A MODULAR SOFTWARE PACKET ROUTER FOR ANDROID

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Computer Science and Engineering

by

Dirk C. Van Bruggen,

__________________________

Aaron Striegel, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
November 2012
A MODULAR SOFTWARE PACKET ROUTER FOR ANDROID

Abstract

by

Dirk C. Van Bruggen

Cellular mobile devices have become an increasingly large part of society, permeating almost every aspect of life. Consequently, researchers have been searching for ways to distribute communication and data usage across not only the cellular networks but also infrastructure and ad-hoc WiFi connections. Current methods for evaluation of network protocols include network simulators and mobile testbeds but suffer from inaccurate models of physical interactions and time consuming network programming. Our work aims to create an easy to use, highly configurable software framework which will allow for the quick implementation of network experiments on mobile devices utilizing features such as packet interception, modification and injection. Furthermore, our results indicate that the framework allows researchers to create experimental prototypes with minimal performance overhead.
CONTENTS

FIGURES .................................................. iii

TABLES ................................................... v

CHAPTER 1: INTRODUCTION .............................. 1

CHAPTER 2: RELATED WORK .............................. 4
  2.1 Ad-hoc networking .................................. 4
  2.2 Network simulators .................................. 6
  2.3 Mobile testbeds ..................................... 8
  2.4 Software routers ................................... 10

CHAPTER 3: ARCHITECTURE ............................. 13
  3.1 Mobile network stack ............................... 14
  3.2 Network access ..................................... 22
  3.3 Modular architecture ............................... 32
  3.4 Logging ............................................. 32
  3.5 Timers .............................................. 34
  3.6 Configuration ...................................... 36
  3.7 Graphical interface ................................. 36

CHAPTER 4: CASE STUDIES .............................. 41
  4.1 Case study 1 - OLSR (Optimized Link State Routing) protocol .. 41
  4.2 Case study 2 - Link quality emulation .................. 42
  4.3 Performance evaluation ............................. 45

CHAPTER 5: CONCLUSION ............................... 51

BIBLIOGRAPHY ............................................ 53
# FIGURES

2.1 Infrastructure network topology ............................................... 5
2.2 Ad-hoc network topology ....................................................... 5
2.3 Portion of an NS3 configuration script establishing topology ........ 7
2.4 ORBIT node configuration script ............................................. 9
2.5 Click configuration file for counting UDP packets ....................... 11

3.1 Flow of control for packets through the Linux kernel ................. 16
3.2 Socket buffer data structure (sk_buff) .................................... 18
3.3 Simplified diagram of networking components in Linux network stack 23
3.4 Example packet flow with framework enabled ............................. 25
3.5 Diagram of packet compression .............................................. 30
3.6 Example node (CountNode.h and CountNode.cpp) ...................... 33
3.7 An example of defining the debugging level and logging a string to the log file .......................................................... 34
3.8 Example call to create a timer ............................................... 35
3.9 ASPR framework configuration file (Config.xml) which accepts all packets ................................................................. 36
3.10 Main GUI screen to create experiment ...................................... 37
3.11 Module configuration: dragging module to configuration area .......... 38
3.12 Completed module configuration ............................................ 39
3.13 Running the experiment ....................................................... 40

4.1 Processing chain configuration for the delay injection experiment . 44
4.2 Experiment hardware setup .................................................... 45
4.3 File size vs. download time with three different configurations. Time represents the average time it took to download one file of the given file size. .......................................................... 47
4.4 File size vs. normalized overhead. To normalize, the results for both “accept all packets” and “filter tcp / xor” were divided by the time for the associated “no framework” configuration.
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>NETWORK LAYERS PER THE OSI MODEL</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>NETLINK QUEUE PACKET COPY MODES</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>FIELDS PASSED BETWEEN MODULES</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>OLSR NODES</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>MEAN DOWNLOAD TIME, STANDARD DEVIATION AND 95% CONFIDENCE INTERVALS FOR 512 MB FILES.</td>
<td>49</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Cellular mobile devices have become an increasingly large part of society, permeating almost every aspect of life. Over the last decade, the number of mobile subscribers in the United States alone has more than doubled with 2011 seeing the number of mobile subscriptions surpass the number of people in the United States [16]. In addition to the growth in subscribers, the data usage of mobile devices is predicted to grow by over 1000x in the next decade [13]. Provided the continued growth, exclusive usage of cellular networks for data access will become problematic.

Researchers have been searching for ways to distribute communication and data usage across not only the cellular networks but infrastructure and ad-hoc WiFi connections as well [5, 6, 21, 24, 26, 32, 33, 45, 54]. In order to evaluate the performance and effectiveness of new networking methods researchers, including [6, 22, 54], have turned to methods such as simulation to evaluate proposed protocols. While simulation allows for easy evaluation of performance at large scale, current techniques of simulation have proven to be less realistic than physical devices [19]. The problem with a majority of proposed routing protocols is a lack of implementation on physical devices. Instead, protocols are usually tested and evaluated through simulations which have varying results depending on the models used within the simulation software.
An alternative to simulation is the use of network testbeds which have been based on many forms of networking including Ethernet [53], WiFi [14, 53], Zigbee [10, 20, 51], a combination of Ethernet and wireless technologies [53] and a few which even span multiple locations across the planet [12]. Typically, a testbed consists of multiple physical devices in a given layout that represent a physical scenario in which to evaluate a protocol or experiment. Although testbeds provide an environment in which to run repeatable experiments, two main problems arise: First, large, general purpose testbeds can be expensive and difficult to setup. Second, most testbeds require the user to fully implement the protocol within the standard operating system running on the testbed devices. Such a task can be a difficult and time consuming ordeal. Although work such as software defined networking [36] has made implementing new routing techniques easier, the main focus is on providing access to the forwarding plane of routers and does not provide access to the network traffic on end-point devices.

With the increasing complexity of general networking stacks in modern computing devices, modifying the way a device handles networking has become a difficult task. A researcher who wants to experiment with the networking stack is required to explore and modify a large body of code in order to complete the desired task. The goal of the presented work is to create a software framework which allows for easy manipulation of packets on an endpoint device used for network experimentation and prototyping, thereby reducing the burden on the researcher.

The framework is built within the Android operating system due to two main reasons: First, Android devices have been growing in popularity [1] which has resulted in devices becoming much more affordable. The release of the Nexus 7 tablet in July 2012 represented an affordable, high performance Android device
with a retail price of $199. Second, the relative openness with regards to development on the Android platform allows for easy access to the kernel and networking stack in order to provide a usable framework for users. Through use of the presented framework a wide variety of network experimentation is possible on mobile devices. An easy to use networking framework coupled with inexpensive mobile devices allow for relative ease in development of a mobile testbed on which it is possible to quickly prototype new ideas.

Our work accomplishes the following main objectives:

- **Provides easy to use access to the mobile networking stack.** The framework provides an easy to use method of modifying and monitoring incoming and outgoing network traffic to enable the implementation of a variety of networking protocols. Any changes made would occur before the packets are received by the local listening applications or transmitted by the device driver. In addition to modification of existing traffic, packets can be injected both as: outgoing packets destined for a remote host as well as incoming to locally listening applications. These features allow for a wide variety of network applications from firewalls to custom packet handling techniques.

- **Allows for modular configuration of basic components.** All features are abstracted away from their operating system level implementation which creates an easy to use framework for developing custom packet handling methods. Compact and simple functions can be created as individual pieces of code which can then be chained together to create more complex methods. In addition, previously developed methods can be re-used for new ideas, thus reducing the development time.
CHAPTER 2

RELATED WORK

Given the large strain on the current infrastructure of mobile networks, mobile carriers and academic researchers have been interested in ways of offloading data to other means of communication resulting in many research projects that have looked into new ways of networking. Specifically, ad-hoc networking has seen a large number of newly proposed networking protocols, which require testing to evaluate the effectiveness and performance as compared to previous work. Currently, there are three main areas of work that have offered ways of doing easily repeatable testing of newly designed network protocols which include network simulators, mobile testbeds and software routers.

2.1 Ad-hoc networking

Ad-hoc networks consist of multiple devices that connect in a decentralized method, without the use of a central access point to orchestrate communication between devices. In traditional infrastructure-based networking as shown in Figure 2.1, all devices communicate with an access point which is responsible for forwarding the packets to the correct destination. Whereas, in ad-hoc networking as shown in Figure 2.2, each node acts as an independent router, taking care of sending and forwarding packets to the correct destination in addition to the basic networking services provided by end-point devices. Such networking methods
allow devices to dynamically form networks between available peers, allowing for either point to point communication or more complex paths through the use of multi-hop routing techniques. With each device responsible for additional routing related work, many projects have looked at different ways to optimize features ranging from networking performance to power efficiency.

With the uncertainty of the wireless environment, the goal of routing protocols is to provide a reliable channel of communication between two devices. As summarized, there has been a plethora of work looking at different routing protocols for use in an ad-hoc network situation with the majority of protocols falling into two main categories: proactive \[\textbf{[11, 27, 37, 40, 42]}\] and reactive \[\textbf{[15, 29, 39, 41, 46]}\].
Proactive routing usually keeps a table, or multiple tables, of the routing information associated with all of the other nodes within the network and chooses a route to the destination before sending the packet. In order to keep all of the tables updated, routing information is exchanged between all of the nodes in different ways depending on the given protocol. While there is an overhead created by keeping track of all of the routing information between all of the nodes within the network, when a packet needs to be sent out, there is no delay and the packet is transmitted immediately.

On the other hand, reactive routing attempts to reduce the initial overhead present in proactive protocols by saving route discovery until a packet needs to be sent. Commonly, a route discovery packet is sent out looking for the destination and once it finds the destination sends back a report to the source on how to find the destination. The original packet then follows the path that was discovered. In this way, there is no overhead to keep track of all of the links within the network, but there is a per-packet delay that occurs while an appropriate path is discovered. The large variety of ad-hoc routing protocols described is just a sample of the wide number of approaches being explored within ad-hoc networking research.

2.2 Network simulators

Network simulators have existed in many forms over the last three decades and include such works as OPNET Modeler [38] which was released in 1986, NS (Network Simulator) [7] which has been under development since 1989, NIST Net [9] which has been around since 1997, and The One simulator [30] which was released in 2007. The main goal of these software applications is to allow for complex network experiments to be setup and network protocols to be defined and
... NodeContainer c;
c.Create (7);
NodeContainer n0n2 = NodeContainer (c.Get (0), c.Get (2));
NodeContainer n1n2 = NodeContainer (c.Get (1), c.Get (2));
NodeContainer n5n6 = NodeContainer (c.Get (5), c.Get (6));
NodeContainer n1n6 = NodeContainer (c.Get (1), c.Get (6));
...

Figure 2.3. Portion of an NS3 configuration script establishing topology

tested. Most simulators have a proprietary format for defining network simulations as well as how each node will interact with the network. A researcher commonly defines both the physical layout and the network protocols of a given experiment. An example of the NS3 configuration file which defines the topology of a network is provided in Figure 2.3. Seven nodes are created and then five NodeContainers are defined which allow communication to take place between all nodes in each container. Different groups of nodes are placed in the containers and the result is a topology with seven nodes and ten links.

Movement of devices is simulated by using either pre-recorded movement traces or algorithmically defined movement models. Examples of movement traces include the Haggle [50] and MobiClique [43] datasets which were released in 2009 and based on bluetooth sightings of devices carried by people in office and conference settings. Another example is a trace of taxi cabs moving through San Francisco collected using GPS receivers in each cab [44] which was also released in 2009. A summary of mobility models is provided in [8] and include random waypoint (RWP) first proposed in [29], random direction [49], and random walk [17].
The accuracy of simulation software is limited by the algorithms which model different constraints of the network. Movement models, RF noise, and hardware anomalies cannot always be predicted by the simulation algorithms, thus giving a skewed view of the performance of a tested protocol. As discussed in [19], simulation is difficult as the complexity of the constraints and the interaction of different factors is hard to model. Thus, while simulations are relatively easy to setup and use to perform experiments, by removing the physical devices involved in the networking experiment, measured performance may not match performance observed in real world scenarios.

2.3 Mobile testbeds

A method for obtaining a more accurate / realistic view of networking performance is to use a mobile network testbed. While network simulators run experiments entirely within software, testbeds attempt to bring repeatable experiments closer to real world scenarios by performing experiments on physical devices. Mobile testbeds are commonly instrumented in ways to allow repeatable movement or emulation of movement patterns while allowing the physical devices to handle the networking in much the same way as if the devices were not in a testbed.

Testbeds which focus on mobility have taken multiple approaches to providing movement in a repeatable and testable way. An example configuration used in the ORBIT testbed to define a simple node can be seen in Figure 2.4. While testbeds make use of the same types of mobility models and movement traces used in network simulators different testbeds have approached the problem in different ways. Research works such as [14,47,48] provide methods to emulate movement to devices through modification of wireless signals. In contrast, [28] places mo-
bile devices on robots which then move in the predefined pattern removing any problems that may occur from movement emulation. Some testbeds, such as [52], forego movement altogether and provide a fixed layout deployment of sensor nodes which removes the constraints of movement and focuses solely on allowing easily deployable mobile experiments to run across the set of devices.

Software defined networking methods such as OpenFlow [36] have focused on providing access to the forwarding plane of network switches and routers. Research such as [18] has shown how OpenFlow could operate on end-devices in a mesh network by providing an additional OpenFlow network stack that operates along side the standard Linux network stack. A researcher who wanted to create a new protocol that was dependent on the content within traditional network packets would not be able to utilize the secondary OpenFlow stack for such a purpose as
the secondary stack is separated from the standard Linux network stack.

While mobile testbeds reduce the overhead associated with coordinating a large number of networked devices physically interacting, the software to implement the tested protocol is left to the researcher to develop within the standard operating system environment. Implementing new routing protocols on end devices can be time consuming and difficult which prevents quick prototyping and testing. Instead, researchers spend a large portion of time developing applications to run on mobile devices in addition to the time associated with setting up an experiment on a testbed.

2.4 Software routers

Due to the complex nature associated with implementing routing methods, software routers, including such work as Click [31] and XORP [25], were designed as an alternative to classic hardware based routers. The major goal of software routers is to provide an easily configurable, dynamic router for experimentation and industry use. Hardware routers perform predefined routing well but are not easily modified or extended. Software routers allow for custom routing software to be defined which enables researchers and industry users alike to customize the way the routers are working within a network environment. The Click Modular Router employs a modular system which allows multiple routing “elements” to be tied together to create a custom routing system. Figure 2.5 shows a sample configuration file for the Click router which counts UDP packets. Unlike the NS simulator and ORBIT testbed, the Click configuration file is only defining how packets are handled. Two classifiers are defined, one which filters IP packets and another which filters UDP packets destined to port 1234. The two classifiers are
chained together to funnel UDP packets to a counter which then keeps a running count of UDP packets that pass through.

Software routers have been designed to fully emulate any and every possible routing protocol in the most efficient manner possible in order to perform as stand alone routers in production environments. Our work aims to provide a quick and easy prototyping platform that users can employ to test out an idea or troubleshoot a networking problem in a given application. Once a prototype is created and evaluated, additional work can concentrate on finding the most efficient ways to implement the idea.

Summary

Although simulation, mobile testbeds and software-based routers offer solutions for many scenarios, all three have related challenges. First, while simulators address the need for large scale evaluation that may not always be available in
real world scenarios, simulations are constrained by the models used. Therefore, simulated results may not accurately represent real world scenarios. Second, mobile testbeds allow for experimentation on real world devices but, without an abstracted framework for network programming, can be difficult and time consuming to program. Third, while software routers provide an abstraction to network stack programming, the routers have been engineered for large scale deployments and thus result in a large code base. Hence, our framework fills a need by allowing researchers to quickly and easily program for the network stack while also allowing for running experiments on real world, inexpensive hardware.
CHAPTER 3

ARCHITECTURE

The goal of our work is to create an easy to use, highly configurable software framework which will allow for quick implementation of network experiments on mobile devices. In addition to providing features such as packet interception, modification, and injection the framework will reduce the burden of development time on the researchers. To this end, we created the Android Software Packet Router (ASPR) package which makes use of stock Linux functionality including both the Netfilter framework \cite{netfilter} and virtual network kernel devices. The framework allows users to quickly and easily create complex network experiments by combining simple, modular pieces of software called Nodes into more complex packet processing chains. Packets are intercepted within the network stack and passed through processing chains which can choose to modify the data before returning the packets to the network stack or deciding to drop the packets altogether.

In addition to handling packets, the framework provides a built in logging system to record data from individual nodes for analysis during and after the execution of network experiments. Finally, users can create applications with functions that are triggered in one of two ways: First, processing can be defined to act on packets when the packets either arrive from the network or are ready for departure out to the network. Second, processing functions can be defined to be called based on the expiration of timers. Through the use of these features, almost
any network protocol or experiment can be quickly and easily implemented and run on a physical device.

The chapter first describes how network traffic travels through the network stack on Android devices in order to provide a reference as to how our framework fits into the existing networking structure. Following the in-depth look at network fundamentals, an overview of the framework architecture is presented in detail. Furthermore, a graphical user interface was developed for easy configuration and is explained in more detail.

3.1 Mobile network stack

Android is a popular mobile operating system [1] powered by the Linux kernel, resulting in the majority of network functionality being drawn from the Linux network stack. Network communication has been abstracted into multiple different layers as defined by the OSI (Open Systems Interconnection) model as detailed in Table 3.1 and therefore the networking stack has multiple locations with which to interact. Figure 3.1 documents the flow of packets through the network stack in intricate detail in order to provide a detailed context in which the ASPR framework is functioning. The following overview will identify pieces of software that could be modified to accommodate the framework.

**Physical and data link layers**

Physical and data link layers make up the first two layers in the OSI networking model. These layers are responsible for physically transporting the data across the networking medium and taking care of communication between devices on the local area network. Network cards operate within these layers, utilizing the device driver associated with the network card. Here, data units called frames are
<table>
<thead>
<tr>
<th><strong>Application</strong></th>
<th>Interfaces with applications that implement communications.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Presentation</strong></td>
<td>Responsible for formatting data for the application layer.</td>
</tr>
<tr>
<td><strong>Session</strong></td>
<td>Takes care of establishing sessions between endpoints.</td>
</tr>
<tr>
<td><strong>Transport</strong></td>
<td>Responsible for network connections between endpoints and taking care of flow-control, reliability and error control.</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>Provides a means for packet forwarding between devices and multiple networks.</td>
</tr>
<tr>
<td><strong>Data Link</strong></td>
<td>Takes care of communication between devices on the local area network as well as physical addressing.</td>
</tr>
<tr>
<td><strong>Physical</strong></td>
<td>Responsible for physically transporting the data across the networking medium.</td>
</tr>
</tbody>
</table>
Figure 3.1. Flow of control for packets through the Linux kernel
transmitted between nodes in the local network which involves the usage of MAC (media access control), or hardware based, addresses which identify each of the network devices and allow communication between devices in the local network. Also, the device driver is responsible for detecting and handling any collisions that may occur on the transmission medium.

Hardware interrupts initiated by the device driver are caused by different events, including when a frame is received, and are handled by a special function within the device driver. When a frame is received, the device driver will allocate buffer space in memory represented as a socket buffer (sk_buff) structure and copy the frame into the buffer. After copying the data into the buffer, parameters in the socket buffer such as the layer three protocol (e.g. IP, ARP) values are initialized so the correct protocol handler can be called as control passes higher up the stack.

Network data (frames, packets, etc) is represented within the network stack through the use of the socket buffer data structure. Figure 3.2 gives an overview of some of the more important fields in the socket buffer data structure. An important set of fields include (transport_header, mac_header, network_header, data_len, mac_len, hdr_len) which represent data associated with the protocol headers encountered at each level of the network stack. The main reason for usage of the structure is it is more efficient to append data to the structure as it passes between layers than it is to copy the entire packet between each layer. As a packet travels down the network stack, each layer appends the header associated with that layer. In the reverse direction, instead of removing headers, a pointer to the payload data of the packet (@data) is moved to point to the appropriate position within the packet. For example, as a packet moves from the data link
* @next, @prev: Linked list pointers to other socket buffers
* @sk: Socket we are owned by
* @tstamp: Time we arrived
* @dev: Device we arrived on/are leaving by
* @transport_header: Transport layer header
* @network_header: Network layer header
* @mac_header: Link layer header
* @skb_dst: Destination entry
* @cb: Control buffer, for use by every layer for private vars
* @len: Length of actual data
* @data_len: Data length
* @mac_len: Length of link layer header
* @hdr_len: Writable header length of cloned skb
* @csum: Checksum (must include start/offset pair)
* @csum_start: Offset from head where checksumming should start
* @csum_offset: Offset where checksum should be stored
* @fclone: skb_buff clone status
* @priority: Packet queueing priority
* @protocol: Packet protocol from driver
* @truesize: Buffer size
* @head: Head of buffer
* @data: Data head pointer
* @tail: Tail pointer
* @end: End pointer

Figure 3.2. Socket buffer data structure (sk_buff)

layer to the network layer, the payload pointer moves from the initial position at
the start of the MAC header to the start of the network_header data field.

Other fields in the socket buffer include layout variables. The Linux kernel
keeps a doubly linked list of socket buffers and each socket buffer keeps a few
variables that allow it to quickly search through the list and find the beginning
of the list. Other variables include fields that record timestamps, the device the
packet was received on or about to be transmitted out on. Each layer stores
information in the socket buffer for private use as well. For example, at the IP
layer, information about fragmentation is stored in the private space, while TCP uses it to store information about segments.

Once the socket buffer has been setup the driver issues a soft interrupt request which is handled by the kernel through a pre-registered interrupt handler \texttt{(netif\_receive\_skb)} which resides in the operating system in kernel space. The interrupt handler marks the beginning of the network layer processing code.

**Network layer**

The network layer provides a means for packet forwarding between devices and multiple networks. The forwarding is done by providing host based addresses and routing support, mainly through the IP protocol. In addition to routing support and decisions, the network layer section of code is also where the Netfilter \footnote{\url{https://netfilter.org/}} framework has access to packets moving through the device in multiple locations.

Layer three protocol handlers can register with the interrupt handler to be notified when packets with a given protocol value are received by the interrupt handler \texttt{(netif\_receive\_skb)} which then passes copies of the packet to registered protocol handlers. In addition, network bridging is handled within the network layer as well, allowing packets to travel between two network devices that connect two separate networks together. Again, \texttt{netif\_receive\_skb} is responsible for passing a copy of the socket buffer off if bridging has been enabled on the receiving device.

The remaining packet processing throughout the rest of the section follows a similar pattern. First, a small handler takes care of running sanity checks on the packet before switching control over to the Netfilter framework. After passing the approval of registered Netfilter hooks, the packet moves on to a larger handler that takes care of more in-depth processing of the packet.
As IP is a common layer three protocol, the operating system takes care of registering `ip_rcv` as a layer three handler that is interested in any packets of the IP protocol. While it is common to only have a single protocol handler registered for each protocol, there are edge cases where multiple handlers may be registered (e.g. IPSec). The initial `ip_rcv` function is primarily responsible for performing sanity checks on the packet such as ensuring the packet is addressed for the current host, the entire packet was copied into the buffer, and whether or not a local copy needs to be made of the packet if other copies exist.

Once the checks have been performed, control is passed to Netfilter through the `NF_IP_PRE_ROUTING` hook point. All callback functions that are registered with the hook point can make a decision to either, accept, drop or steal a packet. The difference between dropping the packet and “stealing” the packet is that when a packet is stolen the socket buffer associated with that packet is left intact whereas dropping the packet causes the kernel to take care of deleting the buffer and reallocating the memory space.

After Netfilter is finished with the packet, the larger handler takes care of checking if the packet is destined for the local host or should be forwarded, updating the QoS statistics for the device, and processing any IP options which exist. Packet forwarding is handled via `ip_forward` which again performs a few sanity checks before passing control to Netfilter through the `NF_IP_FORWARDING` hook point. Finally, `ip_forward_finish` takes care of setting the packet up to be forwarded and finally hands the packet over to `ip_output` which will be described later.

Finally, given that the packet was destined for the local host, `ip_rcv_finish` hands control over to `ip_local_deliver` and `ip_local_deliver_finish`. Net-
filter has one final hook here before the packet moves on to the layer four handlers through the NF_IP_LOCAL_IN hook point. The layer four handlers process the respective headers and hand the packet off to the correct application / service running on the given port. In addition to layer four handlers, ip_local_deliver_finish takes care of raw ip sockets that may be interested in handling the packets directly at the IP layer.

In the opposite direction, packets are created locally and travel down the network stack and are transmitted across the network. Packets can either be generated through sockets in applications, or through raw IP sockets and each method has a different way of initially sending the packet depending on whether features such as fragmentation or multicast need to be considered. The family of methods are all responsible for setting up the IP header and associated values in the socket buffer and end by calling the NF_IP_LOCAL_OUT netfilter hook point which passes control on to dst_output if necessary.

For fragmented packets, the hook point is only called once for all of the fragments instead of not once for each fragment of the packet. The dst_output function passes control to ip_output and between both functions takes care of routing lookups and decisions about how packets are to be transmitted from the device including multicast and fragmentation decisions. ip_output is also the location where packets that were received and need to be forwarded join up with the transmission chain of events. Packets are passed on to ip_finish_output which takes care of calling the NF_IP_POST_ROUTING hook point which is the last chance Netfilter has to touch a packet before it is transmitted.

If accepted, control is passed to ip_finish_output2 which interfaces with the neighboring subsystem (ARP). The neighboring subsystem is responsible for
translating the layer three address (IP address) into the appropriate layer two (MAC) address, and is mainly accomplished through the Address Resolution Protocol (ARP). Packets are then queued to be transmitted through dev_queue_xmit which interfaces with the traffic control subsystem before finally passing the packets off to the device driver. The driver is responsible for wrapping the packet with a frame and finally transmitting the data out over the specific network medium.

3.2 Network access

Given the goal to have complete control over packets traveling into and out of the device, several features in the Linux kernel make the goal much easier. Options such as libPcap or a virtual kernel network device (TUN/TAP) allow a user to design an application that can observe packets traversing through an associated real device. Although, the options do not allow for modification of packets or the ability to prevent the packets from continuing on to the listening application or onto the network. The limitation is due to the way that libpcap accesses the network data through the netif_receive_skb method which creates a copy of the socket buffer structure to hand off and the original version continues along the path unmodified and uninterrupted. The virtual network devices require modification of the routing table and correctly establishing bridging between network devices in order to function correctly.

The Netfilter framework is a firewall system which provides a set of hooks into the kernel network stack and allows firewall applications to observe, modify and make decisions on the fate of each packet that passes through the stack. A summary of the Netfilter hook points is presented in Figure 3.3. The functionality is provided by allowing kernel modules to register callbacks with the hook which
Figure 3.3. Simplified diagram of networking components in Linux network stack
are called any time a packet travels through the hook point. The callback function is able to return a decision on what to do with the packet before it continues up the stack (e.g. accept, drop, steal, etc). Additionally, if modifications are made to the socket buffer, these modifications will be passed along if the packet is accepted. Since buffers can be shared between multiple handlers, a check must first be performed to see if the packet has been shared and if so, a copy must be made and that copy will be what is passed along the standard processing chain.

Netfilter also allows for the packets to be queued up to user-space through the NetLink [2] infrastructure which provides an IPC (inter-process communication) mechanism for communication between kernel and user processes. The NetLink queue API (i.e. libnfnetfilter_queue) allows a kernel module to queue packets up to a user-space application which then returns a decision about the future of each packet and also is able to perform modifications to a packet. The API provides a 16-bit queue ID which allows for 65,536 simultaneous netfilter queues communicating between the kernel space and the user space. The queue system allows the kernel to continue processing packets and performing other tasks while the user-space application makes the decisions and modifications necessary. Additionally, the queue also allows for most of the processing of the application to be done in user space which lessens the likelihood of a kernel panic and allows for a more stable system.

Kernel vs. user space

The Linux kernel provides a method for dynamically loading new features / code into kernel space through the use of kernel modules. The modules allow code that requires kernel level execution to be dynamically loaded into the kernel. The Netfilter system allows kernel modules to register callback functions with hook
Figure 3.4. Example packet flow with framework enabled

points in the kernel network processing system. In order for our application to interact with the Netfilter system, a kernel module was created to register such a callback function. Specifically, five different modules were created, each with a different hook point which allows packets to be intercepted at any of the five hook points and pulled into our processing system. The kernel modules are dynamically loadable into the kernel at runtime so the software can be turned on and off without restarting or recompiling the system which allows for easy experimentation.

Consider a simple example in which the framework is enabled and set to accept all packets as depicted in Figure 3.4. Communication between the kernel module and the main application (ASPR.o) which runs in user-space is done through the
Netfilter NetLink infrastructure. The infrastructure allows packets to be queued in kernel space and then dequeued in a user-space application. The function that dequeues the packets then returns the accept decision and the packet is passed back to kernel space where it continues up the network stack and on to a listening application. On the return trip, a similar scenario unfolds before the packet is transmitted across the network.

**The Netfilter queue (NF_QUEUE)**

Instead of accepting or dropping a packet, the kernel module can also queue the packet to user space. The queue allows a user space application to listen and operate on packets after the arrival in the queue. Once the user space application is done with the packet, it must set a verdict on the packet. That is, it needs to tell the kernel module that queued the packet whether the packets should be accepted or dropped. There are other, more complicated options as well, such as re-queueing. The NetLink queue subsystem is a library that is produced by Netfilter and is present in most Linux operating systems. The queue system used to be called ipq, but that system has since been deprecated and replaced by Netfilter NetLink. In order to build applications that support NFNetlink, the libraries need to downloaded, included, and linked against in the developed applications. The kernel module does not need to access these libraries but the user space application does in order to access the queues and process the packets.

To access the queues, the kernel module returns the value NF_QUEUE instead of NF_ACCEPT or NF_DROP. The default queue id for queueing the packet is 0. If multiple queues or a different queue id number is desired the queue id is returned in the high 16 bits of the returned NF_QUEUE 32-bit value.

These queues are then accessed in the user space by creating a queue connection
TABLE 3.2

NETLINK QUEUE PACKET COPY MODES

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFQNL_COPY_NONE</td>
<td>No packet data will be copied.</td>
</tr>
<tr>
<td>NFQNL_COPY_META</td>
<td>Only packet headers are copied.</td>
</tr>
<tr>
<td>NFQNL_COPY_PACKET</td>
<td>The entire packet is copied.</td>
</tr>
</tbody>
</table>

handler, a queue handler and a callback function. A connection to a queue is created, and a callback function is registered with the queue. Then, the application makes a call to “recv” to receive a packet from the queue connection. When a packet is received, it is passed onto “nfq_handle_packet” along with a reference to the callback function.

The queue connection handler also allows for a packet copy mode to be set, which determines how much data from the received packets is copied up to user space for access by user level applications. All of the options are outlined in Table 3.2 although currently ASPR exclusively uses the NFQNL_COPY_PACKET mode in order to provide access to all data within a given packet to the processing modules. Depending on the needs of a given application, the copy mode that is used could be adjusted to improve performance.

The callback function is then called for each packet and the data specified is passed as well. The function receives the parameters outlined in Table 3.3. Inside the callback, different functions are used to grab the packet payload, which starts with the IP header, but may also depend on the amount of data copied for the packet. Information is also grabbed from the nfnetlink headers as well in order to provide additional context to the received packet. One important value that
TABLE 3.3
FIELDS PASSED BETWEEN MODULES

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nfq_q_handle *myQueue</td>
<td>Represents the queue handler</td>
</tr>
<tr>
<td>struct nfgenmsg *msg</td>
<td>Represents a struct containing the AF_XXX family, nfnetlink version and resource id</td>
</tr>
<tr>
<td>nfq_data *pkt</td>
<td>Netlink packet data handle, used by the informational functions to access the data</td>
</tr>
<tr>
<td>void *cbData</td>
<td>Custom data that can be passed to the callback function when the queue is created.</td>
</tr>
</tbody>
</table>

needs to be retrieved is the id of the packet within the queue. The packet id is used to notify the queue when the packet needs to be accepted or dropped.

Queue performance

An interesting question that arises is how the queue would perform if packets were allowed to be accepted / dropped in an out of order fashion. Such a case may occur if a packet was to be left in the queue while artificial delay was added to the delivery of the packet. According to the documentation of the dequeue() function which removes packets from the queue, the function approaches a runtime of O(n) as it searches for a packet within the queue in order to remove it from the queue. Thus, if the functionality of the framework is such that packets could be passed along to another module while the processing of another packet is started before the processing on the first packet finished, then the performance of the queue would start to decrease.

The queue size has a default maximum of 1024 items, and after it is full,
packets start to get dropped. Statistics about the performance of the queue can be checked using

```
cat /proc/net/netfilter/nfnetlink_queue
```

which outputs fields including Queue ID, the number of currently queued packets and the number of packets dropped due to failures. Hence, the software was designed such that individual nodes are passed the packet and control. A packet traverses a chain of nodes until a decision about the packet is reached and then control is returned back up the chain until it reaches the main function which is receiving packets off of the queue. The decision is passed along with any modifications to the packet back down to kernel space through the queue.

**Packet injection**

While the Netfilter system allows for modification and interception of packets, it does not afford a way to inject a new packet from scratch. Packet injection is a beneficial feature to provide in our framework as it allows for the sending of control plane information outside of the normal operation of networking on the host system. That is to say, while networking can operate as normal, injecting control plane packets allows for routing control operations to be performed at the user-space level.

Another useful example of injection is the compression of multiple packets which are all headed to the same destination and is outlined in Figure 3.5. In the described scenario, multiple packets are compressed on one end, sent as a single packet across the network and then decompressed on the other end. To perform packet compression, the receiving end would need to break apart the single packet and inject multiple small packets to a listening application.
Packets are condensed to a single super packet.

Super packet is sent over the network.

Upon arriving at the destination, the super packet is broken up into the original packets which are then injected up to the listening application.

Figure 3.5. Diagram of packet compression
To create the injection functionality, options such as raw sockets and virtual tun/tap devices were investigated. While raw sockets work well for injecting network traffic that is destined to remote devices, any attempt at injecting packets that should be received by some listening application on the local host do not work. The reason is due to how raw sockets pass off packets to the networking stack. Referring again to Figure 3.1 when raw sockets inject packets, the outgoing packets have no way to be handed off to the received packet processing code and on to a locally listening application. Since the device driver does not re-route packets back up the network stack using raw sockets is out of the question.

Therefore, virtual network kernel devices (TUN/TAP) were used which are virtual network adapters and allow a software application to write packets to and read packets from the device. When a packet is written to a virtual device, it appears to the operating system that the packet was just received from the network, not the local host, and is processed accordingly. Hence, packets that were saved from previous communication can be re-injected through the use of a tun/tap device. The packets are then queued up to be read by an application from the virtual device.

A feature of the Linux kernel that affects the performance of packet injection is that of “reverse path filtering”. The filtering option, if enabled, cases the kernel to drop any packets that arrive on a network device if the src ip address cannot be reached through the same device the packets are received on. The reverse path filtering feature is designed to prevent IP spoofing by ensuring a symmetric communication channel. For the purposes of our work the filtering option must be disabled as the goal of packet injection is similar to that of IP spoofing.
3.3 Modular architecture

Modularity is provided through an abstraction of processing software called a **Node**. An example of a node which counts packets is provided in Figure 3.6. All **Nodes** consists of three main methods: First is the **input** which handles input from previous nodes in the chain. Second is the **processPackets** method which performs modifications or other operations based on the data within the packet. Third is the **output** function which transfers control to another **Node** by making a return call on the **input** method of the next node in the chain.

Developers creating their own nodes need to only modify the **processPackets** as the other two functions perform the same task for every node. **Nodes** are stateful, that is, all nodes are initialized at the start of the software and can keep state throughout the running of the software. Currently, there is no way for nodes to share information between each other. To limit the complexity of configuring processing chains, nodes have been limited to two distinct output slots. The limitation forces nodes to perform simple tasks and push packets along one of two outputs which is similar to the basic decision of accepting to dropping a packet.

3.4 Logging

A large part of networking experiments involve the measurement of performance and different aspects of the networking protocol being tested. In order to allow easy measurement of a multitude of aspects, the framework was designed with a logging feature. Every node is configured to be able to log any data it sees fit to a specified file. This creates a file for each module currently active in the framework. As data needs to be logged, it is written to file.

When a node is created, it creates a log file specific to that node. Thus, a node
Figure 3.6. Example node (CountNode.h and CountNode.cpp)
Figure 3.7. An example of defining the debugging level and logging a string to the log file

which counts passing TCP packets is able to log a timestamped count to a log file allowing easy access to the desired specific data. After the experiment is finished, all data files can be retrieved and examined. In addition to post-experimentation analysis, since the files are written to throughout the experiment, the files can be used as a status report as well. This would allow a researcher to observe the current status of each node throughout the experiment.

The framework also allows for a debug level to be defined which controls how verbose the logging output is. A higher debug level indicates a more verbose logging mode. Figure 3.7 shows an example of defining both the debugging level and making a call to the \texttt{log} function. While the current implementation requires individual files to be examined, future work could address a more centralized method for processing the logs.

3.5 Timers

While the previously described parts of the framework allow for networking tasks to be performed on individual packets, such tasks can only be triggered when packets arrive at the networking stack. As can be seen from previously described methods in the ad-hoc networking related work section, certain tasks may need to be performed independently of standard networking traffic. For example, in a

```c++
int Logging::debugLevel = 3;
Logging::log("logs/UdpCountLog","ASPR.cpp","Packet Count: 10");
```
link state routing protocol, messages related to advertising links need to be sent at a pre-specified interval, which would require a timer to keep track of.

To provide this functionality, timers were created which are based on POSIX timers, which create a custom thread to handle the timer callback any time the timer expires. A single POSIX timer is created as part of the BaseTimer object. All timers used within the framework are part of a queue of timers contained within BaseTimer which follows the singleton pattern, allowing all references to act on the single base timer. The base timer has a fine granularity expiration time and decrements the timer value in all of the timers present in the queue. In order to allow timers to perform some function, a callback function must be provided when scheduling the timer as shown in Figure 3.8. Once the timer in the queue reaches the expiration time, the callback function is called. In addition to the callback function, a data pointer can be optionally passed if data needs to be shared with the callback function. For example, if a timer is scheduled with the goal of adding 50 milliseconds delay to a packet, a pointer to the saved packet must be passed along with a function that injects that packet when the timer is expired.
3.6 Configuration

To enable easy configuration of packet processing chains, an XML formatted configuration file is used which allows developers to string together nodes in any arbitrary way. An example of the XML configuration file is shown in Figure 3.9.

Once a node is created within the framework, it is added to a growing library of nodes that can then be referenced from within the configuration files. The library allows developers to create new nodes which can be shared with anyone who is using the ASPR framework. The modular nature of the configuration system allows for quick and easy prototyping of new networking software as well which reduces the bootstrap time of new experiments.

3.7 Graphical interface

A graphical user interface was created to allow easy configuration of the modules. While the framework makes it easy for a user to quickly test their ideas,
Figure 3.10. Main GUI screen to create experiment

execution must be performed on the command line interface (CLI). Experiments that needed to use multiple devices would require the researcher to manually run the experimental software on each device or to use SSH to login to each device and start the experiment. To remedy this, an Android GUI application was created that allows for the configuration and running of the experiments across multiple devices.

Figure 3.10 displays the main features of the GUI, allowing a user to create and run an experiment. The main menu is set up as a nested menu, with multiple layers of menus extending from left to right with columns of buttons representing each layer of menu. In the given figure, “Create a New Experiment” has been pressed and the second layer of menus has expanded towards the right allowing for modules and data logging to be enabled and finally for the experiment to be launched.
Figure 3.11. Module configuration: dragging module to configuration area

The GUI makes configuration of the processing chain of nodes even easier by providing a drag and drop interface as demonstrated in Figure 3.11. A library of nodes is created at runtime and displayed to the user in the leftmost column, allowing the nodes to drug into the desired location on the ride side of the screen with available targets appearing as needed. When a node is picked up, dotted lines appear showing the available locations to be dropped with red lines showing the connections between the different nodes. A example of a completed, more complex configuration is shown in Figure 3.12. The example configuration counts both TCP packets and non-TCP packets while dropping all non-TCP packets.

The GUI also makes use of push notification messages to control multiple devices within the experiment. Devices are listed in the experiment by IP address and requires that each device have the GUI application turned on. Once all
devices are running the GUI application, only one device needs to be used to define node configuration. The push notification messages are used to distribute the `config.xml` file to all the other devices as well as to send control messages. The control messages consist of start and stop messages which enable and disable the framework respectively.

Finally, when an experiment is launched, the activity depicted in Figure 3.13 is shown. There are buttons which allow for starting and stopping the experiment as well as a text box which displays the output of the framework. If modules are configured to output text to the screen, it will be displayed here. For example, in the figure provided, all packets are printed out in hex, acting as a packet trace of sorts.

**Summary**

Overall the provided framework allows for users to easily create a variety of net-
work experiments in a modular fashion by operating in the middle of the existing mobile networking stack. On going network traffic can be observed and modified as needed with new traffic generation available through packet injection and timer techniques. Also, with the addition of the graphical interface, multi-device experiments are much easier to execute. Given the framework, the following chapter will demonstrate two case studies of how to apply the framework in addition to a performance analysis.
In order to demonstrate the capabilities and functionality of the presented framework, two case studies are presented. The first case study shows how a newly proposed routing protocol may be implemented and evaluated using the framework. The second case study considers emulating network performance anomalies, in this case modifying link qualities through use of the framework. Finally, the performance of the framework itself is evaluated and the associated overheads associated with the framework are presented. Overall, the case studies and performance evaluation demonstrates that the framework is easy to use, versatile, and results in minimal overhead cost.

4.1 Case study 1 - OLSR (Optimized Link State Routing) protocol

This case study considers a researcher who wants to prototype a simple link-state routing algorithm. What follows is an example of how to implement an algorithm based on the Optimized Link State Routing protocol (OLSR) in the presented framework. The OLSR protocol has two main features: Broadcasting HELLO messages and distributing link lists throughout the network. The purpose of the HELLO messages is to indicate the node’s presence to all surrounding neighbors. These messages allow nodes to discover their one-hop neighbors. In
addition to the hello messages are the neighbor list messages. All nodes in the network send out lists of all of their neighbors. These messages are flooded through the network and allow nodes to establish a topology of the network and maintaining a routing table that lists the best path for all destinations within the network.

To implement the OLSR protocol in the framework, five nodes need to be created and would perform the tasks specified in Table 4.1. Each node performs a simple task and either passes the packets onto another node or returns the accept or drop decision. Normal network traffic from applications would be accepted while routing related messages such as HELLO messages and neighbor lists would be processed by the framework and dropped. In addition to the five nodes, a timer is used to schedule the periodic sending of HELLO messages. With these steps, a basic implementation of OLSR can be implemented with minimal coding effort.

4.2 Case study 2 - Link quality emulation

In a situation where a researcher does not have access to a mobile testbed which can discreetly modify link qualities of different nodes, the researcher may want to evaluate the effect of different delays on a given networking method. In this case, the provided framework can be used to inject artificial delay into the network. In order to accomplish this goal, a module would be created which caches each packet and re-injects the packet after the desired amount of delay.

In order to cache the packets, a priority queue is defined in the module which will hold all of the packets that are to be cached. The priority field of the queue is represented by the amount of delay, with lower delay correlating to a higher priority. After each fringing of the timer, the delay field is decremented, which those packets reaching zero being re-injected back into the network. This re-
<table>
<thead>
<tr>
<th>NODE NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterTrafficNode</td>
<td>Sends all outgoing traffic through the WrapNode and all incoming traffic through FilterHelloNode.</td>
</tr>
<tr>
<td>FilterHelloNode</td>
<td>Filter out HELLO messages which are sent to UpdateNeighborNode and send all other traffic through FilterNeighborNode.</td>
</tr>
<tr>
<td>UpdateNeighborNode</td>
<td>Updates the neighbor list and then broadcasts a NeighborList message using packet injection.</td>
</tr>
<tr>
<td>WrapNode</td>
<td>Wrap all outgoing packets with a custom tcp header. The custom header indicates the next hop destination, while keeping the original tcp header wrapped within the packet.</td>
</tr>
<tr>
<td>UnwrapNode</td>
<td>Unwrap all incoming packets, determine if the packets are locally destined or should be forwarded. Packets which are destined for local applications are allowed to continue up the stack without the custom header. Packets that need to be forwarded are re-wrapped with a new custom header and injected back into the network.</td>
</tr>
</tbody>
</table>
injection utilizes the `injectPacket` function which uses a virtual network device to inject the packet back into the network.

In order to avoid additional delay being added to packets which have already been delayed, a custom wrapper is applied to each packet before it is re-injected to the network. Then, a module which checks for the presence of this custom wrapper is inserted before the delay module in the packet processing chain. All packets with the custom wrapper are unwrapped and allowed to continue through the network stack, whereas all other packets are added to the delay queue. Figure 4.1 represents the processing chain of the given modules showing the simplicity in configuring such a setup once the nodes have been created.
4.3 Performance evaluation

While some performance is sacrificed in order to allow an easier / simpler method of implementing a network protocol, this case study aims to show that minimal harm is done and thus the decrease in performance can be justified. For this study, five different text files ranging in size from 32MB to 512MB incrementing in powers of two were transferred from a web server to a tablet across a local wireless network. The web server Apache 2.4 running on an IBM Thinkpad X41 laptop using Ubuntu 10.04 as the operating system. The tablet was a Samsung Galaxy Tab 10.1 using a stock Android Honeycomb rom which had been rooted and had the BusyBox tools installed.

The physical hardware setup, as depicted in Figure 4.2, consisted of a Netgear WGR614v8 wireless router, the laptop connected via ethernet and the tablet con-
nected wirelessly. In order to reduce wireless interference, other wireless networks were observed and the experimental network was setup on channel that did not overlap the other networks.

The following three software configurations were tested:

• No framework. This configuration did not have the framework running and used only the stock features.

• Accept all packets. This configuration had the framework and kernel modules loaded, but simply accepted a packet as soon as it arrived.

• TCP filter and xor. This configuration had the framework running, filtered out tcp packets and computed a running xor of all the bytes in the packet. Both tcp and non-tcp packets were accepted.

For each of the software configurations, a bash script was used to download each of the files from the web sever 25 times. The number of downloads (i.e. 25) was chosen due to the fact that initial experiments indicated a full run of the experiment with 25 downloads would take eight hours to run. The version of \texttt{wget} included in the BusyBox toolset was used to download the files and a five second delay was used between each successive file download. Each of the calls to \texttt{wget} was measured with the \texttt{time} command to record the execution time of the download. The three software configurations combined with the five files resulted in 15 configurations, which all consisted of 25 separate downloads.

Average download times for each configuration are displayed in Figure 4.3. All three configurations follow a similar pattern, with download time increasing as file size increases. Additionally, the “filter tcp and xor” configuration was always
Figure 4.3. File size vs. download time with three different configurations. Time represents the average time it took to download one file of the given file size.
Figure 4.4. File size vs. normalized overhead. To normalize, the results for both “accept all packets” and “filter tcp / xor” were divided by the time for the associated “no framework” configuration.
TABLE 4.2

MEAN DOWNLOAD TIME, STANDARD DEVIATION AND 95% CONFIDENCE INTERVALS FOR 512 MB FILES.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Mean</th>
<th>Overhead</th>
<th>Std Dev</th>
<th>95% Conf. Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Framework</td>
<td>186.818s</td>
<td>0</td>
<td>0.817s</td>
<td>(186.498, 187.138)</td>
</tr>
<tr>
<td>Accept All Packets</td>
<td>191.344s</td>
<td>4.526s</td>
<td>3.120</td>
<td>(190.121, 192.568)</td>
</tr>
<tr>
<td>Filter TCP / XOR</td>
<td>195.016s</td>
<td>8.198s</td>
<td>1.956</td>
<td>(194.249, 195.783)</td>
</tr>
</tbody>
</table>

longer than the “accept all packets” configuration which was always longer than the “no framework” configuration. For the configurations with the framework enabled (i.e Accept all packets, filter tcp / xor) as the file sizes, and therefore the number of packets used, increase, the overhead of using the framework also increases.

To explore the relationship between the three configurations closer, Table 4.2 shows the mean, standard deviation and confidence interval for the three configurations with a 512MB file. The simplest form of the framework in which all packets are accepted increased download times on average by 4.526 seconds for a 512MB file while filtering and xor operations more than doubled the overhead to 8.198 seconds. Figure 4.4 shows the overhead normalized by the “no framework” configuration times. The increases for the 512MB file result in a 2.365 and 4.284 percent increase in download times. Additionally, once the file sizes reach 128MB, the overhead appears to level out.

Summary

Overall, the two previous case studies have shown the wide variety of appli-
cations possible with the provided framework. In addition to the applications, the performance evaluation demonstrates the associated increases in processing time for different configurations of the framework software. When examining the normalized results, the tested scenarios increase the processing time by between two and four percent. This overhead is a low cost for the ability to be able to quickly prototype and develop a network protocol or experiment. By saving time, tests can be run quicker and allows for adjustments to be made much quicker, thus justifying the associated costs.
CHAPTER 5

CONCLUSION

This work provides an easy to use framework which allows users to quickly implement a wide variety of network experiments on a physical Android device. While other work has provided multiple methods to test new protocols, the work either overly simplifies networking conditions or does not address the complicated tasks involved in implementing a new network protocol. The presented software framework addresses this problem. For an optimal networking testbed which allows quick implementation and repeatable experiments, the provided framework could be deployed to one of the previously mentioned mobile testbeds. Doing so would offer the best of both worlds, controllable movement and coordination of multiple nodes and a quick and easy way to implement the desired protocol.

Finally, we believe this work represents a useful system and there are three areas that would benefit from future work: First, there may be a few opportunities for improving the performance of the framework, such as using a caching mechanism when logging data to a file or command line. As shown in the performance evaluation, a minimal overhead was introduced when logging data to the command line which could be reduced with further work. Second, to facilitate more complex experiments, a system of communication could be established between packet processing nodes. Such a system would allow the framework to change behavior farther down the processing chain based on observations from other nodes.
For example, different parts of a chain could be disabled if the network traffic reached a high rate. Third, the logging system may benefit from centralizing all of the logs in to a more easily accessed format, whether that is a database or a method of summarizing all logs.
BIBLIOGRAPHY


