Design and Implementation of an Educational Platform for Hosting Virtual Wireless Networks

Ahmed Youssef

Department of Electrical & Computer Engineering
McGill University
Montreal, Canada

August 2015

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Engineering.

2015/08/16

Abstract

With the ubiquity of wireless devices and their diverse and evolving applications, wireless systems are becoming an integral part of Computer Networking courses. Due to the importance of experimentation in the learning process of computer networking concepts, various computer networking toolkits and platforms have been developed that are targeted for educational purposes. However, most of these current toolkits and platforms are focused on wired networks. The toolkits that have wireless networking support do so through network simulation. While simulation is acceptable and often times preferred when dealing with performance evaluation of new algorithms in large-scale scenarios, they do not possess the necessary realism required in educational settings when dealing with wireless networks. Physical platforms are developed to address this issue. However, the current physical platforms fall short of one or more of the design criteria that are critical to their adoption in most educational institutions. The physical platforms are often prohibitively expensive, hard to setup, inflexible, and/or complicated to use. In this thesis, we aim to address these issues.

We describe Wireless GINI, a wireless platform for hosting virtual networks. Wireless GINI allows each virtual network to define its own topology and network configuration, while amortizing costs by sharing the physical infrastructure. The platform also creates mechanisms to integrate commodity wireless devices into a deployed virtual network. Wireless GINI provides a user-friendly interface that makes the physical setup process completely transparent to the user. A centralized server is used to provide this transparency, handle user requests, and automatically provision the shared physical infrastructure. We describe the design and implementation of Wireless GINI and suggest several educational experiments that can be conducted on this new platform. A detailed survey of the existing toolkits and platforms is also provided.

Sommaire

Avec l'omniprésence des appareils sans files et la diversité de leurs applications, les systèmes sans files sont devenus une partie intégrante des réseaux informatiques. Vu l'importance des expériences dans la procédure d'apprentissage des réseaux informatique, plusieurs outils et plateformes ont été développés pour des fins d'apprentissage. Cependant, la plupart de ces outils considère les réseaux filaires. Bien que les simulations soient préférées pour l'évaluation rapide des performances de nouveaux algorithmes, elles ne sont pas assez réalistes pour modéliser les systèmes sans files. Des plateformes physiques sont développées pour remédier à ce problème. Mais ces plateformes sont difficile à manier, et trop cher. Tout au long de cette thèse, nous proposons de résoudre ce problème.

Nous décrivons la plateforme sans file GINI pour les réseaux virtuels. La GINI permet à chaque réseau virtuel de définir sa propre topologie et de configurer son propre réseau, tout en réduisant le coup de développement en partageant l'interface physique. La plateforme crée des mécanismes pour intégrer les appareils sans fil au réseau virtuel. La GINI propose une simple interface qui rend la procédure d'installation complétement transparente à l'utilisateur. Un serveur centralisé est utilisé pour apporter cette transparence, rependre aux demandes des utilisateurs et gérer l'interface physique commune. Nous décrivons le désigne et l'implémentation du GINI et nous suggérons plusieurs expériences qui peuvent être effectuées sur cette plateforme. Nous donnons aussi une description détaillée des outils déjà existant.

Acknowledgments

In the name of God, The most Merciful, The most Compassionate.

I would like to start by thanking God Almighty, the One whom through His infinite blessings achievements are attained.

Although words are not enough to describe the amount of gratitude I hold for my supervisor Prof. Maheswaran, it is only right to attempt to convey my gratitude. I am truly thankful to have Prof. Maheswaran as my supervisor and mentor, as I embarked on this intricately satisfying, humbling, and overall splendid journey at McGill. I would like to sincerely thank Prof. Maheswaran for being a source of inspiration with his insightful knowledge, his amiable character, and his unwavering passion. I will always cherish the great moments we had from the delightful Boston trip to the inspiring discussions.

I am truly indebted to my lovely sister Lamis Youssef for championing the implementation of the local database and XML parser. I greatly appreciate her sincere commitment and hard work even when it meant losing out on enjoying the city of Montreal.

I am grateful to my lab mate and friend Syed Ahmed for his valuable input on several aspects of the Wireless GINI design. I would also like to thank Amir Helmy for his meticulous review of my thesis drafts, and for his invaluable friendship, encouragement, and motivation. Outside the academic context, I am thankful to Moataz El-Naghi for his sound advice, his selfless help, and his enjoyable company.

Last and certainly not least, I am genuinely blessed to have a loving and supporting family. I am deeply indebted to my amazing parents for their unassuming support and unconditional sacrifice. I only hope that they are as proud of me as I am of them.

Contents

List of Figures

List of Tables

List of Acronyms

WLAN	Wireless Local Area Network
API	Application Programming Interface
DHCP	Dynamic Host Configuration Protocol
UDP	User Datagram Protocol
AP	Access Point
MAC	Media Access Control
SSID	Service Set Identifier
IoT	Internet of Things
IPC	Inter-Process Communication
GUI	Graphical User Interface
CLI	Command-Line Interface
GINI	GINI Is Not Internet
UML	User-Mode Linux
RPC	Remote Procedural Call
MBSS	Mesh Basic Service Set
HWMP	Hybrid Wireless Mesh Protocol
OS	Operating System
UCI	Unified Configuration Interface
VAP	Virtual Access Point
VN	Virtual Network
NIC	Network Interface Card
GRE	Generic Routing Encapsulation
QoS	Quality of Service
TSF	Topology Specification File

VPL	Virtual Private Link
SQL	Structured Query Language
COA	Care-Of Address
VM	Virtual Machine
PC	Personal Computer
SDN	Software Defined Networking
TCP	Transmission Control Protocol
IP	Internet Protocol
ICMP	Internet Control Message Protocol

Chapter 1

Introduction

Recent years have witnessed a proliferation of wireless devices and this trend is predicted to further increase in the future. This is fueled by the continuous introduction of diverse technologies such as Wireless Sensor Networks (WSNs), Internet of Things (IoT), Machine Type Communication (MTC) and Heterogeneous Networks (HetNets). The applications of these technologies span numerous industries including home automation, utilities, health, transportation, environmental monitoring, and consumer electronics. Considering the diversity of wireless devices which ranges from consumer electronics such as smart phones, tablets, and wearable, to smart embedded devices such as smart thermostats, home appliances, and lightings, it comes as no surprise that there will be 24 billion interconnected devices by 2020 compared to the 9 billion connected devices today [2]. The market opportunities from IoT alone are predicted to be around $8.9 trillion according to recent estimates by the International Data Corporation (IDC).

With the ever growing field of wireless networks along with its applications, it seems prudent that the educational tools that are used to teach Computer Networking should also evolve to allow the students to experiment with and grasp the diverse wireless technologies. Unfortunately, this has not been the case. Current network emulators that are targeted for educational purposes are still centered around wired networks. The fundamental differences between these two physical mediums, such as device mobility, and channel characteristics, are not exposed to the students.

Moreover, these didactic network emulators generate networks that are either purely *Testbed-oriented*, or purely *Process-oriented*. Testbed-oriented emulators require an existing

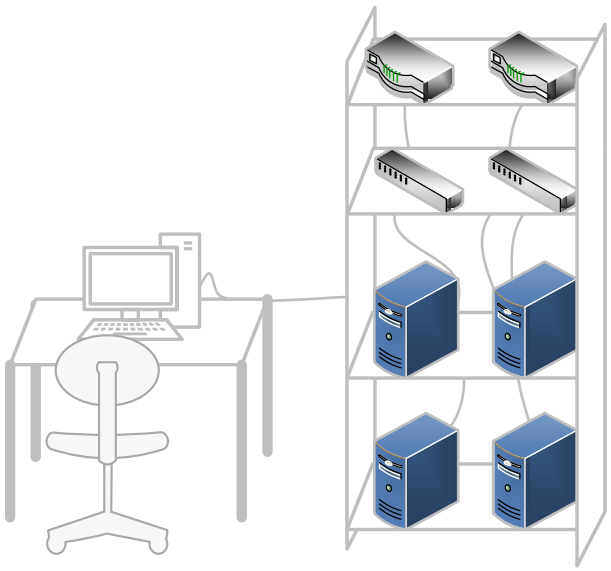


Fig. 1.1 A physical setup example of a testbed-oriented emulator [1]

physical network setup. This physical network setup commonly consists of multiple general purpose workstations, routers, and switches. These small-scale physical networks are then used to perform network experiments. The physical testbeds can be shared by multiple students (although not at the same time) by using an appropriate time sharing policy where each student is allowed to use the testbed for a reserved time slot [1]. One of the key issues with testbed-oriented emulators lies in their expenses due to the substantial cost of equipment purchase and maintenance. Another key issue is that they are harder to setup since to bring up an experimental network, one must log in to every device in the network and manually configure each one which can become tedious and error-prone. Yet another key issue is the difficulty to use those testbed-oriented emulators by the students. Finally, the testbeds are hard to scale. A single testbed can only be used by a single student at a time. Hence, to support multiple students at a given time (such as the case in a course lab), one must purchase duplicate equipment which scales poorly. These issues make testbed-oriented emulators less favorable for educational purposes. Testbed-oriented emulators include The Internet Lab [1] and [3]. An example of the physical setup for a

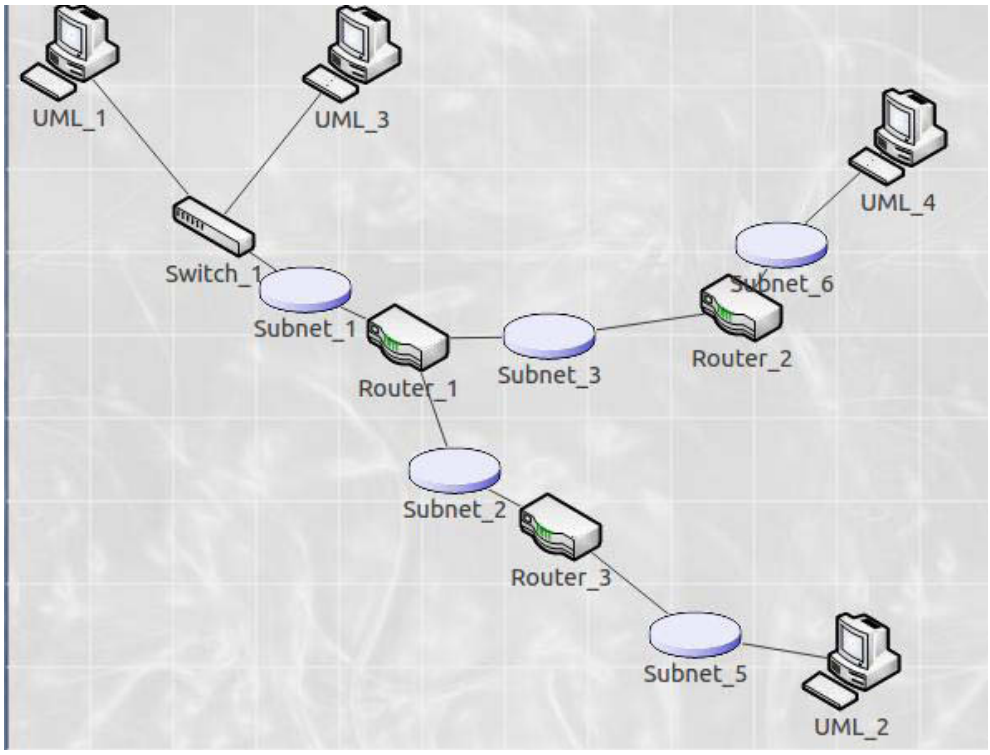


Fig. 1.2 An example of a custom network created using GINI.

testbed-oriented emulator in [1] is shown in Figure 1.1.

The majority of the most popular emulators are process-oriented. Process-oriented emulators run the network components, such as routers, switches, and workstations, as processes inside a host machine. These processes are then "connected" using an appropriate Inter-Process Communication (IPC) mechanism such as Unix Sockets. The custom network components are often times all hosted on a single workstation although some emulators, such as CORE [4], have the capability to distribute parts of the custom network over multiple hosts. Others such as Junosphere Classroom [5] and Cisco Learning Labs [6] provide a cloud-based service for hosting the user's custom network on the third-party's cloud (for a not very modest price). Network emulators can provide a Graphical User Interface (GUI) or a Command Line Interface (CLI) to allow the user to specify his/her custom network. CLI-based emulators include Netkit [7], VNX [8] and Mininet [9]. GUI-based emulators include CORE [4], Cloonix [10], and GINI [11]. An example of a network topology created in GINI (GINI is Not Internet) is shown in Figure 1.2.

There is a lack of emulators, however, that have the capability to support a *hybrid*

custom network, one which can accommodate process-oriented network components along with physical network components on the same topology. This may have been unnecessary for conventional emulated wired networks due to the limited variety in wired computing devices in the past and the ability to sufficiently abstract the wired connection. However, for wireless networks, capturing the mobility of the components and integrating with the new plethora of wireless devices into the emulated network is rather essential.

Finally, there is a lack of emulators that have an open-source router component that allows the user to view the internal workings of the router and modify its source code to implement new functionality. The routers in the emulators act as black boxes that can only be interfaced using the console that the router exposes. This is either due to the propriety nature of the router used in the emulator (such as Cisco's Learning Lab or Juniper's Junosphere) or due to the use of a router's image which is the case with the emulators that use User-Mode Linux (UML) [12] as routers in the custom network. UMLs are lightweight virtual Linux machines that run as a user-level process on an actual physical computer [13]. Even if the source code of the UML is provided, the reprogramming of the kernel networking module of the UML is not a straightforward feat especially for the average networking student.

1.1 Motivation

This thesis tries to address the need for a user-friendly, low cost, scalable wireless platform for educational purposes by presenting Wireless GINI. Wireless GINI is an extension to an open-source process-oriented network emulator toolkit developed by the Advanced Networking Research Lab (ANRL) at McGill University. Wireless GINI extends the existing GINI toolkit to support the creation of hybrid process-oriented networks and testbed-oriented wireless networks. This allows users to benefit from the simplicity of process-oriented networks while also benefitting from the realism needed for wireless networks. This enables students to perform innovative and practical experiments that touch upon the ever-growing application space of wireless networks.

As mentioned earlier, testbed-oriented emulators suffer from 1) higher cost 2) harder setup 3) complexity 4) poor scalability. Wireless GINI addresses those key issues to provide a low cost, user-friendly, and scalable wireless testbed for educational experiments. To address the cost issue, we used low cost ($65) embedded wireless devices called the Arduino

Yun [14], as opposed to expensive heavy workstations and commercial routers. In addition, there is no extra costs due to cables or peripherals (switches, hubs, monitors, etc.) since the testbed is connected via a wireless mesh network. Moreover, the devices are robust and require little maintenance since they run a lightweight Linux-based embedded Operating System (OS) that can be re-flashed to original factory settings by merely holding a button.

The second key issue that is faced with physical testbeds is that they are hard to setup and deploy. Indeed, current testbeds are setup in a fixed location and cannot be easily moved around to try out new experiment settings. In contrast, the Arduino Yuns are compact (2.8" x 2.1") and hence can be stored and placed anywhere with ease. The devices are connected via a wireless mesh network using the 802.11s standard [15]. This negates the need for intermediate cables for connectivity. The 802.11s standard for wireless mesh networking provides the perfect means for creating scalable wireless networks. The auto-discovery and distributed layer-2 routing used in 802.11s allows these wireless devices to automatically connect to and reach all the other wireless devices without user intervention. The wireless mesh network along with the lightweight and compact sizes of the devices makes the testbed easy to setup and deploy in a variety of locations.

The third issue faced by physical testbeds is that students find them hard to use. In process-oriented emulation software, the network topology can easily be created, expanded and modified by merely dragging and dropping network elements in case of a GUI-interface, or by writing scripts in case of a CLI interface. Furthermore, they provide a unified interface where one can easily configure a network element at runtime by double-clicking on the element in case of GUI-interface, or by clicking on its terminal window in the case of a CLI-interface. With physical testbeds, however, one has a fixed set of network elements at his/her disposal and there is no intuitive graphical representation of the topology that can be generated on demand. Moreover, interfacing with a network device requires either using special equipment such as a Keyboard, Video, and Mouse (KVM) switch, or by remote login. To address this issue, Wireless GINI uses an application server that automatically deploys custom virtual networks on the wireless testbed. Remote Procedure Calls (RPCs) are used to interact with the application server. To deploy a virtual network, the user can invoke the appropriate call to the server passing a simple XML document that describes the custom network. Future work on Wireless GINI will work on the auto-generation of the XML document via the GUI-interface. The server then uses this XML file to automatically configure and deploy the user's virtual network.

The fourth issue that testbed-oriented platforms face is the issue of scalability. The issue of scalability pertains to two concerns. The first concern is in relation to the number of students that can experiment at a given time. This is mostly done by purchasing and deploying identical physical networks. For example, in the case of University of Victoria's lab setup [3], 18 identical physical setups were deployed to support up to 18 students at a time in a given lab session. The lab sessions were divided across different time slots and days to accommodate all the students in a given class. The second concern is in relation to the size of the network for a given student. Increasing the network in terms of the number of elements is difficult due to the cost and space requirements. To overcome the first concern, we leveraged wireless network virtualization techniques [16] to enable us to efficiently share the same physical platform among multiple custom networks concurrently. The architecture and implementation of Wireless GINI allow for complete isolation of the concurrent networks. It also does not allow one network to constrain the configuration of another network. The application server manages the deployment of the custom network and maintains the necessary parameters for each running network via a SQLite database implemented using Python. Figure 1.3 depicts an example of two overlay networks deployed on the same physical testbed.

To overcome the second concern which is in regards to the size of the testbed, we developed Wireless GINI to allow for easy integration of existing wireless devices into the user's custom network. The user can request one or more Wireless Local Area Networks (WLANs) to be deployed on-demand on any given wireless device that is associated one's network. Other wireless devices can then connect to this WLAN via its broadcasted Service Set Identifiers (SSIDs) and become part of the custom internet. This effectively allows for out-of-the-box integration of wireless devices into the virtual network thereby enabling innovative and flexible experimentation with new technologies. We describe some of these experiments in Chapter 5.

1.2 Thesis Contribution

We designed and implemented Wireless GINI, an educational platform for hosting virtual wireless networks. Wireless GINI provides a cohesive and unified framework for deploying custom internets on a shared wireless platform. We developed the network virtualization mechanisms that enable the sharing of the wireless infrastructure. These mechanisms

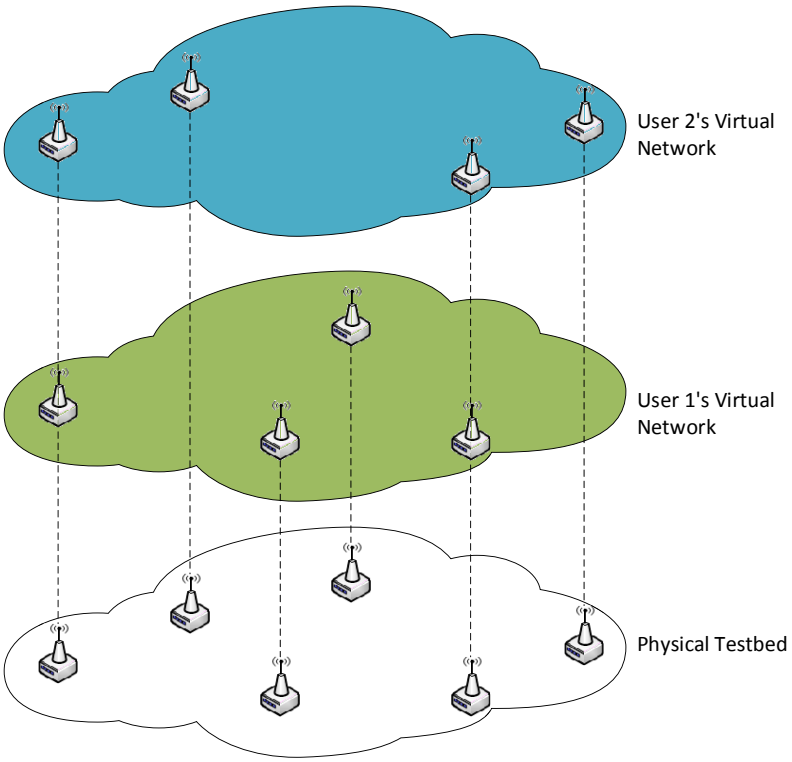


Fig. 1.3 An overview of two overlay networks running on top of the same physical testbed.

provide proper isolation between the hosted wireless networks and allow for custom configuration of the user's internet. We developed an application server, called the WGINI server, that provides a simple interface for the user to deploy his/her custom internet. It handles all of the complexity of deploying the user's topology, and the complexity of provisioning the wireless infrastructure. We developed the platform to support a diverse combination of network elements that are all integrated into one custom internet. The platform seamlessly integrates process-emulated components running on the user machine, wireless mesh overlays deployed on the wireless platform, and physical wireless devices connected to the user's custom network.

1.3 Thesis Organization

Chapter 2 provides the relevant background information for the thesis. We discuss the various techniques for network virtualization which are used to share multiple virtual net-

works on the same physical network. We also give an overview of the GINI toolkit. In addition, we provide an overview of wireless mesh networks and the 802.11s standard. We also describe OpenWrt, which is the operating system used on the wireless devices, and discuss its capabilities that aid in wireless virtualization and the implementation of Wireless GINI. Chapter 3 delves into the design considerations that were employed when designing Wireless GINI. It also describes the architecture of Wireless GINI, and the various techniques used to enable its features. Chapter 4 presents the details of the Wireless GINI implementation. We evaluate the performance of Wireless GINI in chapter 5 and present two applications in the realm of mobility management and IoT that can be enabled on the platform. Chapter 6 provides an overview of the various network emulators that are available in the literature.

Chapter 2

Background

2.1 The GINI System

In this section, we provide an overview of the GINI toolkit [11]. GINI is an open-source process-oriented network emulation toolkit that is targeted towards educational purposes. It provides a GUI by which users can design, run, inspect and stop their custom topologies. The topologies can consist of workstations, routers, and switches. The network elements are spawned as user-level processes running inside the host machine. Figure 2.1 shows the graphical interface for creating topologies. Users create their topologies by dragging-and-dropping network elements from the menu on the left-hand-side of the canvas.

An overview of the GINI architecture is shown in Figure 2.2. GINI uses a custom lightweight version of UML [12], called *gLinux*, to emulate the workstations. Compared to the default UML version, *gLinux* has a much smaller disk memory footprint (80 MB vs 600 MB) and a much faster boot up and shutdown times. Each *gLinux* instantiation contains its own network stack and process tables which are completely isolated from the host's tables and stacks as well as any other *gLinux* instances running on the host. GINI uses the UML switch daemon to emulate a Layer-2 switch. To emulate a router, GINI uses an open-source software-based router called *gRouter* developed for GINI. The *gRouter* is written in the C programming language. *gRouter* fully supports the processing of the Internet Protocol (IP). Having a completely open-source router provides users with practical hands-on experience in software development in the Computer Networking field. Students can modify and extend the *gRouter* to implement and test new algorithms and networking protocols.

Fig. 2.1 The GINI GUI.

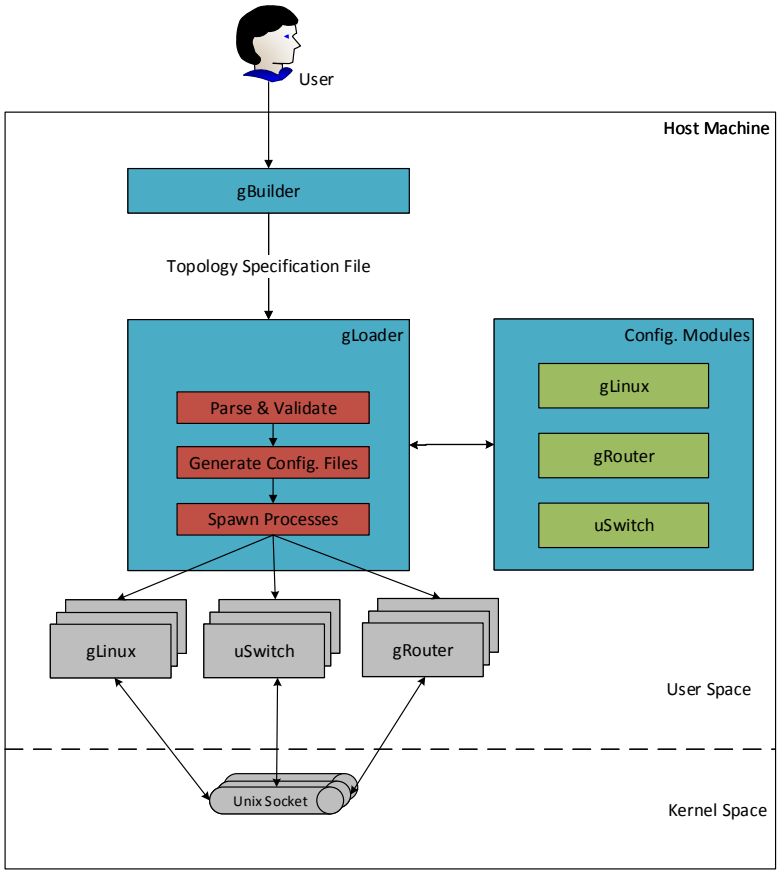


Fig. 2.2 An overview of the GINI architecture.

GINI uses Unix sockets to connect the network components of a given topology. A Unix Socket is an IPC mechanism for exchanging data between processes that are running on a given host machine. The Unix socket APIs provided by the OS are similar to the APIs provided for creating Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) sockets, with the notable difference being that all communications occur entirely within the host's OS kernel.

The user interacts with GINI through *gBuilder*, which is the front-end component of GINI that provides the graphical interface. *gBuilder* allows for a user-friendly way of creating network topologies. *gBuilder* uses the graphical topology supplied by the student to construct an XML topology specification file. The topology specification file is passed to the *gLoader* module to instantiate, configure, and run the network components along with their appropriate socket connections.

2.2 Wireless Mesh Networks and the 802.11 Standard

In this section, we give an overview of the 802.11s standard [15] which is the protocol used to connect the wireless devices that make up our wireless platform. The 802.11s is a wireless mesh networking protocol that provides the mechanisms for discovery, dynamic routing, authentication, and forwarding/relaying packets in the wireless mesh. The protocol accounts for power consumption, packet collision avoidance, and security in the wireless mesh. It was developed to account for the need of providing wireless access in places where connecting wireless devices to the backbone via wires was unfeasible or inconvenient. Having a wireless mesh network entails numerous benefits over having a wired connection from a switch to each access point including:

- **Flexibility**. In a typical 802.11 WLAN, each access point must be connected to the back-haul network via a wired connection (typically using ethernet). With a wireless mesh setup, only one station needs to have a wired connection to the back-haul network. This station is called the mesh portal. A wireless mesh network can have one or more mesh portals. The other mesh stations' packets can then reach the back-haul network by having the intermediate stations relay the packet to the appropriate mesh portal using the wireless medium. This allows the setup to cover larger areas without being constrained to being in range of a switch (typically $<$

100 m). Moreover, it provides a greater path redundancy to the back-haul network brought about by the re-routing of wireless packets through another path if one of the nodes in the existing path fails. In case of a wired connection, any failure in the single wired connection to the back-haul causes the wireless network to become unreachable. An example of a typical wireless mesh network is shown in Figure 2.3.

- **Self-Formation**. This is in regards to the automatic discovery of new mesh stations that get in range of an existing mesh station, and the ability to dynamically update the routing table in response to changes in the topology. This feature allows for a simple expansion and setup of the wireless mesh. A wireless mesh is officially called the Mesh Basic Service Set (MBSS).

- **Self-Healing**. The is in regards to the ability of the mesh station to detect several possible paths to a destination and automatically choose the best path in case of a failure or congestion in an intermediary mesh station. This feature allows for a seamless maintenance of the network without a dedicated specialist having to continuously monitor and reconfigure the network in response to a failure.

The MBSS is an 802.11 WLAN that consists of autonomous mesh stations that establish peer-to-peer wireless links. Messages can then be transferred between Mesh stations that are not in direct communication with each other by having the intermediate nodes relay the message. Hence, a Mesh station may act as a source, a sink, or a relay of the message. It is important to note that mesh stations do not communicate with non-mesh stations [17]. To provide communication with non-mesh 802.11 devices or to an 802.3 device, a separate interface must be set up on the given mesh station. The two interfaces can then be *bridged* together by the mesh station's kernel if a layer-2 networking is required, or forwarded by the station's IP routing modules if a layer-3 network is applicable.

Mesh stations that are on the same MBSS must be on the same channel in order to be able to detect and exchange packets one another. Channel access coordination is managed by the Mesh Coordination Function (MCF) which combines aspects of contention-based and scheduled-access methods [17]. MCF uses Enhanced Distributed Channel Access (EDCA) which is similar to Distributed Coordination Function (DCF) but has the ability to prioritize packets by reducing their back off interval or Arbitration Inter-Frame Space (AIFS). It also supports scheduled access features by allowing stations to reserve channel time by propagating this intent to their neighbours.

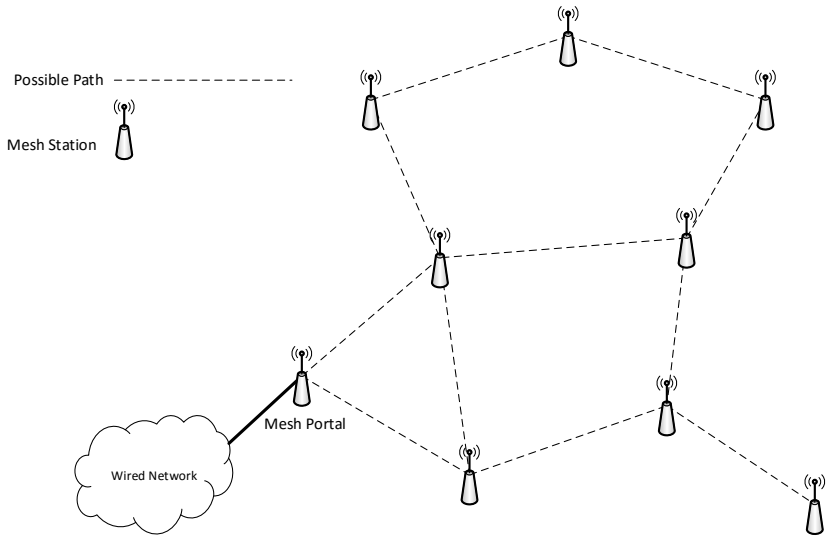


Fig. 2.3 A basic mesh topology.

802.11s uses active and passive scanning for neighbour discovery. Mesh stations actively broadcast beacon frames as well as send probe requests in response to received beacon frames. After discovery, the stations can form a peering session with each other provided they have the same *mesh profile*. A key feature of the mesh peering mechanism is that it is completely decentralized, and non-hierarchical. Having the same mesh profile entails having a matching mesh ID, and security credentials if security is enabled on the MBSS. Once peering is established, the stations become part of the same MBSS and, hence, the routing tables of the other mesh stations on the MBSS can be updated to account for the new station.

802.11s uses the Hybrid Wireless Mesh Protocol (HWMP) to perform routing functionality [18]. HWMP is an Ad-hoc On-demand Distance Vector (AODV) layer-2 routing protocol which is the default routing protocol in 802.11s mesh networks. To minimize congestion on the wireless channel, HWMP uses a default reactive approach to path discovery by initiating path discovery only when transmitting to an unknown mesh station. In this approach, a tree-based routing mechanism is employed to find the best path to the unknown station. Once the path is established, it is maintained by having each station in the path remember its next hop address. This continues until a disruption occurs in the path in which case a new path discovery is initiated. HWMP also supports a proactive approach where a mesh station discovers the best path to a given node in the mesh before

the need to transmit to that node arises. HWMP takes into account the channel quality in calculating the link metric (cost). A formula that takes into account the link costs as well as the hop count is used to calculate the best path to the destination.

2.3 OpenWrt and Arduino Yun

In this section, we provide an overview of the Arduino Yun which is the embedded wireless device used to create the Wireless GINI testbed. We also provide an overview of OpenWrt [19] which is the Linux distribution running on the Arduino Yun. Figure 2.4 shows a picture of the Arduino Yun. The Yun contains the AR9331 wireless system-on-chip (SoC). AR9331 includes a MIPS processor along with a 802.11b/g/n wireless device. The Arduino Yun also contains a micro USB port, an Ethernet port, and a micro SD card slot. The micro SD card slot can be used to extend the storage capacity of the device. The board is compact in size, having a length of 2.7 inches and a width of 2.1 inches, and weighs 32 grams.

The MIPS processor on the AR9331 SoC runs the OpenWrt OS. OpenWrt provides a convenient framework for deploying customized wireless networks on embedded devices. The lightweight nature of the embedded OS (the OS image is well under 100 MB) and its robustness accelerated its adoption in numerous industrial applications and mesh network communities. Among its most notable features are a writable root file system, a package manager, a *Unified Configuration Interface* (UCI), and comprehensive and flexible support for network-related features such as mesh networking, stateful firewalls, wireless functionality and security, Domain Name System (DNS) servers, and Dynamic Host Configuration Protocol (DHCP) servers. The UCI is well suited for providing a unified and simple configuration of the entire system programmatically. The wireless functionality includes Virtual Access Point (VAP) support where the same wireless device can be shared among multiple basic service sets (BSS). A BSS is the basic building block of an 802.11 WLAN. Each BSS can operate in a number of modes (infrastructure, ad hoc, mesh, etc.) and has its own networking stack.

2.4 Network Virtualization

Network virtualization has become a hot topic in recent years among researchers and industry professionals. The novelty as well as the diversity of perspectives among practitioners

Fig. 2.4 A picture of the Arduino Yun.

and researchers in this area has caused ambiguity into what "network virtualization" actually *is*. Indeed, the term has been overloaded and a precise yet encompassing definition has not been established yet [20]. The most elegant definition we came across is found in [21] which states that "Network virtualization is the technology that enables the creation of logically isolated network partitions over shared physical network infrastructures so that multiple heterogeneous Virtual Networks (VNs) can simultaneously coexist over the shared infrastructures". Network virtualization aims at increasing the utilization of the physical infrastructure, and provides flexible and extensible networks for tenants (the term tenant is used to refer to the owner of a VN). This is achieved by providing proper isolation between the different VNs that are sharing the same physical infrastructure and providing the necessary abstractions for seamless integration by making the complexity of the underlying network transparent to the tenant.

Before delving into the techniques used for network virtualization, let us enumerate the components that constitute a network. In the most abstract sense, a network is composed of nodes, links, and a topology. Although it can be argued that a topology is a derived component from the nodes and links, a topology is distinct due to the management and provisioning needed when discussing the network as a whole. Nodes can be the end hosts, routers, switches or any other device in the network. The links include the interfaces, and the medium (wireless, optical, twisted copper, etc.) that connects the interfaces together.

2.4.1 Node Virtualization

Node virtualization aims at sharing a network node among the different VNs while providing the necessary abstractions for isolation and configuration. Several techniques can

be used to accomplish this task. These techniques fall under two general approaches. The first approach, the hypervisor-based approach, provides a virtualization layer, called the hypervisor, that can support multiple Virtual Machine (VM) instances at the same time. Each VM can have its own virtual OS. The second approach, the container-based approach, provides each virtual host with a user-level sandboxed environment.

Figure 2.5 provides an illustration of node virtualization using the first approach. A hypervisor, also known as a Virtual Machine Monitor (VMM), is a virtualization layer that enables VMs to share the same hardware. Each VM runs its separate router OS with its network stack isolated from the other VMs running alongside it. The hypervisor provides the necessary abstraction for virtualizing the Network Interface Card (NIC) to isolate the ingress/egress traffic of each VM. This can be accomplished using a software-based solution where the hypervisor exposes a virtual NIC (vNIC) abstraction to the guest VM. The hypervisor then uses the link abstractions discussed in Section 2.4.2 to isolate the traffic and route incoming traffic to the correct VM. Hypervisor vendors, such as VMware and Xen, provide support for the vNIC abstraction. Another solution for supporting vNICs is through hardware support where the NIC's firmware can support multiple network stacks on the same physical interface such as the case with VAPs.

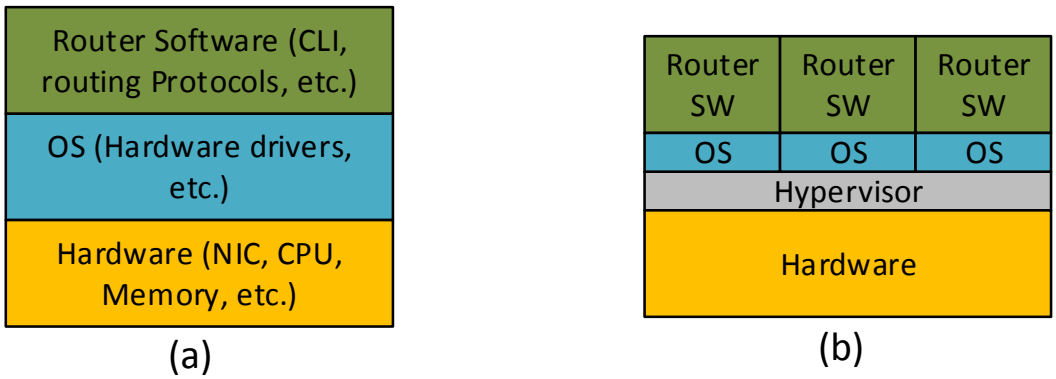


Fig. 2.5 Router virtualization using hypervisors.

The second technique for node virtualization is the container-based approach. This technique provides isolation by supporting multiple user spaces on the same kernel. Each user space instance has its own OS abstractions such as process tables, file systems, and network stack. Since the kernel is shared between containers, this approach provides higher

performance over the hypervisor-based approach. However, sharing the kernel also leads to less flexibility since only a subset of the operating system resources and functions is virtualized. For example, processing non-IP packets requires modification to the kernel data structures. This can be achieved in the hypervisor-based approach by modifying the guest OS, while a container-based approach does not provide this flexibility since all the containers share the same kernel data structures. Several implementations are available for container-based virtualization including VServer [22], and NetNS [23]. Trellis [24] is an example of a virtualization platform that uses the container-based approach for providing node virtualization. Trellis uses VServer and NetNS containers to provide node virtualization and uses Generic Routing Encapsulation (GRE) tunnels (discussed in the next section) to provide link virtualization. Based on experimental results, Trellis reports a significantly higher performance throughput in comparison to the hypervisor-based approach.

2.4.2 Link Virtualization

In this section, we describe the techniques used to virtualize the links that connects the nodes on a network. Similar to node virtualization, link virtualization aims at sharing the transport medium among multiple VNs while providing isolation between them [25]. Several link virtualization techniques have been developed that target various layers in the protocol stack. Generally, the link virtualization techniques fall under two broad categories: 1) Physical layer virtualization 2) Data path virtualization.

Physical layer virtualization techniques partition the physical medium among the VNs such that every VN can own a dedicated share of the link's bandwidth. Examples include the User Controlled LightPaths Project (UCLP) [26] which is a virtualization framework for Synchronous Optical Networks (SONETs). UCLP provides software tools for partitioning the bandwidth of the optical links among the VNs. Another example is the Multi-Purpose Access Point (MPAP) which partitions the wireless spectrum and allows the tenants to obtain a dedicated channel bandwidth on the wireless spectrum [27]. The advantage of physical layer virtualization is that tenants can have custom network stacks that may be different from the TCP/IP protocol stack. Another advantage is that the tenants own a dedicated portion of the physical link bandwidth leading to more fine-tuned Quality of Service (QoS) guarantees. The disadvantage of physical layer virtualization is that dedicated and expensive network equipment must be purchased to support it, and as in the case of

MPAP, the flexibility comes at the expense of having a bottleneck in the virtualization layer.

Instead of dealing with physical layer manipulation, data-path virtualization aims at providing virtual links through the manipulation of the data at the higher layers of the protocol stack. Data-path virtualization is generally what is meant when discussing network virtualization. It allows for virtual link support on commodity hardware. Data-path virtualization is implemented in software which makes it easier to develop, test, deploy, manage, and modify as compared to physical layer virtualization. The goal is to append the data from the virtual host with unique information such that the underlying network can process that information and route the packet to the destination virtual host. Three general approaches are used for data-path virtualization:

- *Labeling.* This technique assigns a unique ID to each VN. Packets belonging to a given VN are labelled or *tagged* before being sent out to the network. The nodes are aware of the labels and can use them in their forwarding table to determine the correct interface to route the packet through. Examples of this technique is 802.1q Virtual Local Area Network (VLAN) tags and Multi-Protocol Label Switching (MPLS).

- *Tunneling.* With labelling, the physical nodes must have support for processing the labels since these labels are not part of the default networking stack. Hence, this technique cannot be used on an existing IP network without modifying all the nodes of that network to add support for label processing. To overcome this, tunneling is another technique that is widely used for implementing virtual links. Tunneling is the method of encapsulating the data packet in an IP packet before sending it over the network. The encapsulated data packet can be a layer-2 packet, a layer-3 packet or any other packet format. This data packet is encapsulated in an IP packet so that it can be forwarded by the existing *underlay* IP network to the next *overlay* node. Once the packet reaches the overlay node, the packet is decapsulated and passed to the virtual host for processing. Various tunneling protocols are widely used including GRE tunnels, Internet Protocol Security (IPsec) tunnels, and Virtual eXtensible LAN (VXLAN) tunnels [28].

- *Flow Space Partitioning.* Another technique brought about by Software Defined Networking (SDN) is partitioning the entire "flowspace" into disjoint non-overlapping

subsets. A flowspace is a subset of the entire set of possible packet headers [29]. This technique is used by FlowVisor [29] which is a network virtualization layer for OpenFlow. OpenFlow decouples the control plane from the forwarding plane and allows packets to be forwarded based on forwarding rules, or *flow entries*, in the switches' tables. These flow entries are configured by an OpenFlow controller. Forwarding occurs based on matching the packet header fields with the bit strings of the flow entries. If a match exists, the packet is routed based on the destination port of the flow entry. Otherwise, a control packet is sent to the controller to determine the action. OpenFlow uses 10 fields to perform the matching operation with a total of 256 bits. These fields span the Media Access Control (MAC) address all the way to the TCP port numbers. FlowVisor is responsible for isolating the flowspace of each VN and for forwarding the ingress packets to the correct virtual switch. OpenRoads [30] is an example of an OpenFlow platforms that uses FlowVisor to virtualize the network and is targeted towards mobile and wireless applications.

In addition to resource sharing, data-path virtualization techniques are also used for connectivity, such as in Virtual Private Networks (VPNs), and providing new services such as in Peer-to-Peer (P2P) applications.

It is important to note that network virtualization is sometimes used to describe network simulation and network emulation. The reasoning behind this is that the network nodes are not physical but are emulated in software, as discussed previously, and hence can be considered "virtual". However, this usage is not encouraged since it does not pertain to sharing a physical network or providing resource abstractions as noted in [20]. This is the reason we opted to use the terms "process-oriented" and "testbed-oriented" to refer to the different network emulation techniques rather than referring to them as virtual emulation and physical emulation, respectively.

Chapter 3

Wireless GINI: Design and Architecture

3.1 Design Considerations

In this section, we identify the high-level design requirements for creating a hybrid emulator that is geared towards computer networking education. There are several aspects to take into account when creating such a platform. More specifically, the adoption of the platform is dependent upon the experience of 3 main groups: 1) the student, 2) the instructor, 3) the university.

Each group has its own distinct requirements and considerations. From the student's perspective, the platform should be user-friendly, robust, and contains minimal emulator-specific technicalities. The student should have an intuitive way of specifying the topology without being distracted by the underlying procedures of setting up and running the virtual network. Ideally, the student should only need to create a network specification file of a VN. The specification file should have an easy format and only contains the parameters that are relevant to the VN. The learning curve of the emulator should be low and the emulator-specific commands that the student is required to learn should be minimal.

From the instructor's perspective, the platform should support innovative experimental scenarios that compliment and enhance the learning experience. It should also be flexible enough to give the instructor the freedom to tailor the labs as needed. The platform should be realistic to help students tie the theoretical background presented in class with practical

real-life working experiments that the students can relate to.

From the university's perspective, the lab should be affordable, easily deployable, and maintainable. Ideally, the testbed should have the flexibility to be deployed anywhere without needing a dedicated space. It should also be easy to setup and extend.

3.2 An Overview of Wireless GINI

As mentioned in Chapter 1, our aim is to address the need for an educational platform to computer network experimentation. To facilitate the adoption of the platform, our design philosophy was shaped by the considerations presented in the previous chapter. In this section, we provide an overview of the Wireless GINI architecture and the components that comprise the Wireless GINI platform. We start by discussing the physical setup of the Wireless GINI platform. Figure 3.1 illustrates an overview of this physical setup. As can be seen from the figure, the physical setup contains three main physical components: 1) The user machine (Lab PC or Laptop) 2) The Wireless GINI server 3) The Wireless Mesh Network. The user machine could be a Lab PC provided by the university or a user's laptop that has GINI installed. The Lab Personal Computer (PC) is connected to the same LAN that the Wireless GINI (WGINI) server is connected to. The laptop can be connected to this LAN via an Access Point (AP).

The user machine serves two purposes: 1) it allows the user to interface with the WGINI system, 2) it hosts the process-emulated part of the user's topology. Wireless GINI must seamlessly integrate with the existing GINI toolkit to allow the software to be extended in a user-friendly manner. With the Wireless GINI system, users can integrate their process-emulated networks running on the user machine with a custom VN deployed on the wireless mesh network. The "client software", the GINI software running on the user machine, is extended to include client code for interfacing with the WGINI server.

Figure 3.2 shows an example of a topology specified by the user. The topology contains the usual *gLinuxes*, *gSwitches*, and *gRouters*, as well as a new component called the *yRouter*. A *yRouter* refers to a virtual node that runs on a mesh station. As can be seen from Figure 3.2, the user's topology can be broken down into two subsets; the process-emulated network subset, and the wireless VN subset. The process-emulated network subset contains the *gLinux*, *gSwitch*, and *gRouter* nodes of the topology. This subset runs on the user's machine as described in Section 2.1. To run the wireless VN subset, the user sends a request to the

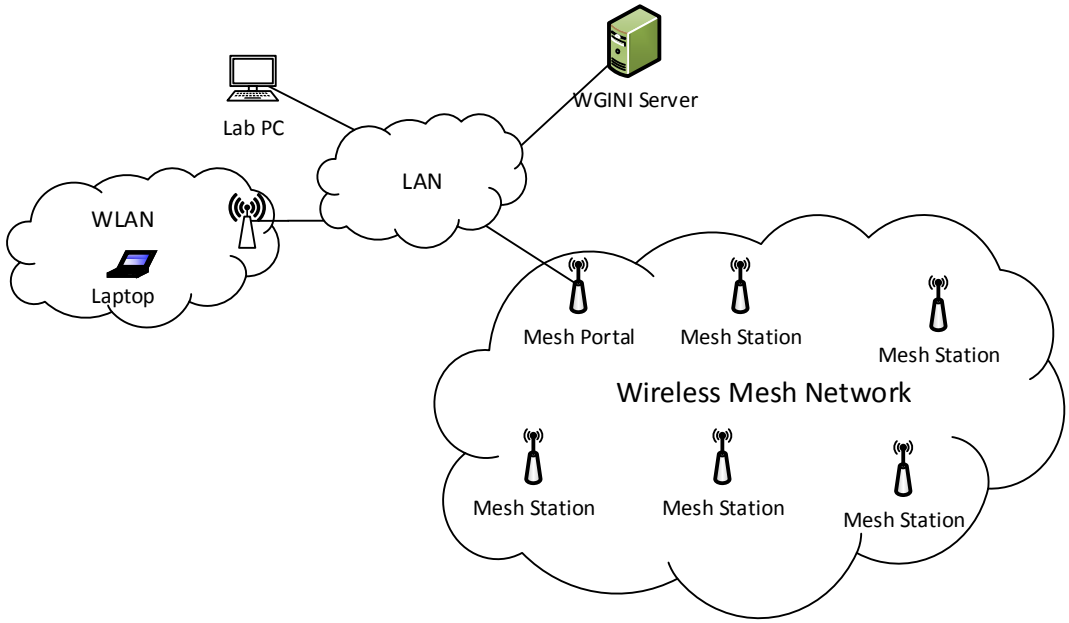


Fig. 3.1 The physical setup of the Wireless GINI platform.

WGINI server to deploy the VN on the wireless mesh. The request is sent using a RPC. The request includes the Topology Specification File (TSF) of the user's VN. The TSF is an XML file that captures the VN graph as well as the network configuration parameters of the VN. These network configuration parameters include the IP address, subnet mask, and MAC address of each interface in the VN, and the routing table information of each *yRouter*. Currently, the user must manually create the TSF. The extension of the *gBuilder* and *gLoader* modules to support the auto-creation of the TSF is being developed as part of a student project and is outside the scope of this thesis.

The second component, the WGINI server, is responsible for the management of the WGINI testbed and for deploying and deleting VNs on the wireless mesh platform. Figure 3.3 provides an overview of the WGINI system architecture. As can be seen from the figure, the WGINI server contains an RPC server that accepts commands from the RPC client running on the user machine. Three APIs are exposed to the user; *Check*, *Create*, and *Delete*. The *Check* API is used to provide information about the wireless platform to the GINI software running on the user machine. It returns the number of mesh stations available, and indicates which mesh station is the mesh portal. It queries the local database

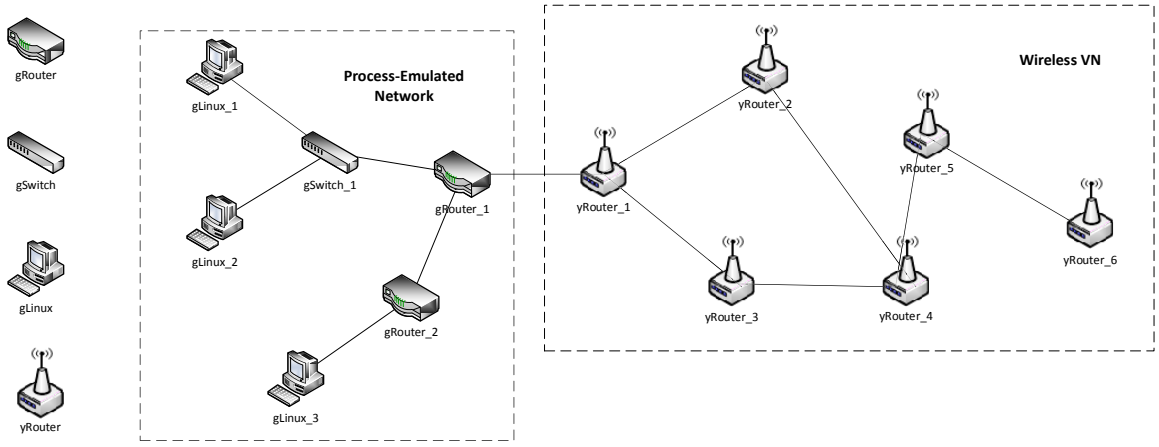


Fig. 3.2 An example of a hybrid topology specified by the user.

to retrieve this information. The local database is used to keep track of the number of mesh stations available, and the VNs that are currently deployed. The information returned by *Check* is used by the user to insure that the number of *yRouters* to be used in his/her topology does not exceed the number of the mesh stations available. It is also used by the user to identify the mesh portal. The *Create* API is used to deploy the user's VN on the wireless mesh. It takes as input the TSF provided by the user. It then parses and validates the file to make sure that the VN is deployable. If the TSF passes the validation check, the WGINI server updates the local database to include the user's VN information and deploys the VN on the wireless mesh. Once the user terminates the experiment, the user invokes the *Delete* API to free up the resources used by the user's VN. The database is queried to retrieve information about the resources used by the user and uses this information to delete all the components of the user's VN. Once complete, the database is updated accordingly.

The third component of the Wireless GINI system is the wireless mesh network. The wireless mesh network is the physical testbed upon which the VNs will be deployed. The physical testbed is comprised of several mesh stations that are connected using the 802.11s protocol. The Arduino Yuns are used as the mesh stations. The mesh portal is connected to the LAN that connects to the WGINI server. Every mesh station is pre-configured to boot up with a mesh interface that has the same mesh profile as all the other mesh stations. The IP address of each mesh station on the MBSS is assigned statically and is

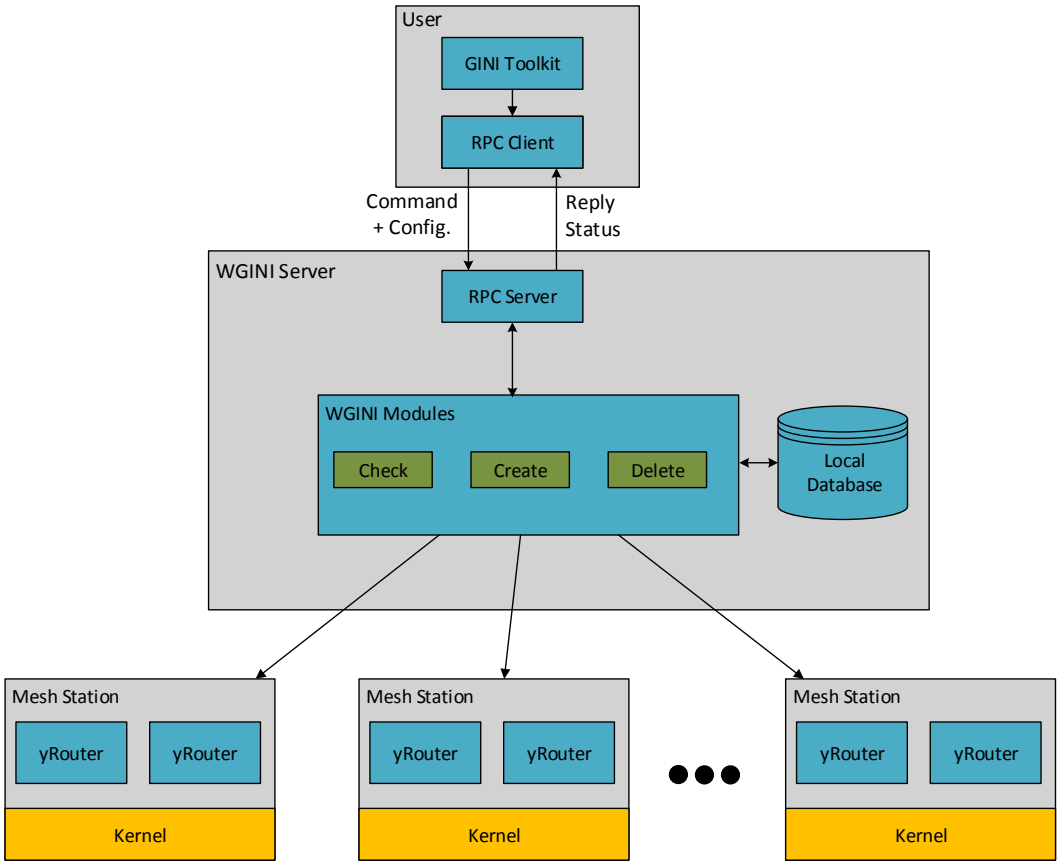


Fig. 3.3 Overview of the Wireless GINI design.

stored beforehand in the local database.

3.3 Supporting Concurrent Topologies

Now that we have a general overview of the WGINI system, it is time to dig deeper into the platform. We start with the wireless mesh network and examine the techniques used to create VNs on the physical wireless mesh network setup. As mentioned in Section 2.4, to support network virtualization, we must implement node virtualization and link virtualization. In this section, we describe how node virtualization and link virtualization are achieved to create overlays on the wireless mesh network for each requesting user.

3.3.1 Node Virtualization

Figure 3.4 illustrates node virtualization in Wireless GINI. As shown in the figure, each *yRouter* runs as a user-level process in OpenWrt. A *yRouter* is essentially a *gRouter* that has been extended to support virtual links and to run on the OpenWrt distribution. Each *yRouter* can have multiple tunnel interfaces. Each tunnel interface connects to another *yRouter* that is running on a different mesh station but is part of the same VN.

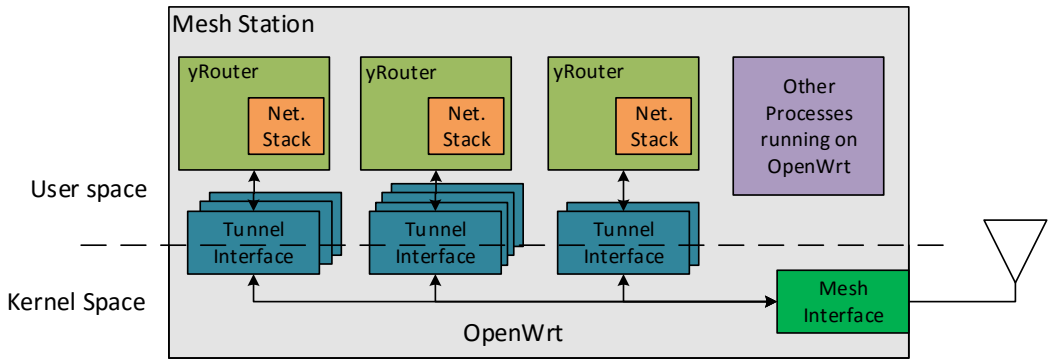


Fig. 3.4 Node Virtualization in Wireless GINI.

Each user can run at most one *yRouter* on a given mesh station. The *yRouters* are isolated from each other using the process abstraction provided by the OpenWrt OS. Each *yRouter* has its own networking stack that is residing in user space. This negates the need for an OS level isolation support such as the one provided by container-based or hypervisor-based virtualization. Moreover, the other isolation features, such as filesystem isolation, and process table isolation, are not needed since the user can only run one process on a given mesh station and the *yRouter* does not need filesystem support to function. Avoiding the resource-expensive OS virtualization layer allows for a more lightweight implementation which, in turn, allows the resource-constrained mesh station to support a larger number of VNs concurrently. Moreover, it leverages the existing OpenWrt firmware, available on the Arduino Yun out-of-the-box, which allows for an easier setup.

3.3.2 Link Virtualization

In this section, we demonstrate how link virtualization is achieved in Wireless GINI. As mentioned previously, a VN consists of *yRouters* running on various mesh stations. These

mesh stations are connected together by virtue of being on the same MBSS. To connect *yRouters* together, we opted to use the tunnelling technique to create the virtual links. Unlike the other link virtualization techniques, such as physical layer virtualization or flowspace partitioning, tunnelling allows us to utilize the underlying physical setup without resorting to additional hardware. This makes the platform easier to setup and more affordable. Moreover, it provides us with the flexibility of decoupling the physical setup from the VN setup. This allows the VNs to be deployed on other physical setups, such as newer mesh networking protocols, without affecting the VN setup.

To support link virtualization, a new device driver was developed on the *yRouter* called the *tun* driver. This *tun* device driver resides in the Virtual Private Link (VPL) module of the *yRouter*. The VPL module is also where the Unix socket device driver resides. Figure 3.5 provides an overview of the *yRouter*. As can be seen from the figure, the VPL module is responsible for emulating the physical layer by pushing packets from one *yRouter* instance to the other.

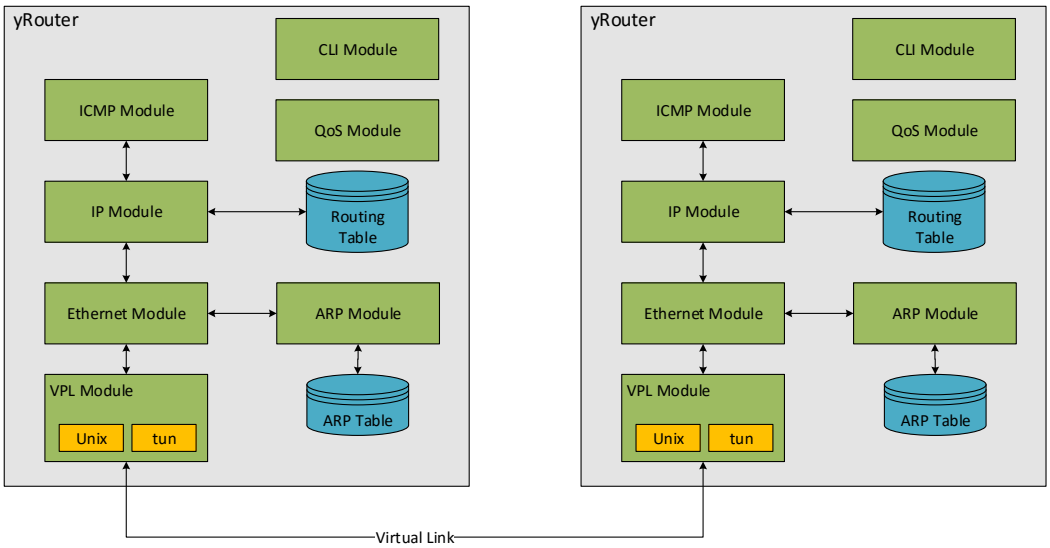


Fig. 3.5 An overview of the *yRouter*.

Packets are tunnelled from one *yRouter* to the other using UDP Sockets. Each interface on a *yRouter* has its own UDP socket. This UDP socket binds to a unique UDP port on the mesh interface. The WGINI server is responsible for allocating this unique UDP port during the deployment of the VN.

Moreover, the WGINI server configures each *tun* interface with the destination IP address and UDP destination port. This destination IP address is the MBSS IP address of the mesh station where the destination *yRouter* resides. The destination UDP port is the unique UDP port that the *tun* interface on the other end of the virtual link is listening on. Configuration of the *tun* interface's UDP source port, destination IP address, and destination UDP port is handled automatically by the WGINI server and requires no user intervention. The user merely has to specify that his/her topology requires a connection between two different mesh stations on the wireless mesh.

When a packet is sent over the *tun* interface, the packet gets encapsulated inside a UDP packet and is sent over the wireless channel to the destination mesh station via the mesh interface. The packet is then decapsulated at the destination mesh station and sent to the destination *yRouter's tun* interface on the other end. Consider the example shown in Figure 3.6 where the user specifies a connection between mesh station 1 and mesh station 2. A *yRouter* is instantiated on each mesh station and a virtual link is established between them. The user specifies the network parameters of the virtual interfaces as shown in the figure. Let us go through the example where an Internet Control Message Protocol (ICMP) ping packet is sent from yRouter1 to yRouter2. Figure 3.7 illustrates the encapsulation and decapsulation procedure. The ICMP module creates the ICMP ping request packet and pushes it to the IP module. The IP module, in turn, encapsulates the ping request in an IP datagram. Notice that the IP header information pertains to the virtual link interfaces. The IP module then passes the packet over to the ethernet module. Similarly, the ethernet module encapsulates the packet in an ethernet frame with the ethernet header populated with the MAC addresses of the virtual interfaces. The ethernet module then passes the frame over to the interface's device driver. The interface's device driver is essentially a pre-configured UDP socket. From there, the packet is encapsulated in a UDP datagram and sent over the mesh interface. As mentioned previously, the WGINI server insures that this UDP socket is correctly configured to send the packet to the correct *yRouter* (in this case yRouter2). This entails that the packet should reach the correct mesh station (mesh station 2) and from there reach the correct virtual interface. To achieve this, the destination IP address and the UDP destination port number of the UDP socket are pre-configured to be the IP address of mesh station 2 on the MBSS and the port number that yRouter2's *tun0* interface is listening on, respectively.

This same link virtualization technique is used to connect the *gRouter* running on the

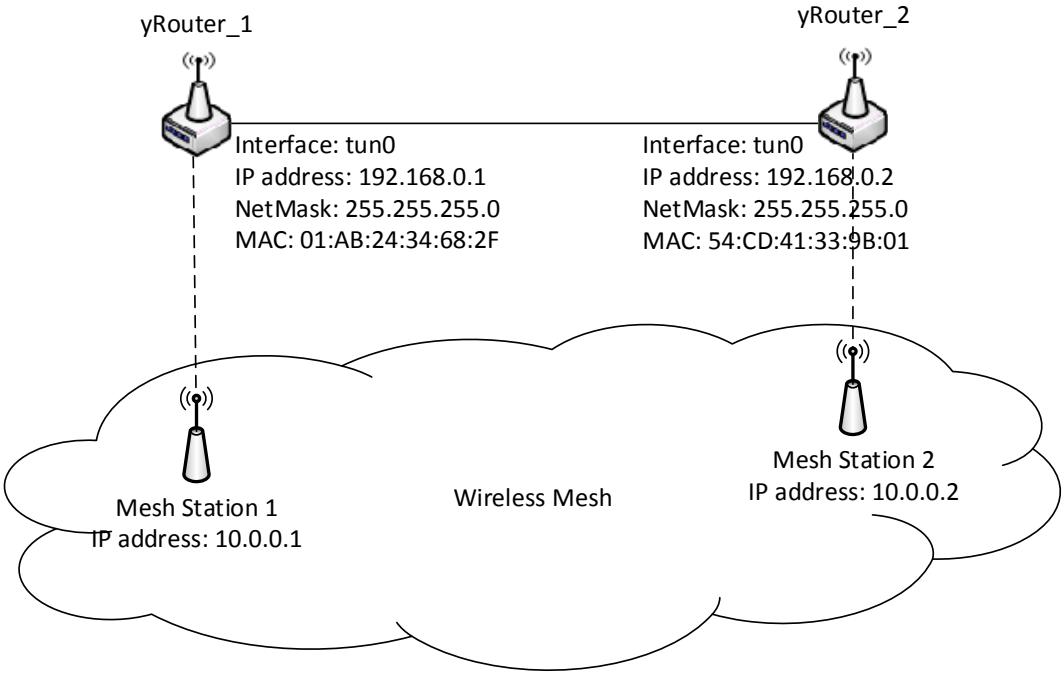


Fig. 3.6 An example of a simple VN.

user's machine to the *yRouter* running on the mesh portal. The only difference is the underlying physical network. The virtual link between the user's machine and the mesh portal runs over the wired LAN while the virtual link between *yRouters* runs over the wireless mesh.

3.4 Integrating Physical Devices

We have seen in the previous section how the *tun* device driver facilitates the communication between *yRouters* on the mesh network. We have also seen the underlying network virtualization techniques of this *tun* interface. We have seen that the network virtualization techniques solves two issues: 1) It provides us with resource abstractions that facilitate the sharing of the wireless mesh 2) It allows us to connect the process-emulated part of the topology to the Wireless VN part. We now shift our focus to the following question: how can we integrate wireless devices that are not part of the wireless mesh network into the user's topology?

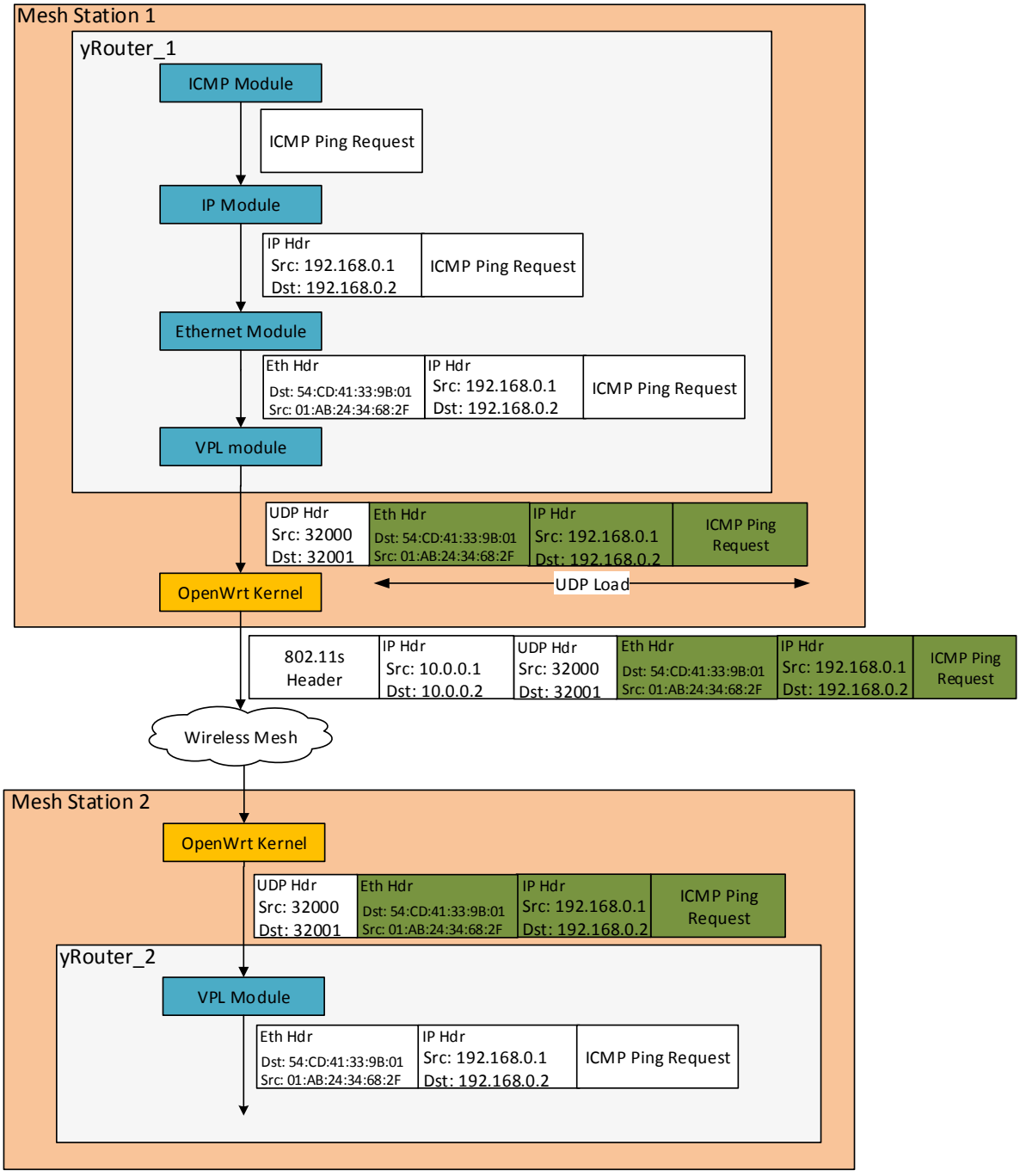


Fig. 3.7 An illustration of the tunnelling process.

Adhering to the design considerations set forth in Section 3.1, we require that this integration be easy to setup, and require no additional hardware. We expect that external wireless devices can integrate into the user's topology out-of-the-box. No modification to the wireless device's software or installation of additional software/hardware should be required. The aim should be to extend the platform to have the feature of connecting generic wireless devices into the topology in a familiar way.

To achieve this, we utilize OpenWrt's VAP feature. As mentioned in Section 2.3, the VAP feature allows the kernel to share multiple interface profiles on the same wireless device. These profiles could have different modes of operation (mesh, ad-hoc, infrastructure) and networking stacks residing in kernel space. The Arduino Yun's wireless device can support up to 8 different interface profiles at a given time. We leverage this VAP feature to deploy infrastructure mode 802.11 WLANs that generic Wi-Fi devices can connect to. Infrastructure mode WLANs is the familiar WLAN mode that is used by 802.11 APs to provide wireless clients with network connectivity by broadcasting their SSIDs.

We extended the WGINI platform to take into account this feature. Specifically, the *Check* API was updated so that it returns the number of available wireless profiles left on each mesh station. We also updated the *Create* API to accept an updated TSF format. The updated TSF format allows the user to specify the mesh stations that he/she would like to deploy a WLAN on. A user can deploy a WLAN on each mesh station that is part of his/her VN. However, a user can only deploy one WLAN on a given mesh station. The TSF allows the user to specify the network parameters of the WLAN interface such as the IP address, and the SSID. The *Delete* function was updated to account for the deletion of the WLAN interfaces that the user deployed to free up the resource so that it can be used by other users. The database was also updated to keep track of the WLANs deployed on each mesh station. This allows the WGINI server to check whether a mesh station has a WLAN available before deploying the WLAN.

The *yRouter* has also been extended to integrate the WLAN into the user's topology. A new device driver was created in the VPL layer called the *wlan* device driver. Upon creation of a *wlan* interface on a *yRouter*, the *yRouter* executes a bash script that creates a new VAP interface in OpenWrt. The bash script takes as input the IP address, MAC address, and SSID provided by the user. The bash script then uses UCI, discussed in Section 2.3, to create the VAP interface. The script uses the UCI API to configure the network parameters, wireless parameters, and firewall rules of the new interface. The script also sets up a DHCP

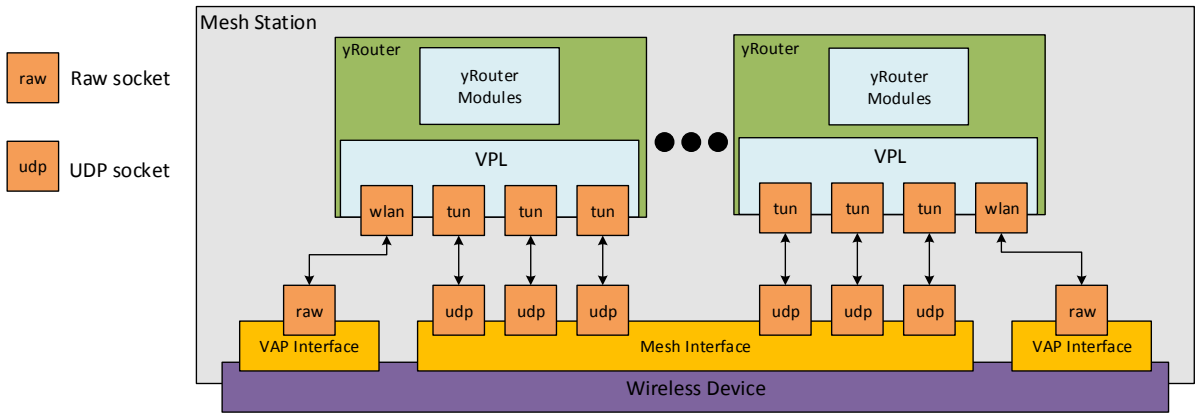


Fig. 3.8 WLAN support on the *yRouter*.

server on the VAP interface. The DHCP server allows wireless devices connected to the VAP to be assigned IP addresses automatically. Once the new VAP interface is brought up, the *yRouter* creates a *raw* socket that binds to the VAP interface. A *raw* socket is a socket type provided by the Linux kernel that is used to capture incoming packets and inject packets to an interface while bypassing the normal TCP/IP processing done at the kernel. Since a copy of all the incoming packets are captured by the raw socket and sent to the *yRouter*, the firewall rules on the VAP interface are configured such that incoming packets are dropped by the kernel. This is done so that the packets destined to the VAP do not interfere with the kernel networking stack. Once the packet is captured by the *yRouter*, the packet is processed by the networking stack of the *yRouter* and routed to the correct destination thereby becoming part of the user's VN. Figure 3.8 provides an overview of a *yRouter* that contains the *wlan* interface support. Once the user terminates his/her VN, a bash script is executed that uses the UCI API to delete the VAP interface residing in the kernel.

Chapter 4

Wireless GINI: Implementation and Deployment

4.1 The Wireless GINI Interface

Now that we have a comprehensive picture of the WGINI system, it is time to go over some of the implementation details. In this section, we go over the implementation details of the sub components of the WGINI system. We also go over the WGINI interface. Specifically, we discuss the RPC client-server implementation as well as the TSF format.

4.1.1 RPC Client-Server Implementation

The xmlrpclib [31] Python library is used for the RPC client-server implementation. xmlrpclib uses the XML-RPC protocol to implement RPCs. XML-RPC runs over HTTP and uses XML to specify the API to be invoked and to pass the input parameters of the API. When a user machine loads up GINI, an RPC client is initialized. The RPC client is passed the IP address of the WGINI server and the TCP port that the RPC server on the WGINI server is listening on. It uses this information to connect to the RPC server so that subsequent RPC calls can be performed directly using the RPC client connection. The RPC server handles one request at a time using a single-threaded implementation. The server queues any request that arrives while another request is being serviced. This avoids overcomplicating the design, caused by implementing techniques to avoid race conditions and stale information, based on a multi-threaded implementation. The downside is the poten-

tial increase in latency in servicing a request when more than one request is queued. This is due to the decrease in server utilization especially during long I/O tasks, since request executions are not overlapped. However, given the typical class size and the anticipated frequency of requests, we predict that the increased complexity of a multi-threaded implementation is currently not warranted. This decision may be revised in the future based on feedback from adopters.

Currently, the user must manually specify the IP address and the TCP port of the RPC server. Future work aims at making this procedure transparent to the user by using the multicast Domain Name System-Service Discovery (mDNS-SD) standard to advertise the WGINI services. mDNS-SD is a protocol defined in RFC6762 [32] for advertising and discovering services on a LAN using DNS-like operations without the requirement of a conventional DNS server running on the local network. mDNS-SD are easy to setup and require no adminstration or dedicated infrastructure. Using mDNS-SD, the WGINI server can advertise its IP address and port number of its service on the LAN. Avahi and Bonjour are free implementations of the mDNS-SD standard that run natively on Linux and MAC OSs, respectively.

4.1.2 The Topology Configuration File

As mentioned previously, the TSF is an XML file that captures the graph and network parameters of the deployed VN. It allows the user to specify his/her topology transparently without being concerned with the underlying virtual node and link implementation. The Document Type Definition (DTD) of the TSF is provided in Appendix A.1. DTD is a set of markup declarations that is conventionally used to define the structure, elements, and attributes of XML documents. Figure 4.1 illustrates the tree structure of the TSF. The The + sign indicates that the child element must have one or more occurrences. The $*$ sign indicates that the child element must have zero or more occurrences. The ? sign signifies that the child element must have zero or one occurrences.

The following is a description of the elements found in the TSF:

- *Station*. This element specifies that a *yRouter* should be deployed on the mesh station that has the unique ID specified by the child element *ID*. Each mesh station has a pre-configured unique ID. Multiple *Station* elements can be declared in a TSF.

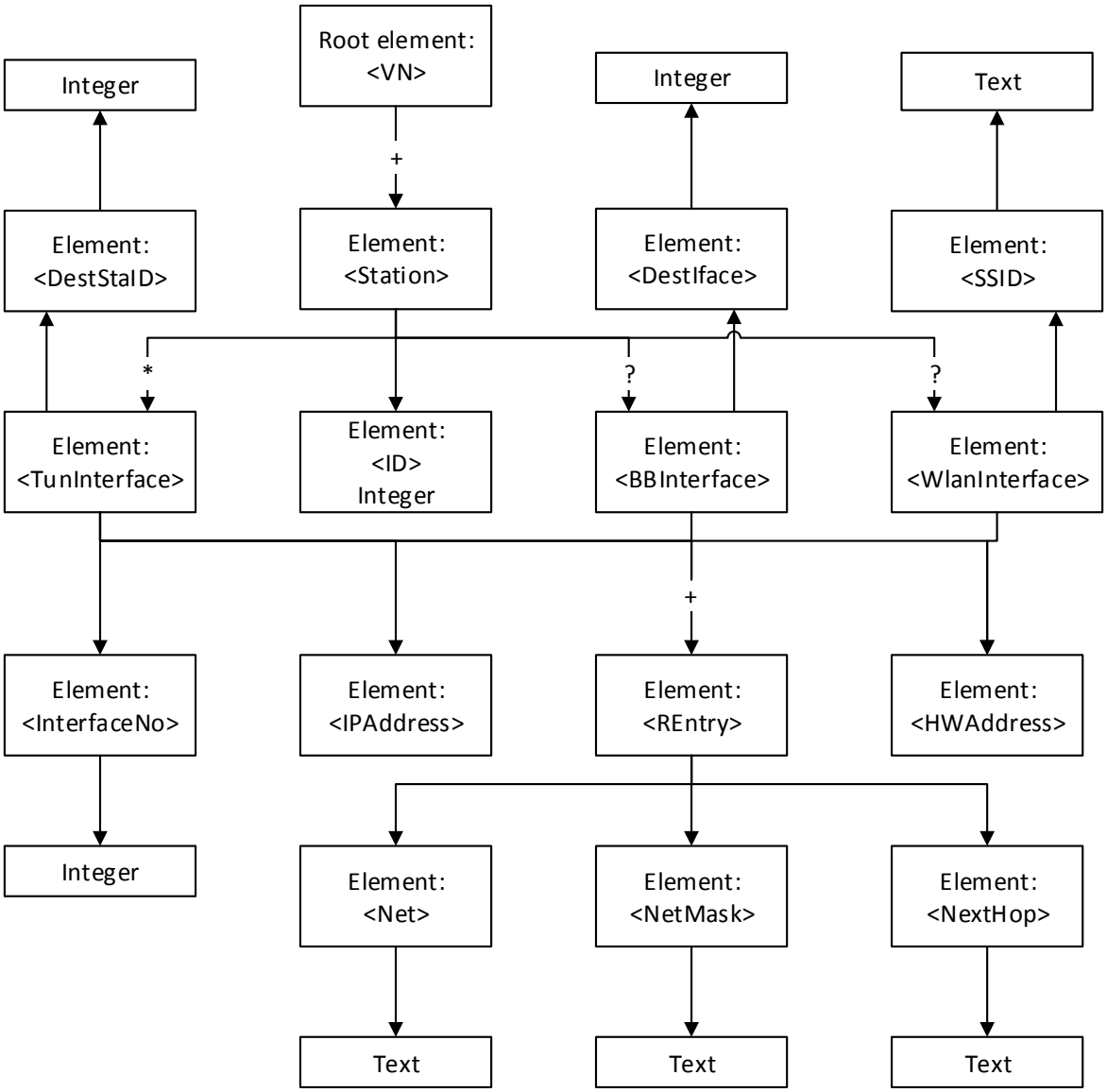


Fig. 4.1 On overview of the XML Tree of the TSF.

- *TunInterface*. This element specifies that a *tun* interface should be instantiated on the *yRouter*. The IP address and MAC address of this virtual *tun* interface is supplied by the *IPAddress* and *HWAddress* elements, respectively. The *DestStaID* parameter signifies the mesh station ID that this interface should connect to.

- *BHInterface*. This element specifies that this interface connects to a *gRouter* on the user machine. Only a mesh portal can have a *BHInterface*.

- *WlanInterface*. This is an optional element. If this element is present, a *wlan* interface is set up on the *yRouter*. The IP address and SSID of the *wlan* interface are specified by the *IPAddress* and *SSID* elements, respectively.

- *REntry*. This element specifies the routing entries for the interface.

A snippet of a TSF file is shown below. Note how the user only specifies the network parameters relevant to his/her VN in the TSF. The user does not need to consider any of the physical setups such as the UDP ports, physical IP addresses, firewall rules, etc. All these underlying configurations are handled automatically by the WGINI server. We will see how this is handled in the next section.

```
<VN>
  <Station>
    <ID>1</ID>
    <TunInterface>
      <InterfaceNo>0</InterfaceNo>
      <DestStaID>2</DestStaID>
      <IPAddress>192.168.0.1</IPAddress>
      <HWAddress>1a:1a:1a:1a:1a:10</HWAddress>
      <REntry>
        <Net>192.168.0.0</Net>
        <NetMask>255.255.255.0</NetMask>
        <NextHop>192.168.0.2</NextHop>
      </REntry>
      <REntry>
      ...
      </REntry>
```

```
   </TunInterface>
  <TunInterface>
     ...
  </TunInterface>
  <WlanInterface>
     <InterfaceNo>2</InterfaceNo>
     <IPAddress>192.168.3.1</IPAddress>
     <SSID>MyWirelessVN</SSID>
     <REntry>
       <Net>192.168.3.0</Net>
       <NetMask>255.255.255.0</NetMask>
       <NextHop>None</NextHop>
     </REntry>
   </WlanInterface>
 </Station>
 <Station>...
 </Station>
</VN>
```

4.2 The WGINI Server

The WGINI server is comprised of three components: 1) The RPC server 2) The local database 3) The WGINI APIs. We've already gone over the RPC server in Section 4.1.1. In this section, we provide an overview of the implementation of the other two components.

4.2.1 The Local Database

We start with the implementation of the local database. The local database was implemented using SQLite [33]. SQLite is an open-source implementation of a relational database management system. SQLite implements most of the Structured Query Language (SQL) standard. Unlike many other database management systems that use a client-server database engine, SQLite is disk-based allowing the database to be easily embedded into the applications by having its library linked in with the program. SQLite is easy to setup, lightweight, and reliable making it the de-facto choice for local data storage for individual

applications.

The local database contains three tables; the Station table, the Topology table, and the Interface Table shown in Tables 4.1, 4.2, and 4.3, respectively.

Table 4.1 The Station Table.

Column Name	Type	Key	Remarks
ID	Integer (Autoincrement)	Primary Key	Surrogate Key
IPAddress	Text	No	
MaxWlanInterfaces	Integer	No	
isPortal	Integer	No	Boolean

Table 4.2 The Topology Table.

Column Name	Type	Key	Remarks
ID	Integer (Autoincrement)	Primary Key	Surrogate Key
HostIP	Text	No	

Table 4.3 The Interface Table.

Column Name	Type	Key	Remarks
StationID	Integer	Foreign Key Alternate Key (1 of 3)	Station.ID
DestStaID	Integer	Foreign Key Alternate Key (2 of 3)	Station.ID
TopologyID	Integer	Foreign Key Alternate Key (3 of 3)	Topology.ID
Type	Text	No	Tun|Wlan|BH
InterfaceNo	Integer	No	

The Station table stores the relevant information of the mesh stations. For each mesh station in the MBSS, the table stores its unique ID, its physical IP address on the MBSS, the maximum number of *wlan* interfaces it can support, and whether or not this mesh station is the mesh portal. The maximum number of *wlan* interfaces can be increased by adding another USB wireless dongle to the Arduino Yun or by using another embedded device altogether. The Station table is pre-configured by the instructor or lab administrator during setup. Future work aims at automating this process by having the mesh stations communicate directly with the WGINI server when they enter and leave the MBSS.

Each VN that is currently deployed is assigned a unique ID. When a user sends a request to deploy a VN, the IP address of the user is captured by the WGINI server. The Topology

table maps the unique ID of the requesting user's VN to the IP address of the requesting user.

The Interface table stores information about all the interfaces that are currently deployed on the wireless mesh. Each interface entry contains the ID of the mesh station that this interface is deployed on, the topology ID that this interface belongs to, the interface number of this interface on the *yRouter*, and the interface type. The interface type could be of type *Tun*, *Wlan*, or *BH*, signifying whether this interface is a *tun* interface, *wlan* interface, or a "back-haul" (BH) interface, respectively. A BH interface is the interface that connects the wireless VN to the process-emulated portion of the topology that is running on the user machine. In case of a *tun* interface, the Interface table additionally stores the ID of the mesh station that contains the *yRouter* that this *tun* interface connects to.

4.2.2 The WGINI API

In this section, we discuss the implementation details of the WGINI server API modules. Specifically, we discuss what occurs when the user invokes any of the three APIs exposed, namely, *Check*, *Create*, and *Delete*. These APIs provide a user-friendly and comprehensive way of creating and deleting VNs.

The Check API

The *Check* API is called by the user to retrieve information about the wireless mesh network. The user does not need to pass any input parameters to the function. When the *Check* API is invoked, it queries the local database to retrieve all the entries in the Station Table. It then creates an array of objects that are returned to the user. Each object contains the mesh station ID, the maximum number of *wlan* interfaces available, and a boolean specifying whether this mesh station is a mesh portal. The object also contains the current number of *wlan* interfaces deployed on each station. The WGINI server retrieves the number of *wlan* interfaces deployed on a given station by querying the Interface table. The user can then use this returned object array to determine the number of stations available, and whether or not a *wlan* interface can be deployed on a given station.

The Create API

Figure 4.2 illustrates the *Create* API procedure. The *Create* API takes as input the TSF file as well as the IP address of the requesting user. The IP address is automatically retrieved when the user sends the request. The API starts by parsing the TSF file to extract its contents into Python objects. It then validates the TSF file to insure that it adheres to the design requirements. Specifically, if a backbone interface is specified, it insures that it is present on the mesh portal, it insures that at most one *wlan* interface is specified on a given station, and it insures that a *wlan* interface is available on each station where a *wlan* interface was specified. If the TSF passes the validation stage, an entry is added to the Topology table for the requesting user's IP address. The Interface table is also updated to include all the interfaces of the user's requested topology. Once this step is complete, the topology is ready for deployment. The next stage is to create the configuration scripts for each *yRouter* in the topology. The configuration scripts are executed by the *yRouter* upon start up. The configuration scripts contain *yRouter* specific commands to set up the required interfaces on the *yRouter*. The scripts also contain the routing information of each *yRouter*. Currently, the GINI toolkit uses static routing to populate the routing tables of the *gRouters* and *yRouters*. Future work aims at developing support for dynamic routing protocols, such as Open Shortest Path First (OSPF), on the routers.

The *tun* interface is automatically assigned a UDP source port based on the following equation:

$$UDPSourcePort = 50000 + 100 \times TopologyID + InterfaceNo \quad (4.1)$$

50000 is an arbitrary number that falls within the range of $49152 - 65535$. This range is dedicated to private ports that cannot be registered by the Internet Assigned Numbers Authority (IANA) [34]. The maximum number of topologies that can be supported at a given time is set to 100. The maximum number of interfaces on a given *yRouter* is also set to 100. Hence, for a given *tun* interface on a given topology, the UDP source port will be 5XXYY where XX is the topology number and YY is the interface number. This partitioning is somewhat similar in concept to flowspace partitioning, described in Section 2.4.2, albeit at a rather smaller scale.

The *tun* interface creation command on the *yRouter* takes as input the virtual IP

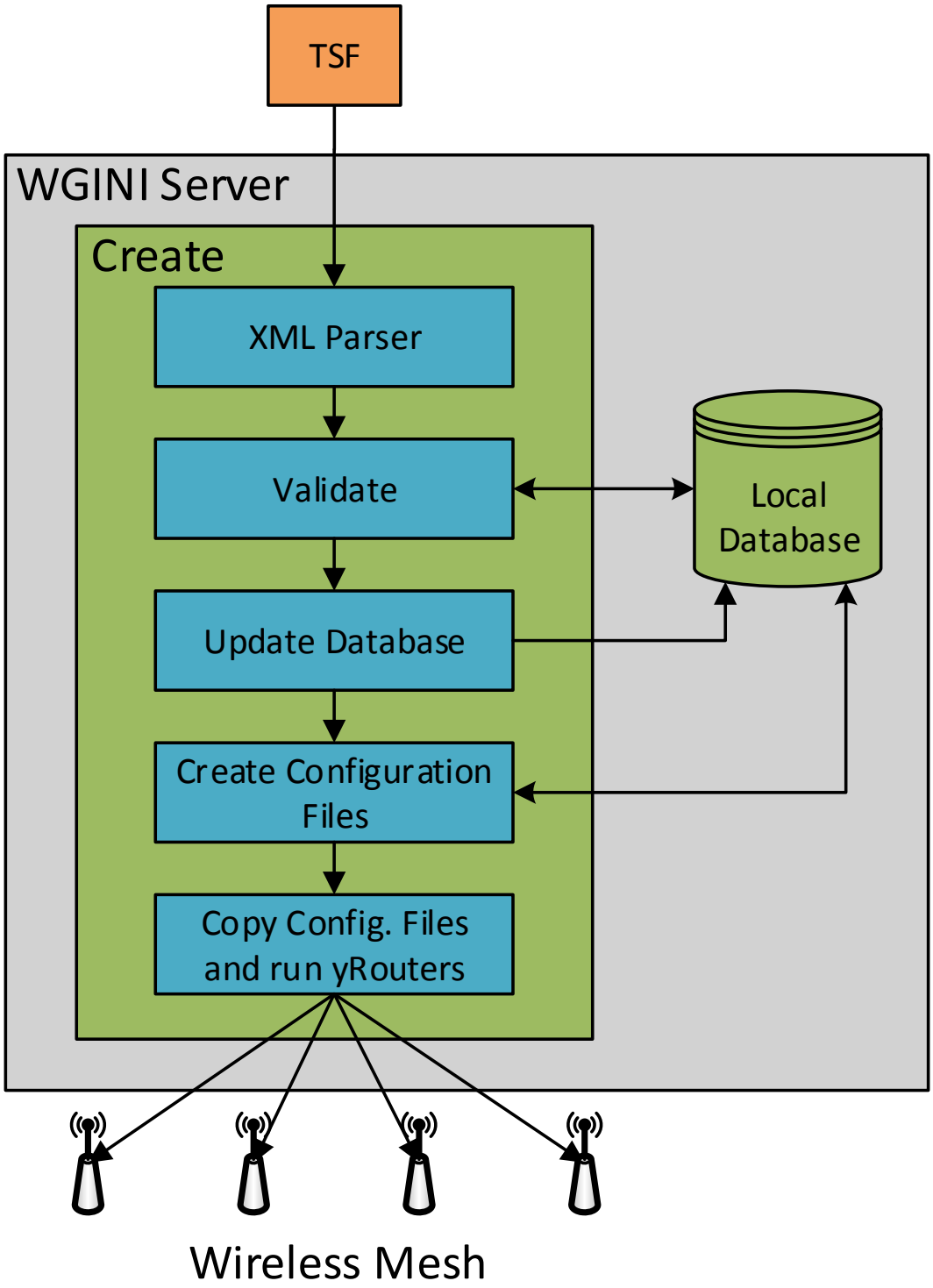


Fig. 4.2 The *Create* API procedure.

address and virtual MAC address of the interface. It also takes as input the physical IP address and the interface number of the *tun* interface on the other end of the virtual link. The Station table is queried with the destination station ID to retrieve the physical IP address of the destination mesh station. The Interface table is queried with the topology ID, and destination Station ID to retrieve the interface number of the destination interface. Equation 4.1 is then used to calculate the destination port number.

The *wlan* interface creation command on the *yRouter* takes as input the IP address, MAC address, and SSID of the *wlan* interface. It then forks a process that runs a bash script that creates a new VAP interface in OpenWrt and assigns it the input IP address, MAC address, and SSID. As mentioned in Section 3.4, the bash script also sets up the firewall rules, and a DHCP server on the VAP interface. The *yRouter* then binds a Raw Socket to the newly created VAP interface.

Once the configuration file for a *yRouter* is created, the next step is to copy this configuration file to the appropriate mesh station and run the *yRouter*. The WGINI server uses the Secure Copy (scp) command to copy the configuration file to a designated directory on the mesh station. It then uses screen [35] to run the *yRouter* passing it the path of the configuration file. Screen is a virtual console multiplexer that is used to separate programs from the Unix shell that started the program. This allows the program to run as a daemon by "detaching" the virtual console. The console can be re-attached by invoking screen and passing it the name of the virtual console. We assign the name of the virtual console based on the topology number to insure that the name is unique and can be easily retrieved. Re-attaching the virtual console allows us to access the standard input/output of the *yRouter* console during run-time.

The Delete API

Once a user is done with an experiment, the user invokes the *Delete* API to free up the resources that are held by the VN. Figure 4.3 illustrates the *Delete* API procedure. The *Delete* API starts by retrieving the user's topology ID from the user's IP address. It then queries the Interface table to retrieve all the mesh stations that have a *wlan* interface deployed by the user. A list of station IDs of all the matching entries is returned. The WGINI server then queries the Station Table to retrieve the IP address of each mesh station on the MBSS. It uses the IP addresses to ssh into each station, and runs a bash

script that deletes the *wlan* interface from the kernel. Once this step is complete, it is time to kill all the *yRouters* that are part of the user's topology. The interface table is queried again for all entries that have a matching topology ID but this time without specifying the interface type. Note that the SQL query used does not return duplicate entries. A list of all the station IDs used by the topology is returned. Similarly, this list is used to ssh into each station on the list and kill the *yRouter* process pertaining to the user's topology. The *yRouter* is programmed to save a file that contains the *yRouter's* process ID in a designated directory. The file is named with the user's topology ID. This file is read by the bash script to retrieve the process ID to invoke the *kill* command on the OS. Once complete, the local database is updated by removing the user's interface entries in the Interface table and the user's topology entry in the Topology table.

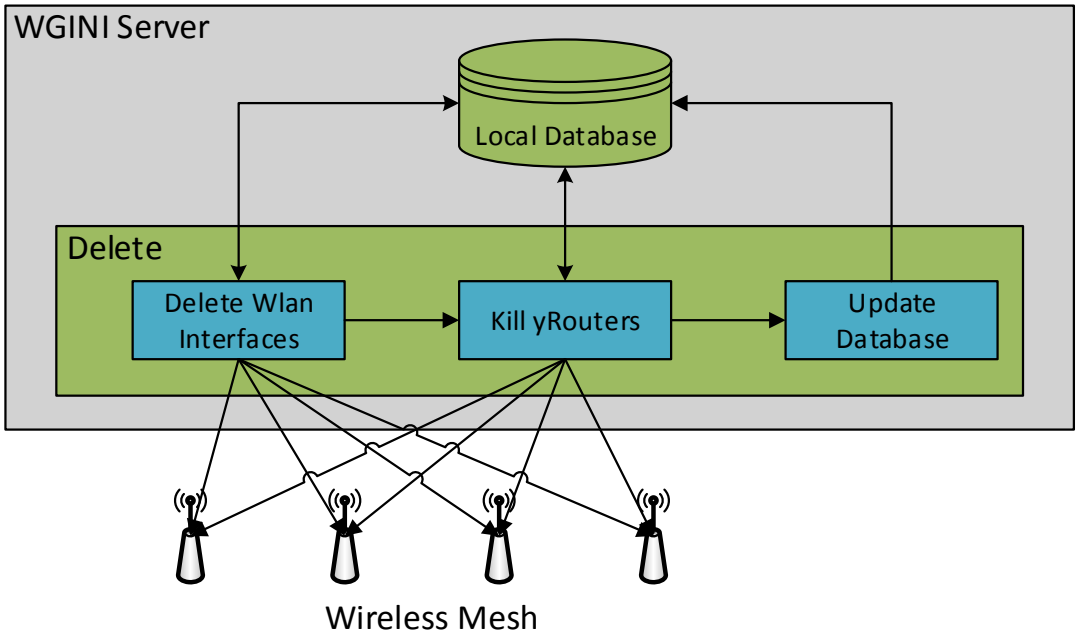


Fig. 4.3 The *Delete* API procedure.

Chapter 5

Wireless GINI: Evaluation and Use Cases

5.1 Performance Evaluation

We've seen in the previous chapters the technologies behind Wireless GINI that enable isolation and connectivity. In this section, we focus on the performance aspect of the system, or, more specifically, latency and bandwidth. Primarily, we evaluate the latency and bandwidth effects from three dimensions. The first dimension is how latency and bandwidth are impacted in relation to the size of an individual VN. The second dimension is how latency and bandwidth are impacted by the activity of other VNs deployed on the platform. The third dimension is the packet processing overhead incurred by using the *yRouter*. It is important to note that our goal is not demonstrate how our environment outperforms other environments. Indeed, this is not possible since no other platform with the same features exists in the literature. As elaborated in Chapter 6, each existing system has its unique design choices and features that provide the ability to perform certain kinds of experiments, but which limit the ability to perform other kinds of experiments. The Wireless GINI system is no different. Hence, our goal in this chapter is to investigate the performance characteristics of the system with respect to certain important aspects of the design.

5.1.1 Impact due to Topology Size

In this section, we evaluate the end-to-end latency and bandwidth as we increase the number of *yRouters* between the end hosts. The experimental setup is shown in Figure 5.1. The physical setup consists of one PC, one switch, eight mesh stations, and one laptop. The PC is a Dell Studio machine, with a 2.4 GHz Intel Core 2 Quad processor and 4 GB 800 MHz DDR2 RAM, running Ubuntu 12.04. The laptop is a Toshiba Satellite, with a 2.0 GHz AMD Turion X2 Dual-Core processor and 3 GB 400 MHz DDR2 RAM, running Ubuntu 14.04. The stations are the Arduino Yuns discussed in Section 2.3. We evaluate the latency and bandwidth between the *gLinux* and the laptop as the hop count between the two end hosts gradually increases from 1 *yRouter* to 8 *yRouters*. The *iperf* command is used to measure the bandwidth between the end hosts while the *ping* command is used to measure the latency. The average of 30 samples is calculated to determine the latency. The maximum standard deviation was measured at 13.29, which occurs when the hop count is 8.

The laptop is initially connected to the WLAN deployed on yRouter1. We measure the bandwidth and latency at this setup. As can be seen from the figure, the traffic goes through 1 *gRouter* and *yRouter*. We then disconnect the laptop from yRouter1's WLAN and connect it to yRouter2's WLAN. The traffic now goes through 1 *gRouter* and 2 *yRouters*. Similarly, we measure the latency and bandwidth at this new setup. We repeat the same procedure for yRouter3 up to yRouter8. The results are shown in Table 5.1. The plots of the Round Trip Time (RTT) and bandwidth are shown in Figures 5.2a and 5.2b, respectively. As can be seen from the figures, the latency linearly increases with respect to the hop count (at an average rate of about 20%), while the bandwidth linearly decreases (at an average rate of about 5%).

5.1.2 Impact due to Other Virtual Networks

Since the VNs share the same processor, wireless device, and wireless channel, it is inevitable that the activity of one VN effects the performance of the other VNs that are simultaneously deployed on the platform. We investigate the inter-VN impact on latency and bandwidth by deploying two VNs simultaneously as shown in Figure 5.3. The smartphone used is an iPhone6 running iOS 8.3. The first topology is used to inject packets into the network to emulate activity. We use the *hping2* command to inject ICMP echo request packets with

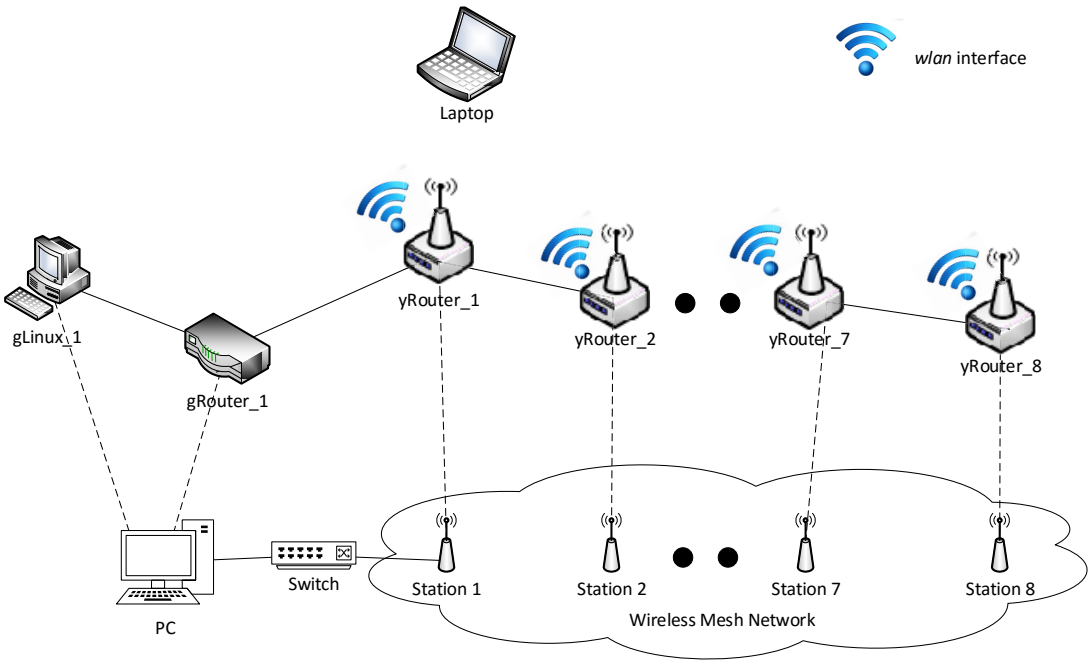
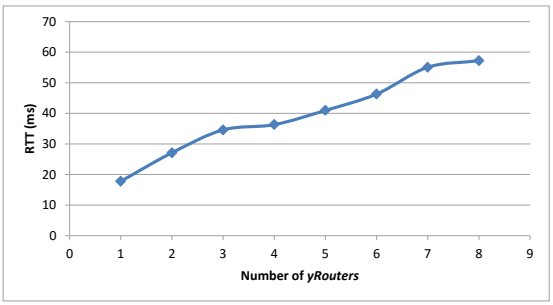


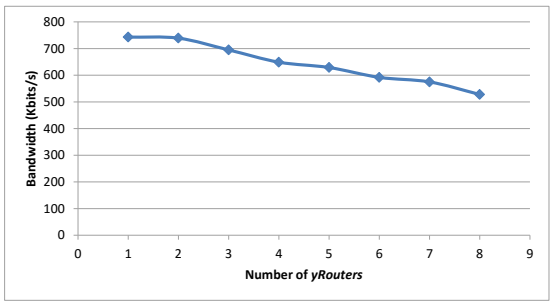
Fig. 5.1 Experimental setup for evaluating topology size effects on bandwidth and latency.

Table 5.1 Latency and bandwidth measurements due to hop count increase.

Number of *yRouters*	RTT (ms)	Bandwidth (Kbits/s)
1	17.780	743
2	27.104	739
3	34.583	695
4	36.305	649
5	40.965	629
6	46.297	592
7	55.066	575
8	57.206	528



(a) RTT vs Hop Count.



(b) Throughput vs Hop Count.

Fig. 5.2 The bandwidth and latency plots with respect to the hop count.

a payload of 500 bytes. The packets are generated by gLinux1 and destined to yRouter3. We gradually increase the traffic intensity by decreasing the wait time between packet injections. Topology 2 is used for performance measurements. We measure the latency and throughput on Topology 2 between the smartphone and the laptop at several packet injection rates of the *gLinux* running on Topology 1. Table 5.2 shows the results. The RTT and bandwidth plots are shown in Figures 5.4a and 5.4b, respectively.

As shown in Figure 5.4a, the RTT time on Topology 2 slightly increases as we increase traffic on Topology 1. On the other hand, the bandwidth gradually decays before plateauing at a certain bandwidth. The reason for this plateau is that beyond this point, the two *yRouter* processes are equally competing for CPU utilization. At low injection rates, the *yRouters* on Topology 1 are mostly idle. Hence, during the bandwidth measurements, most of the processing power at the stations and the bandwidth at the wireless channel are dedicated to the *yRouters* of Topology 2. As the injection rates increase, the two *yRouters* on a given station start competing for CPU time and channel bandwidth. Eventually, the packet injection rate becomes greater than the *yRouter*'s packet processing rate. As a result, the incoming packets get queued at the input buffer of Topology 1's *yRouters*. The input buffer eventually becomes full and all the extra incoming packets are dropped by the *yRouters*. In other words, the bottleneck becomes the processing rate of the *yRouters*. Since the OS's process scheduler is responsible for distributing CPU time among the running processes, each *yRouter* process gets a share of the processing time based on the scheduler's policy. Generally, each *yRouter* will get an equal portion of the processing time.

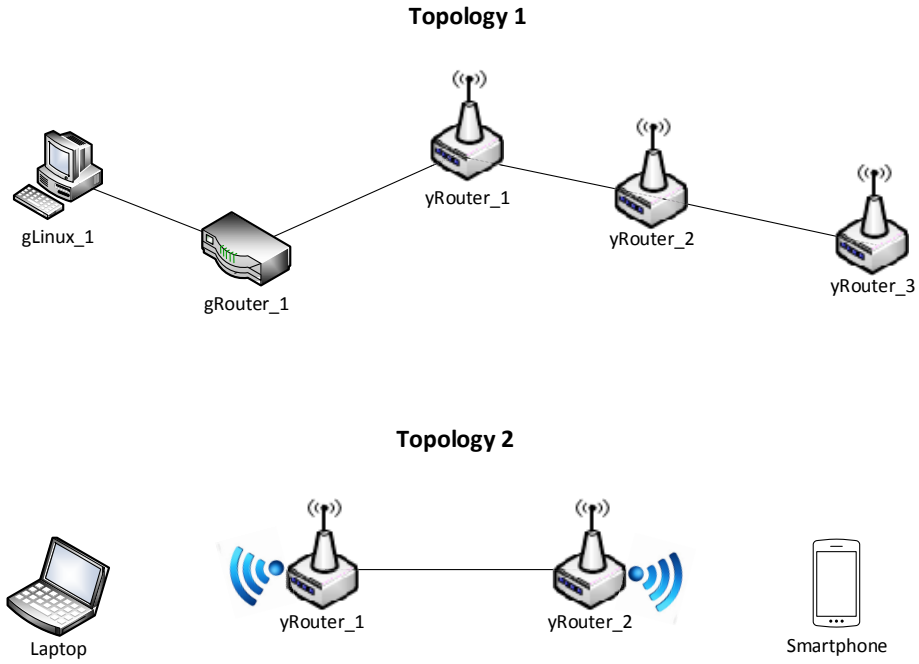
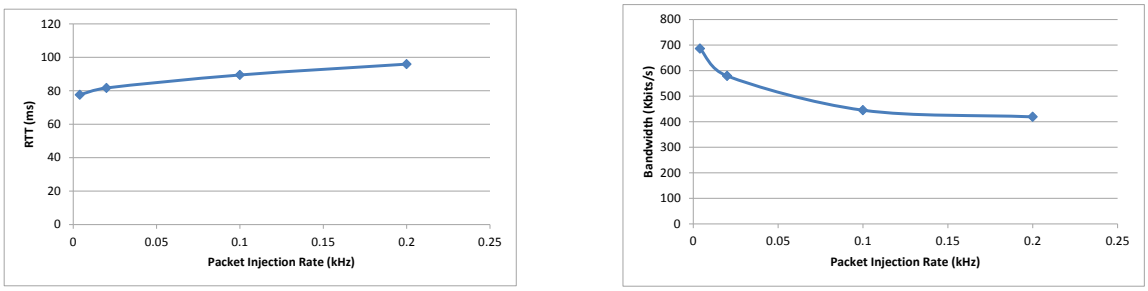


Fig. 5.3 Experimental setup for evaluating inter-topology effects on bandwidth and latency.

Table 5.2 Latency and bandwidth measurements with respect to the various packet injection periods.

Packet Injection Period (ms)	RTT (ms)	Bandwidth (Kbits/s)
250	77.611	696
50	81.691	579
10	89.433	445
5	95.924	419



(a) RTT vs packet injection rates.

(b) Throughput vs packet injection rates.

Fig. 5.4 The bandwidth and latency plots between the two end hosts on Topology 2 with respect to the packet injection rates caused by the end hosts on Topology 1.

5.1.3 Virtualization Overhead

We now investigate the overhead incurred due to the virtualization techniques employed. There are two main sources of overhead: 1)The overhead due to processing the packets in user space as opposed to the kernel 2) The overhead due to encapsulation and decapsulation. The experimental setup is shown in Figure 5.5. Two experiments are conducted. For each experiment, we measure the bandwidth and latency between the laptop and the smartphone. In the first experiment, we use *yRouters* to connect the two WLANs of the laptop and smartphone. In the second experiment, we configure a *wlan* interface on each mesh station similar to the setup of the *yRouter* scenario. However, the kernel routing tables of the mesh stations are configured to directly route packets destined to the WLANs. No firewall rules are employed. Hence, when a packet reaches the *wlan* interface of a mesh station, it is processed by the kernel and routed directly to the mesh interface. Thus, there is no overhead due to copying the packet to user space, processing it in user space by the *yRouter*, and encapsulating the packet when sending it over the *mesh* interface. Similarly, when the packet reaches the *mesh* interface of the mesh station, it is forwarded directly to the *wlan* interface. Table 5.3 shows the results of the experiment. As can be seen from the table, there is a 46% decrease in latency when kernel routing is employed. In addition, there is a 10 fold increase in bandwidth. This is due to the bottleneck incurred on the *yRouter* due to the processing speed which simply cannot match that of the kernel due to the extra overhead related to sending the packet from kernel space to user space and back again to the kernel.

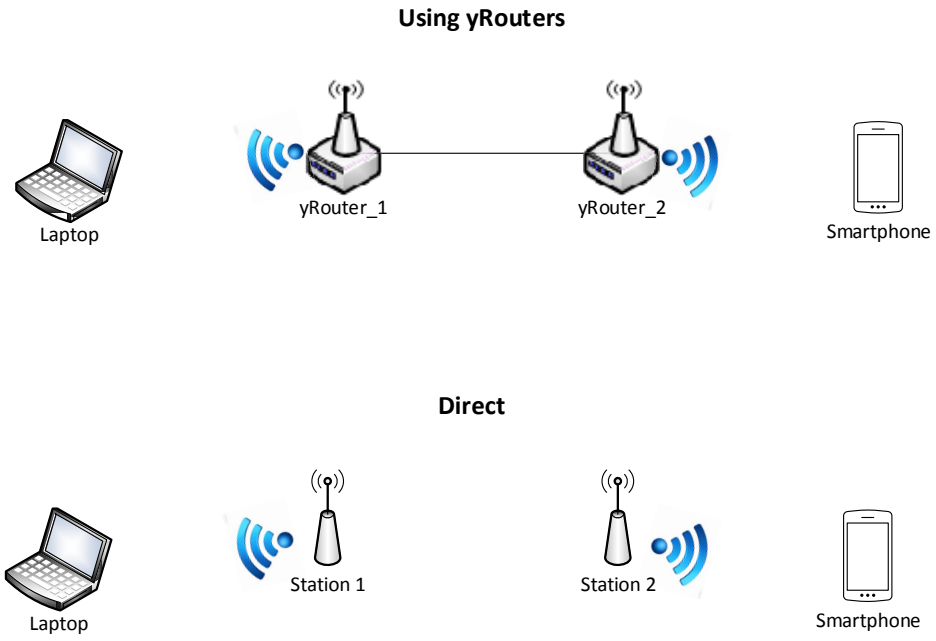


Fig. 5.5 Experimental setup for evaluating overhead on bandwidth and latency due to the network virtualization techniques.

Table 5.3 Latency and bandwidth measurements for the virtualization overhead experiment.

Setup	RTT (ms)	Bandwidth (Kbits/s
Using *yRouters*	76.805	667
Direct	53.231	7510

5.2 Use Case Scenarios

In this section, we discuss some of the applications that could be conducted on the Wireless GINI platform. We provide two experiments that can used to provide a hands-on learning experience for students. The first suggested experiment relates to mobility in wireless networks. Specifically, we provide an experimental scenario for the implementation of Mobile IP [36]. The experiment helps students understand the design and implementation aspects of the Mobile IP protocol. An overview of Mobile IP and the suggested experimental scenario is provided in Section 5.2.1. The second experimental scenario is related to IoT applications. We provide an overview of the IoT experimental scenario in Section 5.2.2.

5.2.1 Mobile IP

Many Applications, such as Voice Over IP, experience a significant degradation in service due to abrupt changes in network connectivity as users transition from one network to the other. This usually occurs in situations where wireless clients roam between overlapping wireless networks. Mobile IP aims at reducing this degradation by maintaining the TCP connection between the mobile client and the static server without modification to the underlying TCP/IP protocol. Mobile IP is an Internet architecture and protocol standard that aims at providing a solution to the issue of mobility management on the Internet [37]. Mobile IP aims at making mobility transparent to the higher layer protocols. It allows application code to serve mobile and non-mobile connections alike without being concerned with the potentially changing IP address of the mobile user. Indeed, an entire book has been written on this subject (see [38]), so our modest goal here is to provide a general overview and to illustrate a common-case scenario of its use on the Wireless GINI platform.

Mobile IP introduces a new architecture to provide location transparency that allows mobile nodes to have two addresses; a permanent address, and a Care-Of Address (COA). A permanent address is the address of the mobile user at its home network. We can think of the home network as the permanent network of the mobile node. This is the network where the mobile node seems reachable by the rest of the Internet even when the mobile node is on a different network. The COA is the address of the mobile node on a foreign network. The foreign network is the network that the mobile node is attached to when it is not on the home network.

The central idea behind Mobile IP is that the mobile node is assigned a permanent address that does not change as the mobile node goes from one network to the other. This permanent address is assigned by the home agent. A home agent is a node on the home network that is responsible for insuring that the mobile node can be reachable by its home address even when the mobile node is on a foreign network. When a mobile node goes from the home network to another network (the foreign network), the mobile node is assigned another IP address at the foreign network (the COA). A foreign agent is responsible for assigning the COA to the mobile node. The foreign agent is also responsible for communicating with the mobile node's home agent to register the mobile node's COA at the home agent. This allows the mobile node to be reachable by the home agent at any given time.

An entity that wishes to communicate with a mobile node is called the correspondent node in the Mobile IP nomenclature. Now let us look at what happens when a correspondent node wishes to send a packet to the mobile node. Figure 5.6 provides an example that illustrates this procedure. The correspondent addresses the IP packet to the mobile node's permanent address regardless of whether the mobile node is attached to its home network or to a foreign network. In other words, mobility is completely transparent to the correspondent. Once the packet reaches the home network, it is intercepted by the home agent (step 1 in Figure 5.6). The home agent looks up the COA of the mobile node that has a permanent address that matches the destination address of the intercepted packet. It then tunnels the packet to the foreign agent using the mobile node's COA (step 2 in Figure 5.6). Once the tunnelled packet reaches the foreign agent, it is decapsulated and the original packet is sent to the mobile node (step 3 in Figure 5.6). The mobile node can send a packet directly to the correspondent node (step 4 in Figure 5.6). The mobile node uses its permanent address as the source address and the correspondent's address as the destination address.

We can summarize our discussion by listing the three main components of the Mobile IP standard:

- *Agent Discovery.* This refers to the protocols used by a home or foreign agent to advertise its services and for mobile nodes to solicit the services of a foreign agent or a home agent.

- *Registration with the home agent.* This refers to the protocols used by the foreign

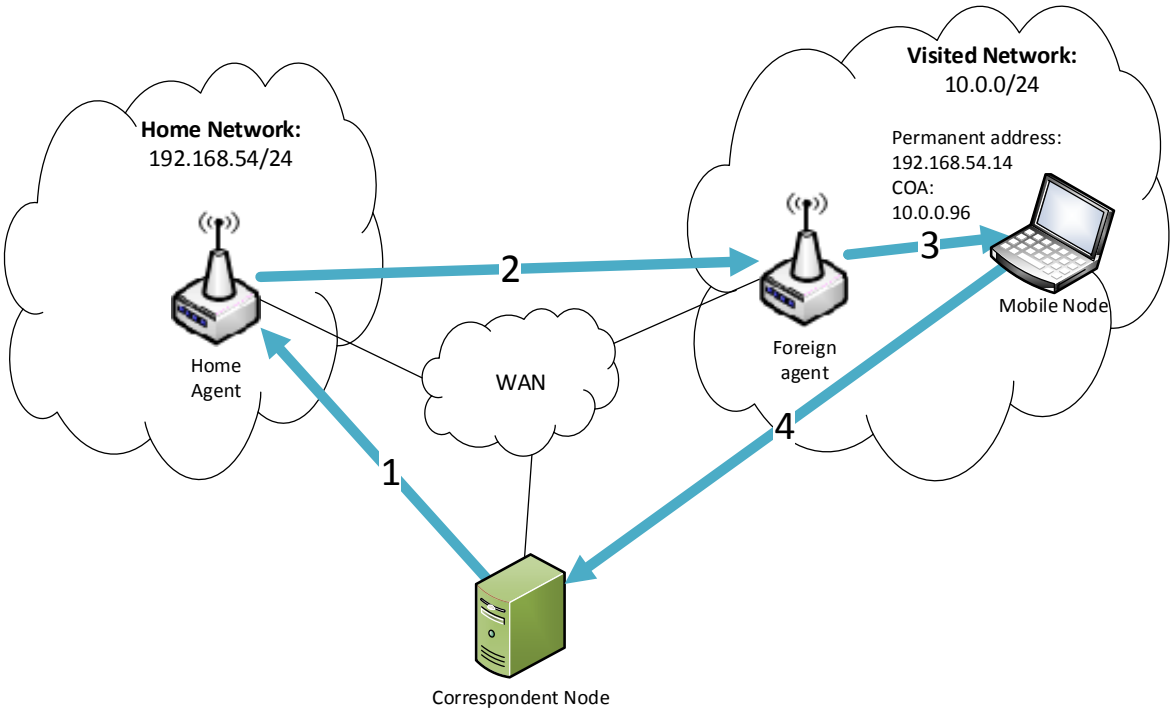


Fig. 5.6 The Mobile IP routing procedure.

agent to register the mobile node's COA with the mobile node's home agent.

- *Indirect packet routing.* This refers to the protocols used by the home node to forward packets to the mobile node.

A programming assignment can consist of implementing these components on the *yRouters* and *gRouters*. Since the *yRouter* implementation is open-source, the students can modify the source code to implement these functionalities. They can then test their implementation on the Wireless GINI platform. The user can setup a topology that contains multiple overlapping WLANs on the platform. The user can then use one's wireless device to transition from one WLAN to the other while maintaining a TCP connection to an emulated server. The emulated server can be implemented by running a service on a *gLinux* that is running on the host machine.

5.2.2 IoT Applications

Having Wireless GINI enables a whole new dimension of experiments in the IoT application space. The IoT phenomenon has attracted great interest from the research and industrial communities. The decreasing costs of embedded devices, coupled with increases in their computing performance and energy efficiency, have paved the way for a vast number of applications in numerous areas. These areas include home automation, security, e-Health, environmental monitoring, and industrial control to name a few. It comes as no surprise that technology research and advisory corporations predict that the number of IoT devices will reach 26 billion by 2020 [39]. Students can leverage the Wireless GINI system to explore this new realm of exciting opportunities.

We believe that the best IoT assignment for students on Wireless GINI should allow students to utilize their creativity to create, implement, and test their novel ideas. We believe that this will provide a more interesting and engaging learning experience. It also allows students to exercise their creativity to come up with potentially viable and experimentally tested solutions and applications in the growing realm of IoT. With that said, we suggest a viable IoT experiment in this section that can be conducted on Wireless GINI to help get started in that venue.

One of the experiments that could be implemented on the Wireless GINI platform pertains to smart lighting applications. In this experiment, a smart Wi-Fi enabled LED light bulb could be connected to the student's VN via the *wlan* interface. The smart light bulb's luminosity and color can be controlled over Wi-Fi. The setup is shown in Figure 5.7. In this setup, the student implements a home automation system where the light bulb is controlled using his/her smart phone. The student can develop a small mobile application on one's smart phone to interface with the smart lighting system or could develop a web application hosted on a *gLinux* node in his/her topology. The *gLinux* could host an HTTP application server that takes input from the student's laptop or smart phone and uses it to configure the lighting parameters. The system can be implemented such that the light bulb automatically turns on when the client's smart phone or laptop connects to the same WLAN that the smart light bulb is connected to and automatically turn off once the client's device disconnects from this WLAN. The application server could be implemented such that it remembers the client's lighting preference and automatically sets the light bulb to this saved preference when the client joins the WLAN. As an extension, a multi-client

lighting system could be set up that saves the preferences of multiple clients and configures the smart light bulb based on the preference of the client that is currently connected to the WLAN on the student's VN. When multiple clients are connected to the WLAN at the same time, the student can implement a brokage system with a pre-configured policy. This policy can be priority-based, where each client is assigned a priority level and the client with the highest priority gets to have his lighting preference supplant that of the other clients currently connected. The policy could also be voting-based or a combination of policy modes that are specified by the owner of the VN. The application server should be implemented to uphold the configured policies.

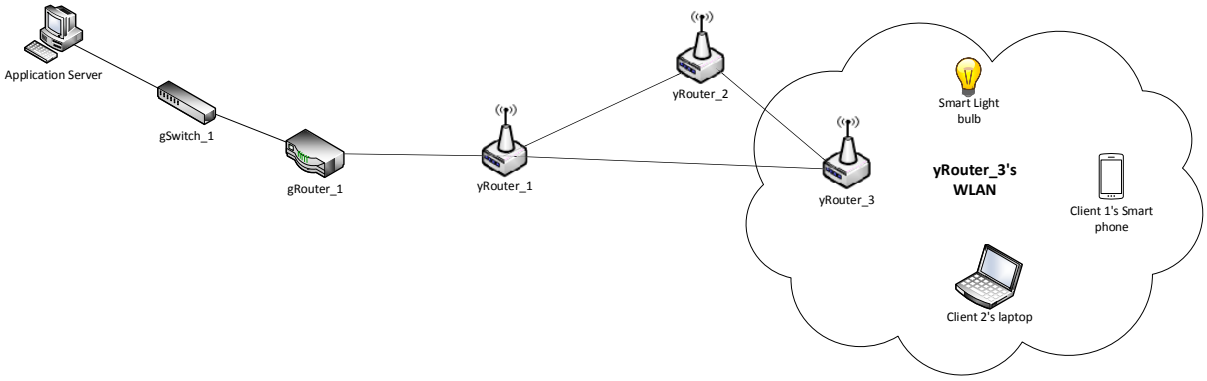


Fig. 5.7 An example of a smart lighting system that can be implemented on Wireless GINI.

Chapter 6

Related Work

Before delving into the current state-of-the-art network emulation toolkits available today, it is important to make the distinction between network *emulation* and network *simulation*. In simulation, the goal is to develop an analytical model that computes the internal state of the target while abstracting away the functional features that are not relevant in the computation. These functional features include the filesystem, the command line interfaces, and the message formats. Simulation is well-suited in testing the performance of new algorithms in large-scale scenarios. It is important to note that, unlike emulation, the experiments need not be conducted in real-time and indeed it is not uncommon for large-scale network simulation experiments to last several hours/days depending on the size of the simulation and the performance of the hardware running those simulations. Popular network simulators include ns-2 [40], OPNET [41], and OMNet++ [42].

On the other hand, emulation aims at replicating the observable behaviour of a target to match its intended functionality. Functional details are therefore exposed to the user to accurately replicate the real-life operation of the target. Unlike simulations, the user has the ability to observe in real-time the actions of the emulated device. This comes at the cost of an increase in the computational and storage requirements as compared to simulation. Due to its realism and observability, network emulation has been the method of choice for a pedagogical approach to computer network education.

There is a plethora of network emulation toolkits and testbeds that are available for computer networking experimentation. They can be divided into two groups: testbed-oriented emulation, and process-oriented emulation. As mentioned in Chapter 1, testbed-

oriented emulators emulate a real-life network by providing a miniature physical setup that would replicate an actual internet. On the other hand, process-oriented emulators run the components that compromise the network as a set of processes that communicate via IPC mechanisms such as sockets and pipes. Each network component, such as workstations, routers, and switches, is emulated by running a process that functions similar to a lightweight version of the component and often runs the same network services that an actual component would run. For example, a UML is often used to emulate a workstation in the network. UML is a user-space virtual machine that runs a lightweight version of Linux and supports basic networking functions such as **route**, **ifconfig**, and **ping**.

Due to the importance of experimentation in systems-oriented courses in computer science curricula such as Computer Networks, numerous network emulation toolkits have been developed over the years. Each network emulator is developed based on a certain design philosophy. Section 6.1 reviews some of the more popular process-oriented network emulation toolkits that are available. Section 6.2 reviews the testbed-oriented toolkits.

6.1 Network Emulation Toolkits

Process-oriented emulation toolkits can further be divided into two categories: CLI-based, and GUI-based. CLI-based emulators require the user to provide configuration files that are written in a format that is specific to the emulator in order to describe the user's topology. The file type of a configuration file can range from regular text files, to XML files, to code written in a popular scripting language. Afterwards, the user can use the CLI commands provided by the emulator to run the topology (also called laboratory or lab for short), execute commands on a given node in the topology, and stop a running topology. GUI-based emulators provide a graphical interface for creating and running a topology by allowing the user to drag-and-drop the available network components onto the GUI's canvas. Users can double-click on a component during run-time to display the component's console.

6.1.1 CLI-based Network Emulators

An example of CLI-based emulator is Netkit [7]. In Netkit, the user must obey Netkit's file directory structure and place the configuration files in the appropriate directories. For example, the start-up scripts for each node, and the topology descriptor file must be placed

in the top-level directory of the lab. The topology descriptor file is a regular text file that is written in a format that is understood by Netkit to describe the connections between various nodes in the lab. The start-up scripts are normal shell scripts that are executed as soon as the topology starts. Each node in the lab then has its own directory where other configuration files that are specific to that node can be placed to run a particular service on that node. For example, an Apache configuration file can be placed in a given node's directory to be able to run on a Web server on that node.

Another CLI-based emulator is Mininet [43] which is targeted towards creating OpenFlow-based [44] networks. Users can create custom labs in Mininet by creating Python scripts that utilize Mininet's Python package. The Mininet Python package contains APIs for assigning nodes to a topology, specifying the node type (i.e. host, switch, controller), and assigning the links between the nodes. Mininet emulates hosts by leveraging network namespaces that are supported in the Linux Kernel 2.2.26+. Each network namespace contains its own unique copy of the network stack. Mininet uses network namespaces to emulate network nodes by grouping the user-level processes of each node into separate network namespaces instead of using VMs to emulate each node. This approach allows Mininet to be more lightweight albeit at the expense of requiring root privileges and an increased difficulty to setup and install. The standard distribution image of Mininet requires installing a full-blown virtual machine image on Virtualbox or VMware.

Virtual Networks over LinuX (VNX) [8] is another process-oriented emulator that uses XML files to specify a topology. It uses UML to emulate workstations and routers and uses the UML switch to emulate an ethernet switch. It is an evolution of Virtual Networks User Mode Linux (VNUML) [45].

Manage Large Networks (MLN) [46] is another CLI-based network emulator that runs process-oriented networks using UML, Xen [47], or VMware Server [48] as the VMs. MLN uses the proprietary *mln configuration language* to specify the topology. The language is based loosely off of Perl and is used to configure, run, and test the labs.

Overall, CLI-based network emulators have multiple drawbacks in common. Firstly, these emulators require the user to master tools, languages, APIs, and/or commands that are specific to the given emulator. This provides a steep learning curve for the user and distracts from the primary goal of providing a tool to enhance the learning process of networking concepts. Secondly, these emulators are hard to use, time-consuming and often error-prone since the user must manually configure all the parameters for each interface at

each node on the network using the proprietary format. Thirdly, they do not provide an intuitive visual illustration of the topology which makes it harder to reason about the flow of packets through the network especially for larger networks.

6.1.2 GUI-based Network Emulators

In this section, we provide an overview of the more popular GUI-based process-oriented network emulators available. GUI-based network emulators provide a unified and simple means of configuring a network topology as well as a visual feedback of the topology as it is being created. GUI-based emulators provide a more intuitive and user-friendly experience towards topology creation. An example of one is GNS3 [49] which uses Dynamips [50] to incorporate proprietary Cisco routers into the topology. Dynamips is an emulator for Cisco routers that is capable of accurately emulating the Cisco routers on an instruction-by-instruction basis. It does so by directly booting an actual firmware image of the Cisco router, Cisco IOS. However, the Cisco image typically uses up 256 MB of RAM along with considerable CPU time. This makes the emulator fairly resource intensive which makes it unsuitable for running more than a small sized network on an average workstation.

Other platforms that use proprietary routers for their labs are Junosphere Classroom [5] and Cisco Learning Labs [6] provided by Juniper and Cisco, respectively. They provide a commercial Software as a Service (SaaS) platform where the user's labs are hosted on the vendor's cloud. Users are charged based on the amount of time the SaaS platform is used and on the number of nodes used in a lab. This can prove costly especially for an average class size through a 4-month duration. Another commercial platform is Estinet [51] which puts more emphasis on SDN.

Common Open Research Emulator (CORE) [52] is another GUI-based process-oriented emulator that uses the same network namespace technique used by Mininet in order to emulate the nodes in the topology. CORE has the capability of also emulating a wireless node by simulating a wireless channel based on a user-configurable statistical channel model between two virtual wireless nodes. The user can specify the channel parameters of each wireless link in the topology independently. Another feature of CORE is the ability to distribute the topology across multiple hosts using a C daemon called Span that tunnels packets between the two physical hosts. However, when using distributed emulation, the tunnel configuration must be manually setup by the user on each host that is part of the

distributed system before it can be used. Moreover, there is no multiple topology support on a given host machine. IMUNES [53] uses the same approach as CORE but uses the FreeBSD kernel and does not support distributed emulation.

Marionnet [54, 55] uses UML to emulate its network components. It also supports a "virtual external socket" which can be used to access the physical network of the host. However, no isolation mechanism is employed making it unsuitable for integrating it in typical physical lab settings. Cloonix [10] is another GUI-based emulator that also has the physical network access feature along with supporting Dynamips in addition to UML.

6.2 Network Experimentation Testbeds

In this section, we provide an overview of the testbed-oriented emulation platforms available in the literature. Testbed-oriented emulation platforms have the advantage of realism over process-oriented counterparts. However, they suffer from the drawbacks discussed in Section 1.1, namely cost, setup and maintenance time, scalability, and ease-of-use.

Emulab [56] is a testbed-oriented emulator that can allocate components of the physical platform to a user remotely. The platform consists of workstations, routers, switches, wireless access points, and wireless clients. However, each component can only be used by one user at a time. Emulab also supports a hybrid topology where the physical components can be connected to virtual nodes running on a physical machine that is part of the testbed. The user must manually configure the host machine and setup certain proxies to allow the traffic to reach the virtual nodes. Emulab uses OpenVZ [57], a container-based VM for Linux, to run the virtual nodes. However, the virtual nodes are rather limited. They do not provide an interface, such as a console, by which the user can run commands during run-time. Moreover, there is no isolation between the host's network stack and that of the virtual nodes being hosted.

PlanetLab [58, 59] is another testbed-oriented emulator that is aimed towards scientific research in distributed systems. It consists of a global network that consists of 1353 nodes scattered around 712 sites around the world [60]. It allows users to create custom overlay networks on top of the global physical network. Similar to PlanetLab are Global Environment for Networking Innovation (GENI) [61] and Smart Applications on Virtual Infrastructure(SAVI) [62]. GENI, which is sponsored by the U.S. National Science Foundation, is used by researchers and educators to perform research experiments and laboratory

assignments. It consists of nodes distributed around fifty sites in the U.S. It utilizes network virtualization techniques to allocate "slices" of the physical network to users. The sites can contain virtualized computation and storage resources, SDN enabled network resources and/or virtualized cellular wireless communication resources. Its Canadian counterpart, SAVI, which is sponsored by 9 Canadian universities and several industry partners, follows a similar architecture but also includes virtualized reconfigurable hardware resources such as the NetFPGA and the BEE4.

Another emulator that allows the user to remotely configure a physical network is the Open Network Laboratory (ONL) [63]. The testbed consists of 4 routers and 40 PCs and is hosted in one site. The network components are divided into clusters with each cluster consisting of 1 router and 10 PCs connected via a switch. No network virtualization is supported and the user is constrained into using a topology that is a subset of the physical topology. Moreover, only one user can use the platform at a given time.

ORBIT [64] is a testbed-oriented emulator that aims at facilitating research experiments of wireless network protocols. The testbed consists of 64 wireless nodes equipped with 802.11a/b/g wireless cards. The user can use the library APIs provided by the emulator to hook up the user's applications and/or MAC layer modifications. The testbed also provides channel measurement equipment to measure the channel parameters and artificial interference sources to modify the channel characteristics. ORBIT uses the time sharing approach where each experimenter can reserve the testbed for up to two hours and has complete control of the testbed during that time.

An example of testbed-oriented emulators that are targeted for educational purposes is the internet lab [1]. The testbed consists of four Cisco routers, four PCs, and a few Ethernet hubs. A Keyboard-Video-Mouse (KVM) switch is used for multiplexing the keyboard, mouse, and display among the four PCs. The lab setup is used as a complement to the book [1] to aid students in grasping the concepts presented in the book by suggesting experiments to perform on the lab that are related to the concepts discussed in the book.

Pan [3] proposes another testbed-oriented emulator that is used by the University of Victoria in their Computer Networking courses. The platform aims at being low cost and simple to setup and hence contains only a PC connected via ethernet to a Linksys router running the OpenWrt firmware. Users can Secure Shell (SSH) into the router and run various commands for experimentation. Each platform can be used by one user at a time.

Chapter 7

Conclusion and Future Work

7.1 Concluding Remarks

The computer networking field along with its diverse technologies and applications are growing and changing at an unprecedented rate. It is up to educational institutions to keep pace with this evolving field by evolving their platforms to help bridge the gap. We hope that Wireless GINI is a step in this direction. Wireless GINI provides an adoptable, flexible and extensible platform that will be used by the students, who some may be the future practitioners and researchers of the computer networking field, to gain a practical hands-on experience in the current trends and technologies of the computer networking field.

We conclude by summarizing the main architecture and implementation aspects of the Wireless GINI platform that address the design considerations set forth to provide a cohesive and unified framework for hosting virtual wireless networks. Specifically, the main features are:

- **User-Friendly**. This addresses the needs of the user of the platform. The WGINI server provides a unified and simple interface for deploying wireless networks. It handles all the complexity of deploying the user's topology and provisioning the wireless network.

- **Cost-effective**. This addresses the needs of the institution that will be hosting the platform. The network virtualization techniques employed allow for proper isolation and custom configuration of the VNs. This allows the wireless platform to be shared

by multiple users and thus provides a cost-effective means for servicing multiple users simultaneously.

- **Flexible**. This addresses the needs of the person responsible for assigning the experiments who could also be the user, or could be a course instructor. The Wireless GINI platform supports a diverse combination of network elements that are all integrated seamlessly into one custom internet. We saw how physical wireless devices, wireless mesh overlays, and process-emulated networks can all by seamlessly integrated into one custom internet that the user can easily deploy. This provides the flexibility and simplicity needed to perform a wide variety of interesting and innovative experiments. In addition, the source code of the *yRouter* is freely available. This adds another layer of flexibility whereby the students can directly modify the source code to implement existing protocols and systems, or create a whole new protocol/system.

7.2 Future Work

Although we have built a working prototype of the Wireless GINI system, several improvements can be made to further enhance the established design goals of providing a user-friendly, flexible, and cost-effective platform. We will look at each aspect in more detail and suggest possible improvements. Several improvements could be made to enhance the user experience of the system. This relates to providing a more automated and seamless setup of the Wireless GINI platform. The more effortless the setup and deployment is, the more time students can have in developing more complex experiments and the more adoptable the system will become. To that end, integrating the WGINI server interface into the *gBuilder* and *gLoader* provides a seamless setup and deployment of the system. The students can leverage the intuitive GUI of the existing GINI platform to automate the TSF creation and RPC invocation. Moreover, as mentioned previously, the implementation of the mDNS-SD functionality into the RPC client-server architecture will automate the discovery and connectivity of the user's client application to the WGINI server application. Another automation feature mentioned is in regards to the wireless mesh setup. A discovery and management protocol can be implemented that allows the WGINI server to automatically keep track of mesh stations that enter and leave the MBSS. Implementing these suggested features will provide a transparent and intuitive way of creating topologies

on the Wireless GINI system.

Another improvement relates to making the Wireless GINI system more secure. Security can be implemented in various aspects of the Wireless GINI system. Starting with the interface, an authentication system could be employed such that users cannot accidentally (or maliciously) delete or modify the topologies of other users by impersonating their identity. Another aspect for security lies in implementing security features on the wireless mesh network so that unintended devices will not be able to become part of the MBSS. A third avenue for security pertains to securing the deployed WLANs of the users. The fourth aspect is securing the mesh stations from unauthorized remote access to their shells by the users. The fifth aspect pertains to providing a more sandboxed environment where defected or malicious code on a *yRouter* will not harm the other *yRouters* running on the station. A way for allowing users to deploy their custom *yRouter* binaries while providing proper isolation between *yRouter* instances will provide for a more robust and flexible platform.

On the flexibility dimension, the Wireless GINI can be made even more flexible by having the 802.11 networking stack integrated into the *yRouter* code. This requires extending the *yRouter* to process the 802.11 protocol in user space. An 802.11 module can be implemented below the Ethernet module in the *yRouter* architecture. The 802.11 module will be responsible for the implementation of the 802.11 authentication, management and data processing features. In order to capture the 802.11 frame received by the wireless device before it is converted to an ethernet frame by the kernel, the wireless interface mode should be set to monitor mode. Monitor mode is one of the modes supported by hardware devices that captures all 802.11 packets on the wireless channel. A *raw* socket could then be setup to capture all incoming packets on the wireless channel and pass them to the 802.11 module for further processing. With this setup, no kernel support for the *wlan* interface is needed. Thus, the limit on the number of VAP is no longer imposed since only one monitor mode interface is needed. Any *yRouter* that wishes to deploy a WLAN can simply attach a *raw* socket on the monitor mode interface and process the incoming packets in the *yRouter*. This also allows for further experimentation with the 802.11 protocols and the implementation of new wireless features for experimentation directly on the *yRouter*. However, this setup could be computationally expensive and slow. Setting up a *wlan* interface through the kernel allows the kernel to leverage the hardware-assisted processing features provided by the wireless device through the device's OS driver. An example of a hardware-assisted feature includes automatically discarding packets that have a BSSID that does not match

any of the configured BSSIDs of the wireless device. By using monitor mode, each packet will reach all the way to the user space *yRouters* before they are discarded. If this proves too slow, only the hostapd [65] daemon's functionality could be integrated into the *yRouter*. Hostapd is a user space daemon that is used by OpenWrt for 802.11 management and authentication. With this setup, the limit on the VAP will still be imposed but the 802.11 user space processing will occur on the *yRouter*. This allows us to experiment with new 802.11 features such as mobility management and handover features, and QoS features.

Appendix A

Wireless GINI Documentation

In this chapter, we provide the documentation of the top-level Wireless GINI functions. The source code of the Wireless GINI system is open source. The source code of the *yRouter* can be found at: https://github.com/ahmed-youssef/yRouter. The source code of the WGINI server can be found at: https://github.com/ahmed-youssef/WirelessGINI.

A.1 Topology Specification File Document Type Definition

```
<!DOCTYPE VN [
  <!ELEMENT VN (Station+)>
    <!ELEMENT Station (ID, Interface*, BHInterface?, Raw_Interface?)>
      <!ELEMENT ID (#PCDATA)>
      <!ELEMENT Interface (InterfaceNo, DestStaID, IPAddress, HWAddress,
          REntry+)>
      <!ELEMENT BHInterface (InterfaceNo, DestIface, IPAddress, HWAddress,
          REntry+)>
      <!ELEMENT WlanInterface (InterfaceNo, IPAddress, SSID, REntry+)>
        <!ELEMENT InterfaceNo (#PCDATA)>
        <!ELEMENT DestStaID (#PCDATA)>
        <!ELEMENT IPAddress (#PCDATA)>
        <!ELEMENT HWAddress (#PCDATA)>
        <!ELEMENT DestIface (#PCDATA)>
            <!ELEMENT SSID (#PCDATA)>
```

```
            <!ELEMENT REntry (Net, NetMask, NextHop?)>
               <!ELEMENT Net (#PCDATA)>
               <!ELEMENT NetMask (#PCDATA)>
               <!ELEMENT NextHop (#PCDATA)>
]>
```

A.2 WGINI API

Below are the documentation of the WGINI server API discussed in Section 4.2.

- **Stations = Server.Check()**: Creates a New VN.

  - Returns the *Stations* object array.

  - *Station.ID*: The ID of the mesh station.

  - *Station.CurrWlan*: The current number of *wlan* interfaces deployed on the station.

  - *Station.MaxWlan*: The maximum number of *wlan* interfaces that this station can support

  - *Station.IsPortal*: A boolean that specifies whether or not this station is the mesh portal.

- **Status = Server.Delete(UserIP)**: Deletes the user's VN from the wireless mesh platform.

  - *UserIP*: The IP address of the user who wishes to delete his/her VN.

  - *Status*: A status code that determines whether or not the operation succeeded.

- **Status = Server.Create(UserIP, TSFstring)**: Deploys the user's VN as specified by the input TSF.

  - *UserIP*: The IP address of the user who wishes to deploy the VN.

  - *TSFstring*: The TSF file input as a regular string.

  - *Status*: A status code that determines whether or not the operation succeeded.

A.3 yRouter API

- **ifconfig add -dev tun0 -dstip dst_ip -dstport portnum -addr IP_addr hwaddr MAC**: Creates a *tun* interface on the *yRouter*.

  - *dst_ip*: The IP address of the destination mesh station on the MBSS.

  - *portnum*: The interface number of the *tun* interface on the other end of the virtual link.

  - *IP_addr*: The virtual IP address of the *tun* interface.

  - *MAC*: The virtual MAC address of the *tun* interface.

- **ifconfig add -dev wlan0 -addr IP_addr -ssid SSID**: Creates a *wlan* interface on the *yRouter*.

  - *IP_addr*: The IP address of the *wlan* interface.

  - *SSID*: The SSID of the *wlan* interface.

A.4 Bash script API

- **del_iface top_num**: deletes the *wlan* interface on OpenWrt.

  - *top_num*: The topology number that the *wlan* interface belongs to.

- **kill_yrouter top_num**: kills the *yRouter* process.

  - *top_num*: The topology number that the *yRouter* process belongs to.

- **new_iface top_num IP_addr SSID**: Creates a new *wlan* interface on OpenWrt.

  - *top_num*: The topology number that the *wlan* interface belongs to.

  - *IP_addr*: The IP address of the *wlan* interface.

  - *SSID*: The SSID of the *wlan* interface.

References

[1] J. Liebeherr and M. E. Zarki, *Mastering Networks: An Internet Lab Manual*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645 – 1660, 2013.

[3] J. Pan, "Teaching computer networks in a real network: the technical perspectives," in *Proceedings of the 41st ACM technical symposium on Computer science education*, pp. 133–137, ACM, 2010.

[4] J. Ahrenholz, C. Danilov, T. Henderson, and J. Kim, "Core: A real-time network emulator," in *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pp. 1–7, Nov 2008.

[5] "Junosphere Classroom." `http://www.juniper.net/us/en/products-services/software/junos-platform/junosphere/`. [Online; accessed 05-June-2015].

[6] "Cisco Learning Labs." `https://learningnetworkstore.cisco.com/cisco-learning-labs`. [Online; accessed 05-June-2015].

[7] M. Pizzonia and M. Rimondini, "Netkit: network emulation for education," *Software: Practice and Experience*, 2014.

[8] T. U. of Madrid (UPM), "Virtual Networks over linuX (VNX)." `http://web.dit.upm.es/vnxwiki/`. [Online; accessed 13-June-2015].

[9] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pp. 253–264, ACM, 2012.

[10] "The MLN Project." `http://clownix.net/`. [Online; accessed 13-June-2015].

[11] M. Maheswaran, A. Malozemoff, D. Ng, S. Liao, S. Gu, B. Maniymaran, J. Raymond, R. Shaikh, and Y. Gao, "Gini: a user-level toolkit for creating micro internets for teaching & learning computer networking," in *ACM SIGCSE Bulletin*, vol. 41, pp. 39–43, ACM, 2009.

[12] "The User-mode Linux Kernel Home Page." `http://user-mode-linux.sourceforge.net/`. [Online; accessed 05-June-2015].

[13] J. Dike, *User mode linux*, vol. 2. Prentice Hall Englewood Cliffs, 2006.

[14] "Arduino Yun." `https://www.arduino.cc/en/Main/ArduinoBoardYun?from=Products.ArduinoYUN`. [Online; accessed 05-June-2015].

[15] "IEEE standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE P802.11-REVmb/D12, November 2011 (Revision of IEEE Std 802.11-2007, as amended by IEEEs 802.11k-2008, 802.11r-2008, 802.11y-2008, 802.11w-2009, 802.11n-2009, 802.11p-2010, 802.11z-2010, 802.11v-2011, 802.11u-2011, and 802.11s-2011)*, pp. 1–2910, March 2012.

[16] C. Liang and F. R. Yu, "Wireless network virtualization: A survey, some research issues and challenges," *Communications Surveys & Tutorials, IEEE*, vol. 17, no. 1, pp. 358–380, 2015.

[17] D. D. Coleman and D. A. Westcott, *Cwna: certified wireless network administrator official study guide: exam Pw0-105*. John Wiley & Sons, 2012.

[18] I. F. Akyildiz, X. Wang, and W. Wang, "Wireless mesh networks: a survey," *Computer networks*, vol. 47, no. 4, pp. 445–487, 2005.

[19] "Open Wireless Router - OpenWrt." `https://openwrt.org/`. [Online; accessed 04-July-2015].

[20] A. Wang, M. Iyer, R. Dutta, G. N. Rouskas, and I. Baldine, "Network virtualization: technologies, perspectives, and frontiers," *Lightwave Technology, Journal of*, vol. 31, no. 4, pp. 523–537, 2013.

[21] S. Jeong and H. Otsuki, "Framework of network virtualization," *FG-FN OD-17*, 2009.

[22] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 275–287, ACM, 2007.

[23] "Linux containers network namespace." `http://lxc.sourceforge.net/network.php`. [Online; accessed 08-July-2015].

[24] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford, "Trellis: A platform for building flexible, fast virtual networks on commodity hardware," in *Proceedings of the 2008 ACM CoNEXT Conference*, p. 72, ACM, 2008.

[25] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[26] E. Grasa, G. Junyent, S. Figuerola, A. Lopez, and M. Savoie, "Uclpv2: a network virtualization framework built on web services [web services in telecommunications, part ii]," *Communications Magazine, IEEE*, vol. 46, no. 3, pp. 126–134, 2008.

[27] Y. He, J. Fang, J. Zhang, H. Shen, K. Tan, and Y. Zhang, "Mpap: virtualization architecture for heterogenous wireless aps," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 475–476, 2011.

[28] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks," tech. rep., 2014.

[29] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[30] K.-K. Yap, M. Kobayashi, R. Sherwood, T.-Y. Huang, M. Chan, N. Handigol, and N. McKeown, "Openroads: Empowering research in mobile networks," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 125–126, 2010.

[31] "XML-RPC Python Library." `https://docs.python.org/2/library/xmlrpclib.html`. [Online; accessed 15-July-2015].

[32] Cheshire and Krochmal, "Multicast DNS," RFC 6762, RFC Editor, February 2013.

[33] "SQLite Home Page." `http://sqlite.org/`. [Online; accessed 16-July-2015].

[34] T. W. Cotton, Eggert and Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry," RFC 6335, RFC Editor, August 2011.

[35] "GNU Screen." `http://www.gnu.org/software/screen/`. [Online; accessed 17-July-2015].

[36] Perkins, "IP Mobility Support for IPv4, Revised," RFC 5944, RFC Editor, November 2010.

[37] C. E. Perkins, "Mobile networking through mobile IP," *Internet Computing, IEEE*, vol. 2, no. 1, pp. 58–69, 1998.

[38] C. E. Perkins, S. R. Alpert, and B. Woolf, *Mobile IP; Design Principles and Practices*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[39] "Gartner, Inc. - Newsroom." `http://www.gartner.com/newsroom/id/2636073`. [Online; accessed 19-July-2015].

[40] "The Network Simulator - ns-2." `http://www.isi.edu/nsnam/ns/`. [Online; accessed 28-June-2015].

[41] "OPNET." `http://www.openet.com/`. [Online; accessed 28-June-2015].

[42] "OMNeT++ - Discrete Event Simulator." `https://omnetpp.org/`. [Online; accessed 28-June-2015].

[43] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19, ACM, 2010.

[44] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[45] F. Galán, D. Fernández, W. Fuertes, M. Gómez, and J. E. L. de Vergara, "Scenario-based virtual network infrastructure management in research and educational testbeds with vnuml," *Annals of telecommunications-annales des télécommunications*, vol. 64, no. 5-6, pp. 305–323, 2009.

[46] "Cloonix: dynamical topology virtual networks." `http://mln.sourceforge.net/`. [Online; accessed 26-June-2015].

[47] "The Xen Project Hypervisor." `http://www.xenproject.org/developers/teams/hypervisor.html`. [Online; accessed 26-June-2015].

[48] "The VMware Server." `https://my.vmware.com/web/vmware/info/slug/infrastructure_operations_management/vmware_server/2_0`. [Online; accessed 26-June-2015].

[49] "GNS3 - Graphical Network Simulator." `http://www.gns3.com/`. [Online; accessed 26-June-2015].

[50] "Cisco router simulator (Dynamips)." `https://github.com/GNS3/dynamips/tree/4697d8488204787c79a8a8c550e96ca65fe79708`. [Online; accessed 26-June-2015].

[51] "EstiNet network simulator and emulator." `http://www.estinet.com/`. [Online; accessed 26-June-2015].

[52] "Common Open Research Emulator (CORE)." `http://www.nrl.navy.mil/itd/ncs/products/core`. [Online; accessed 27-June-2015].

[53] M. Zec, "Implementing a clonable network stack in the freebsd kernel.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 137–150, 2003.

[54] J.-V. Loddo and L. Saiu, "Marionnet: a virtual network laboratory and simulation tool," in *First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, 2008.

[55] "Marionnet: a virtual network laboratory." `http://www.marionnet.org/EN/`. [Online; accessed 28-June-2015].

[56] "Emulab - Total Network Testbed." `http://www.emulab.net/`. [Online; accessed 13-June-2015].

[57] "OpenVZ." `http://wiki.openvz.org/Main_Page`. [Online; accessed 28-June-2015].

[58] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir, "Experiences building planetlab," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 351–366, USENIX Association, 2006.

[59] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[60] "Emulab - Total Network Testbed." `http://www.emulab.net/`. [Online; accessed 13-June-2015].

[61] M. Berman, C. Elliott, and L. Landweber, "Geni: Large-scale distributed infrastructure for networking and distributed systems research," in *Communications and Electronics (ICCE), 2014 IEEE Fifth International Conference on*, pp. 156–161, July 2014.

[62] J.-M. Kang, H. Bannazadeh, and A. Leon-Garcia, "Savi testbed: Control and management of converged virtual ict resources," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pp. 664–667, May 2013.

[63] J. DeHart, F. Kuhns, J. Parwatikar, J. Turner, C. Wiseman, and K. Wong, "The open network laboratory," in *ACM SIGCSE Bulletin*, vol. 38, pp. 107–111, ACM, 2006.

[64] 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, pp. 1664–1669 Vol. 3, March 2005.

[65] "Hostapd - Home Page." `https://w1.fi/hostapd/`. [Online; accessed 19-July-2015].