *Mobile Networks and Applications*, 2003.