Design and Implementation of the Flexible Internetwork Stack (FINS) Framework

Jonathan M. Reed

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

Allen B. MacKenzie, Chair

Luiz A. DaSilva

Godmar Back

May 7, 2014

Blacksburg, Virginia

Keywords: Network Programming, Network Protocols, Mobile Ad hoc Networks, Mobile Computing.

Copyright 2014, Jonathan M. Reed

Design and Implementation of the Flexible Internetwork Stack (FINS) Framework

Jonathan M. Reed

(ABSTRACT)

This thesis describes the Flexible Internetwork Stack (FINS) Framework, an open-source tool to facilitate experimental research in wireless networks on multiple platforms. The FINS Framework uses a module-based architecture that allows cross-layer behavior and runtime reconfiguration of the protocol stack. Version 1.0 of the framework makes use of existing physical and data link layer functionality, while enabling modifications to the stack at the network layer and above, or even the implementation of a clean-slate, non-layered protocol architecture. Protocols, stubs for communicating with intact layers, and management and supervisory functions are implemented as FINS Framework modules, interconnected by a central switch. This thesis describes the FINS Framework architecture, presents an initial assessment along with experiments on Android and Ubuntu enabled by the tool, and documents an intuitive mechanism for transparently intercepting socket calls that maintains efficiency and flexibility.

This material is based upon work supported by the National Science Foundation under Grant Nos. 0916300 (Virginia Tech).

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Allen B. MacKenzie, for guiding and supporting me during my graduate study at Virginia Tech and throughout the course of this thesis research. I am grateful to Dr. MacKenzie for the opportunity to work with him in his lab, as well as for his mentoring and advice shown over these last three years.

I would also like to thank my other committee members, Dr. Luiz A. DaSilva and Dr. Godmar Back, for serving in my committee and for their time and co-operation in reviewing this work. Special thanks to Dr. DaSilva for his continuing advice and support over the lifetime of this project.

I would like to thank the fellow students in the Networking Lab for their companionship and guidance, especially Ph.D candidate Abdallah S. Abdallah. Abdallah was one of the founding members of this project and helped me greatly as I familiarized myself with the field and this research. Furthermore, he has provided important discourse and constructive suggestions throughout the implementation of this research. I wish him luck with his own research and the continuation of this project.

Finally, special thanks to my family for their loving support and encouragement, without whom none of this would have been possible.

Contents

1 Introduction **1**

1.1 Background and Motivations 2

1.2 Contributions 4

1.3 Outline 6

2 Review of Experimental Wireless Network Tools **7**

2.1 Click Modular Router 8

2.2 Ad Hoc Protocol Evaluation Testbed 9

2.3 Monitor for Mobile Ad hoc Networks 10

2.4 Autonomic Network Architecture 12

2.5 X-Layer 13

2.6 Implementing Radio In Software 14

2.7 Emstar 16

2.8 WiFu 18

2.9 Open Access Research Testbed for Next-Generation Wireless Networks . . . 19

2.10 Conclusions 21

3 Architecture, Design, and Implementation 24

3.1 System Architecture 24

3.1.1 Structure 25

3.1.2 Modules 26

3.1.3 FINS Framework Frames 27

3.1.4 Switch and Linking Table 29

3.2 FINS Framework Version 1.0 : Design and Implementation 30

3.2.1 FINS Framework Core Process 32

3.2.2 FINS Framework Socket Stub Module 35

3.2.3 FINS Framework MAC/PHY Stub Module 40

3.2.4 FINS Framework RTM module and FINS Framework Console 41

3.2.5 Traffic Flow Walkthrough 43

4 **Results and Discussion** **47**
4.1 Performance Evaluation . . . 47
4.1.1 Experiment 1 . . . 48
4.1.2 Experiment 2 . . . 50
4.1.3 Forward for Experiments 3-4 . . . 52
4.1.4 Experiment 3 . . . 52
4.1.5 Experiment 4 . . . 55
4.1.6 Conclusions . . . 56
4.2 Experimental Scenarios . . . 57
4.2.1 Simple Scenarios . . . 57
4.2.2 Complex Scenarios . . . 58
4.2.3 Conclusions . . . 59
5 **Conclusion** **60**
5.1 Summary of Work and Contributions . . . 60
5.2 Future Work and Closing . . . 62
Bibliography **65**

List of Figures

2.1 The overall flow of the MMAN system, H. Kazemi, G. C. Hadjichristo, and L. A. DaSilva, "MMAN - a monitor for mobile ad hoc networks: Design, implementation and experimental evaluation," in *Proceedings of the Third ACM International Workshop on Wireless Network Testbeds, Experimental evaluation and CHaracterization (WiNTECH)*, 2008, Used under fair use, 2014. 11

2.2 Proposed architecture with interlayer resource broker, S. P. Aaron Beach, Mike Gartrell and R. Han, "X-Layer: An experimental implementation of a cross-layer network protocol stack for wireless sensor networks," Department of Computer Science University of Colorado at Boulder, Tech. Rep., December 2008, Used under fair use, 2014. 14

2.3 Architecture of Iris, "IRIS (CTVR, Trinity College Dublin)," April 2014. [Online]. Available: http://www.crew-project.eu/iris, Used under fair use, 2014. 15

2.4 Decomposition of the Emstar framework into layers and interfaces, L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks," *ACM Trans. Sen. Netw.*, vol. 3, no. 3, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1267060.1267061, Used under fair use, 2014. 17

2.5 WiFu Transport architecture, R. Buck, R. Lee, P. Lundrigan, and D. Zappala, "WiFu: A Composable Toolkit for Experimental Wireless Transport Protocols," in *IEEE MASS 2012, the 9th IEEE International Conference on Mobile Ad hoc and Sensor Systems*, 2012, pp. 299-307, Used under fair use, 2014. . . 19

3.1 Comparison of architecture between stacks. 25

3.2 Structure of the FINS frames. 28

3.3 FINS Framework implementation and flow of data 31

3.4 Steps of example interaction between the MAC/PHY stub & ARP modules. 35

3.5 Implementation and data flow for the FINS Framework Socket Stub Module 37

3.6 UDP/IP example stack using the FINS Framework 42

4.1 The observed throughput through the socket stub module normalized by the data rate attempted by iperf. 49

4.2 The packet loss rate through the MAC/PHY stub module at varying data rates. The traffic was sent over the local loopback of a laptop to reduce packet loss from channel effects. 51

4.3 Experimental setup for Experiment 3. Note the blue arrow represents a 100 Mbps LAN cable connecting the nodes. The black arrow represents the flow of UDP datagrams or TCP data segments. 53

4.4 Experimental setup for Experiment 4. The test was conducted at Virginia Tech using tablets in a setup similar to that of Experiment 3 (Fig. 4.3). Depicted in the figure is an iperf client on top of a FINS Framework (blue) sending over Wi-Fi through an IEEE 802.11n access point (black) to an iperf server on top of the traditional stack (red). The black arrow indicates the flow of iperf traffic in UDP datagrams or TCP data segments. 55

List of Tables

3.1 A list of possible Operation Codes and their use. 29

3.2 An example of configuration records from the FINS Framework linking table 36

4.1 UDP Results for Experiment 3 (laptops, LAN) and Experiment 4 (tablets, Wi-Fi). 54

4.2 TCP Results for Experiment 3 (laptops, LAN) and Experiment 4 (tablets, Wi-Fi). 54

Chapter 1

Introduction

The widespread rise of tablets, mobile devices, and new mobile network architectures incorporating heterogeneous technologies has encouraged a resurgence in research on device-to-device communications and mobile ad hoc networks (MANETs). Furthermore, the availability, small form factor, and long battery life of mobile devices enable the evaluation of protocols in truly mobile experiments. Despite this, the vast majority of MANET research still relies primarily on simulation, with only a fraction of research studies employing implementation-based experiments. Wireless networking research has arguably relied too much on simulation, which has serious limitations regarding fidelity to real-world conditions [1–5]. While simulation-based studies are useful in developing and testing ideas, we believe they should not be a terminal step in the research process. In this thesis, we introduce a new software tool for wireless research that aims to reduce the barriers researchers encounter with wireless experiments and enable rapid wireless network prototyping, experi-

mental protocol evaluation, and complex cross-layer behavior. This thesis will also describe a mechanism used to accomplish this that exploits the nature of the traditional stack to efficiently intercept socket calls and seamlessly support applications.

1.1 Background and Motivations

Researchers attempting to conduct experimental studies in wireless networking encounter logistical, evaluation, and implementation challenges.

- Logistical challenges include the need for significant quantities of people, equipment, and space.
- Evaluation challenges involve the amount of repeatability and standard benchmarking scenarios.
- Implementation challenges include the availability of software tools and the cost of protocol prototyping and implementation.

For simulation-based studies the logistical and evaluation challenges are typically negligible as simulations do not require a physical environment or the need to use actual devices. This allows for low-cost testing, high scalability in scenarios, and a controllable and flexible environment, which lessens the burden placed on researchers and expedites the testing of research questions. High fidelity simulators like ns-2 [6], GloMoSim [7], and OMNeT++ [8] offer the ability to easily and quickly evaluate protocols with high repeatability of results.

However, the lack of a physical environment also opens up simulation to common user error (typically bad simulation practices or poor model assumptions [2]) and problems when the software is not sophisticated enough to accurately reflect complex device implementations, protocols, or scenarios, such as the inaccuracy observed in multi-hop simulations [4,5]. While simulation is certainly a powerful tool that can be useful for sufficiently simple scenarios, it is limited to only providing estimates for more complex situations, especially in the case of MANETs [9].

Emulation tools have been increasingly used to mitigate many of the problems simulation exhibits as their use of real network stacks and hardware in combination with simulators provide higher fidelity. Tools such as ns-3-click [10], ORBIT [11], or MeshTest [12] add a degree of control over the network to retain the low logistical and evaluation costs of simulation while increasing fidelity to real conditions. This is commonly accomplished by simulating lower layers, miniaturizing the network, or causing predictable mobility patterns for mobile nodes such that the ease of deployment, scale, and repeatability for emulators is similar to that of simulations. However, emulation also inherits some of the disadvantages of simulation environments [13] as the methods of adding control inherently decrease emulation's ability to faithfully capture physical and MAC layer characteristics observed in larger networks [9].

This leads researchers who require high accuracy to rely on experiments and experimental wireless network tools, since they evaluate protocols and scenarios in real-world conditions using the actual target platforms. In doing so, implementation-based studies avoid issues related to imperfect modeling at the cost of being harder to prototype, deploy, and test. How-

ever, in the case of MANETs, mobile devices significantly reduce logistical challenges related to preparing, using, and powering equipment. Furthermore, this thesis introduces a new experimental wireless network tool, called the Flexible InterNetwork Stack (FINS) Framework, with the goal of addressing evaluation and implementation challenges in an effort to make experimental wireless networking research easier to conduct. The FINS Framework was inspired by challenges encountered during the MANIAC Challenge project [14], challenges which plague most experimental research in wireless networking.

1.2 Contributions

The first contribution of this thesis is the introduction of a tool, the FINS Framework, that enables wireless networking experimentation from the data link to the application layer, with particular support for new scenarios involving cross-layer behavior and context-aware applications. The FINS Framework is an open-source, flexible networking subsystem meant to reduce evaluation and implementation barriers in order to encourage experimental research. Implementation costs are lessened by moving the rigid traditional network stack into user-space as a module-based architecture to enable fast prototyping of new protocols and additional cross-layer interactions. Other tools have similarly implemented user-space stacks to avoid the high cost of debugging and modifying the kernel; however, they still strictly adhere to the layered architecture of the traditional stack, making research in alternative architectures and cross-layer interactions difficult. The FINS Framework distinguishes itself

through a flat architecture of generically interfaced modules to allow high reconfigurability and complex cross-layer behavior within the protocol stack. Evaluation challenges are addressed through transparent logging enabled by the architecture, management or supervisory modules, and backwards compatibility with existing applications without modification. In addition, the FINS Framework was created with deployment to mobile devices in mind, such that researchers may capitalize on the logistical benefits of mobile devices by developing protocols on laptops and then effortlessly deploying them to a tablet or handheld device for testing. The framework can support existing networking applications in typical use cases at connectivity speeds up to IEEE 802.11a/b/g.

The second contribution is a mechanism for backwards compatibility that exploits the nature of the traditional network stack to efficiently intercept socket calls and seamlessly support applications. Interfacing with applications and providing support for the BSD socket API is complicated as this typically requires either recompiling an application or dynamically interposing system calls. This can limit the available test software or cause problems when encountering statically linked libraries. We present a mechanism that links with the kernel through the network subsystem such that it avoids needing to alter applications and automatically intercepts all networking system calls, while also avoiding re-implementing file- and socket-related functionality provided by the kernel. By intercepting system calls in the kernel and shuttling them to user-space for processing, this mechanism is similar to the FUSE operating system mechanism [15]–a tool that enables virtual file systems–and allows for custom user-space implementations of sockets. In addition, our mechanism avoids many

of the tradeoffs encountered by other interception methods, while retaining flexibility and code reusability.

1.3 Outline

This thesis is partitioned into five chapters. The next chapter, Chapter 2, surveys existing tools used for experimental research in wireless networking, discusses advantages and disadvantages, and lists desirable characteristics. In Chapter 3, we describe the architecture, design, and implementation of the FINS Framework and show how the framework achieves the desired goals. Chapter 4 presents performance results of the FINS Framework in a simple IEEE 802.3-based network use scenario on laptops and a simple IEEE 802.11-based network on mobile devices, followed by discussion about a workshop and the possible experimental scenarios presented using the framework. This thesis concludes with Chapter 5, which summarizes our work and suggests new areas for future work.

Chapter 2

Review of Experimental Wireless Network Tools

In this chapter we will review existing tools in the field through their context and purpose. For this thesis, *experimental wireless network tools* refers to tools used to perform or collect data during an experiment in a wireless network where the experiment components are implemented or prototyped using the actual target platforms. More detail on classes and categories for experimental wireless network tools can be found in [16]. For tools with emulation components, we use the common classifications of *physical layer emulators* and *MAC layer emulators*, based on up to which layer is simulated through software [1].

2.1 Click Modular Router

Originally released in 2000 for modular forwarding paths, the *Click Modular Router* [17] is a software framework that has been widely embraced by the networking community and expanded to include protocol development and testing. The Click Router creates a virtual platform-independent layer to run on top of the traditional stack, sacrificing some performance for the sake of a unified implementation regardless of the platform [18,19]. The modularity and code reusability of the tool has led to its adoption by many researchers; however, reconfigurability of the virtual stack is limited during runtime and the Click Router does not directly support the BSD socket interface, instead intercepting and augmenting outgoing network traffic of the traditional stack through the local loopback interface. These nuances introduce issues when evaluating experimental protocols with traffic from context-aware and real-time applications as the Click Router is limited by the traditional stack.

Support from the community has expanded to even include emulation as the *ns-3-click* [10, 20] project creates a MAC layer emulator by integrating the Click Router into the *ns-3* [21] simulator. Generally considered one of the most popular discrete event network simulators, *ns-3* [21] is the third major revision of an open-source project initially created for wired networks in 1989 that has grown in documentation, support, and community to cover a wide range of protocols and networks, including wireless networks and MANETs. ns-3 is written in C++ and Python and improves much of the functionality of its predecessor, such as redesigning the core for scalable expansion, providing graphical user interfaces (GUIs) for

creating scenarios and statistical analysis, and supporting integration with real devices [22]. Connecting the Click Router with ns-3 allows researchers to sanity test experimental protocols in a virtual environment before conducting experiments with a physical MANET. Drawbacks include a limit on the fidelity due to simulating lower layers, increased memory consumption, and that the community surrounding ns-3 (hence ns-3-click) is still maturing.

2.2 Ad Hoc Protocol Evaluation Testbed

The *Ad Hoc Protocol Evaluation Testbed* (APE) [23] is a framework designed to achieve test repeatability and result reproducibility by choreographing experiments through a distributed software package of build scripts and source code. In operation, participants carry around laptops with the package pre-installed while following directions and entering in commands assigned in movement scenario files. These movement scenario files are processed by the framework in order to control traffic generators, routing protocols, and inform participants when and where they should move during the scenario [24]. After an experiment all of the collected data can be congregated and analyzed using tools written in Perl that are provided by APE. The framework also provides a visualization tool called *APE-view* that processes experiment logs to animate the topological configuration of nodes during test runs.

The creators of APE further reduce evaluation challenges by providing a build system to create customized *APE distribution packages* that self-install encapsulated execution environments (essentially small Linux distributions) with all the necessary drivers, tools, and

scripts. In effect, APE automates the deployment and execution of experiments to the point that the participants simply need to move and follow directions. This approach greatly reduces the work for the both organizers and participants of a wireless experiment, but also introduces minor drawbacks in that the encapsulated execution environments are tuned to operate on specific platforms with special drivers modified for APE. This increases the implementation costs when researchers desire to use the framework with other hardware, such as handheld or mobile devices. In addition, minimalistic environments make it difficult to troubleshoot problems during the experiment and may be unstable if built incorrectly. It is important to note that APE does enable experimental protocol design and has officially added support for several protocols since its release in 2002. Unfortunately, its biggest drawback is that experimentation has been limited to only routing protocols with little to no development of protocols in other layers.

2.3 Monitor for Mobile Ad hoc Networks

Intended to reduce evaluation challenges, the *Monitor for Mobile Ad hoc Networks* (MMAN) [25] framework is a distributed software package for unobtrusively monitoring MANET experiments. The framework is intended to be deployed to a set of passive nodes, called Monitoring Units (MUs), that observe wireless network traffic during an experiment and then aggregate the data over a wired connection. In using passive nodes that communicate over a separate wired network, MMAN avoids taxing the wireless network and interfering with the exper-

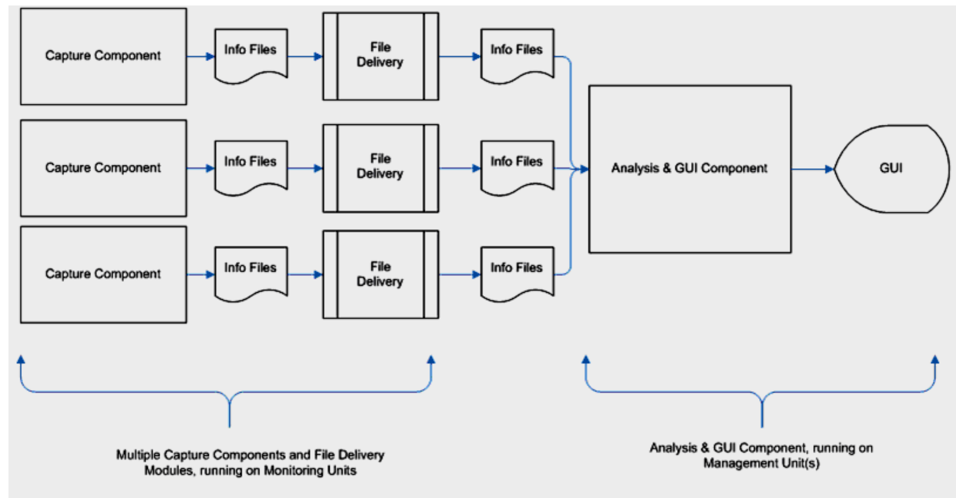


Figure 2.1: The overall flow of the MMAN system, H. Kazemi, G. C. Hadjichristo, and L. A. DaSilva, "MMAN - a monitor for mobile ad hoc networks: Design, implementation and experimental evaluation," in *Proceedings of the Third ACM International Workshop on Wireless Network Testbeds, Experimental evaluation and CHaracterization (WiNTECH)*, 2008, Used under fair use, 2014.

iment. The framework is separated into three major independent components: a *Capture* component, a *File Delivery* component, and an *Analysis & GUI* component 2.1. Each MU collects wireless traffic for an area through the Capture component using JPCAP [26] into a *info file* which is then delivered to a central node using the File Delivery component for processing or visual presentation (Analysis & GUI component).

The separation between components ensures minimal overhead or loss of traffic during the experiment, with more complex and resource intensive processing able to be completed afterwards to create network-level and node-level snapshots. The GUI provided by MMAN is able to display the dynamic network topology and node connectivity observed in the

experiment (network-level) as well as traffic information and an assessment of cooperation in forwarding packets at a node-level. The MMAN framework offers a simple and efficient tool for collecting and analysing data in wireless networking experiments; however, does not extend beyond that core functionality.

2.4 Autonomic Network Architecture

The *Autonomic Network Architecture* (ANA) [27] project incorporates the trend of virtualization and attempts to foster self-learning, self-configurable, and self-forming networks by replacing the traditional stack and adding a common presentation layer. This experimental wireless network tool takes inspiration from peer-to-peer file sharing networks, with nodes able to invoke remote services or alternative network resources provided by peer nodes in RPC-like (remote procedure call) access through its presentation layer [28]. Appropriately, ANA supports high flexibility and runtime reconfigurability to the point of multiple network stacks running on top of the same physical node.

This level of flexibility is achieved through a central core called *MINMEX* that modular components (aka *bricks*) attach to in order to provide the necessary functionality. In user-space bricks are compiled as shared library objects and dynamically loaded into the framework, which enables MINMEX to load functionality as needed. A notable feature of ANA is the ability to prototype new components in user-space and then seamlessly test them in kernel-space through a dual build system and ANA wrappers for common functions. For

kernel-space, the bricks are built as loadable kernel modules (LKMs) and use the modularity provided by the Linux kernel to join the components. However, as a clean-slate tool ANA requires researchers to re-implement some protocols (e.g. TCP) which is non trivial and modify existing applications to be compatible with ANA's presentation layer.

2.5 X-Layer

Following a more conventional approach, *X-Layer* [29] is an tool designed for wireless sensor networks (WSNs) that uses a condensed version of the traditional stack, but also allows additional cross-layer behavior. The layered architecture of the stack suits the low-resource environment of WSNs, which prioritizes efficiency over flexibility, and a carefully chosen subset of cross-layer links were incorporated to allow cross-layer protocol design from the physical layer to the application layer. The kernel level modification of the stack shows the efficiency possible through direct augmentation of the traditional stack, but at the cost of minimal flexibility and significant development challenges when coding and debugging. In addition, X-Layer's cross-layer optimization has been shown to be limited as the authors found that simple cross-layer interactions in the layered stack was not sufficient.

To improve the potential cross-layer behavior, the authors also proposed a new architecture (Fig. 2.2) that modified the layered stack to include an *interlayer resource broker*. In receiving cross-layer information from every layer, the resource broker is able to more efficiently share information through a publish/subscribe mechanism that filters all informa-

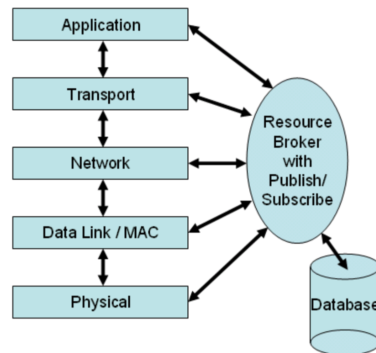


Figure 2.2: Proposed architecture with interlayer resource broker, S. P. Aaron Beach, Mike Gartrell and R. Han, "X-Layer: An experimental implementation of a cross-layer network protocol stack for wireless sensor networks," Department of Computer Science University of Colorado at Boulder, Tech. Rep., December 2008, Used under fair use, 2014.

tion and only relays necessary data. Unfortunately, to our knowledge the authors have not introduced a tool using this architecture and X-Layer is no longer maintained.

2.6 Implementing Radio In Software

Focusing more on the physical and medium access layers, *Implementing radio in software* (Iris) [30, 31] is a flexible framework written in C++ for building reconfigurable cognitive nodes on a software defined radio (SDR). Although the project has heavily focused on issues related to hardware concerns and SDR, the framework does provide a generic design for implementing a whole cognitive network stack. The architecture of Iris (Fig. 2.3) supports a core cognitive processing engine (*radio engine*) which manages the flow of data among a set of functional blocks (*components*), each of which are implemented using the same generic pat-

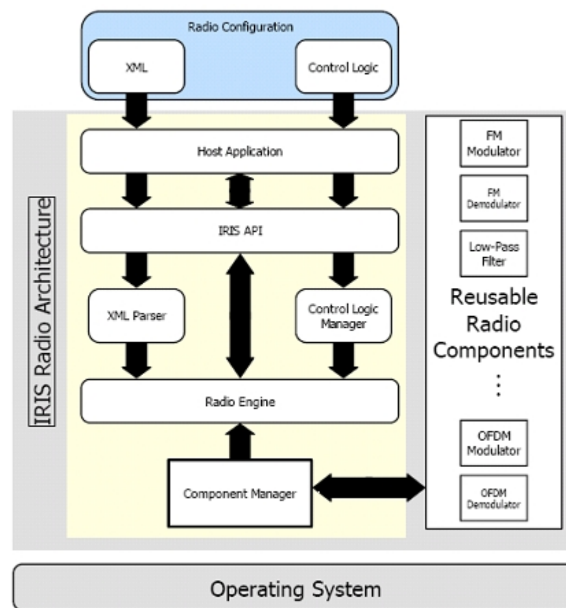


Figure 2.3: Architecture of Iris, "IRIS (CTVR, Trinity College Dublin)," April 2014. [Online]. Available: http://www.crew-project.eu/iris, Used under fair use, 2014.

tern [32]. Events in these components are listened to by controllers (*control logic manager*), which modify parameters in components based on *control logic* set in XML configuration files. Similarly, the inputs, outputs, and parameters of a component and the structure of components in a radio engine are given by XML files (*XML*). In this architecture, protocols would be implemented as components with an engine representing a complete network stack and controllers configured to perform cross-layer or cognitive algorithms. Alternatively, controllers could be used by context-aware applications (*host application*) to observe network conditions, report important contextual information, and augment the engine's behavior accordingly.

Meant for building cognitive radios, a distinguishing characteristic of Iris is its high flex-

ibility and reconfigurability at runtime, such as restructuring of the stack through linking and de-linking components or dynamic reading and modification of parameters within protocols. Initially released in 2006, the framework underwent a major redesign in 2008 that introduced dynamic memory allocation, parallelism support, and modular design for heterogeneous platforms. In addition to the framework, the CTVR (Telecommunications Research Centre, based at University of Dublin, Trinity College) provides researchers with remote access to a testbed of computers and universal software radio peripherals (USRPs) for testing. While Iris provides a large number of components for the physical and media access layers, its main drawback is the complete lack of components for higher layer protocols, which greatly increases implementation costs for network or transport layer research.

2.7 Emstar

Emstar [33] is an software tool designed for the simulation, physical emulation, and deployment of WSNs using microservers. This tool is distinctive as its current implementation uses a multi-process service architecture of libraries and daemon that communicate through device file inter-process communication (IPC). Each service is connected with others through loosely coupled interfaces that are accessible through the file system using FUSD [34], a kernel module that allows device files to be implemented by user-space daemons, which Emstar uses for IPC. These services offer a variety of functionality from the physical layer to the application layer, including link-layer routing protocols, time syncing between neighboring

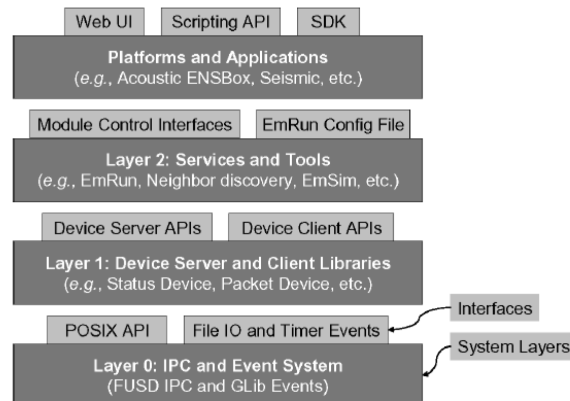


Figure 2.4: Decomposition of the Emstar framework into layers and interfaces, L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks," *ACM Trans. Sen. Netw.*, vol. 3, no. 3, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1267060.1267061, Used under fair use, 2014.

nodes, and a remote resource sharing. Figure 2.4 shows a decomposition of the services by system layer; note that while Emstar provides functionality for data collection from the physical layer and device drivers the framework itself resides entirely in user-space.

Main priorities in the project have been to provide increased system robustness to sensor network applications, as well as fault isolation, fault tolerance, and system visibility. These are provided by the multi-process architecture and visibility inherent to FUSD. Unfortunately, the multi-process architecture and use of IPC adds non trivial overhead that they attempt to mitigate through microservers. In addition, while the services of Emstar provide a wide range of functionality, there are noticeable gaps when compared to the traditional network stack (e.g. TCP) and some services are simply pass throughs to the underlying

platform.

2.8 WiFu

A more recent tool, *WiFu* [35] is an open source user-space toolkit released in 2012 for developing experimental wireless transport protocols that retains high efficiency by building off of the network layer of the traditional stack and connecting to legacy applications through a user-space static library. The toolkit is implemented as *WiFu Transport*, which replaces the transport layer, and *WiFu Core*, a background service that intercepts and modifies packets to enable cross-layer support. The architecture of WiFu Transport uses a central entity called the *Event Dispatcher* to direct data flows and control signals among different functional components [36].

Fast prototyping of TCP flavors is encouraged through a decomposition of TCP into separate libraries for connection management, reliability, and congestion control. Users create protocols that receive traffic from the Event Dispatcher, perform method calls to the appropriate library, and push any traffic back to the Event Dispatcher (Fig. 2.5). WiFu provides high flexibility and efficiency with opportunities to create cross-layer behavior; however, because it only replaces the transport layer it is limited to the existing cross-layer interactions in the traditional stack. Furthermore, in using raw sockets to connect with the traditional stack to send and receive packets, WiFu is unable to expand experimentation to additional layers without considerable implementation challenges. Finally, the implementa-

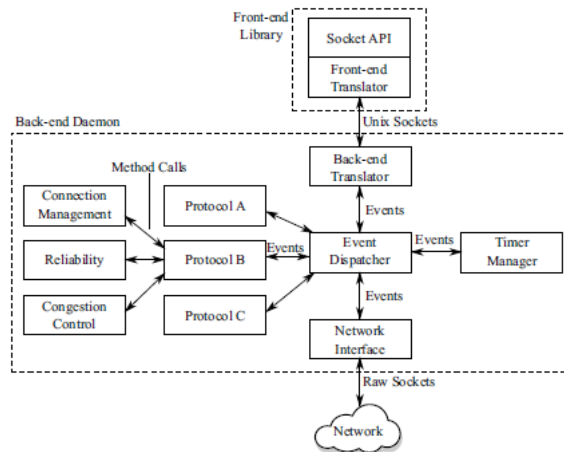


Figure 2.5: WiFu Transport architecture, R. Buck, R. Lee, P. Lundrigan, and D. Zappala, "WiFu: A Composable Toolkit for Experimental Wireless Transport Protocols," in *IEEE MASS 2012, the 9th IEEE International Conference on Mobile Ad hoc and Sensor Systems*, 2012, pp. 299-307, Used under fair use, 2014.

tion of its user-space frontend as a static library requires applications to be recompiled and WiFu Transport currently does not support critical functionality of the BSD socket API (e.g. poll).

2.9 Open Access Research Testbed for Next-Generation Wireless Networks

The *Open Access Research Testbed for Next-Generation Wireless Networks* (ORBIT) [11] is a publicly accessible testbed that provides an indoor radio grid for physical layer emulation and an outdoor field trial network for experiments. Researchers can access a joint gateway

for both, remotely run user code to prototype protocols on the emulator, and then easily transition to the outdoor network for high-speed cellular (3G) and 802.11 wireless research in a real-world setting. The emulator reduces logistical challenges by dynamically mapping virtual nodes in a scenario to physical radios in a grid [37], which are then used to simulate mobility by rebinding virtual nodes to different radios and causing discrete changes in signal power [38]. The stationary radios transmit at low power levels to miniaturize the emulator and allow automated testing of scenarios in a laboratory environment, which ensures greater control over the environment, high repeatability of results, and avoids the costs of implementing a complex analytical model for lower layers. The ORBIT emulator has expanded past the laboratory-based, 20-by-20 main grid to include 9 smaller "sandboxes" with varying hardware and environments (indoor and outdoor) to approach conditions closer to real experiments; however, the use of miniaturization, simulated mobility, and artificial interference limits the fidelity of the tool, prevents emulating complex physical environments, and forces researchers to deeply understand how the emulated environment maps to a real-world one [39].

The outdoor field trial network is comprised of numerous outdoor and vehicular nodes deployed on or around the Rutgers campus. Coverage for the testbed roughly spans a region 5 km wide by 2 km long and contains both urban and suburban areas. Similar to the indoor emulator, devices in the network operate under a framework called OMF (cOntrol Measurement Framework), which allows remote users to configure the network stack of a node, control traffic through the network, and collect statistics during scenarios.

The ORBIT testbed is a part of the global environment for network innovation (GENI) [40], a worldwide collection of computers, emulators, and testbeds integrated together in order to provide researchers more accessibility to networking resources. The outdoor field trial network avoids many of the problems its emulator counterpart experiences as nodes consist of devices that operate under normal conditions in an unmodified environment. This increases fidelity to real-world levels at the cost of repeatability as environmental conditions may affect experiments (weather, traffic, etc). As a testbed, its main drawback is that researchers are limited by the available scenarios provided, which does not include rural areas.

2.10 Conclusions

In reviewing existing experimental wireless network tools, we found many beneficial concepts and characteristics. Almost all of the widely successful tools exhibited high flexibility in components and configurations, the ability to reconfigure the stack at runtime, and code reusability. These factors provide researchers with a range of elements with which to conduct research and enough flexibility to construct whatever scenarios they are interested in. In addition, well used tools typically branched out and integrated building their framework for multiple situations, such as ANA's dual build system or how scenarios for the Click Router are reusable for ns-3-click. We wish to emulate this by incorporating build support for Ubuntu and Android, enabling our users to develop protocols or applications on laptops and then easily deploy them to mobile devices for experiments. To our knowledge none of

these tools have official branches for Android, which we think is a major opportunity for encouraging experimental research in wireless networking.

While each tool had some desirable qualities, we found that all of them lacked sufficient support for cross-layer interactions and context-aware applications. This is in part because most of experimental wireless network tools keep the strict, layered architecture of the traditional stack, which is at odds with the nature of cross-layer optimization–low layer information is typically most useful for upper layers and vice versa. Other tools offered some cross-layer support but were limited to simple interactions between protocols, such as the Click Router where design decisions about control connections between modules limited it's efficacy. Greater support for cross-layer optimization is especially important to MANETs and adaptive algorithms as the time-varying nature of wireless channels permits sporadic usage of the link, which encourages higher efficiency in transmitting information. We wish to enable more of the possible cross-layer designs and flows presented in [41] and [42].

Furthermore, many of the tools had poor support for backwards compatibility, often requiring modifications to any application a researcher might wish to use. While in a tightly controlled scenario this is not as much an issue, it adds additional cost to evaluating protocols and fragments benchmarking scenarios. To assuage this, we desire our tool to have seamless support of the BSD socket API and enable researchers to use common networking applications, tools, and generators.

Finally, we found that the level of logging and managerial functionality varied greatly. Some tools, such as APE, showcased extremely effective methods of documenting scenarios

and reducing evaluation challenges; however, others provided only the most basic functionality, e.g. printing traces. We wish to provide better interactivity with documenting scenarios, but also with applications, such that researchers are able to test context-aware applications with a large amount of control and information about the protocol stack.

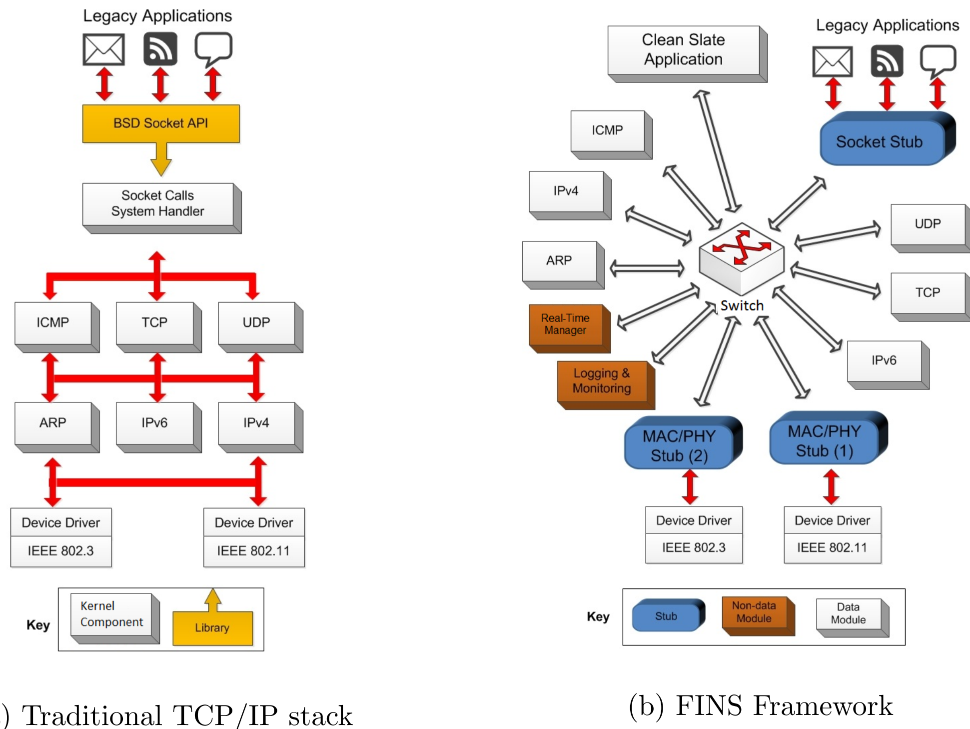
Chapter 3

Architecture, Design, and Implementation

In this section, the architecture, design, and implementation of the FINS Framework is discussed, followed by a step-by-step walkthrough of an example traffic flow through the system.

3.1 System Architecture

The FINS Framework is a hybrid experimental wireless network tool that attempts to break the strict layering seen in a traditional stack (Fig. 3.1a) for a more flexible approach. Fig. 3.1b shows a visual representation of the architecture using the two main types of components: *modules* and *the switch*. Modules are independent, self-contained components that commu-



(a) Traditional TCP/IP stack

(b) FINS Framework

Figure 3.1: Comparison of architecture between stacks.

nicate with each other through the switch, which acts as a multiplexing component. Not shown in the figure but equally important are *FINS frames*, which encapsulate and shuttle data between modules and the switch, and the *linking table*, which specifies how frames flow through the framework.

3.1.1 Structure

The hybrid structure of the FINS Framework builds off of the traditional data-link layer and introduces a flat approach for the network to application layers, while maintaining seamless backward compatibility with legacy network applications. The FINS Framework

thus allows users to avoid the high cost of re-implementing hardware-specific low-level layers and enables experiments that use realistic traffic scenarios, namely real-world applications instead of custom-tailored traffic generators. The non-layered nature of the architecture means any module can communicate with any other module, allowing for greater access to *meters* and *knobs* across layers as well as support for cross-layer protocol design.

Our architecture also allows experimenters to implement applications that are connected directly to the FINS Framework without using the BSD or glibc socket API in a clean-slate fashion (Fig. 3.1b). The ability to implement clean-slate applications allows researchers to explore new experimental scenarios, such as implementing and testing a context-aware application that varies its traffic pattern based on network conditions. The ability to connect directly to the framework not only allows context-aware applications through access to the internal conditions of protocols, but also cognitive applications that can control some protocol parameters or even direct management features within the FINS Framework. This is helpful for researchers working on cognitive nodes and networks [43].

3.1.2 Modules

There are three types of modules in the framework: *data*, *non-data*, and *stub*.

- Data modules act on ingress and egress network traffic as it is processed in the node (e.g. ARP, TCP, UDP).
- Non-data modules observe network traffic and interact with other modules; these mod-

ules may not act on or modify network data (e.g. Logging & Monitoring).

- Stub modules are modules that enable integrating the FINS Framework with existing mechanisms outside of the framework (e.g. Socket Stub, MAC/PHY Stub).

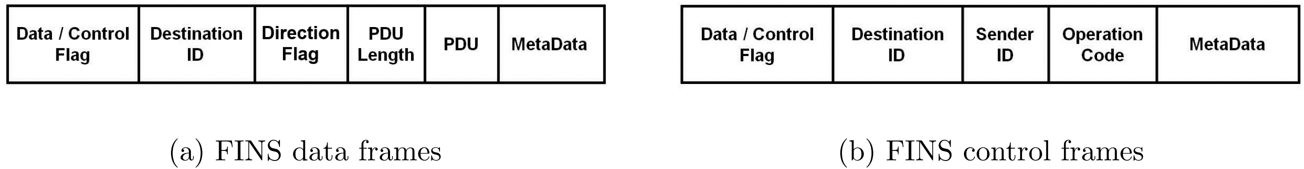
Modules may vary greatly, but must all follow common guidelines and interface with the switch in the same way. They are meant to interact with other modules through loosely coupled generic interfaces and to be implemented at differing levels of granularity, such as at the protocol, algorithm, or library level. This aligns with the concepts of generic interfaces and code reusability mentioned previously.

3.1.3 FINS Framework Frames

Communication between modules is accomplished through two types of frames based on the traffic type: *data frames* and *control frames*.

- Data frames encapsulate ingress or egress network traffic that flows through the framework (Fig. 3.2a).
- Control frames encapsulate messages exchanged between the modules in the framework that do not carry network traffic; these are typically used for management or supervisory functions (Fig. 3.2b).

Data and stub modules are able to send both data and control frames; however, since non-data modules may not act on or modify network data they can only send control frames.



(a) FINS data frames (b) FINS control frames

Figure 3.2: Structure of the FINS frames.

Both data and control frames share a first field used to differentiate frame type, followed by a second field holding the value of the destination ID. Subsequent fields vary depending on frame type and other parameters.

For data frames the direction flag is the third field and is processed internally by protocol modules to determine whether the frame contains ingress or egress network data. This is a helpful feature for researchers who are interested in reusing traditional protocol implementations with minimum modifications. The next field is the length of the protocol data unit (PDU), which is followed by the PDU data field, where the encapsulated data is stored. The last field is metadata, which is used to exchange key-value pairs between modules. The metadata field is also used to collect information recovered from headers as the frame passes through the framework, reducing reprocessing and enabling frame-by-frame logging.

For control frames the third field is the sender ID of the frame, which is necessary since a module's internal processing of the frame depends in part on which module it comes from. The next field is the operation code, which determines what operation is requested of the receiver module (Table 3.1). Once again, the last is field metadata, which contains special key-value pairs that are associated with the sender, operation code, and other factors.

Table 3.1: A list of possible Operation Codes and their use.

Operation Code	Use
Parameter Read Request	Request the value of a module's parameter
Parameter Read Reply	Respond with the value of the module's parameter
Event Listen	Register to be notified when an event occurs in a module
Event Alert	Notify a listening module the event occurred
Parameter Write Request	Request to change the value of a module's parameter
Parameter Write Confirmation	Confirm whether the parameter's value was changed
Procedure Execute Request	Request the execution of a procedure
Procedure Execute Reply	Respond with the result of the procedure
Error Log	Report that a non-termination error has occurred

Control frames expose *meters* for other modules to measure through two modes: *polling mode* and *event-driven mode*. Polling mode is performed using the Parameter Read Request and Parameter Read Reply operations, while the event-driven mode is accomplished using Event Listen and Event Alert. The other half of the concept, *knobs* through which modules can be affected, is realized through Parameter Write Request and Parameter Write Confirmation.

3.1.4 Switch and Linking Table

All of the modules within the framework communicate using frames that pass through the switch, with the path of frames controlled by the linking table. This table, which is used like a routing table, specifies the receiver module(s) of a frame based on the sending module and virtual link it is sent over. This allows both the user and module designer to set and

change how data flows through the system, as both can reconfigure the stack through the linking table. Virtual links are typically differentiated by the type of communication, traffic behavior, and/or direction through the protocol suite (egress or ingress). For example a module might have separate links for sending control frames and data frames to the same module or possibly separate links to send UDP and TCP traffic to different modules. Finally, the use of a central switch adds the ability to perform runtime reconfiguration through changing the linking table and the effect of "pausing" the stack by halting the routing of frames.

3.2 FINS Framework Version 1.0 : Design and Implementation

This section details the implementation of the architecture in version 1.0, with most discussion about the three major parts of our implementation: the *FINS Framework core process*, the *FINS Framework socket stub module*, and the *FINS Framework MAC/PHY stub module.* In this version we strove to reproduce the existing capabilities offered by the traditional stack with the goal that subsequent versions would exhibit improved support for configurability at the MAC/PHY layer and more interactivity through additional modules. As such, the switch and modules for the socket stub, Real-Time Manager (RTM), ICMP, TCP, UDP, IPv4, ARP, and the MAC/PHY stub have been implemented and connected as shown in Fig. 3.3. Note that each of the protocols provided are new implementations following the

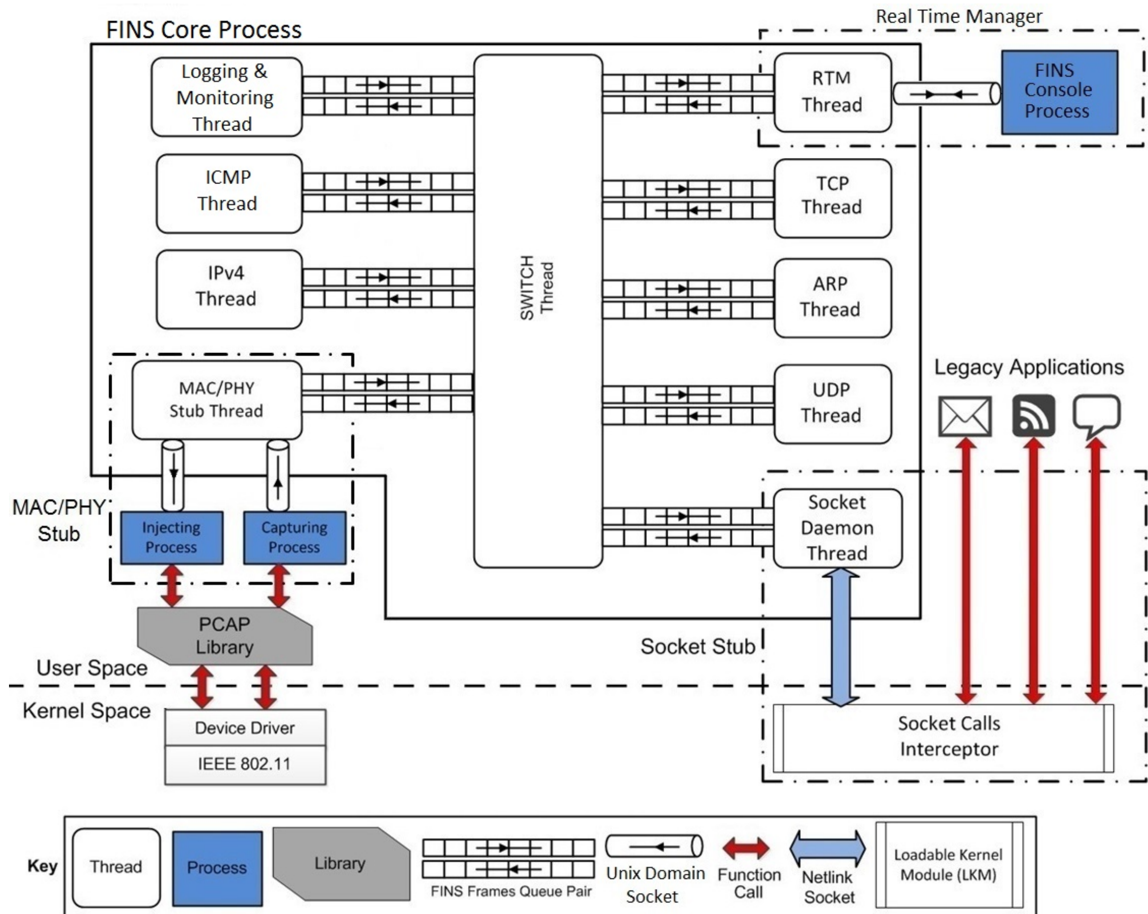


Figure 3.3: FINS Framework implementation and flow of data

appropriate RFCs and standards. Most of the commonly used options and functionality has been provided, such as ARP retransmissions, IPv4 fragmentation, and ICMP error handling. In particular, the TCP module offers an implementation of TCP New Reno with support for many socket options, such as buffer sizes, window scaling, and delayed acknowledgements.

The FINS Framework is written in C and implements most of the platform in user-space on top of the Linux kernel. The following two platforms served as target platforms:

- Laptop computers running Ubuntu 12.04 and Linux kernel 3.2 with an IEEE 802.11

wireless interface,

- and Nexus 7 tablets running Android 4.2.2 and Linux kernel 3.1.

Implementing most of the FINS Framework in user-space means users avoid needing kernel-space programming, are able to reuse existing libraries and/or FINS Framework code, and recover more quickly from mistakes, collectively speeding development. Finally, C is portable to Android handheld devices, as the Android OS is built on top of a version of the Linux kernel and has built-in support for running native applications.

3.2.1 FINS Framework Core Process

A majority of the components in the architecture are implemented as part of the core process, which is a single, multi-threaded process in user-space. The switch is implemented as a single thread and each module is implemented using at least one thread. The modules and switch interact through pairs of input (switch-to-module) and output (module-to-switch) queues, with a pair associated to each module. A module is expected to receive FINS frames through its input queue, process them, and send any response or internally generated frames through its output queue, thus providing a unified interface between the switch and each module.

The switch is expected to read frames from the module output queues using a round robin mechanism, determine the receiver module(s) for each frame based on the configuration stored in its linking table, and push the frame onto the corresponding module's input queue. The linking table should be configured with all ingress and egress data traffic paths as well as

control traffic paths between modules. Table 3.2 shows an example linking table configured to rebuild the traditional stack using the modules presented in Fig. 3.3.

The linking table determines a frame's receiver(s) using its sender and destination ID. For a simple example, consider the case when the MAC/PHY stub thread intends to send an Ethernet frame, but lacks the corresponding MAC address (Fig. 3.4):

C1) The MAC/PHY stub thread recognizes it needs a MAC address and creates a Read Parameter Request control frame with the appropriate metadata to communicate with the ARP module. The MAC/PHY stub module inserts the frame into its output queue with the destination ID set to its link ID associated with ARP traffic (02).

C2) The switch thread reads the frame from the MAC/PHY stub output queue and uses the sender and destination ID (02) to search the linking table for the receiver module IDs (7 and 8). Since there are multiple destinations the frame is copied and the switch pushes one into each input queue of the receiver modules (ARP and Logging & Monitoring).

C3) The ARP thread reads the frame from its input queue and searches the locally cached MAC addresses using the metadata provided in the frame. If no up-to-date MAC address is found, it sends out ARP requests using data frames and resolves the address. After resolving the address, the ARP thread changes the control frame's operation code to Read Parameter Reply, adds the appropriate information to the metadata, and inserts the frame into its output queue with the destination ID set to its link ID associated with MAC/PHY stub traffic (01).

C4) The switch thread reads the frame from the ARP output queue, uses the sender and destination ID (01) to search the linking table for the receiver module IDs (6 and 8), since there are multiple destinations the frame is copied and the switch pushes one into each input queue of the receiver modules (MAC/PHY stub and Logging & Monitoring).

C5) The MAC/PHY stub thread reads the frame from its input queue, retrieves the necessary information from the metadata, and completes the Ethernet frame.

This example illustrates the use of control frames to communicate between modules and the interaction between the linking table, switch, and modules. The frames sent to the Logging & Monitoring module were not discussed as the module simply logs their contents; however, they showcase how transparent logging is possible through simple manipulation of the linking table.

It is important to note that in version 1.0 the linking table has been optimized and implemented in a distributed manner, with each module receiving a subset of the linking table, performing the search for module ID(s) when sending, and copying frames when multiple receivers are found. Thus the destination ID field in frames stores the module ID of the receiver and the switch simply forwards frames to their destination. A subsequent optimization is that replies to control frames (e.g. Parameter Read Reply, Parameter Write Confirmation, Procedure Execute Reply) can be sent to the module ID contained in the sender ID field without a lookup in the local linking table. Later step-by-step examples will describe traffic flow using the distributed implementation of the linking table.

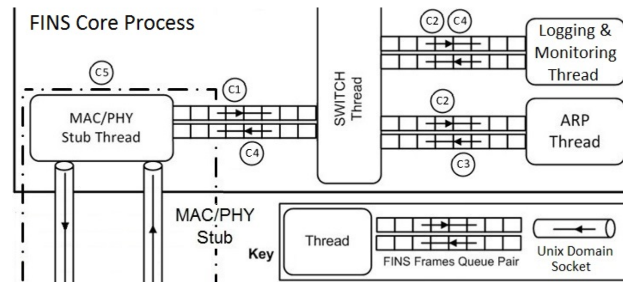


Figure 3.4: Steps of example interaction between the MAC/PHY stub & ARP modules.

Implementing the core of the FINS Framework as a single process greatly simplifies development and reduces overhead from traffic between modules, which would be non trivial in a highly modular configuration. At the same time, the use of independent modules interfacing through generic input/output queues ensures module reusability and the possibility of alternative implementations for the core process. Inspiration could be taken from tools like Emstar, such that each module is implemented as a separate background process that communicates through Unix Domain sockets or shared memory, trading efficiency in module-to-module communication for stability and fault tolerance. While this does not currently suite our goals, our implementation leaves these alternatives as viable possibilities.

3.2.2 FINS Framework Socket Stub Module

Supporting backwards compatibility of unmodified legacy applications is an important factor in implementing the socket stub module. Many of the reviewed tools performed tradeoffs between efficiency, ease of development, and flexibility by implementing their mechanism in either kernel-space or user-space. The socket stub module (Fig. 3.3, bottom right) merges

Table 3.2: An example of configuration records from the FINS Framework linking table

Sender Module	Sender Module ID	Destination Link	Receiver Module	Receiver Module ID
Socket Stub	1	01	ICMP	2
Socket Stub	1	02	TCP	3
Socket Stub	1	03	UDP	4
ICMP	2	01	Socket Stub	1
ICMP	2	02	TCP	3
ICMP	2	03	UDP	4
ICMP	2	04	IPv4	5
TCP	3	01	Socket Stub	1
TCP	3	02	ICMP	2
TCP	3	03	IPv4	5
UDP	4	01	Socket Stub	1
UDP	4	02	ICMP	2
UDP	4	03	IPv4	5
IPv4	5	01	ICMP	2
IPv4	5	02	TCP	3
IPv4	5	03	UDP	4
IPv4	5	04	MAC/PHY stub	6
MAC/PHY stub	6	01	IPv4	5
MAC/PHY stub	6	02	ARP	7
MAC/PHY stub	6	02	Logging & Monitoring	8
ARP	7	01	MAC/PHY stub	6
ARP	7	01	Logging & Monitoring	8

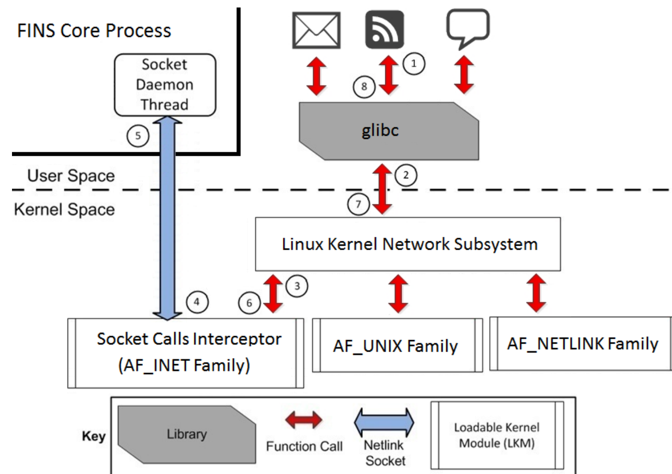


Figure 3.5: Implementation and data flow for the FINS Framework Socket Stub Module

the benefits of both by being implemented as two separate components: the *socket calls interceptor* in kernel-space and the *socket daemon thread* in user-space.

The socket calls interceptor is meant to be a lightweight component that intercepts pertinent socket system calls and relays the necessary information to and from the socket daemon thread in the core process. To do this the socket calls interceptor is implemented as an LKM that unregisters the traditional internet socket family (AF_INET) and replaces it with our own family. This utilizes the behavior of the kernel to direct only appropriate socket calls to the socket calls interceptor and avoids re-implementing functionality provided by the kernel, such as handling cloned sockets or interactions with file descriptors. The socket calls interceptor communicates with the socket daemon thread over a Netlink connection (Fig. 3.3, blue arrow) in client-server fashion, whereby the inherently parallel system calls are serialized and passed through the outgoing Netlink connection in order of arrival.

The socket daemon thread within the core process is attached to the other end of the

Netlink connection and handles tasks that were originally implemented by the socket system call handlers and the network subsystem inside the kernel, such as maintaining the status and structures of opened sockets. An example walk-through of the steps in intercepting a socket call is as follows (see Fig. 3.5):

1. In user-space, an application calls socket() to create a socket for ICMP, TCP or UDP transfer.

2. Glibc converts the socket() call into a system socket() call that goes to the network subsystem in the Linux kernel.

3. In kernel-space, the network subsystem demuliplexes each call to its respective family and for the AF_INET family the call is directed to the socket calls interceptor. Through this, other types of socket communication are left untouched and are directed to their traditional handlers, e.g. Netlink, Unix domain sockets, etc.

4. The socket calls interceptor catches the call and creates the minimum necessary kernel socket objects, setting the socket operations (bind(), connect(), etc) to corresponding functions in the socket calls interceptor. This enables the kernel to track the socket and directs future system calls to the appropriate socket calls interceptor function (bind() → FINS_interceptor_bind()). The socket calls interceptor serializes the call, forwards it to the socket daemon thread in the core process through the Netlink connection, and waits for a response.

5. In user-space, the socket daemon thread processes the socket() request, creates a new

socket record in the FINS Framework socket database, and constructs any socket-related objects. The result of processing the call (for socket() a success/failure status) is serialized and transmitted back to the socket calls interceptor through the Netlink connection.

6. In kernel-space, the socket calls interceptor receives the result, deserializes and handles it accordingly, and then returns an associated value to the network subsystem.

7. Depending on the return value, the kernel returns either a socket descriptor or an error to glibc.

8. In user-space, glibc returns the result to the application.

Once a FINS Framework socket is created, future system calls to the socket pass through the network subsystem to a corresponding function in the socket calls interceptor. A similar process is conducted to serialize and shuttle the call to the socket daemon thread with an accompanying procedure to deserialize and handle the socket daemon thread's return.

Of the many possible mechanisms for intercepting system calls, we found the two-component LKM solution to be the least invasive and most advantageous. By replacing AF_INET using a lightweight LKM, interception is seamless without modifying any kernel code, meaning that there is no need to recompile the Linux kernel. This avoids many of the issues related to kernel-space programming and allows for the same socket calls interceptor code to be used for kernel versions 2.6 to 3.8 on both Ubuntu and Android with only negligible changes. In addition, this intercepts all calls to internet sockets whether from dynamically or statically

linked applications, which was a problem for an earlier user-space attempt. Unfortunately, some minor overhead is introduced for each call as all necessary information must be shuttled from kernel-space to user-space and vice versa.

3.2.3 FINS Framework MAC/PHY Stub Module

The MAC/PHY stub module (Fig. 3.3, middle left) connects to the network and achieves portability across platforms. Implementation of the MAC/PHY layers differs significantly among platforms due to interactions among the kernel, the device driver, and the network adapter's firmware. The current MAC/PHY stub module is implemented using the Pcap library [44], a portable C/C++ library for network traffic capture and injection. Using the Pcap library allows the user to capture and inject Ethernet frames, and get/set some basic network adapter parameters.

The MAC/PHY stub module is implemented as three components: the *MAC/PHY stub thread*, the *capturing process*, and the *injecting process*. The MAC/PHY stub thread is a thread within the core process that works as a multiplexer/demultiplexer of the traffic between the switch thread and a networking interface. It is connected through a Unix domain socket to the capturing process and through a second Unix domain socket to the injecting process. In order to accelerate the processing of frames and enhance the overall performance of the FINS Framework, the generation and serialization of Ethernet frames is implemented in the MAC/PHY stub thread, while their injection and capture are isolated into the two

processes outside the core process (Fig. 3.3, lower left). In effect, to send an IP packet the MAC/PHY stub thread generates an appropriate MAC frame header, encapsulates the packet, serializes the frame, and sends it over the domain socket to the injecting process. In turn, the injecting process forwards the stream to the network adapter's buffer by calling the Pcap library injection function. The operation is reversed when capturing traffic from a network adapter. Note that Unix domain sockets were used in this module instead of pipelines in order to facilitate easy deployment to mobile devices as file permissions in Android severely limit the use of pipelines.

3.2.4 FINS Framework RTM module and FINS Framework Console

Two minor but essential parts of the FINS Framework are the *Real-Time Manager (RTM) module* and the *FINS Framework console*. The RTM module is a non-data module that receives input from outside of the framework and allows runtime management and supervision of the framework. The FINS Framework console is a command line application that connects to the RTM through a Unix domain socket and can either act as an interactive prompt or simply receive status updates from the framework. Through the RTM and console, advanced monitoring and logging is possible as well as runtime reconfiguration of the linking table and modules.

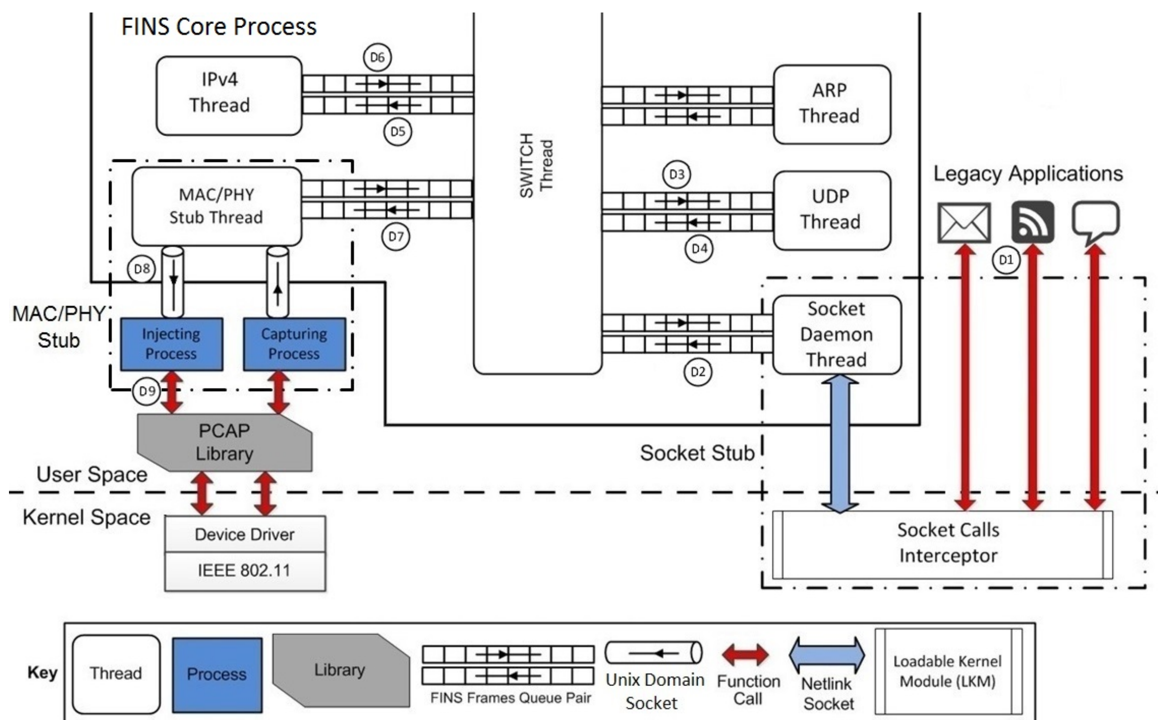


Figure 3.6: UDP/IP example stack using the FINS Framework

3.2.5 Traffic Flow Walkthrough

Fig. 3.6 illustrates a typical traffic flow using an example stack that has rebuilt the traditional TCP/IP stack. This is the same stack mentioned earlier whose configuration is given in Table 3.2. The steps of the outgoing traffic flow are indicated using circles labeled D1-D9 and will be discussed thoroughly, while only differences will be mentioned for the incoming traffic flow. For the outgoing traffic flow, consider the case when a running UDP application sends data:

D1) A UDP application sends a buffer of raw data through the glibc socket API.

D2) The socket daemon thread receives the data from the socket calls interceptor, encapsulates the data into a data frame, and uses its link ID associated with UDP traffic (03) to search its local subset of the linking table for a receiver module ID (4). The socket stub module inserts the frame into its output queue with the destination ID set to the UDP module (4).

D3) The switch thread reads the frame from the socket stub output queue and pushes it into the input queue of the UDP module as directed by the destination ID (4).

D4) The UDP thread reads the frame from its input queue, decapsulates the raw data, and extracts necessary information from the metadata, such as the sending and receiving IP addresses/ports. The UDP thread performs its UDP related functions, creates a UDP datagram, and inserts it into the data frame. It then uses its link ID associated

with traffic going down the stack (03) to search its local subset of the linking table and pushes the frame onto its output queue with the destination ID set to the IPv4 module (5).

D5) The switch thread reads the frame from the UDP output queue and pushes it into the input queue of the IPv4 module as directed by the destination ID (5).

D6) The IPv4 thread reads the frame from its input queue, extracts the encapsulated UDP datagram and any required metadata. Then, a new IPv4 packet gets built based on the metadata forwarded from the UDP module and is encapsulated into the data frame. After searching its linking table using its link ID associated with downward traffic (04), the IPV4 thread pushes the frame onto its output queue destined for the MAC/PHY module (6).

D7) The switch thread reads the frame from the IPv4 output queue and pushes it into the input queue of the MAC/PHY stub module as directed by the destination ID (6).

D8) The MAC/PHY stub thread reads the frame, extracts the IPv4 packet and the metadata. Then, it builds an Ethernet frame to be sent through the Pcap library. The MAC/PHY stub thread attempts to resolve the MAC address using its internal copy of previously found MAC addresses, contacting the ARP module if the corresponding MAC address is not found (Fig. 3.4). Once the MAC address has been acquired it is used to finish the Ethernet frame, which is then pushed into the injection Unix domain socket that carries data to the injecting process.

D9) The injection process reads the Ethernet frames buffered into its input Unix domain socket. Then, it sends each frame over a separate call to the Pcap library, which pushes the frames to the interface device driver.

Note that these steps may not occur directly after each other, as the modules may process other frames in between or conduct module-to-module communication to retrieve necessary information. This is referenced in Step D8, where conditions may necessitate a control flow to request the MAC address.

The incoming data flow is generally the reverse of the outgoing flow taking into account the following:

- The Pcap library captures the incoming Ethernet frames after the device driver replaces the original MAC and PHY header with a generic Ethernet header. The frames are pushed into a special Pcap buffer space which is read using a specific call back function.
- The capturing process serializes each captured frame and pushes it into the Unix domain socket connecting the capturing process to the core process.
- The steps then generally proceed in reverse order of the outgoing flow explained earlier.
- Eventually, the incoming data is pushed into the socket daemon thread's input queue. After reading the frame from the queue, the frame metadata is used to search the socket database for the target socket. If the destination socket is found, the frame is pushed into a separate internal receiving queue, which is one of the socket-layer related

objects that was created when the socket was created.

- Whenever the socket stub detects a socket receiving call from the application, the next data frame is read from the internal queue. The encapsulated raw data within the frame is serialized and sent over the Netlink connection to the socket calls interceptor. Then the socket calls interceptor returns the data to the application through the Linux kernel socket API.

Chapter 4

Results and Discussion

This chapter is segmented into two parts: an evaluation of performance of the FINS Framework in basic experimental scenarios on Ubuntu and Android, and a discussion about more complex experimental situations enabled by the FINS Framework.

4.1 Performance Evaluation

While flexibility is a primary goal of the FINS Framework, a minimum level of performance is needed for it to be usable in most scenarios. Our goal was to support data rates up to 54 Mbps, the maximum data rate of IEEE 802.11g. The equipment used in the following tests included:

- A Netgear N300 IEEE 802.11b/g/n wireless router access point.

- Two Dell Inspiron 1525s laptops with Intel Core 2 Duo processors running at 2 GHz, 3 GB of RAM, Broadcom IEEE 802.11b/g wireless interfaces, and Marvel fast Ethernet controllers. Both laptops were running Linux Ubuntu 10.04 with Linux kernel version 3.2.0.

- Two Google Nexus 7 tablets produced by ASUS. Both tablets were running Android 4.2.2 with Linux kernel version 3.1.10.

For each of the following experiments, the end-to-end throughput was measured at the application level using the common networking tool *iperf*.

4.1.1 Experiment 1

This experiment was intended to evaluate the socket stub module and determine the maximum throughput of application data through the module and verify it does not limit end-to-end goodput. This is important since socket calls must be shuttled to the core process and a bottleneck in this procedure would slow the execution of applications, potentially changing their behavior. To evaluate this, Experiment 1 utilized a single computer not connected to the network that was running a FINS Framework which consisted of only the switch, the socket stub module, and a custom logging module that reported throughput. The switch linking table was configured so that frames from the socket stub module traveled to the logging module, which simply measured the throughput of application data. On top of this stack ran an instance of the iperf application in client mode, which was set to generate 10

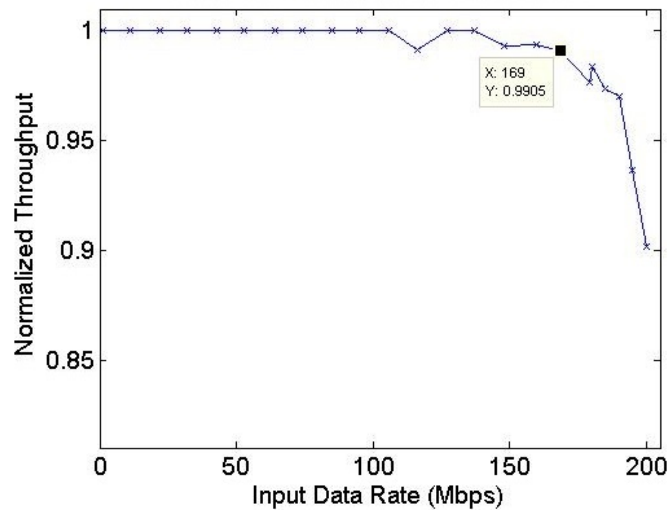


Figure 4.1: The observed throughput through the socket stub module normalized by the data rate attempted by iperf.

second trials of UDP traffic sent through the local loopback in 1.47 kB datagrams.

The results of this experiment (Fig. 4.1) show that the socket stub module is able to handle throughput data rates up to 169 Mbps with negligible overhead. In addition, throughout the entire testing range (1-200 Mbps) the FINS Framework exhibited a drop rate for system calls and UDP datagrams of 0%. This suggests that the socket stub module and the data transfer between the socket calls interceptor and the socket daemon thread is reliable and will not create a bottleneck for the FINS Framework in networking experiments. However, it is important to note that Netlink connections have been shown to operate at speeds an order of magnitude faster, indicating that the limiting factor for the socket stub module is most likely the processing done in the socket daemon thread. Since the core process is implemented as a single process with many threads, the throughput of the socket stub module may decrease

in practice due to scheduling constraints.

4.1.2 Experiment 2

The goal of Experiment 2 was to evaluate the current implementation of the MAC/PHY stub module and observe the maximum throughput of application data that our packet capturing mechanism can support without dropping frames. This is pertinent because the Pcap library will overwrite previously captured frames in its buffer if the capturing process does not poll and process the frames fast enough. To evaluate this the traditional stack and the FINS Framework were run in tandem and iperf was used to send traffic through the traditional stack to the local loopback address. When these packets were "received" by the node, the FINS Framework was able to intercept these packets through the Pcap library. As such, the setup for this experiment consisted of a single, non-networked computer that was running a FINS Framework which consisted of only the switch, the MAC/PHY stub module, and a custom logging module that reported throughput. The switch linking table was configured so that frames from the MAC/PHY stub module traveled to the logging module, which measured the throughput of application data. Since this particular FINS Framework excluded the socket stub module the traditional TCP/IP stack was still accessible and was used to run the iperf application. Once again, iperf ran in client mode and was set to generate 10 second trials of UDP traffic sent through the local loopback in 1.47 kB datagrams. The local loopback was used to avoid the increased probability of dropping frames due to a wireless channel or potential throughput limitations in using a LAN cable.

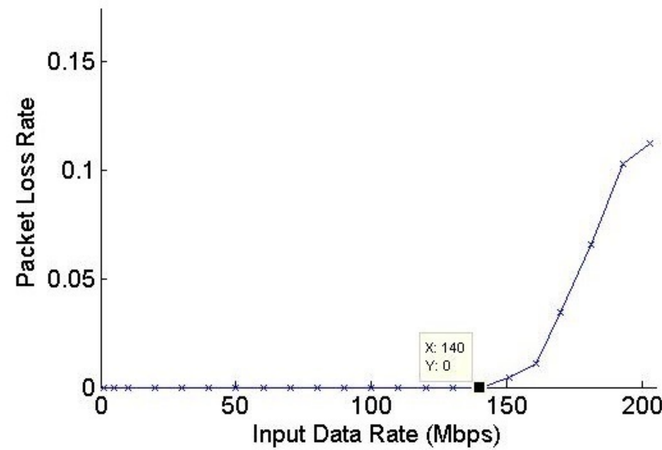


Figure 4.2: The packet loss rate through the MAC/PHY stub module at varying data rates. The traffic was sent over the local loopback of a laptop to reduce packet loss from channel effects.

Fig. 4.2 shows that the MAC/PHY stub module maintained a packet loss rate of 0% at incoming data rates up to 140 Mbps. The packet loss rate appeared to increase rapidly at rates above 140 Mbps. Throughput measurements from the logging module showed that zero percent (0%) of the packet loss occurred internal to the FINS Framework, indicating that the packet drops shown in Fig. 4.2 occurred at the Pcap library level. Our analysis suggests that this is caused by the capturing process blocking on the Unix domain socket connecting it to the MAC/PHY stub thread, which effectively couples the processing delay of handling a frame in the MAC/PHY stub thread with how quickly the capturing process can retrieve frames from Pcap. As with the socket stub module, this processing delay may increase in times of high scheduling demand.

4.1.3 Forward for Experiments 3-4

The following two experiments use similar setups to contrast the maximum performance achievable by the FINS Framework (found using a wired setup) to a more realistic performance observed in a wireless environment. The UDP and TCP results recorded in both experiments are shown in Tables 4.1 and 4.2, respectively. In addition, benchmark values for the traditional Linux stack were recorded for reference and are also shown. During Experiment 4, the results of the traditional stack were used to estimate the wireless channel, which is why the drop rate is not listed.

4.1.4 Experiment 3

In the third experiment, we used iperf to measure the maximum end-to-end goodput achievable by the FINS Framework when sending or receiving data. To observe the sending rate we used the two laptops networked together using a 100 Mbps LAN cable (Fig. 4.3). On one of the computers an iperf client was run on top of a FINS Framework, while the other computer ran an iperf server on top of the traditional Linux stack. The FINS Framework used for the experiment was the rebuilt stack depicted in Fig. 3.6 and whose linking table was shown in Table 3.2. Finally, all processes that were not required by the operating system were killed on both computers for the duration of this experiment. The experiment was performed over a period of days with each trial consisting of a 5 minute iperf traffic stream. For the receiving rate the same setup was used with the traffic stream reversed – the iperf server was run on

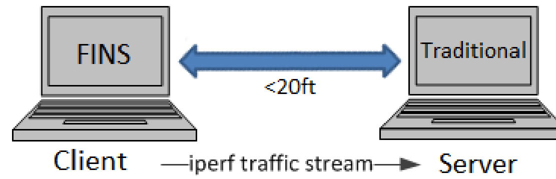


Figure 4.3: Experimental setup for Experiment 3. Note the blue arrow represents a 100 Mbps LAN cable connecting the nodes. The black arrow represents the flow of UDP datagrams or TCP data segments.

the FINS Framework and the iperf client application on the traditional stack.

In Experiment 3, it was found that version 1.0 of the FINS Framework is currently able to send a maximum of 87.1 Mbps through UDP and 83.6 Mbps using TCP. These values are close to the observed throughputs for the traditional stack (95.4 Mbps and 94.4 Mbps respectively) when limited by a 100 Mbps LAN cable. Receiver results where more nuanced, with the FINS Framework currently able to receive at a rate equal to that of the traditional stack for UDP (95.4 Mbps) and a significant fraction for TCP (48.1 Mbps). The major performance difference between UDP and TCP receive results is likely due to our current implementation of the TCP module and not the stack as a whole. In any case, the results demonstrate the ability to send and receive at speeds that would be enough to support the theoretical data rates of IEEE 802.11a/b/g and may be enough for the rates practically achieved by IEEE 802.11n technologies.

Table 4.1: UDP Results for Experiment 3 (laptops, LAN) and Experiment 4 (tablets, Wi-Fi).

		FINS Framework			Traditional Stack		
Network	Action	Goodput (Mbps)	Jitter (ms)	Drop (%)	Goodput (Mbps)	Jitter (ms)	Drop (%)
Exp. 3	Sending	87.1	0.242	0.0615	95.4	0.222	0.00610
	Receiving	95.4	0.242	0.135	95.4	0.222	0.00610
Exp. 4	Sending	15.7	1.40	0.0575	15.7	1.36	-
	Receiving	15.6	1.74	0.397	15.7	1.36	-

Table 4.2: TCP Results for Experiment 3 (laptops, LAN) and Experiment 4 (tablets, Wi-Fi).

		FINS Framework	Traditional Stack
Network	Action	Goodput (Mbps)	Goodput (Mbps)
Exp. 3	Sending	83.6	94.4
	Receiving	48.1	94.4
Exp. 4	Sending	13.0	13.0
	Receiving	13.0	13.0

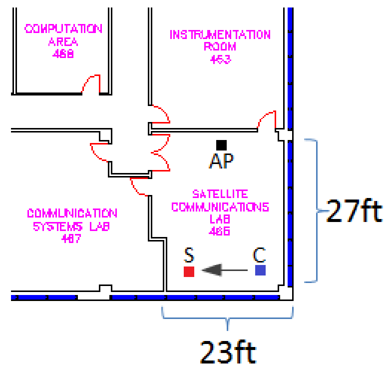


Figure 4.4: Experimental setup for Experiment 4. The test was conducted at Virginia Tech using tablets in a setup similar to that of Experiment 3 (Fig. 4.3). Depicted in the figure is an iperf client on top of a FINS Framework (blue) sending over Wi-Fi through an IEEE 802.11n access point (black) to an iperf server on top of the traditional stack (red). The black arrow indicates the flow of iperf traffic in UDP datagrams or TCP data segments.

4.1.5 Experiment 4

This experiment was designed to observe the maximum end-to-end goodput achievable by the FINS Framework in a wireless environment. As such, the previously described Android tablets were set to work in Wi-Fi infrastructure mode and comprised a basic service set (BSS) in conjunction with the access point (Fig. 4.4). Similar to Experiment 3, when sending one of the computers ran an iperf client on top of a FINS Framework (Fig. 3.6), while the other computer had an iperf server on top of the traditional Linux stack. The traffic stream was reversed when collecting receiving rates.

Using the traditional stack, an estimate of the maximum goodput possible between end nodes in this wireless setup was found to be 15.7 Mbps for UDP and 13.0 Mbps for TCP.

Within these constraints, the FINS Framework achieved a maximum sending rate of 15.7 Mbps for UDP and 13.0 Mbps for TCP through the iperf server application. For receiving, the framework performed equally well (15.6 Mbps and 13.0 Mbps) with the observed packet loss ($< 0.4\%$) occurring at the Pcap library level. Throughout this entire experiment the FINS Framework maintained a zero percent (0%) dropping rate within the stack, with the low maximum goodput estimates provided by the traditional stack caused by the noisy wireless environment. Analysis of the observed traffic and the good wireless performance for the FINS Framework relative to the traditional stack suggest that while our TCP module may have flaws receiving at higher rates it responds robustly to moderate segment loss.

4.1.6 Conclusions

Experiments 1 and 2 tested the two FINS Framework stub modules and established that they operate efficiently by themselves, setting high upper limits when transferring data to existing network layers. Experiments 3 and 4 tested the overall performance of the framework to show that version 1.0 is currently able to support implementation-based Wi-Fi networking experiments for IEEE 802.11a/b/g standards on Ubuntu and Android. The lesser performance seen when receiving a TCP datastream suggests further improvements to the TCP module may be necessary if testing at high throughput on LAN. However, considering wireless environments are the desired use case and the available flexibility of the framework, the performance losses are acceptable and the FINS Framework meets the minimum requirements.

4.2 Experimental Scenarios

Various experimental scenarios have been successfully implemented and tested using the FINS Framework in qualitative trials that occurred in the lab and at a small workshop. At the most basic level, extensive testing with many networking applications was conducted in the lab to ensure the transparent support of tools commonly used in research. Each of the tools were successfully used unmodified and without recompilation; examples include: *ifconfig*, *ping*, *iperf*, *dig*, *traceroute*, *Telnet*, *Wireshark*, and *Firefox*. Testing involved

- using many of the common command line options,
- connecting to other nodes running the traditional stack or FINS Framework,
- running multiple instances of a tool or several different tools simultaneously, and
- connecting to sites through the Virginia Tech network.

Furthermore, many of the tools were purposely used in the workshop described below.

4.2.1 Simple Scenarios

The workshop brought together about 20 participants for a handful of attendee-driven hands-on exercises that demonstrated the benefits of the framework through experimental scenarios that ranged in complexity and implementation challenge. Simple examples included creating a new module and inserting it into the linking table to intercept conventional outgoing traffic

flow to allow for routing decisions on a packet-by-packet basis. This low cost mechanism was used to conduct group experiments in a ring topology, highlighting how easy it is to reconfigure the framework's behavior without having to augment other modules. One logical extension of this exercise might match the MANIAC competition, where teams compete by each implementing an experimental forwarding scheme as a module that is simply inserted to configure a node.

4.2.2 Complex Scenarios

More challenging scenarios involved modifying a Broadcom Wi-Fi device driver to allow for direct communication with the FINS Framework to provide real-time MAC information for a context-aware network application. Through the RTM, attendees were able to view real-time MAC information with more detail and granularity than possible through the traditional stack, an important factor for context-aware applications that are time sensitive, such as adaptive video streaming. Trials by attendees involved simultaneous transmission of 10+ data streams each governed by a context-aware application competing for optimal performance. Similar to the previous example, development costs related to the FINS Framework were low, modifying the driver made up the bulk of the work.

Other exercises focused on the ability to monitor and control network behavior at runtime, potentially for educational uses. For instance, one scenario involved controlling TCP behavior and variables (e.g. Fast retransmit, GBN, timeouts, etc.) while observing the

impact on performance in real-time. Another exercise showcased Android and the ease of use when performing mobile experiments. The planned extension introduced collecting data from the FINS Framework as well as on-board sensors to better characterize the relationship of movement and throughput in a MANET.

4.2.3 Conclusions

The qualitative evaluation showed that the FINS Framework is able to support many networking applications without modification and thus provide standard benchmarking tools for wireless networking experiments. This reduces the ancillary overhead in using the FINS Framework by allowing researchers to use familiar tools. The scenarios tested at the workshop showed the practical utility of the framework in an uncontrolled environment and with faux researchers, as well as demonstrating the flexibility of the platform through simple and complex cases.

Chapter 5

Conclusion

5.1 Summary of Work and Contributions

In this thesis we presented the FINS Framework, an open-source tool with the goal of facilitating implementation-based experiments in wireless networking research. The FINS Framework aimed to address evaluation and implementation challenges by moving networking functionality into an open, user-space architecture and supporting existing applications with additional management functionality.

In Chapter 2, we surveyed existing experimental wireless network tools, described their intended use, and listed relative advantages and disadvantages. From this we chose desirable characteristics and identified a lack of sufficient support for cross-layer optimization and context-aware applications, functionality which the FINS Framework intends to provide.

Chapter 3 described the architecture, design, and implementation of the framework in depth and showed how we incorporated these characteristics to meet our goals. In this chapter we also detailed a mechanism for utilizing the traditional network stack to support existing applications and provide backwards compatibility without recompilation or re-implementing functionality for file management or process control.

Using unmodified network applications, initial performance limits for the stub modules and the framework overall were collected in Chapter 4. Results showed that version 1.0 of the FINS Framework is capable of supporting experiments requiring IEEE 802.11g hardware speeds and operating on both Ubuntu laptops and Android tablets. Later in the chapter we discuss qualitative results from a workshop using the FINS Framework and explore possible experimental scenarios enabled by our new tool.

The work from this thesis is reported in the following publications:

- Michael S. Thompson, Abdallah S. Abdallah, Jonathan M. Reed, Allen B. MacKenzie, and Luiz A. DaSilva, "The FINS Framework: An Open-Source, Userspace Networking Subsystem for Linux," accepted to *IEEE Network Magazine*, special issue on "Open Source for Networking: Tools and Applications".

- Jonathan M. Reed, Abdallah S. Abdallah, Allen B. MacKenzie, Luiz A. DaSilva, Michael S. Thompson, "The FINS Framework: Design and Implementation of the Flexible Internetwork Stack (FINS) Framework," under review for *IEEE Transactions on Mobile Computing.*

5.2 Future Work and Closing

In releasing Version 1.0 of the FINS Framework and submitting papers presenting the tool, we feel that many of our design goals have been accomplished; however, there still exists many areas for improvement and opportunities where the framework could be used to explore additional research fields.

From the results observed in Chapter 4, our implementation of the TCP module needs improvements to better handle traffic at higher speeds. While performance improvements are expected for the TCP module and the FINS Framework overall, there are major changes to the TCP module that we see as possible future work. As an initial version meant to replicate the base functionality of the traditional stack, the TCP protocol was implemented as a single module with the connection management, reliability, and congestion control components distinct but in the same module. One approach to creating a new TCP implementation would be to decompose the module into multiple pieces, similar to the concepts presented in the WiFu project. Encapsulating specific components of the protocol into modules will allow for researchers to more easily create multiple versions of TCP.

Another area of possible future work includes improving the logging and supervisory modules currently available. While the RTM provides advanced logging and monitoring functionality to external applications through direct and in-depth access to the meters and knobs of modules, the logging module included in the latest version is meant only for basic tasks, such as traces, throughput measurements, and metadata related statistics. Intro-

ducing a logging module that actively polled modules for information (similar to the RTM module) or one that automatically stored information it collected into a database would reuse existing functionality to improve passive logging with minimal cost. Furthermore, there are many alternative ways that supervisory modules could aid researchers. Providing GUIs for managing the configuration of the protocol stack or simply visualizing traffic in real-time would aid researchers or educators in exploring scenarios.

A possible research opportunity includes experimenting with and creating an API to connect to applications that is radically different from the BSD socket API in order to better customize it to context-aware applications in wireless environments. Projects like ANA have already started exploring modifications to or functionality beyond the BSD socket API with intriguing results. Through the ability for applications to connect directly to the framework and the generic nature of modules, the FINS Framework allows researchers to explore new interfaces with applications and experimental scenarios that would be difficult with other tools.

Already being pursued is the addition of more meters and knobs for cross-layer research by expanding user control and capabilities at lower levels. The currently implementation of the MAC/PHY stub module is elegant but limited. The use of the Pcap library allows the following functionality:

- use of IEEE 802.2/3/11 network adapters,
- disabling/enabling network adapters,

- switching between the basic Wi-Fi modes (infrastructure and adhoc), and
- using the loopback interface.

However, using PCAP does not provide control over IEEE 802.11 MAC details (e.g. retrieving the IEEE 802.11-specific headers), physical layer details (e.g. reading IEEE 802.11 channel parameters or controlling the PHY sending rate), or controlling the networking interface (e.g. controlling transmission power levels, switching to monitoring mode). The plan for future FINS Framework releases is to create a new MAC/PHY stub module implemented as a modified device driver in order to access the previously mentioned desired attributes and controls. This implementation would not modify the design or implementation of the FINS Framework as device drives in the Linux kernel are implemented as LKMs and the modified device driver component could be loaded/unloaded without a need to recompile the kernel or to restart the machine. Currently, a prototype of the new MAC/PHY stub module supported by a modified Intel Wi-Fi device driver is being tested for Linux platforms.

Progress in these areas and others is expected to continue in the future as the FINS Framework matures. For more information on the FINS Framework, including source code, complete user documentation, project progress, and future plans, the reader may refer to the project website at `http://www.finsframework.org`.

Bibliography

[1] W. Kiess and M. Mauve, "A survey on real-world implementations of mobile ad-hoc networks," *Ad Hoc Networks*, vol. 5, no. 3, pp. 324 – 339, 2007.

[2] S. Kurkowski, T. Camp, and M. Colagrosso, "MANET simulation studies: The incredibles," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, pp. 50–61, 2005.

[3] C. A. V. Campos and L. F. M. de Moraes, "Investigating the user mobility in wireless mobile networks through real measurements," in *Proceedings of the 2006 ACM CoNEXT Conference*, ser. CoNEXT '06. New York, NY, USA: ACM, 2006, pp. 58:1–58:2. [Online]. Available: http://doi.acm.org/10.1145/1368436.1368503

[4] A. Rachedi, S. Lohier, S. Cherrier, and I. Salhi, "Wireless network simulators relevance compared to a real testbed in outdoor and indoor environments," in *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, ser. IWCMC '10. New York, NY, USA: ACM, 2010, pp. 346–350. [Online]. Available: http://doi.acm.org/10.1145/1815396.1815477

[5] R. Khattak, A. Chaltseva, L. Riliskis, U. Bodin, and E. Osipov, "Comparison of wireless network simulators with multihop wireless network testbed in corridor environment," in *Wired/Wireless Internet Communications*, ser. Lecture Notes in Computer Science, X. Masip-Bruin, D. Verchere, V. Tsaoussidis, and M. Yannuzzi, Eds. Springer Berlin Heidelberg, 2011, vol. 6649, pp. 80–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21560-5_7

[6] "The Network Simulator NS-2," September 2010. [Online]. Available: http://www.isi.edu/nsnam/ns/

[7] X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks," *SIGSIM Simul. Dig.*, vol. 28, no. 1, pp. 154–161, Jul. 1998. [Online]. Available: http://doi.acm.org/10.1145/278009.278027

[8] "Modular Simulator OMNeT++," April 2014. [Online]. Available: http://omnetpp.org/

[9] S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic, *Mobile Ad Hoc Networking : The Cutting Edge Directions.* IEEE Press and Wiley, 2013, ch. 6. [Online]. Available: http://xplqa30.ieee.org/xpl/bkabstractplus.jsp?bkn=6480473

[10] "Getting started with ns-3-click and nsclick," April 2014. [Online]. Available: http://www.read.cs.ucla.edu/click/nsclick

[11] "Open-Access Research Testbed for Next-Generation Wireless Networks (ORBIT)," April 2014. [Online]. Available: http://www.orbit-lab.org/

[12] T. Clancy and B. Walker, "Meshtest: Laboratory-based wireless testbed for large topologies," in *Testbeds and Research Infrastructure for the Development of Networks and Communities, 2007. TridentCom 2007. 3rd International Conference on*, May 2007, pp. 1–6.

[13] P. De, A. Raniwala, S. Sharma, and T. Chiueh, "Design considerations for a multihop wireless network testbed," *IEEE Communications Magazine*, vol. 43, no. 10, pp. 102–109, Oct. 2005.

[14] V. Srivastava, A. B. Hilal, M. S. Thompson, J. N. Chattha, A. B. MacKenzie, and L. A. DaSilva, "Characterizing mobile ad hoc networks: the MANIAC challenge experiment," in *WiNTECH '08: Proceedings of The Third ACM International Workshop on Wireless network testbeds, Experimental Evaluation and Characterization*, 2008, pp. 65–72.

[15] "FUSE: Filesystem in Userspace," May 2014. [Online]. Available: http://fuse.sourceforge.net/

[16] A. S. Abdallah, A. B. MacKenzie, L. A. DaSilva, and M. S. Thompson, "On software tools and stack architectures for wireless network experiments," in *IEEE Wireless Communications and Networking Conference (WCNC)*, March 2011.

[17] "The Click Modular Router," January 2010. [Online]. Available: http://read.cs.ucla.edu/click/

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, pp. 263–297, August 2000.

[19] E. Kohler, “Click for measurement,” UCLA Computer Science Department, Tech. Rep., February 2006.

[20] “Click Modular Router Integration,” April 2014. [Online]. Available: https://www.nsnam.org/docs/release/3.13/models/html/click.html

[21] “The Network Simulator NS-3,” September 2010. [Online]. Available: http://www.nsnam.org/

[22] P. L. Suresh and R. Merz, “Ns-3-click: Click modular router integration for ns-3,” in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools ’11. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, pp. 423–430. [Online]. Available: http://dl.acm.org/citation.cfm?id=2151054.2151130

[23] “Ape: Ad hoc protocol evaluation testbed,” April 2014. [Online]. Available: http://apetestbed.sourceforge.net/

[24] H. Lundgren, D. Lundberg, J. Nielsen, E. Nordstrom, and C. Tschudin, “A large-scale testbed for reproducible ad hoc protocol evaluations,” in *IEEE Wireless Communications and Networking Conference, 2002. WCNC2002.*, vol. 1, 2002, pp. 412–418.

[25] H. Kazemi, G. C. Hadjichristofi, and L. A. DaSilva, “MMAN - a monitor for mobile ad hoc networks: Design, implementation and experimental evaluation,” in *Proceedings of the Third ACM International Workshop on Wireless Network Testbeds, Experimental evaluation and CHaracterization (WiNTECH)*, 2008.

[26] "Jpcap: A network packet capture library." May 2014. [Online]. Available: http://jpcap.sourceforge.net/

[27] "Ana: Autonomic network architecture," April 2014. [Online]. Available: http://www.ana-project.org/

[28] G. Bouabene, C. Jelger, and C. Tschudin, "Virtual network stacks," in *PRESTO Sigcomm Workshop*, 2008.

[29] S. P. Aaron Beach, Mike Gartrell and R. Han, "X-Layer: An experimental implementation of a cross-layer network protocol stack for wireless sensor networks," Department of Computer Science University of Colorado at Boulder, Tech. Rep., December 2008.

[30] "IRIS (CTVR, Trinity College Dublin)," April 2014. [Online]. Available: http://www.crew-project.eu/iris

[31] "SRS: Software Radio Systems," April 2014. [Online]. Available: https://github.com/softwareradiosystems

[32] L. Doyle, P. Sutton, K. Nolan, J. Lotze, B. Ozgul, T. Rondeau, S. Fahmy, H. Lahlou, and L. DaSilva, "Experiences from the Iris testbed in dynamic spectrum access and cognitive radio experimentation," in *IEEE Symposium on New Frontiers in Dynamic Spectrum Access*, 2010.

[33] L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "Emstar: A software environment for developing and deploying heterogeneous

sensor-actuator networks," *ACM Trans. Sen. Netw.*, vol. 3, no. 3, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1267060.1267061

[34] "FUSD - a Linux Framework for User-Space Devices," May 2014. [Online]. Available: http://www.circlemud.org/jelson/software/fusd/

[35] "WiFu," April 2014. [Online]. Available: http://internet.byu.edu/research/wifu

[36] R. Buck, R. Lee, P. Lundrigan, and D. Zappala, "WiFu: A Composable Toolkit for Experimental Wireless Transport Protocols," in *IEEE MASS 2012, the 9th IEEE International Conference on Mobile Ad hoc and Sensor Systems*, 2012, pp. 299–307.

[37] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh, "Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols," in *Wireless Communications and Networking Conference, 2005 IEEE*, vol. 3, March 2005, pp. 1664–1669 Vol. 3.

[38] K. Ramachandran, S. Kaul, S. Mathur, M. Gruteser, and I. Seskar, "Towards large-scale mobile network emulation through spatial switching on a wireless grid," in *Proceedings of the 2005 ACM SIGCOMM Workshop on Experimental Approaches to Wireless Network Design and Analysis*, ser. E-WIND '05. New York, NY, USA: ACM, 2005, pp. 46–51. [Online]. Available: http://doi.acm.org/10.1145/1080148.1080158

[39] R. Beuran, Y. Tan, and Y. Shinoda, "Challenges of using wireless network testbeds: A case study on orbit," in *Proceedings of the 6th ACM International Workshop*

on Wireless Network Testbeds, Experimental Evaluation and Characterization, ser. WiNTECH '11. New York, NY, USA: ACM, 2011, pp. 11–18. [Online]. Available: http://doi.acm.org/10.1145/2030718.2030723

[40] "Global Environment for Network Innovations (GENI)," April 2014. [Online]. Available: https://www.geni.net/

[41] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *Proceedings of the ACM Symposium on Communications Architectures &Amp; Protocols*, ser. SIGCOMM '90. New York, NY, USA: ACM, 1990, pp. 200–208. [Online]. Available: http://doi.acm.org/10.1145/99508.99553

[42] V. Srivastava and M. Motani, "Cross-layer design: a survey and the road ahead," *Communications Magazine, IEEE*, vol. 43, no. 12, pp. 112–119, Dec. 2005.

[43] D. Friend, M. EINainay, Y. Shi, and A. MacKenzie, "Architecture and performance of an island genetic algorithm-based cognitive network," in *5th IEEE Consumer Communications and Networking Conference*, 2008, pp. 993 –997.

[44] "TCPDUMP & LiBPCAP," April 2014. [Online]. Available: http://www.tcpdump.org/