INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Programming the Enterprise WLAN: An SDN Approach

Lalith Suresh Puthalath

Dissertation submitted in partial fulfillment of the requirements for the European Master in Distributed Computing Programme

Supervised by

Dr. Ruben Merz	Telekom Innovation Laboratories/TU Berlin
Dr. Teresa Vazão	Instituto Superior Técnico

Jury

Head:	Prof. Luís Rodrigues
Examiner:	Prof. Leandro Navarro
Advisor:	Dr. Teresa Vazão
Advisor:	Dr. Ruben Merz

June 22, 2012

Abstract

Enterprise WLANs need to support a wide spectrum of services and functionalities. This includes authentication, authorisation and accounting, policy management, dynamic channel reconfigurations, mobility and interference management, and load balancing. Vendor solutions that address these problems are proprietary, and closed source, leading to vendor lock-in. They restrict flexibility for the network operator. This thesis explores the introduction of programmability into the enterprise WLAN, demonstrated through a prototype system named Odin, with the goal of empowering the network operator, and to make enterprise WLANs extensible. Through Odin, we provide a set of abstractions and interfaces using which different enterprise WLAN services can be implemented as network applications. Odin is based on a software defined networking (SDN) approach, wherein the WLAN is orchestrated through centralised control, as dictated by a set of network applications. Odin's SDN framework also takes into account the unique challenges of WLANs. For instance, clients in the IEEE 802.11 protocol make access point association choices entirely on the basis of local decisions. To this end, Odin builds on a light virtual access point abstraction that takes control of the client's association decisions, and greatly simplifies client management. Additionally, Odin's architecture does not expect any hardware or software modifications on the 802.11 client, and its design supports WPA2 Enterprise. Experimental evaluations conducted within an enterprise WLAN setting demonstrate Odin's feasibility.

Keywords

Keywords

Software Defined Networking

Enterprise WLAN

Network Programming

Odin

Acknowledgements

I would like to express my sincere gratitude to Dr. Ruben Merz, who guided me through this research, was a great source of encouragement, and always made me go that extra mile to solve the various problems that lead to this thesis. I also thank my supervisor, Dr. Teresa Vazão, for supporting me throughout this work. During the course of this research, I also benefited greatly from interactions and discussions with other members from the INET team, namely Julius Schulz-Zander, Prof. Anja Feldmann, and Dr. Cigdem Sengul, to whom I'm deeply indebted. A special mention also goes to Robin Kuck from the BOWL team for helping me setup the testbed used for this research. The IRC conversations with my EMDC classmates also helped me over the course of this work, not only in terms of resolving quick technical difficulties, but also with regards to lightening up.

Last, but by no means the least, I would like to thank my family for their immense support and encouragement, without whom none of this would have been possible.

Contents

List of Figures

List of Tables

1 Introduction

Today's IEEE 802.11 [1] enterprise WLAN deployments range from a few dozens to thousands of access points (APs), which need to serve a large number of users. These users connect to the enterprise WLAN through a multitude of devices, including smart phones, laptops, and tablets. Regardless of their size, these networks need to provide a varied set of services in a scalable manner. These services include support for authentication, access, and accounting (AAA), policy based network management, interference management, mobility management, dynamic channel reconfigurations, load balancing, intrusion detection and prevention, and providing quality of service guarantees. The management of these enterprise WLANs is usually centralised (through a controller). Many vendor solutions exist that cover a set of these features [2, 3, 4, 5]. However, these solutions are usually proprietary, and are closely tied to the hardware provided by the same vendor as well. There is thus a need for an open and flexible software architecture for enterprise WLANs.

Recent years have seen a growing adoption of Software Defined Networking (SDN). In SDN, the network control plane is decoupled from the physical network topology and, instead, uses software to control how traffic is forwarded in the network. For instance, a switch's forwarding tables can be controlled remotely through a software controller (or network operating system). The OpenFlow [6] protocol is considered an enabler of SDN because it provides a standardised protocol that can be used by a controller to manipulate forwarding tables of a network of switches (an analogy would be the x86 instruction set for computer architectures). One can now write *network applications* that can programmatically control the forwarding behaviour of a network by talking to a network controller. Any OpenFlow enabled switch from any vendor will have a common interface for forwarding plane manipulation via a controller (of which there are many open source implementations today). This enables flexible and simplified network management.

In this thesis, we take a step forward in designing extensible enterprise WLANs using an SDN approach, demonstrated through a prototype system named *Odin*. Odin is an SDN framework for enterprise WLANs. Using Odin, different enterprise WLAN services can be programmed and deployed as *network applications*. By providing the means to program the enterprise WLAN, we empower the network operator with a flexible way to address the diverse needs of each individual network. Programmability also avoids the need for developing large monolithic management systems, and protects against a vendor lock-in of the entire network.

WLANs, however, exhibit some specific challenges which are not present in wired networks. As per the 802.11 [1] standard, clients are free to make AP association choices purely on the basis of local decisions. The standard does not define a signalling mechanism to control the client's association through an AP. This complicates client management. To overcome this limitation, Odin builds on a light virtual access point (LVAP) abstraction. LVAPs abstract the association state away from the physical AP itself. An LVAP is a per-client AP. Each client connecting to an Odin-based network sees only its own unique AP regardless of the actual physical AP they are within range of. As we will show in subsequent chapters of this thesis, the LVAP design also facilitates infrastructure controlled handoffs without invoking a state machine change at the client.

1.1 Contributions

This thesis' key contribution is a novel system that brings programmability into enterprise WLANs. It is the first system of its kind that enables the development of different enterprise WLAN network services as applications. This is made possible through the simple, yet effective abstraction in the form of LVAPs. Furthermore, the performance benefits that are achieved through this system are made possible without any modifications to the 802.11 client. The architecture and design of the system is compatible with existing enterprise WLAN security protocols like WPA2 Enterprise.

1.2 Results

The outcome of this thesis is a prototype system named *Odin* which provides a framework for implementing typical WLAN services as "network applications". It builds upon a novel LVAP abstraction that simplifies client management. The practicality of the system is demonstrated through an experimental evaluation conducted within an operational enterprise network setting.

1.3 Research Context

The research described in this thesis was done within the Intelligent Networks and Management of Distributed Systems (INET) research group at Telekom Innovation Laboratories, Berlin. A paper that describes part of this work has been accepted for publication in SIGCOMM Hot Topics in Software Defined Networking (HotSDN) 2012. The paper is titled "Towards Programmable Enterprise WLANs with Odin". A demo titled "Programming Enterprise WLANs using Odin" is due to be presented as part of the SIGCOMM 2012 demo session as well.

1.4 Organisation of this thesis

The remainder of this thesis is organised as follows. Chapter 2 describes the background for this thesis. Chapter 3 provides an overview of the related work. Chapter 4 explains the architecture of Odin. Chapter 5 describes the details of Odin's implementation. Chapter 6 presents results obtained from an evaluation of Odin. Lastly, Chapter 7 concludes the thesis, and describes future work.

2 Background

This chapter describes the concepts that are relevant to this thesis. This includes Software Defined Networking (SDN), and the IEEE 802.11 protocol for Wireless Local Area Networks (WLANs).

2.1 Software Defined Networking

Architecturally, a network switch is composed of a data plane and a control plane. The data plane is concerned with forwarding packets through the switch, and for this reason, is also known as the forwarding plane. The control plane comprises all the logic that the switch needs in order to correctly set up a forwarding plane, that is, the signalling associated with a switch.

Traditional switches have proprietary firmware and control logic implementations, which remain under the control of the vendor. This not only inhibits vendor inter-operability, but also hampers flexibility. Even though these switches can be configured or managed through SNMP or a CLI, there is no means of introducing a new control plane function or protocol into the switch. This makes experimenting with new networking protocols cumbersome. The Software Defined Networking (SDN) approach aims to alleviate these issues by allowing a switch's control plane to be remotely accessible and modifiable using an open protocol (such as OpenFlow [6]). Third-party software can then leverage this open protocol to orchestrate an entire network.

The SDN architecture comprises a logically centralised network operating system, which can communicate with a network of switches (Figure 2.1). A network "application" that runs on top of this network OS can then manipulate the forwarding tables of the switches to implement a particular control function. A clear advantage here is that it simplifies the implementation of control functions.

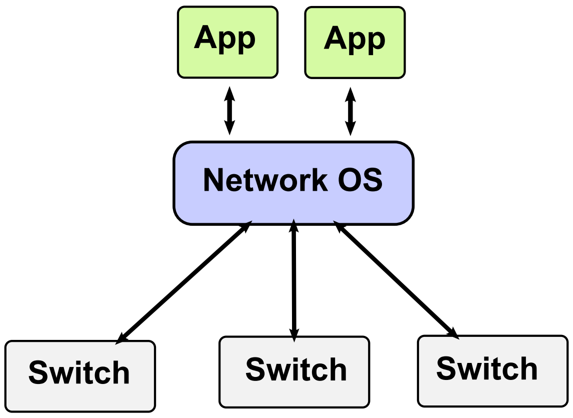

Figure 2.1: Architecture of a Software Defined Network (SDN). The network operating system provides interfaces for applications to control the forwarding tables of a set of switches. Thus, the control plane is split from the data plane. OpenFlow is an open protocol for a network operating system to control the forwarding tables of a switch.

2.2 IEEE 802.11 Media Access Control Layer

The IEEE 802.11 Media Access Control Layer (MAC) [7] defines the protocol for stations to establish connections with each other and transmit data frames.

Medium access in 802.11 is performed by a distributed coordination function (DCF), which uses carrier sense multiple access with collision avoidance (CSMA/CA) to enable random medium access among all contending stations (STAs). Hence, it reduces the amount of collisions. Logically, the MAC is divided into two parts, an upper MAC, and a lower MAC. The upper MAC handles management frames, which include probe, authentication, and association requests and their corresponding responses. The lower MAC handles control frames, which includes acknowledgement (ACK) frames, along with request-to-send (RTS) and clear-to-send (CTS) frames. The frames handled by the lower MAC have real-time constraints. For instance, ACK frame timeouts are within the order of micro-seconds. For this reason, control frames are handled and generated within hardware. Management frames, however, have softer time constraints, and can be handled in software locally (as is the case in Linux systems that use `hostapd` [8]), or remotely (as is the case when using a centralised WLAN controller [9]).

An 802.11 based wireless interface can operate under the following operating modes: STA (client), access point (AP), mesh, ad-hoc and item Monitor mode. The most common mode of operation is the infrastructure mode (which includes enterprise WLAN environments). In this mode of operation, clients connect to the AP using a series of message exchanges in a process called "association". The decision on which AP to associate with is left entirely to the client. Clients learn about APs either passively through beacon frames that are periodically broadcasted by the access points, or actively by performing a probe scan. In a probe scan, clients first send out probe request frames over all channels. APs that receive these frames and are willing to accept a connection from a client respond with a probe response frame. All APs from which the client receives probe responses are candidates for the client to associate with. Next, the client sends an authentication frame, and waits for an authentication response from the AP. This is followed by the client sending an association request, and receiving an association response from the AP. If the network is operating in open authentication mode, the client is considered to be associated at this point, and can now transmit data frames to be forwarded by the AP. If the AP is configured to use WPA, WPA2, or WPA2 Enterprise, the corresponding 802.1X [8] handshake is performed after the association phase before clients can forward data frames through the AP.

3 Related Work

This chapter discusses previous research related to this thesis. It covers work on software defined networking (SDN), enterprise WLAN management, wireless device virtualisation, and infrastructure-controlled handoffs.

3.1 Software Defined Networks (SDN)

SDN research over recent years have spanned many areas. This includes scalability of the control platform and switch implementations, programming models for SDN based networks, network applications, SDN for wireless networks, and testing of SDN applications. We will now cover related work on the areas that are relevant for this thesis.

3.1.1 SDN Scalability

The centrally controlled nature of SDN-based networks raises questions about scalability of the architecture. This includes a scalability bottleneck in the form of the control platform, and a bottleneck with the switch (in terms of forwarding table memory, and the overhead of invoking the control plane). We now describe research efforts that have tried to address these issues.

Hyperflow [10], built as an application on top of the NOX [11] OpenFlow controller, works by having multiple controllers manage different partitions of the network, and having the HyperFlow instances passively synchronise the network views across different controllers. Maestro [12] is an OpenFlow controller that exploits parallelism to alleviate the performance bottleneck at the central controller. Onix [13] is a distributed controller that focuses on reliability. It provides a more general programming API than NOX for network applications. It leaves distributed coordination to the application logic, but provides primitives like leader election and content distribution to the applications in order to handle distributed coordination easily. A key difference between Onix and NOX is that the former does not handle per-packet events, and only registers for less frequent network events. This is a trade-off made for scalability reasons, since a server that runs the controller cannot forward packets faster than a switch's data plane. Also related is ETTM [14], which is a logically centralised network manager that exposes a set of APIs to implement control applications. The manager itself is composed of software running on a number of physical end points, and uses Paxos [15] to ensure consensus among management applications. DIFANE [16] proposes to avoid the bottleneck at the central controller by having the controller distribute rules across a subset of the switches known as 'authority switches', which can then handle forwarding table misses from other switches (instead of the controller itself).

Lastly, DevoFlow [17] highlights the bottleneck at the OpenFlow switch within the context of high-performance networks (like data centers). This occurs because in OpenFlow based networks, if a packet is received for which there is no matching forwarding rule, the packet is forwarded to the controller. Assuming the network performs only reactively, the controller will end up handling the first packet of every flow before inserting a rule. For a switch to send a packet to the controller, it has to move the packet from hardware to the context of its management CPU, before forwarding it. This can be a bottleneck within high performance networks, where there are several hundred thousand flows being generated per-second. The authors propose to tackle the problem by addressing short-lived (mice) and long-lived (elephant) flows separately. Switches only inform the controller about those flows which have lasted beyond a threshold.

We note that in Odin, the scalability of the system is dependent on the scalability of the control platform, but this aspect is orthogonal to the ability to program enterprise WLANs. This thesis attempts to solve the latter.

3.1.2 Network Programming

Apart from looking at scalability issues, Onix [13] also sheds light on primitives the controller should expose to the control applications. Onix exposes a key-value store called the "network information base", wherein control applications can obtain information about the network. In Odin, we use a similar approach to supply applications with statistics.

There have been research efforts to develop programming languages for writing network applications. Frenetic [18] is a language and compiler runtime for network programming that aims to provide high-level abstractions to the programmer. It provides means for network programmers to specify the network's behaviour at a high-level, using a functional and declarative language. An important focus of Frenetic is to keep these constructs compositional, so as to facilitate modularity and easier reasoning about code. Frenetic is based on the NetCore compiler and run- time [19]. [20] addresses the need for consistency guarantees to be provided by the run-time when propagating forwarding table updates to multiple switches in an SDN. Nettle [21] also demonstrates network control using a functional reactive programming approach. Another domain specific language to manage network configurations is the Flow-based Management Language (FML) [22]. With Odin, we explore the abstractions required for programming enterprise WLANs, and language level support as proposed by the above would be a good complement for our system.

3.1.3 Applications of SDN

There has been many research efforts in writing SDN applications. Jose et. al. [23] propose using commodity OpenFlow enabled switches for traffic measurement. The authors propose a framework where a collection of rules are installed on OpenFlow switches, and having a controller track the corresponding flow match counters. The controller can then draw inferences from the counters and dynamically tune the rules as required in order to identify different traffic aggregates.

Resonance [24] uses programmable switches to enforce access control in the network. The authors assert that today's enterprise networks rely on different combinations of middle boxes, intrusion detection systems, and network configurations in order to enforce access control policies, whilst placing a burden on end-hosts in the system to remain patched and secure. The proposed system uses an SDN approach comprising programmable switches and a controller, which together implement a network monitoring framework, a policy specification framework, and the ability to trigger specific actions at the switch level.

OpenSAFE [25] is a framework that enables network monitoring using OpenFlow. It addresses the problem of routing traffic for network analysis in a reliable manner without affecting normal traffic.

Hedera [26] is an adaptive flow scheduling system for data center networks. The premise for Hedera is that existing IP multipathing techniques used in data centers usually rely on per-flow static hashing, which can lead to under-utilisation of some network paths over time due to hash collisions. The system works by detecting large flows at the edge switches of a data center, and using placement algorithms to find good paths for the flows in the network. Experiments performed using simulations indicate significant improvements over static load balancing techniques.

In [27], the authors address the problem of server load balancing using OpenFlow switches. The number of flow entries that can be saved on an OpenFlow switch is much less than the number of unique flows that a switch might need to handle in data center workloads. Thus, micro flow management using per-flow rules is not practical for performing distributing flows between different servers using a switch. The authors thus take advantage of OpenFlow's wildcard based rules capability, and propose algorithms to compute concise wildcard rules that achieve a specific distribution of traffic.

With Odin, we extend the scope of network applications like the above to enterprise WLAN specific functionalities as well.

3.1.4 SDN and wireless

In [28], the authors describe the use of OpenFlow for wireless mesh networks. Many protocols like AODV, OLSR, and BATMAN are typically used for mesh networks, which route on a per-packet basis. However, these protocols cannot handle flow based routing, where flows from different sources are routed differently. The authors show that introducing OpenFlow to wireless mesh networks can help overcome this issue.

The Stanford OpenRoads deployment [29] explores the use of OpenFlow for conducting experiments on wireless networks. By creating network slices differentiated by the wireless network's SSID, different experiments can be assigned different slices.

The OpenRadio project [30] explores the use of an SDN architecture to program the datapath of a frame down till the PHY (for instance, the kind of modulation to be used for transmitting a frame). With Odin, programmability of the PHY as proposed by OpenRadio would be a natural complement.

3.2 Enterprise WLAN management

Enterprise WLAN management systems are characterised by the existence of a large number of users, who can connect to a large number of access points, all of whom are under a central administrative authority. The last property is important, as it makes such networks a natural fit for centralised management systems.

Many systems have been proposed that exploit centralised management to deliver improved services to clients in the network. CENTAUR [31] improves the data path in enterprise WLANs by using centralisation to mitigate hidden terminals and to exploit exposed terminals. Dyson [32] addresses the problem of extensibility in wireless LANs, by defining a set of APIs for clients and access points to be managed by a central controller. The controller can query these nodes for information such as radio channel conditions, form a global view of the network, and then control the network's behaviour to enforce a set of policies. However, CENTAUR, and Dyson propose modifications to the client in order to achieve the said results. But requiring special software or hardware on a client raises questions of practicality, and goes against the design requirements for Odin.

There are also systems that do not modify the client in order to deliver services. In DenseAP [33], channel assignment and association related decisions are made by disconnection message, and forcing the client to scan and connect to the network again. In DAIR [34], the authors take advantage of the fact that an enterprise network usually has good wired connectivity even when the wireless network is loaded. The authors propose equipping desktop machines with USB-WiFi cards for continuous wireless monitoring, and then making use of the collected information to effectively manage the network. DenseAP and DAIR do not modify the client, but this has a trade-off in performance limitations. For instance, both systems cannot work around the inherent delay involved with an 802.11-handover, which can be on the order of several seconds.

Lastly, Rozner et al. describe traffic aware channel assignment in enterprise WLAN deployments [35] with the intention of improving performance in the wireless network. Bahl et.s al. [34] propose the use of desktop computers as monitoring stations in order to collect statistics about the network. These statistics can then be analysed in order to make effective network management decisions.

3.3 Wireless Device Virtualisation

There have been several research efforts around the virtualisation of the 802.11 protocol. In MultiNet [36], clients use a special device driver that makes use of 802.11's power saving mode in order to continuously switch between multiple networks. Spider [37] designs a driver that allows clients to be connected to multiple APs at the same time. The authors in Spider also conclude that using multiple APs can demonstrate better throughput when the APs are on the same channel, even though their system can manage multiple channels.

Wireless virtualisation is also an attractive option for testbeds and experimentation facilities. This is demonstrated by G. Smith, et. al. [38], to perform time division based multiplexing of the air interface in order to support multiple concurrent experiments on the same wireless testbeds.However, all the above methods expect client side modifications.

Researchers have also proposed the use of the 802.11 point coordination function (PCF) and power saving mode (PSM) to implement wireless device virtualisation [39]. PSM allows an STA to instruct the AP to buffer packets destined to the STA. This allows the STA to go into power saving mode. With PSM, an STA can transmit on multiple channels (and be on multiple networks) with the same physical interface. Before switching the interface to a new channel, the STA can instruct the AP of the current network to buffer packets destined to the STA using PSM. However, PSM is a client side mechanism. On the other hand, with PCF, the AP takes the role of a point coordinator (PC), giving it higher access priority. It can then assign access slots to different STAs. This can allow it to receive frames through multiple channels by carefully scheduling transmission times for different STAs on different channels. However, PCF is an optional feature as per the 802.11 standard, and there are no vendors that implement this feature in 802.11 hardware.

3.4 Infrastructure Controlled Handoffs

Reducing handoff latency in wireless networks is a well studied problem. On the basis of extensive measurements conducted in a live deployment using different combinations of vendor hardware, Mishra et. al. [40] explain that the probe scan step in the 802.11 association sequence (see Section 2.2) consumes most of the time in a handoff (upto 90% of the overall duration).

Additionally, the authors assert that not only does the overall handoff duration vary between different vendors, but that it is large enough to lead to a bad quality of service to applications. These delays can be as long as several seconds, and on average, tends to be around 400 ms [40, 41]. A similar conclusion was drawn by [42], wherein an investigation was conducted on how 802.11 handoffs impacted voice traffic. Another study which was based on an analysis of data collected from the 67^{th} Internet Engineering Task Force (IETF) meeting [43] also concluded that current STA side mechanisms that trigger handoffs negatively impact clients.

It is to be noted that unlike 802.11, cellular networks manage to achieve smooth intra-domain handovers by sharing information between towers about a particular client [44, 45], and by having a signalling mechanism to initiate a handover by a client. Since 802.11 networks lack such a mechanism, there have been many proposals to realise fast handovers in such networks. These techniques mostly involve client side modifications such that the client doesn't scan through all channels explicitly in order to find a new association, and receives some directions from the infrastructure itself [46]. SyncScan [47] proposes synchronising clients with the access points beacon transmission cycle so that clients can quickly discover new associations, thus speeding up the handoff, and managing to reduce handover times to as low as 5 ms. An approach that doesn't require any modifications to the client is used by SMesh [48]. In SMesh, a mesh network provides the illusion of a single omnipresent access point to the clients. The mesh nodes work together to proactively monitor all clients in the system, and use a combination of DHCP and gratuitous ARP to make clients perform handovers. The eTIMIP micro-mobility protocol also uses a gratuitous ARP-based handover initiated by the infrastructure [49]. All the mesh nodes work together by means of a distributed protocol. However, all mobile clients in SMesh need to be run in ad-hoc mode, as opposed to running the entire network in infrastructure mode (as is the case in an enterprise WLAN).

The need for infrastructure-controlled handoffs is not only motivated by the need for uninterrupted connectivity for the client, but also has potential for being used to load balance the network. Much of the existing work in this direction forces these handovers through some specialised software on the client which the infrastructure can use to control client associations [46, 50, 51]. The authors in [28] make use of Media Independent Handover (MIH) [52] to realise administrator-initiated client handovers in an OpenFlow-based network. In [53], the authors use cell breathing techniques to regulate the transmission power of the APs in order to force a client to re-associate to another AP. However, cell breathing would affect the signal-to-noise ratio of all the STAs that remain associated with an AP that has just lowered its transmission power, and does not solve the problem of handover delay for the client. The 802.11k standard [54], which is now in development, enables better management of a wireless LAN by introducing a signalling technique into the protocol for APs to assist clients in selecting a new AP to re-associate with, as opposed to having clients blindly pick a new AP on the basis of signal strength, which can potentially lead to overloaded APs.

There are also standardisation efforts that are relevant in the direction of re-association based load balancing. The 802.11k amendment [54] deals specifically with load balancing in WLANs. It requires changes in the client as well. The amendment defines a mechanism by which APs can provide the client with information such that the latter can make better association decisions. Lastly, the 802.11r amendment [55] specifies "fast basic service set transition", wherein a client that is to handoff between two APs can perform a re-association faster. This is done by pre-caching intermediary results of the key negotiation process between the original AP where the client first authenticated, and the authentication server at different parts of the wireless network. This requires client side protocol amendments as well, which Odin explicitly avoids.

4 Designing Odin

In this chapter, we describe the design of Odin. We start by describing the design goals for the system, the high-level architectural overview of the solution, and the concept behind light virtual access points (LVAPs), which is an important component of Odin.

4.1 Design Objectives

To address the specific challenges of enterprise WLANs (discussed in Section 1), we lay forth the following design requirements for our system:

Extensibility: The system should be extensible in order to account for the diverse needs of different enterprise WLAN deployments. We opt to achieve this through network programmability (defined in Section 3.1.2). Hence, the system should provide abstractions for developers to implement network services as applications.

Simplified Programming Model: The programmer of a WLAN network service should not have to deal with WiFi state machine transitions of the client. As described in Section 2.2, the 802.11 client makes association decisions locally, without any signalling from the infrastructure. This means that the client's link to the network can change unpredictably. Requiring the programmer to reactively handle these state machine changes would be cumbersome. Instead, the programmer should always see the client's link to the network as a fixed link. The framework should provide the programmer with primitives to express complex network services as applications.

Client Transparency: The system should not require any hardware or software modifications of the clients, which should operate as legacy 802.11 infrastructure mode clients.

Security: The system's architecture should be compatible with regular 802.11 network security protocols like WPA2 Enterprise.

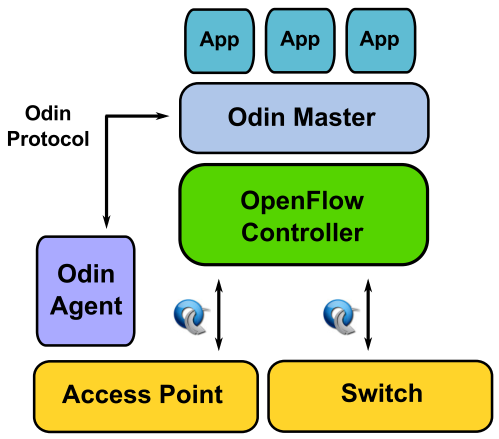

Figure 4.1: Architecture of Odin. The Odin Master (an OpenFlow application), speaks OpenFlow to the switches and the APs, and uses a custom protocol to talk to each Odin Agent running on APs. Odin applications use Odin's primitives to implement enterprise specific services.

4.2 Architecture of Odin

Odin's architecture comprises a logically centralised master, multiple agents, and a set of Odin applications (Figure 4.1). The master is implemented as a network application on top of an OpenFlow controller. This gives the master a full view over the network topology, and allows it to be aware of every flow in the network. However, the OpenFlow protocol does not cover the 802.11 MAC functions (e.g., handling association logic, and configuring wireless interfaces). Odin agents fill this gap. They run on the APs, which are also OpenFlow-enabled switches. Lastly, Odin applications programmatically implement network services through the interfaces provided by the framework.

The interaction between these components is as follows. The agents and the master together implement a WiFi split-MAC (along the lines of CAPWAP [56] and LWAPP [9]), where the MAC layer logic is divided between a central controller and the APs. The master uses OpenFlow to communicate with the switches (including the APs), and an Odin specific protocol to speak with the agents. The master can thus collect standard OpenFlow specific statistics (such as number of packets matched per-flow) from the APs and other switches in the network, and also radio specific information (such as per-frame receiver signal strength and bit-rate) exposed by the agents themselves. The master exposes these statistics along with some primitive operations as interfaces. Odin applications run atop the master and make use of these interfaces in order to implement different network services.

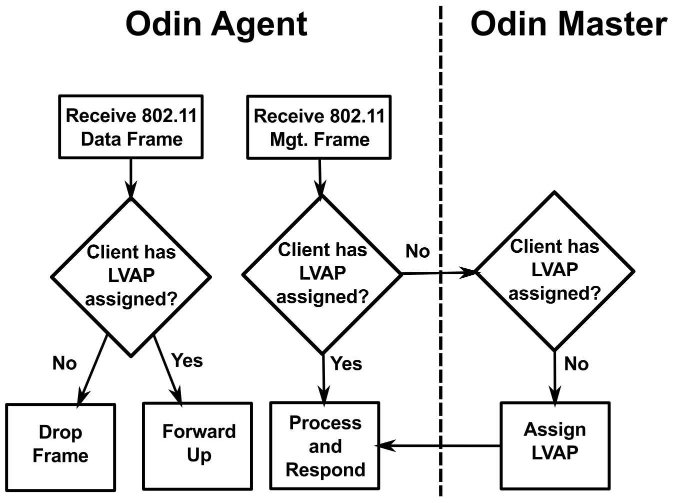
Odin Agent
Odin Master
Receive 802.11 Data Frame
Receive 802.11 Mgt. Frame
Client has LVAP assigned?
Client has LVAP assigned?
No
Client has LVAP assigned?
No
Yes
Yes
No
Drop Frame
Forward Up
Process and Respond
Assign LVAP

Figure 4.2: Processing path for 802.11 frames in Odin. When an 802.11 management frame is received by an agent, it checks whether an LVAP is hosted for the client that generated the frame. If yes, it processes and responds to the frame as per the 802.11 protocol. If there is no LVAP for the client, the agent informs the controller which assigns the client an LVAP. The agent then responds to the client accordingly.

4.3 Light Virtual Access Points (LVAPs)

In an 802.11 setting, when clients in managed mode perform a probe scan in order to find APs, they generate probe request messages. APs responding with probe response messages become potential candidates for the client to associate with. The client then initiates a series of handshakes with the AP that culminates in a successful connection between the two entities. The client can now transmit data frames that will be forwarded by the AP. At this juncture, the infrastructure has no signalling mechanism to instruct the client to handoff to another access point without explicitly disconnecting the client (which is done by sending the client a disassociation frame, and forcing it to repeat all the association handshakes again). This is inconvenient for the client. We tackle this inconvenience through the LVAP abstraction, which enables Odin to take control of a client's association decisions, and leads to a logical isolation of clients with respect to the 802.11 MAC. We now describe the LVAP abstraction and how Odin uses it to achieve the said logical isolation of clients.

In Odin, every client receives a unique BSSID to connect to, i.e., every client is given the illusion of owning its own AP. This client specific AP is the LVAP. A *physical* AP thus hosts an LVAP for each client connected through it. The per-client LVAP is assigned when a probe scan from the client is detected (see Figure 4.2). From the client's perspective, the LVAP is a regular 802.11 AP, with which it first associates with and then forwards data frames through. Removing a client's LVAP from a physical AP and spawning it on another (an LVAP-handoff) achieves the effect of handing off a client without the client performing a re-association, generating additional layer 2 or 3 messages, and most importantly, without requiring any special software or hardware at the client. This is because once a client is associated with an AP, the only protocol level requirement is that the client gets ACK frames for the data frames that it generates from the AP that it is associated with, and that it receives beacons periodically from the AP. At the client's MAC layer, it does not matter if the actual radio that generates these ACK frames changes. By abstracting away the association state of a client's connection away from individual physical APs, Odin's LVAPs thus achieve a form of wireless network virtualisation, where each client sees a logical access point unique to it regardless of the actual physical AP it is communicating with. Intuitively, an LVAP-handoff is equivalent to physically moving an AP whilst retaining all its state.

Thus, using LVAPs, Odin always provides clients a consistent link to the network, and the programmer of an Odin application need not be concerned with the client's link to the network changing. The end-point of a link always corresponds to the client's IP and MAC addresses, along with the unique BSSID assigned by Odin. Furthermore, since the LVAP abstraction preserves the interfaces between the client and the APs, and does not expect any client-side modifications, the design is compatible with enterprise WiFi security protocols such as WPA2 Enterprise.

Section 5.3 explains the details of the LVAP assignment at the protocol level, along with the associated resource utilisation. Note that LVAPs are considerably different from using virtual interfaces on a physical wireless interface (as is leveraged by OpenRoads [29]), wherein networks are sliced by the network name (or SSID). With regular virtual APs, there is a state machine maintained for each association that the infrastructure has no control of.

4.4 Summary

In Odin, the infrastructure's WiFi MAC is decoupled from the individual physical APs. This is done using the concept of a virtual link, realised using LVAPs. That is, the WiFi client always sees a consistent link to the network, regardless of the physical AP it is actually transmitting through. The framework provides primitives for programmers to implement network services as ‘apps’, whilst taking into account the specific needs of enterprise WLANs. Lastly, Odin’s architecture is compliant with the 802.11 protocol, and thus does not require any special software or hardware on the client. In the following chapter (Chapter 5), we describe how this architecture of Odin is realised into an implementation.

5 Implementation

5.1 Odin Master

The Odin master is implemented as an application on top of the Floodlight OpenFlow controller [57]. The master uses the OpenFlow protocol to update forwarding tables on the APs and switches. In its current design, Odin applications execute as a thread on top of the Odin Master. The master uses a separate control channel to invoke commands on the agents. The protocol between the master and the agents is described in Table 5.1. The master listens on a particular port for messages from the agents. When the master receives a `PING` message from an agent for the first time, the former checks to see if there is an OpenFlow switch corresponding to the agent's AP registered as well. If yes, the master "tracks" the agent. The `PING` message from the agents also serves as a keep-alive mechanism. As of now, the system assumes all agents attempting to connect to the master are legitimate. Upon missing a configurable number of `PING` messages from an agent, the master declares the agent to have failed and clears the agent from its agent tracker. An application running on the master can also request for statistics and network information. This can work in both pull-based (active probing) and push based (publish-subscribe mechanism) mechanisms. For the former, the master uses the `QUERY_STATS` protocol message to retrieve a set of statistics that the agent collects. Applications can then view these statistics in a key-value format. The publish subscribe mechanism works with the application expressing "interests" to the master, the master informing the agents about the interests, and the agents notifying the master when a packet is received that matches a subscription. This mechanism is described in detail in Section 5.5.

5.2 Odin Agent

Odin agents run on physical APs, and are implemented using Click [58]. We chose Click because of its flexible and modular packet processing capabilities. As mentioned in Section 4.2,

Table 5.1: Protocol between Odin master and agents

Message	**Arguments**	**Direction**	**Description**
`ADD_LVAP`	`ip_addr`$_{sta}$ `mac_addr`$_{sta}$, `lvap_bssid`$_{sta}$, `lvap_ssid`$_{sta}$	Master $\rightarrow$ Agent	Used by the master to spawn an LVAP at an agent
`REMOVE_LVAP`	`mac_addr`$_{sta}$	Master $\rightarrow$ Agent	Used by the master to remove an LVAP from an agent
`QUERY_STATS`	None	Master $\rightarrow$ Agent	Used by the master to query an agent for statistics
`ADD_SUBSCRIPTION`	`subscription_string`	Master $\rightarrow$ Agent	Insert a subscription for a per-frame event at the agent
`RECVD_PROBE_REQUEST`	`mac_addr`$_{sta}$	Agent $\rightarrow$ Master	Used by the agent to inform the Master that a probe-scan from a client mac_addr$_{sta}$ was detected. This can trigger an LVAP assignment by the master.
`PING`	None	Agent $\rightarrow$ Master	Serves as a heartbeat from the agent to the controller, and is used to detect newly spawned agents
`PUBLISH`	`mac_addr`$_{sta}$, `list`$_{subscription\_Ids}$	Agent $\rightarrow$ Master	Notify the master that a frame was received from the client `mac_addr`$_{sta}$ which matched the subscription Ids in `list`$_{subscription\_Ids}$

the agents contain the logic for the Wi-Fi split-MAC and LVAP handling. The agent runs atop a network interface running in Monitor mode. This allows the agent to receive all frames that the interface can listen to (includes both management and data frames), along with per-frame reception information exposed using a radiotap header [59]. This information includes the signal strength of the reception, the bit-rate or Modulation Coding Scheme for 802.11n at which the frame was transmitted by the source, and the noise. The agent saves this information on a per-source basis, and keeps track of "last heard" timestamps for each source as well. The physical AP hosting the agent also requires a modified Wi-Fi device driver to generate ACK frames for every LVAP hosted on the AP (explained in Section 5.3). Open vSwitch [60] is used to turn the

AP into an OpenFlow enabled switch.

When an LVAP is migrated and a new data-path is used for forwarding a client's traffic, the transport network can issue an ARP request for the client's IP address (since the physical AP runs as a bridge). Again, with the goal of client transparency in mind, the agents intercept ARP requests for the client's IP, and use the LVAP information to answer with the appropriate ARP responses themselves.

Although this aspect of its implementation is still under way, agents will also perform encryption key installation into the WiFi driver as part of WPA/WPA2 authentication. This is explained in detail in Section 5.6.

5.3 Realising LVAPs

The first step in assigning per-client LVAPs is to have Odin agents track probe request frames generated by clients[1]. When an agent receives a probe request from a client for which it is not already hosting an LVAP, the agent informs the master about it using the `RECVD_PROBE_REQUEST` message. If the client is not already associated with the network, the Odin Master spawns an LVAP on the Odin agent which received the probe request (or one of the many agents which received the request). The agent then responds to the client probe request using a BSSID and SSID provided by the master. This BSSID/SSID combination can either be statically assigned on a per-client basis, or generated programmatically by the master. To ensure logical separation between clients and their respective LVAPs, beacons (and probe requests) are only unicast to the clients. Since the client's lower MAC will drop unicast frames that are not destined to it (when operating as stations in infrastructure mode), beacons are only received by the clients to which they are unicast specifically to. Thus, each client can be presented with a unique BSSID to connect to. From this point onwards, the client connects to the newly spawned LVAP as in standard 802.11. An important implementation detail is to ensure that a client is consistently provided the same BSSID. In its current form, we statically assign BSSIDs to clients, but this is easily remedied by generating a BSSID deterministically for the client on the basis of its MAC address. Per-user SSIDs can be statically assigned, or a common SSID can be provided to all

[1]For this discussion, we consider clients that are connecting in open authentication mode. We explain two different modes of authentication that are possible with Odin's architecture in Section 5.6

clients that attempt to connect to the network.

As mentioned earlier, the agent needs to ensure that an ACK is generated for each data packet that the client sends to its LVAP. ACK frames are handled by the hardware because they have a strict real time constraint. Our testbed uses `ath9k` [61] based wireless cards running the OpenWRT Linux distribution [62]. `ath9k` based cards use a register which act as a BSSID mask. This mask is used by the hardware to decide which incoming frames to generate ACKs for. The value that this mask holds is equal to the common bits of all BSSIDs that are being hosted on the device (as a consequence of having multiple virtual interfaces). The algorithm is as follows:

```
$bssid\_mask = 0xffffffffffff$
foreach $bssid$ in $BSSID\_SET$ do
    $bssid\_mask = bssid\_mask \ \& \ !(hw\_mac\_addr \oplus bssid)$
end
```

And on receiving a frame, the hardware verifies whether the incoming frame's destination MAC address matches the BSSID mask. The check performed in hardware is as follows:

```
if $(frame\_dst\_mac\_addr \ \& \ bssid\_mask)$
    $==$ $(bssid\_mask \ \& \ hw\_mac\_addr)$ then
    $//acceptframe$
end
```

Thus, we modified the `ath9k` driver to expose a `debugfs` [63] based userspace interface to allow modification of the hardware's BSSID mask register. The BSSID mask value should include the BSSIDs of all the LVAPs in the mask calculation. This value is computed, updated and applied to the register by the Odin agent whenever an LVAP is added or removed. Thus any data frame destined to a BSSID that matches the said mask will force the access point to generate an ACK from hardware.

An LVAP is represented by the following four-tuple:
$\{\texttt{mac\_address}_{client}, \texttt{ip\_v4\_address}_{client}, \texttt{lvap\_bssid}_{client}, \texttt{lvap\_ssid}_{client}\}$

This takes up a fixed size of 16 bytes, along with the length of the SSID. It is stored on the agent within a hash map keyed by the client's MAC address. With a worst case length of 32 characters for the SSID, the storage involved on the AP with each addition of an LVAP is only 48 bytes. For this reason, hosting an LVAP does not incur a significant memory load on the AP. For each frame that the AP generates to a client, a lookup on this hash map is performed in order to obtain the right BSSID value to be used as the source MAC address in the 802.11 frame. Since this lookup is constant time and is not a function of the number of LVAPs, the processing load on the AP will remain a function of the number of packets the agent has to process (as is the case with a regular AP), and not a function of the number of LVAPs. This is validated through an evaluation in Section 6.

However, since the agent generates beacons corresponding to each LVAP, having too many LVAPs can lead to more beacon generation, which in turn leads to more contention on the channel. Although we're yet to measure the impact of this on a client throughput, this can be easily remedied by using longer beacon intervals.

5.4 Failure Scenarios

Odin's current implementation comprises a single master server. This is a consequence of the underlying OpenFlow controller, Floodlight, not being distributed. Note that the master's reliability follows directly from the reliability of the control platform being used, but this is orthogonal to introducing programmability into the enterprise WLAN. That said, if the Odin master (or the controller) fails, there are the following consequences. First, the system cannot assign fresh LVAPs to newly connecting clients. Second, Odin applications are no longer running, implying that network services like mobility management and LVAP migrations will not function. This implies that connected clients can only transmit via the physical AP on which their LVAP is hosted as of the time at which the master fails.

However, the above issue is easily circumvented. This is because by design, the master does not maintain any state that needs to be preserved after a failure. When the master boots, it waits to receive a `PING` message from the agents in order to track them. If the master is hearing from an agent for the first time, it also probes the agent to see if the latter is hosting any LVAPs. If yes, it records the LVAP information and their binding to the respective agent. Thus, the master always syncs with the rest of the system upon boot. The convergence time depends on the ping interval used by the agents.

This design makes the Odin master a natural fit for running on top of a high availability cluster suite like Linux HA [64] in failover mode. A standby instance of Floodlight and the Odin Master can be launched as soon as the cluster suite detects the failure of the current master.

As mentioned in Section 5.1, the `PING` messages from the agents serve as a heartbeat. When the master misses a configurable number of pings from the agents, it assumes the agent to have failed. It then marks the clients associated with the failed agent as 'unassigned'. At this point, if the client generates data frames that are detected by a neighbouring agent, it can be immediately re-assigned using a mobility manager. If the client generates a probe scan, the master performs a re-assignment.

5.5 Supporting Proactive and Reactive Programming Models

Depending on the network service being implemented, an Odin application will need to work either proactively, reactively or both. An example of a purely proactive application can be an AP load balancer, which looks at the number of clients associated with each AP periodically, and then hands off clients such that the processing load over the APs remains balanced. An example of a purely reactive application can be a mobility manager, which would initiate a handoff if the framework detects that a client is moving.

To support the proactive model, every application executes as a thread on the Odin master which can cycle between sleeping and performing some activity. To support the reactive programming model, applications will need a mechanism to express interest in certain events to the framework, and the framework will need to notify the concerned applications whenever such events occur. This is a natural fit for a content-based publish-subscribe solution, because the master can act as a broker for events published by the agents that are to be delivered to interested applications. We now describe how this is implemented.

First, applications register subscriptions with the master. A subscription is described by the following four fields:

1. `src-addr`: can be a valid unicast MAC Address, or '$*$' (which means 'any'). Only frames with this value for a source address are of interest to the subscriber.

2. `stat`: any statistic being recorded at the agent.

3. `rel`: can hold values `EQUALS`, `GREATER_THAN`, or `LESS_THAN`.

4. `val`: a double value.

The above is interpreted by the master as follows: *"Notify the application if a frame is received from* `src-addr`*, such that the value of the statistic* `stat` *corresponding to the frame should have a relation* `rel` *with the value* `val`*"*

For each subscription that an application registers with the master, it should also provide a callback that is to be invoked as a notification. The callback's arguments include the subscription message that lead to the notification, and context information (such as the source address of the frame which triggered the notification, and the agent at which the event was recorded). The master assigns each subscription a unique identifier, and then informs all concerned agents about the subscriptions. When an agent receives a frame, it performs a check to see if any of the collected statistics corresponding to the packet matches with any of the subscriptions that have been registered with it. If there are matches, the agent then forwards the master a `PUBLISH` message, indicating the source address of the frame that triggered the notification, along with the identifiers of the subscriptions that matched the frame. The master then forwards the events to the concerned applications by invoking the appropriate application provided handlers.

As an example, a mobility manager can register a subscription `["*","signal", GREATER_THAN, 180]`, and associate a handler with the same. This subscription is then pushed to the agents via the master. If the agent receives any frame (regardless of the source address, because of the "*" value for the address) for which the signal strength is greater than 180, the agent sends a `PUBLISH` notification to the master indicating the subscription id against which the frame matched, along with the value of the signal strength that triggered the match. The handler corresponding to the subscription is then invoked, and the relevant context information is passed to the application as well. The context information includes the client that triggered the subscription, the agent at which the notification was generated, and the corresponding value of the signal strength. Within the handler, the application can track the signal strengths of the client as observed by different agents, and use this information to make LVAP-handoff decisions.

5.6 Security

Security is a key component of any enterprise WiFi management solution. The implementation for this is on-going work, but we describe how Odin's architecture is compatible with two approaches to authentication, both of which fit well with the LVAP abstraction scheme.

5.6.1 Post authentication using WPA2 Enterprise

In post authentication, a client is authenticated against the system after it is assigned a LVAP. This can be done using 802.1X [8] and WPA2 Enterprise, the de-facto standard for authentication and security in enterprise networks.

In WPA2 Enterprise, a client (or supplicant) authenticates against an authentication server with the access point acting as an authentication proxy. The first time the client connects to the network, it will begin a probe scan, to which the infrastructure will respond with probe responses from the client's LVAP. In the next step, the client performs open authentication with the access point, and then associates with it. The client is now in the unauthenticated state, and only EAPOL frames from the client are accepted by the access point. The client then uses its credentials to authenticate against the authentication server, mediated by the access point. If the client successfully authenticates, the server provides the client with the master key that is to be used for the session, from which both entities derive a pairwise-master key (PMK). The server then moves the PMK to the authenticator (the access point).

Next, a four way handshake is performed between the access point and the client to derive a session key:

1. the AP generates a random value called the ANonce and sends it to the client.

2. the client generates a random value called the SNonce. It then derives the Pairwise Transient Key (PTK) by making use of the ANonce, the SNonce, and the PMK. It then informs the AP about these two values including a message integrity code (MIC).

3. The AP derives the PTK using the ANonce, the SNonce, and the PMK. The AP then informs the client about the group temporal key (GTK), which is used for encrypting broadcast messages.

4. client installs the PTK and the GTK, then sends an acknowledgement to the AP. The latter then proceeds to install the PTK.

Any entity with the PTK or session key can thus successfully encrypt and decrypt packets in a WPA2 secured session. Including this session key in the LVAP encapsulation, and installing the key in the new AP would suffice to allow secure roaming of the client. Note that this approach is also compatible with the re-keying mechanism in WPA2, wherein the access point can invoke the 4-way handshake again with the client in order to refresh the session key being used.

Furthermore, in the AES-CCMP and AES-CMAC algorithms, a replay counter is used internally in the encryption process to prevent replay attacks. This replay counter is encoded in the WPA2 encrypted packet as a 48-bit packet number (which itself is part of a 64 bit initialisation vector). The only constraint here is that the packet numbers must be monotonically increasing. Thus as long as the packets generated by the new AP satisfy the monotonically increasing sequence property of the same, then the STA's verification of the packet number will not fail. This can be easily done by having the replay counters used by the access points for each client correspond to an NTP synchronised timestamp, or by incorporating the replay counter state in the LVAP handoff. Thus, Odin's LVAP based handoffs can be used even in the presence of standard enterprise grade WiFi encryption.

Thus, the transferability of the transient keys (or the re-computability of it), along with the use of a high enough replay counter value will suffice to implement LVAP handovers in the face of encryption.

5.6.2 Pre authentication for Guest WiFi

In pre authentication, a client is assigned its own LVAP only after it has authenticated against the system. This fits with the common mode of authentication in guest WiFi systems and in public WiFi environments (like in cafés), wherein a user is given an unsecured WiFi connection, and is re- directed to a login page through the web browser. Guest and public WiFi systems usually provide the user with credentials prior to this access attempt. The authors in [65] propose the use of OpenID and OAuth2 [66] for logging into such WiFi networks.

Regardless of the type of credentials used, the authentication mechanism can return a token to the Odin Master for the client after successful authentication by the latter. The LVAP that is then spawned for the client can either be unencrypted, or require the client to negotiate a key with the access point (and thus the Master) over 802.1X, which is then bound with the LVAPs state. Furthermore, in a true Open WiFi scenario with a shared controller for a number of networks, the LVAPs themselves can be re-used, allowing the client to use cached credentials for accessing a WPA2 enterprise protected network.

In enterprises which require the user to provide the IT section with their MAC addresses for access control, pre-authentication can be implemented trivially by having the Odin Master perform a lookup in a database of MAC addresses when Odin agents report a probe scan by a particular client. Only clients which appear in this database are assigned a LVAP by the Master. In its current implementation, Odin supports such a form of pre-authentication.

5.7 Example Odin Applications

In this section, we illustrate example applications that can be built on top of Odin (note that we haven't implemented all of them). As we describe in Section 4.2, these use- cases cannot be handled using only OpenFlow. This is because the scope of the OpenFlow protocol does not cover the 802.11 MAC.

5.7.1 Seamless Mobility

As explained in Section 4.3, an LVAP handoff does not affect the 802.11 state machine at the client. This property can be leveraged by an Odin based application to implement a mobility manager.

Figure 5.1 shows how an LVAP based mobility manager can work. By striving to keep an LVAP as close as possible (on the basis of some metric, e.g. signal strength) to the corresponding client, the latter continues to receive ACKs for data frames that it generates, preventing it from performing a re-scan.

This mobility manager can be implemented using a reactive model, by leveraging the event subscription mechanism of the Odin framework. One way to achieve this would be for the

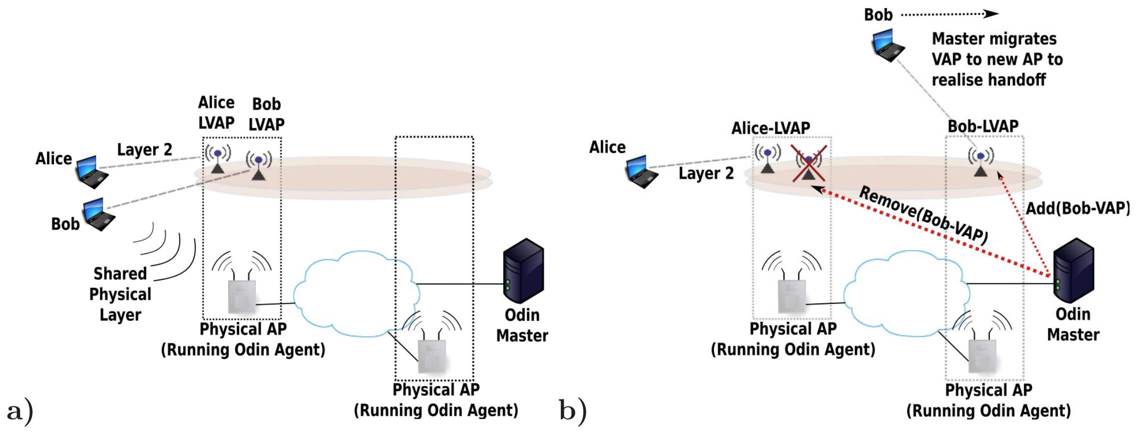
Alice
Bob
Alice LVAP
Bob LVAP
Layer 2
Shared Physical Layer
Physical AP (Running Odin Agent)
Odin Master
Physical AP (Running Odin Agent)
a)
Bob
Master migrates VAP to new AP to realise handoff
Alice-LVAP
Bob-LVAP
Alice
Layer 2
Remove(Bob-VAP)
Add(Bob-VAP)
Physical AP (Running Odin Agent)
Odin Master
Physical AP (Running Odin Agent)
b)

Figure 5.1: LVAP abstraction being used for mobility management: a) Clients connect to LVAPs, with one or more LVAPs being hosted on the same physical AP. LVAPs on top of a physical AP share the same wireless channel. b) The Odin master detects a client movement and performs an LVAP migration to realize a smooth handoff.

application to inform the master to invoke a handler whenever a frame is received at any agent at a signal strength greater than a specific value. Using the context information passed by the master when the handler is invoked, the application can then verify if the subscription was triggered at an agent different from the one where the client is currently associated. It can also determine whether the signal strength at this agent is higher than the one at which the client is currently associated. If yes, then the agent can LVAP-handoff the client to the new agent. An example implementation is shown in Appendix A.1.

5.7.2 Load Balancing

In Section 3.4 we discussed re-association based load balancing, wherein the load across multiple APs can be balanced by dynamically re-assigning clients to different APs. Different metrics (or combinations of them) can be used to determine when and how these re-assignments should be performed. Most existing work employs specialized software on the client, which the infrastructure uses to control client associations (as described in Section 3.4). Since handoffs with LVAPs are inexpensive and fast, re-association based load balancing can be easily implemented as an Odin application (in Section 6, we show that even executing these handoffs every 100 ms does not result in any TCP perceived throughput degradation at the client). One possible approach to implement a load balancer application is for it to run in proactive mode, and periodically inspect the number of active LVAPs per agent, and the processing load on each physical AP. It can then use a heuristic to re- assign LVAPs to even out the processing load on different APs, whilst taking care to ensure that the clients are still within transmission range of the physical APs to which their respective LVAPs are migrated to.

5.7.3 Hidden Terminal Mitigation

Hidden terminal mitigation is a classic problem in enterprise WLANs. Many approaches have been researched to measure and mitigate the impact of hidden terminals in enterprise WLAN environments. These span from adaptive RTS/CTS to centralised scheduling (such as in CENTAUR [31]). However, most approaches require client modifications, due to the lack of control on the clients' association decisions. A hidden terminal mitigation application is a natural fit for Odin because of the centralized view of the network at the master, and the control over the association state of a client. This application would require some extensions to the agents (with regards to information collected about the channel). The application can then provide per client measurements of link impairments (see [67]) such as hidden and exposed terminals, and collisions.

5.7.4 Access Control

Access control is a common feature in enterprise WLANs. A typical use-case is restricting guest users from accessing servers that are part of the internal employee network. This form of access control can also be performed using LVAPs. In Odin's current setup, a set of OpenFlow flow table entries is associated with each LVAP. Every time the master triggers an LVAP-handoff, the corresponding flow table entries are installed on the new physical AP as well. This mechanism can be extended further by having an Odin application associate a set of forwarding table entries that implement access control policies with a particular LVAP. The policies can be differentiated according to the type of user, which can be determined from an authentication server (Section 5.6). Every time the LVAP is migrated between physical APs, the access control policies for the corresponding client would be moved to the new AP as well.

5.8 Summary

In this chapter, we have described the implementation of Odin, based on the design described in Section 4. We've described how the Odin master, and agents work, along with the implementation details of LVAPs. We have also explained how Odin's design is compatible with standard WiFi security protocols. Lastly, we also described some possible applications that can be built on top of the Odin framework. In the subsequent chapter (Chapter 6), we conduct a performance evaluation of this implementation.

6 Evaluation

The overall objective of Odin is to enable programmability of WLANs with an emphasis on not requiring client side modifications. It is important to see the different overheads incurred by the system in order to support programmability of a WLAN. In this chapter, we present an evaluation of Odin to understand how it performs under different workloads. The objective is to understand the practicality of the system.

The evaluations are conducted under the following operational setting for Odin. An Odin based network comprises clients that connect to the network, the agents that host per-client LVAPs, the master that assigns LVAPs, and applications that run atop the master. Apart from the channel conditions, the throughput that a client gets when transmitting data via an LVAP is dependent on the packet forwarding performance of the agent, along with the impact of LVAP handoffs that can be triggered by different Odin applications. Once a client connects to the network, the master installs OpenFlow forwarding rules for that client on the physical AP corresponding to the client's LVAP. This leads to data flows being handled fully by the agent, and not incurring an overhead at the master. The overheads incurred at the master during run time are that of maintaining state for each client and agent in the system, and handling control traffic from agents (caused by OpenFlow messages, subscriptions triggered at the agent, `PING` and `PROBE` messages).

Metrics: Given the above mentioned operational nature of the system, we evaluate the following.

1. **Impact of LVAP handoffs on an Odin client's throughput**. This will help understand if it is practical to have Odin applications perform LVAP handoffs, which is an important primitive for Odin applications. This is evaluated within Section 6.2.

2. **CPU, and memory overhead of the agent when hosting LVAPs**. This includes active LVAPs where the corresponding client is transmitting a data flow, and passive

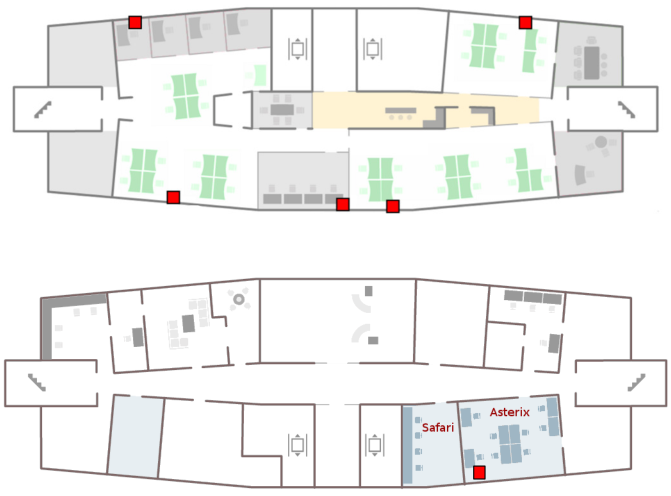

Figure 6.1: The x86 based wireless testbed in an office environment on the 15^{th} (bottom figure) and 16^{th} (top figure) floors of the TEL building at TU-Berlin. The red squares indicate the positions of the APs that comprise the indoor testbed.

LVAPs where there is an LVAP assigned but the client is yet to associate with it and transmit data flows. This will give us an understanding of the performance limits of agents. This is evaluated within Section 6.3.

3. **CPU, memory, and control traffic overhead at the Odin master** under different workloads. This is important to understand the scalability of the master with regards to maintaining client and agent state, and responding to control messages from the agents. This is evaluated within Section 6.4 and Section 6.4.4.

6.1 Testbed description

The testbed for the evaluations consist of six x86 based PC Engines 3d2 WiFi APs equipped with an IEEE 802.11abgn AR9220/AR9280 WiFi card used with the `ath9k` driver. 2dBi "rubber duck" antennas are attached to the WiFi card. The operating system is the OpenWrt "Backfire" release (subversion revision 29984). The nodes run Open vSwitch version 1.0.3, and Click version 2.0.1. The APs are placed across the 15^{th} and 16^{th} floors of the TEL building at the TU-Berlin campus as indicated in Figure 6.1. For running the Floodlight controller and Odin Master, a KVM based virtual machine is used with the specifications outlined in Table 6.1.

Table 6.1: Specifications of virtual machine used to host the OpenFlow controller and Odin master

`Total Memory`	1000M
`Processor type`	QEMU Virtual CPU version 0.12.3
`Number of processing cores`	1
`CPU MHz`	2133.087
`CPU cache size`	4096 KB
`CPU frequency stepping levels`	3

6.2 LVAP-Handoff Feasibility Evaluation

LVAPs offer a mechanism to abstract the association state of a connection (to a client) away from physical APs. As mentioned in Chapter 4, an LVAP migration should not trigger any layer 2 or layer 3 protocol message exchanges between the client and the concerned physical APs. Odin applications can make use of LVAP migrations to implement different network services. It is important that these migrations can be completed fast enough that it does not affect a client's throughput, or interrupt its TCP sessions. In the following experiments, we validate whether LVAP-handoffs satisfy this requirement. If they do introduce significant interruptions to a client's connections, the practicality of the system is limited. Note that we evaluate the CPU/memory load on an agent hosting LVAPs in Section 6.3.

6.2.1 Layer 2 Delay in Legacy WiFi Re-association

Objective: In an Odin network, an LVAP-handoff does not trigger the layer 2 re-association mechanism. The aim of this experiment is to observe the handoff delay at layer 2 in the case of a regular 802.11 handoff across different combinations of channels. It will thus help us understand the delays we can avoid when performing LVAP handoffs. This is our baseline comparison.

Experiment description: The experiment is conducted within the TEL building's 16th floor in the TU-Berlin campus during normal working hours. A Linux based laptop is used as a client to resemble a typical user device, running the `brcsmac` driver on Linux kernel version 3.1.0-030100-generic with Ubuntu 10.04. Two APs of the same specifications as described in Section 6.1 are used. The client is first made to connect to one AP, and then forced to re-associate with the other AP. Using the `nmcli` utility, we observe the layer 2 delay incurred for

Table 6.2: Latencies involved in a layer 2 handoff using open authentication. The latency is higher when handing off from one channel to another than within the same channel. This is because the probe scan duration is lower.

Scenario	Min (s)	Max (s)	Avg (s)	Std. Dev.
Same channel handoff	0.1737	0.3793	0.1897	0.0320
Different channel handoff	0.1767	18.3059	0.3632	1.0260

performing a handoff. This is measured as the time it takes for the client's WiFi driver to transition from the state of having disconnected from the first AP, to having completed the association with the new AP. The experiments were performed in the 2.4Ghz band since most consumer WiFi devices operate within this frequency, providing a noisy environment. Handoffs were performed between all combinations of channels (1 to 11). The experiment is repeated 10 times and the results averaged.

Discussion: The results of this experiment are described in Table 6.2. The average handoff delay of ∼350ms is consistent with the existing literature [68, 40]. The average delay for a handoff within the same channel is ∼190ms. Note that the delay for handing off within the same channel does not apply to Odin because of the LVAP based handoff mechanism (which does not trigger a client side state machine change, and thus no protocol message exchanges). As mentioned in Chapter 5, in case of Odin, an LVAP-handoff is executed by issuing the `REMOVE_LVAP` command to the old AP, and the `ADD_LVAP` command to the new AP. The period of disconnectivity for the client (when it does not get ACKs for data frames) is the time between the old AP having executed the `REMOVE_LVAP` command, and the new AP executing the `ADD_LVAP` command. If this delay is small enough, the client won't observe any disconnectivity and will not disassociate. We demonstrate the performance effects of not triggering the state machine change at the client in the next experiment (Section 6.2.2).

6.2.2 Impact of Single Handoff on HTTP download throughput (Odin vs. IEEE 802.11)

Objective: An LVAP-handoff in an Odin based network does not trigger any layer 2 or 3 protocol messages to be exchanged between the client and the AP. It is thus devoid of the handoff delays shown in Table 6.2. Intuitively, if this LVAP-handoff is performed fast enough, the impact on a client's active data flows should be minimal. This experiment is aimed at validating this claim. We use a static IP for the client and open authentication in this experiment, thus avoiding delays caused by DHCP and authentication. This gives us a lower bound for the delays involved in handoffs. The experiment is repeated 10 times and the results averaged.

Experiment description: We use the same AP setup as in the previous experiment, but we operate the APs within the 5Ghz band, which in our testbed environment, is not as noisy as the 2.4Ghz band. This allows us to compare the effect of a regular 802.11 handoff and an LVAP-handoff on a client's throughput, with a lesser degree of perturbation from channel noise. The client used for this setup is of the same specification as the APs (described in Section 6.1). The client first connects to one AP, and initiates an HTTP download using the `wget` utility. We use HTTP traffic since it is a common client activity. It is then made to handoff after 13 seconds to another AP. When performing a regular 802.11 handoff, the client is made to explicitly re-associate using the `iw` command. When using Odin, an LVAP-handoff is performed instead. We then observe the client's HTTP flow using a `tcpdump` trace, and extract its throughput over one second intervals using `tshark`. The client uses a static IP and open authentication, which means the delay is purely caused by a layer 2 change in case of regular 802.11. The AP and the client use a fixed bit-rate of 54 MBit/sec. Note that in Odin, the LVAP-handoff does not trigger layer 3 or layer 2 messages.

Discussion: Figure 6.2 shows the throughput over time when using a regular 802.11 setup and having a handoff performed during a download session. We observe that the throughput drops to zero for up to 4 seconds before recovering. Note that in the presence of DHCP, and 802.1X authentication, this delay would be much larger. Figure 6.3 describes the same experiment performed with Odin and LVAP-handoffs instead of regular handoffs. We can see that the throughput curve is uninterrupted in spite of the LVAP-handoff. However, Figure 6.3 indicates an overall reduction of throughput (close to 5Mbit/sec) as opposed to that in Figure 6.2. This is because we are using userspace Click to run the Odin agents, leading to slower and jittery forwarding performance on our AP hardware. This forces the client's TCP connection (used for the HTTP download) to throttle down. However, this is orthogonal to maintaining layer 2 and 3 connectivity in spite of performing a handoff, which Odin does achieve.

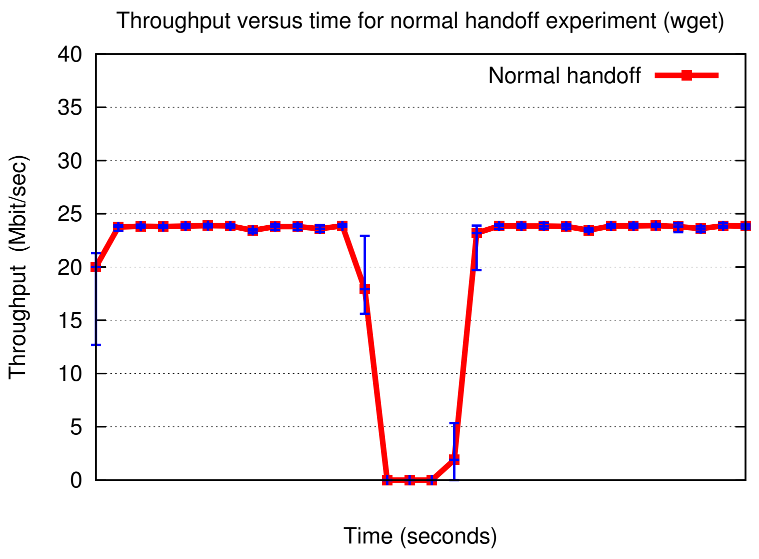

Figure 6.2: Effect of a regular 802.11 handoff without authentication or DHCP on the throughput of a file download over HTTP. There is a period of disconnectivity of up to 4 seconds when such a handoff is made.

6.2.3 LVAP Handoff Micro-Benchmark

Objective: As shown in the previous experiment (Section 6.2.2), a single LVAP-handoff has a negligible impact on a client's download throughput. Since LVAP-handoffs are an important aspect of an Odin, it is important to understand how frequently they can be executed by Odin applications without negatively impacting a client's data flow. This experiment is aimed at observing this.

Experiment description: The same setup as the previous experiment (Section 6.2.2) is used, but an `iperf` based TCP session is executed with the client as the source. The duration of the experiment is 30 seconds, with handoffs being performed repeatedly between the 5^{th} and 25^{th} seconds at a fixed interval. We decrease the interval to observe how the throughput decreases. As with the previous experiment, we use the `tcpdump` and `tshark` utilities to measure the throughput at the client. The experiment is repeated 10 times, and the results averaged.

Discussion: The results of the experiment are shown in Figure 6.4. We observe that in spite of performing an LVAP migration every 100 ms, there is no significant degradation of throughput. Note that it is unrealistic for a network service to have to handoff a client at such a frequency, but serves to validate the inexpensive nature of the LVAP migration.

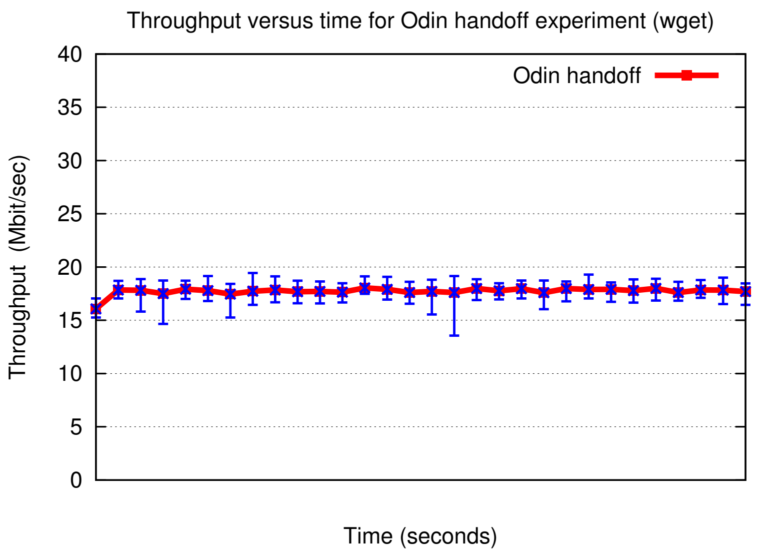

Figure 6.3: Effect of an Odin LVAP-handoff on the throughput of a file download over HTTP. There is no throughput degradation caused by performing a handoff.

6.3 Odin Agent Performance Evaluation

We will now evaluate the CPU and memory overhead on the Odin agent when hosting LVAPs. We observe these metrics on the agent in the presence of passive clients, which are clients that are assigned LVAPs but do not associate with the network, and with active clients, which are clients that do associate with their LVAPs and generate data flows.

6.3.1 Agent Overhead: With Passive Clients

Objective: In this experiment, we observe the CPU and memory overhead on an agent with hosting LVAPs. By conducting the experiment in a regular office setting, we can understand whether an Odin agent can operate with reasonable performance.

Experiment description: A single AP on the 16^{th} floor of the TEL building within the TU-Berlin campus is used for this experiment (Section 6.1). The AP is set to operate on channel 6 of the 2.4Ghz band, which is the most noisy channel in our environment. The AP runs an agent, which connects to the controller node. The specifications of the AP and the controller are the same as in Section 6.1. The agent is left active for a period of 250 seconds. During the 130^{th} second, the master begins pushing subscriptions to the agent every second. Each

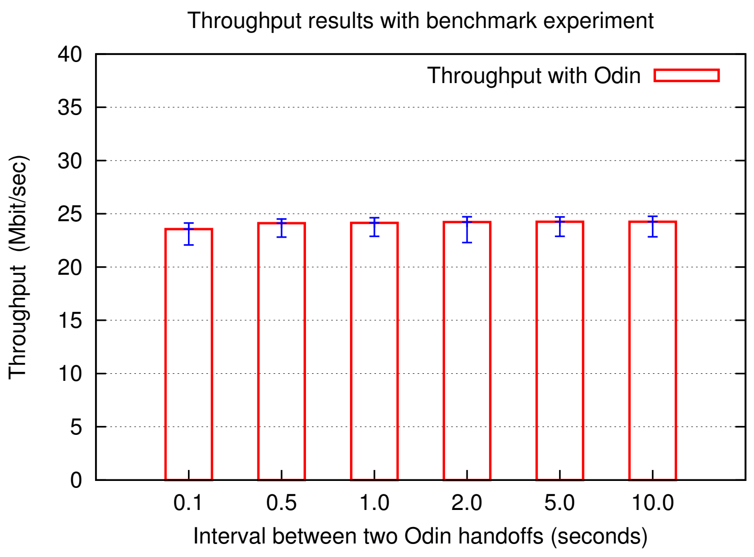

Figure 6.4: Average throughput achieved in the presence of continuous Odin LVAP-handoffs at fixed intervals. The throughput degradation is negligible.

subscription matches against every frame received by the agent. Up to the 130^{th} second, we can observe the baseline performance at the agent when only adding LVAPs without connecting clients. From that point onwards, we can see the effect of the agent triggering an increasing number subscriptions against frames generated by stations in a noisy channel. The CPU and memory overhead at the agent is measured using the `top` utility. The experiment is repeated 10 times, and the results averaged.

Discussion: During the experiment, we used the agent's statistics collection feature to observe the number of stations. The average number of observed stations across all experiments was 82. Figure 6.5 shows the CPU utilisation over time on the AP running the agent. Up to the 130^{th} second, adding of LVAPs have no effect on CPU utilisation. Once the subscriptions are added every second that match against every frame (130^{th} second onwards), we can see that the CPU overhead increases linearly. This is because in its current implementation, the subscriptions are stored as a linked list on the agent, and the entire list is traversed for every frame received. We plan to improve this by using a data structure like an R-tree [69] or a Kd-tree [70] which are more effective for content based subscription matching. Figure 6.6 shows the memory utilisation over time for the same experiment. As mentioned in Section 5.3, the memory overhead associated with each LVAP is a worst case of 48 bytes along with overheads associated with Click's hash table implementation, and other data types used in the LVAP struct (like the

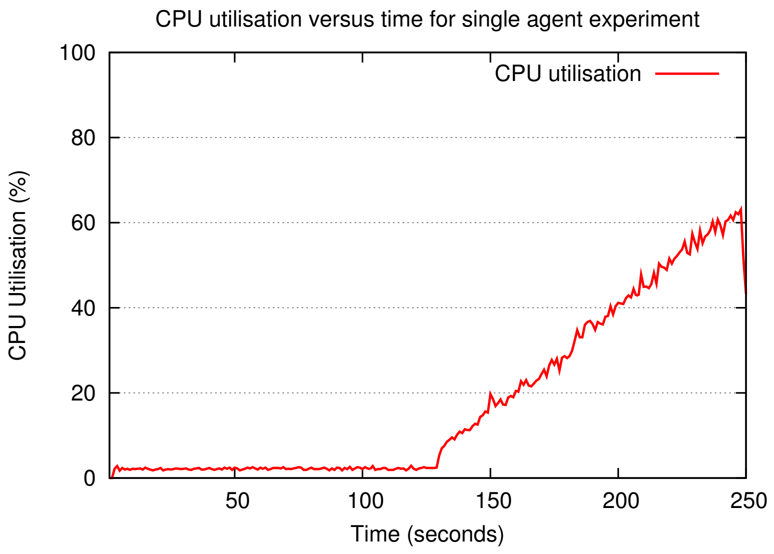

Figure 6.5: CPU overhead versus time for an agent running in a channel with an average of 82 transmitting stations. Note that when there are no subscriptions to match against, there is no CPU overhead associated with an increasing number of LVAPs. As the number of subscriptions increase, the CPU load increases linearly (since the subscriptions are stored as a linked list on the agent, and the list needs to be traversed for each frame received).

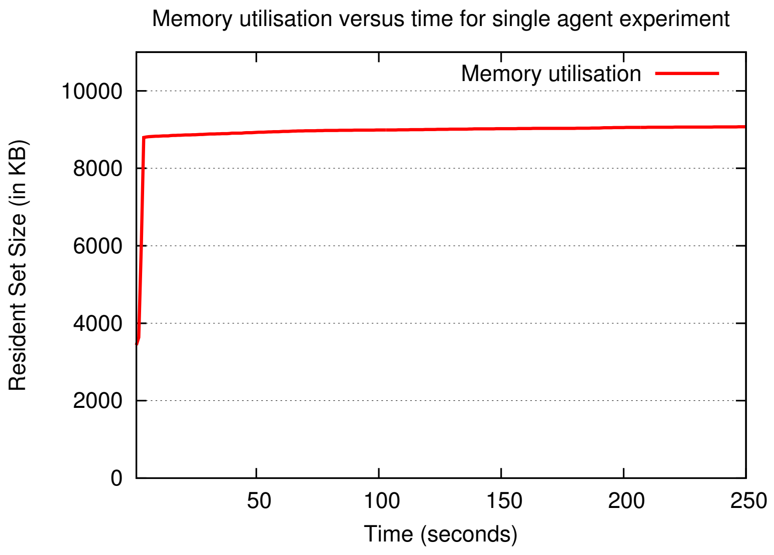

Figure 6.6: Memory overhead versus time for an agent running in a channel with a high number of stations (on the order of 82). As mentioned in Section 5.3, the memory overhead associated with each LVAP is little (worst case of 48 bytes + data type overheads). This is reflected in the curve.

`timestamp` and `EtherAddress` types).

6.3.2 Agent Overhead: With Active Clients

Objective: In this experiment, we observe the CPU and memory overhead on an agent with hosting LVAPs in the presence of active clients. This will help us understand the overheads associated with connected clients that transmit data.

Experiment description: The same setup as the previous experiment is used (Section 6.3.1), except that the single AP is switched to channel 48 in the 5Ghz band. Since this channel has only a few other stations in our testbed (on the order of 10), it will allow us to exclusively observe the agent overhead associated with active clients that are transmitting data. The clients used for the setup are two nodes that are of the same specification as the APs (Section 6.1). The two clients are made to connect to the AP one after the other over a spacing of 20 seconds. The clients begin a large file download over HTTP using `wget` as soon as they associate successfully. At the 130th second, a dummy Odin application begins pushing a subscription per second that matches against every frame received by the AP. The CPU and memory overhead is measured using the `top` utility. The experiment is repeated 10 times, and the results averaged.

Discussion: Figure 6.7 illustrates the CPU utilisation over time with transmitting clients on the AP. The CPU utilisation increases by 6% for every HTTP download being handled (initiated by two different clients) since the agent is processing flows. As already seen in the previous experiment (Section 6.3.1), the increase in processing overhead with the number of subscriptions is linear because of the growing linked list of subscriptions being traversed for every frame received. Figure 6.8 shows the memory overhead during the same experiment. The number of LVAPs being hosted is low, and the memory utilisation only increases negligibly with each subscription added to the agent.

6.4 Odin Master Performance Evaluation

In the following experiments, we will explore the overhead associated with running the Odin master under different workloads. The metrics we will be using for this evaluation are the CPU

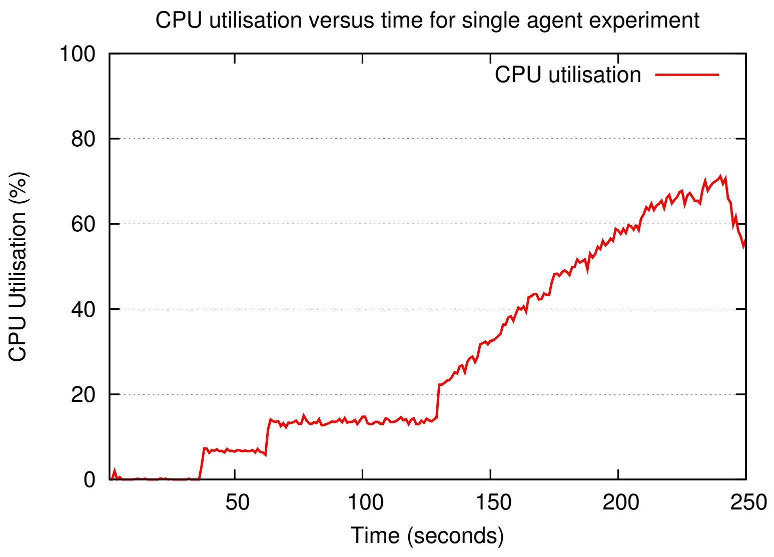

Figure 6.7: CPU overhead versus time on a single AP with two active clients. The jump in CPU utilisation by about 6% per at the 40^{th} and 60^{th} seconds are due to the clients initiating the HTTP downloads. As already observed in the previous experiment (Section 6.3.1), the processing overhead due to subscriptions increases linearly with the number of subscriptions. The load tails off at the end because of the downloads being completed.

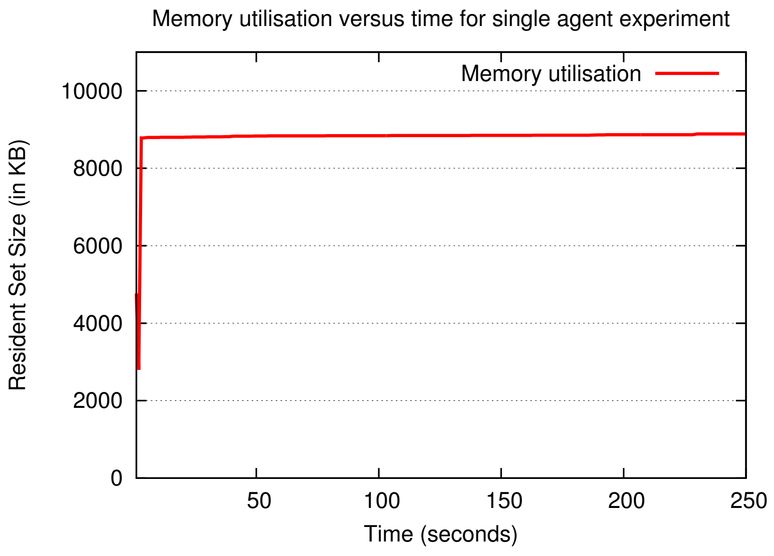

Figure 6.8: Memory overhead versus time on a single AP with two active clients. Since the number of LVAPs being hosted is less, the memory increase over time is negligible, and only increases marginally with the number of subscriptions added.

and memory overheads at the master, and the control traffic generated in the network (as a result of both Odin and OpenFlow messages).

Experimental Setup: The testbed for this evaluation comprises 6 APs distributed across the 15th and 15th floors of the T-Labs building, and a server to run the OpenFlow controller (see Section 6.1).

6.4.1 Master Overhead: No Applications, No Clients

Objective: This experiment aims to observe the CPU, memory, and control traffic overheads that result from the master having to maintain connections and state to a set of Odin agents. The APs are operated in the 5Ghz channel, and the master does not assign any LVAPs, nor run any applications. This eliminates all Odin control traffic with the exception of `PING` messages from the agents. Through this experiment, we will get an indication of the baseline overhead associated with running an Odin setup.

Experiment description: In this experiment, we launch the Odin master, and spawn one agent at a time on the APs over an interval of 10 seconds, starting from the 10th second. There are no applications running on the master, implying that there will be no event subscriptions or statistics queries being executed. Furthermore, the APs are made to operate in the 5Ghz channel which has very little wireless activity so as to not introduce control messages from client scans and data frame transmissions. This setup will indicate the CPU, memory, and control traffic overheads at the master with maintaining connections to each agent. As with the experiments in Section 6.3, we measure the CPU and memory overhead using the `top` utility, and control traffic overhead using `tcpdump` and `tshark`. The experiment is repeated 10 times, and the results are averaged.

Discussion: Figure 6.9 shows the CPU utilisation over time for the controller (and master). After the initial bootstrap period, the CPU utilisation is minimal (less than 10%). The small spikes that are visible at the 40th, 52nd and 67th seconds are due to agent joins and Floodlight's `StaticFlowPusher` component periodically pushing flows onto the Open vSwitch instances. Figure 6.10 indicates the memory utilisation during the same period. Floodlight's use of memory grows up to an average of 140MB during bootstrap before remaining stable. The addition of each agent has a negligible affect on memory consumption. Lastly, Figure 6.11 indicates the

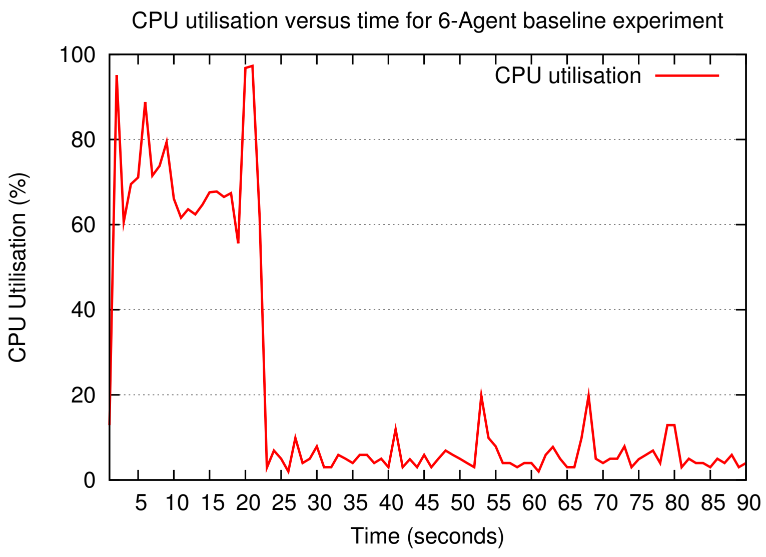

Figure 6.9: CPU utilisation of the master during startup, and synchronisation with each agent. The startup period includes the time it takes for Floodlight to bootstrap, and accept connections from Open vSwitch instances. Agent joins (through the agents' `PING` command), are distributed between the 10^{th} and 80^{th} seconds as can be seen from the small spikes.

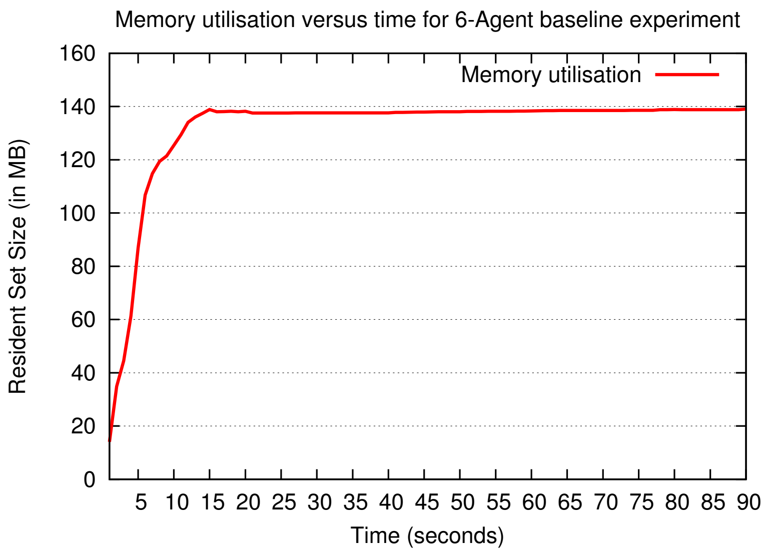

Figure 6.10: Memory utilisation at the master over time. The memory increases during startup as Floodlight instantiates its different components including the master itself, then stabilises at 140Mb.

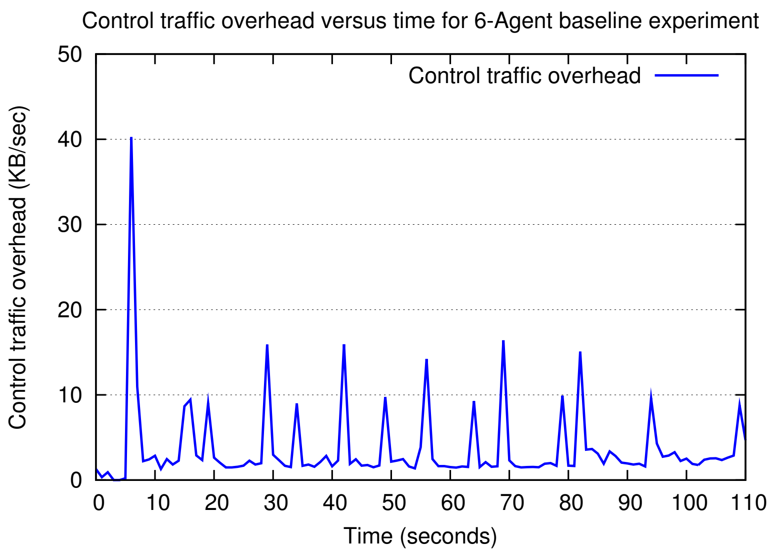

Figure 6.11: Control traffic overhead versus time. This includes the OpenFlow messages exchanged between the controller and the switches, and the protocol messages between the master and the agents. The average value is 3KB/sec in a 6-AP testbed.

control traffic overhead for the system over time. The control traffic includes the OpenFlow messages exchanged between the controller and the Open vSwitch instances on the APs, and the Odin protocol messages exchanged between the master and the agents. The curve indicates an average overhead of 3KB/sec, with periodic spikes which are due to OpenFlow messages. The only Odin protocol messages that contribute to this curve are the `PING` messages from the agents to the master.

6.4.2 Master Overhead: With Applications, Passive Clients

Objective: The aim of this experiment is to observe the additional overhead at the master when assigning LVAPs, and handling a synthetic Odin application. The Odin application pushes a subscription to all the agents every second. The subscription matches against any frame received by an agent. With this setup, we can observe the CPU, memory, and control traffic overhead at the master in an environment with many stations that do not connect to the network.

Experiment description: In this experiment, we launch the Odin master, and spawn one agent at a time on the access points every 40 seconds, starting from the 40^{th} second. The APs operate within channel 6 of the 2.4Ghz band, which is the most noisy channel within our

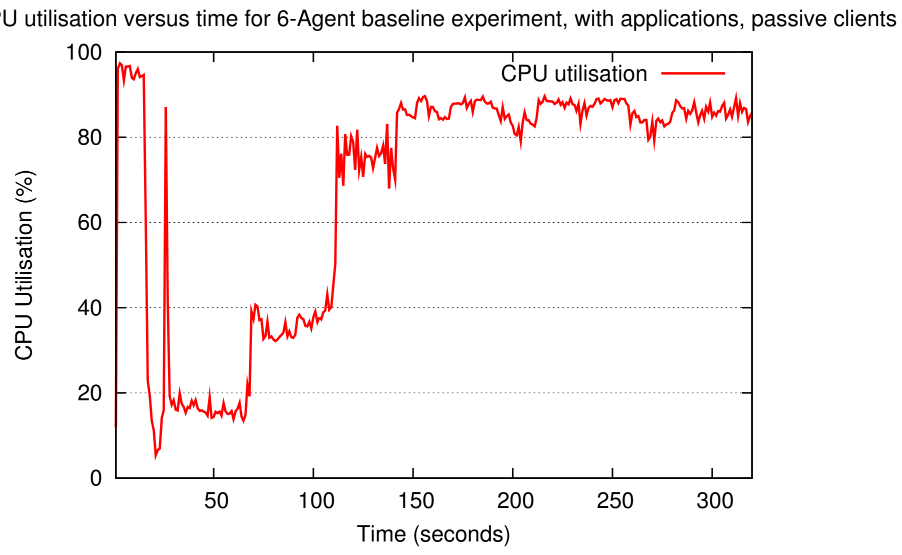

Figure 6.12: CPU utilisation of the master during startup, and synchronisation with each agent. The load on the master increases as the number of agents and the corresponding number of subscriptions increase. The subscription notifications in this experiment are generated by non-associated clients.

lab's environment. The number of unique transmitting stations within this channel is on the order of 250 (across the APS on the 15^{th} and $16th$ floors of our testbed building, described in Section 6.1). We observe this through the statistics collection mechanism of the agents. Odin is run in open access mode, where any station will be assigned an LVAP if it performs a scan. A dummy application runs on the master. The application starts when the master boots, and adds a subscription `["*", "signal", GREATER_THAN, 0]` every second. This subscription will match against any frame received by the agents. Furthermore, If the application has been running for T seconds, every active agent will have T such subscriptions to match every packet against. Ultimately, each frame received by the agent will trigger T subscriptions. This scenario is unrealistic in terms of the number of subscriptions, and in the fact that the subscription matches against any frame. However, it serves to observe the limits of Odin's implementation.

Discussion: Figure 6.12 represents the CPU utilisation over time during this experiment. As can be seen, the processing load on the master increases over time because of the application adding a subscription per second, which matches against frames sent by any of the 250 plus stations in the environment, leading to a burst of `PUBLISH` messages from each agent. With the addition of each agent, the control traffic to be handled by the master increases proportionally. The master server's CPU is a QEMU virtual CPU, which has three stepping states (see Sec-

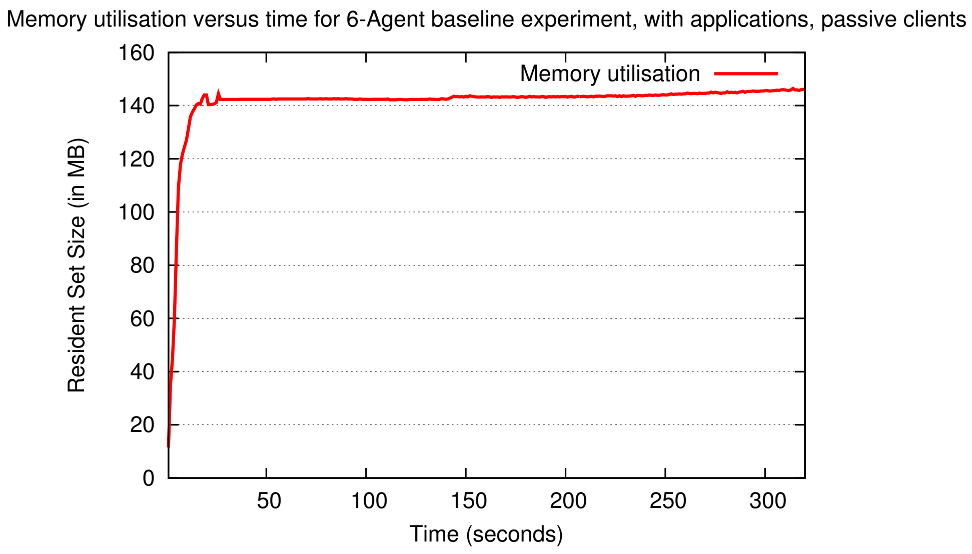

Figure 6.13: Memory utilisation at the master over time. The memory increases during startup as Floodlight instantiates its different components including the master itself. The memory usage does not increase significantly with the increase in number of agents.

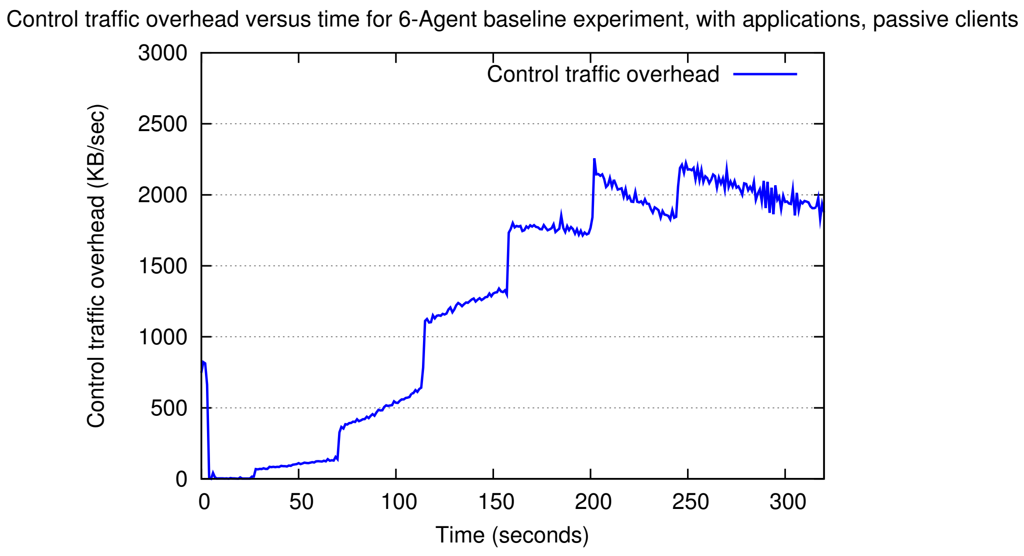

Figure 6.14: Control traffic overhead versus time. This includes the OpenFlow messages exchanged between the controller and the switches, and the protocol messages between the master and the agents. Every frame received in the experiment by an agent matches a subscription. Thus the number of notifications sent to the master by each agent increases with the number of subscriptions added, and increases proportionally with each additional agent. The increase in number of subscriptions at the agent eventually leads to more processing overhead for the agent, and thus the rate of control traffic generated by each agent degrades (the trend is increasingly evident with each additional agent).

tion 6.1). As can be seen in the graph, the server's CPU is pushed to the highest frequency state, and becomes stable at 85%. Figure 6.13 represents the memory utilisation of the master, which increases slowly over time with the number of agents joining the system, and the number of clients that are tracked at the master.

Figure 6.14 represents the control traffic overhead observed at the master over time. The curve's trend can be explained as follows. Each agent handles an increasing number of subscriptions per second, leading to more control traffic being generated per frame received. With each additional agent joining, the overhead increases proportionally. However, as the number of subscriptions to be match a packet against increases, the load on the agent increases as well. This is because the agent's implementation stores the subscriptions as a linked list, which is better suited for smaller numbers of subscriptions. The degradation in the agent's performance can be observed by the decreasing rate of control traffic (from around the 150^{th} second, indicating that each agent on our hardware can perform well up to 150 subscriptions in the presence of passive clients). As mentioned earlier, the agent's matching of subscriptions can be improved further with data structures like R-trees or Kd-trees.

6.4.3 Master Overhead: With Active Clients

Objective: When the master assigns an LVAP to a client, it installs the appropriate OpenFlow based forwarding rules on the corresponding physical AP. Once this flow entry is installed, the switch handles all the data forwarding for the client without the intervention of the master (unless of course, an Odin application requires certain packets of a flow to be forwarded to the controller). By not making the master handle every packet of each flow, the system works around a potential bottleneck at the master. The objective of this experiment is to validate this behaviour.

Experiment description: The same 6-AP testbed is used as the previous experiment (see Section 6.1 for the specifications). There is no application requesting any subscriptions. The APs operate in 5Ghz band which has only on the order of 10 clients present. This will minimise the control traffic generated from probe scans and client connection attempts. As part of the experiment, two clients are made to connect to the same AP and start an HTTP download using `wget`. An AP is launched every two seconds, starting 20 seconds after the master boots. As with the previous experiments, CPU and memory overhead is measured using `top` command,

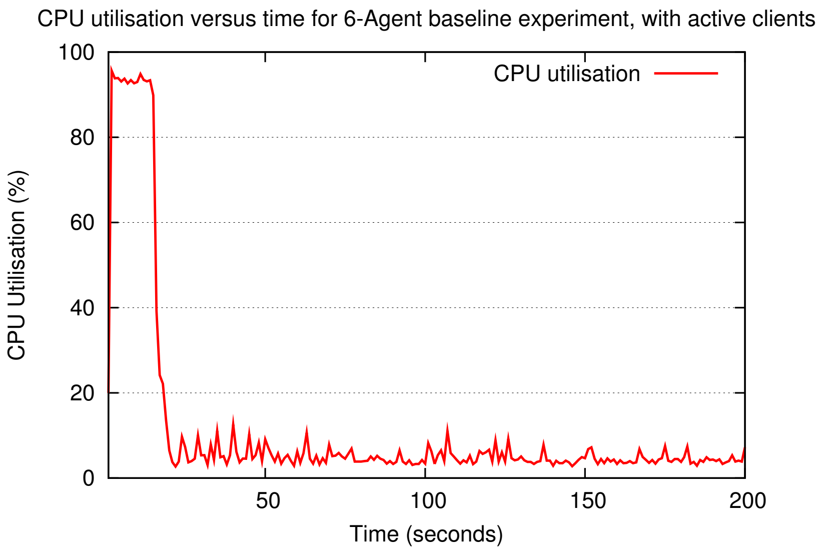

Figure 6.15: CPU overhead versus time at the master in the presence of only active clients and no applications. There is no additional overhead created at the master when there are no subscriptions requested by any application and the clients are transmitting data flows. This is because data flows are handled entirely at the agent.

and control traffic overhead is measured using `tcpdump` and `tshark`.

Discussion: Figure 6.15 represents the CPU utilisation at the master during the course of the experiment. As can be seen from curve, there is no increase in overhead when the clients begin transmitting after the 50^{th} second. Figure 6.16 indicates the memory utilisation which, like in previous experiments, remains stable. Lastly, Figure 6.17 shows the control traffic overhead measured at the master. The periodic spikes are caused by OpenFlow messages exchanged between the controller and the switch, which include `HELLO` messages, and Floodlight's forwarding application which keeps routes on the switches up to date.

6.4.4 Master Overhead: Mobility Manager Application

Objective: In this section, we observe the performance overhead of an example reactive mobility manager application that runs on Odin. The code for the application is described in Appendix A.1 and Section 5.7.1. Note that the effectiveness of the application follows partly from the smooth handoff capability that was evaluated in Section 6.2. The other aspect is the accuracy of the metric used by the application to detect mobility of a client device. We do not evaluate the latter as it is outside the scope of this thesis.

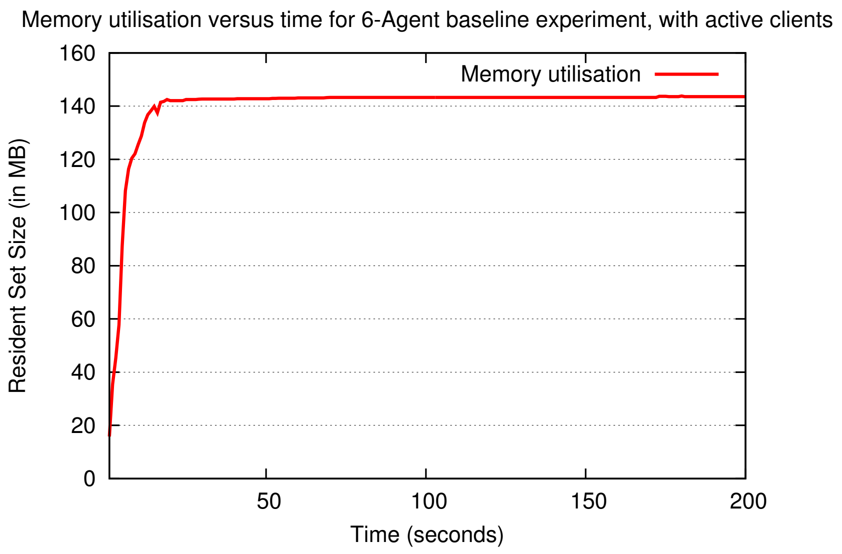

Figure 6.16: Memory utilisation versus time at the master when there are only active clients in the network and no applications. The master does not consume more memory since it does keep track of any state with regards to data flows.

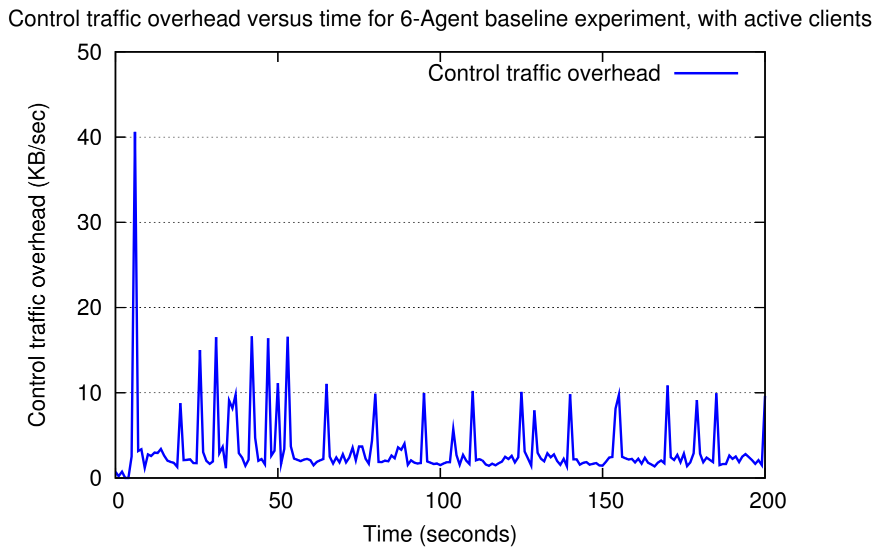

Figure 6.17: Control traffic overhead versus time at the master when there are only active clients in the network and no application. Except for periodic spikes resulting from Floodlight's Forwarding application and StaticFlowPusher component, there is no continuous flow of control traffic generated by the agent when it is handling data flows without any subscriptions to match.

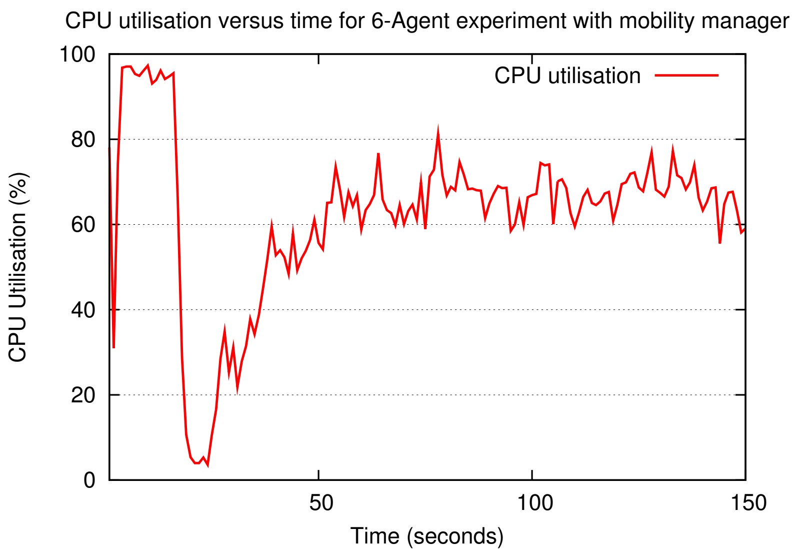

Figure 6.18: CPU overhead on the master versus time from running a mobility manager application. The load increases as the mobility manager has to track more LVAPs, and the agents trigger more subscriptions.

Experiment description: In this experiment, the same 6-AP testbed is used that is described in Section 6.1. The master is run with the mobility manager application. The APs are configured to run on channel 6 in the 2.4Ghz band, which is a noisy wireless channel in our testbed environment. Since there are many transmitting stations, this causes as many LVAPs to be assigned. The mobility manager does not distinguish between active or passive clients, and performs LVAP migrations based on receiver signal strength. This helps us simulate the load created by a mobility manager handling on the order of 250 clients (as observed in our testbed environment through our agent's statistics collection mechanism). The CPU and memory utilisation is measured using the `top` utility. Control traffic overhead is measured using `tcpdump` and `tshark`. The experiment is repeated 10 times, and the results averaged.

Discussion: Figure 6.18 shows the CPU load on the master over time. Once the master and the agents have fully booted, the system begins detecting probe scans from different stations. This leads to LVAPs being assigned. The subscription pushed down by the mobility manager also gets triggered at the agents by different stations, causing the mobility manager's handler to be invoked more often. This leads to the increased CPU usage over time (oscillating between 60% and 80%). Figure 6.19 indicates the memory utilisation over time during the experiment. The increase in memory consumption with time is a result of the information being collected

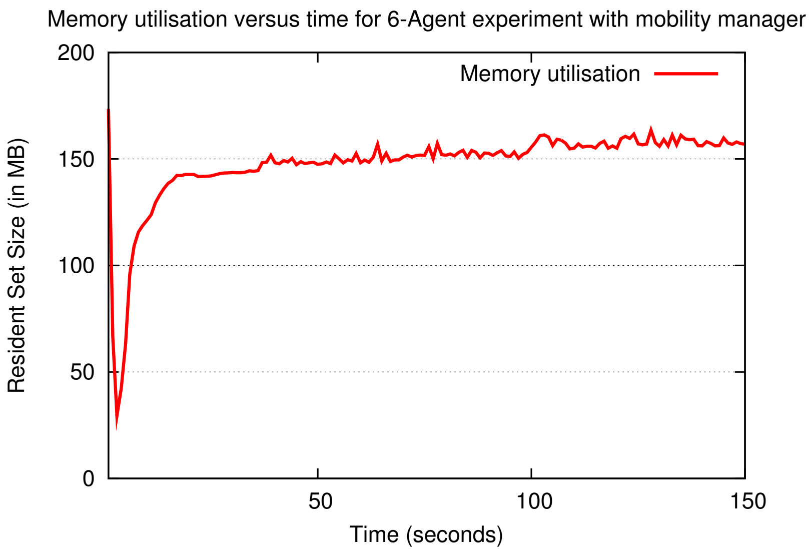

Figure 6.19: Memory utilisation of the master versus time from running a mobility manager application. The memory utilisation increases with time as the application tracks the signal strengths of an increasing number of clients.

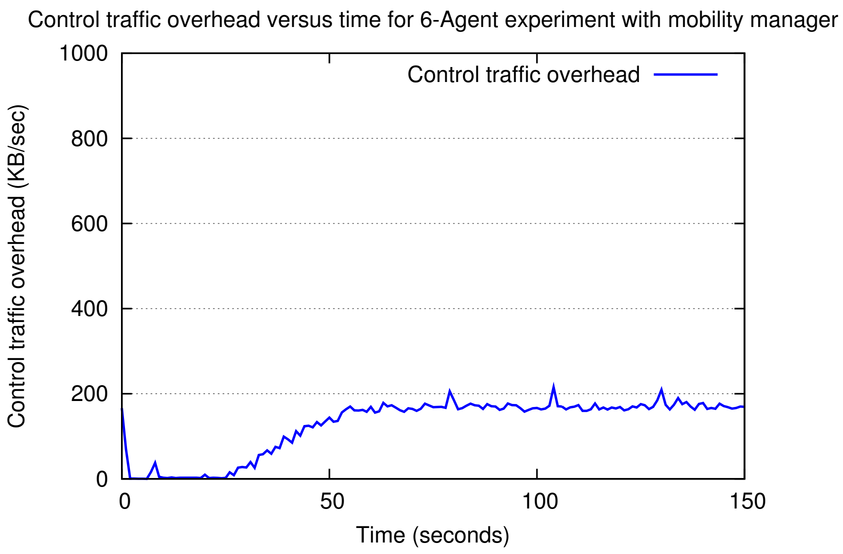

Figure 6.20: Control traffic overhead versus time from running a mobility manager application. The control traffic increases as the system tracks an increasing number of clients, and then stabilises once all of them have been tracked.

by the mobility manager. It maintains a map where it keeps track of the signal strength of the client as observed by the agent that is hosting the client's LVAP. As more stations are detected by the system, more entries are added to the map. Lastly, Figure 6.20 indicates the control traffic overhead observed at the master versus time. The number of subscriptions being triggered increase with each agent booting, leading to the increasing trend of the curve between the 25^{th} and 55^{th} seconds. Once all six agents are running, the control traffic remains stable around 180 KB/sec.

7 Conclusions

The software defined networking approach to orchestrating networks is picking up momentum. By using open interfaces and protocols to programmatically control network elements, it is becoming easier to introduce new functionalities to a network, without being constrained by vendor lock-in. In this thesis, we consider the specific case of introducing programmability to enterprise WLANs. Through the Odin software defined networking framework, we have demonstrated that by using a set of abstractions, it is possible to express common enterprise WLAN services as network applications. This property avoids the need for large, monolithic systems that are difficult to extend. It empowers the network operator with flexibility, as opposed to being restricted by proprietary and closed-source platforms.

7.1 Discussion

This thesis explores the idea of flexible enterprise WLAN architectures wherein network services can be expressed as applications. Using this approach, it is easier to introduce new features and functionalities into such networks. We have taken a step forward in this direction by using the design outlined in Chapter 4. By virtualising the association state of the AP, and introducing the concept of LVAPs, we have provided the network application developer with a simplified programming model. The programmer does not need to keep track of the association state machines at the client and authoriser. This has been achieved without any client side hardware or software modifications. The design allows for developing network applications according to both proactive and reactive programming models. The ideas described are validated through an implementation described in Chapter 5. A performance evaluation of the system (Chapter 6) demonstrates the practicality of the system within the context of an enterprise WLAN setting at T-Labs. It has been tested on a live network using real clients and access points.

7.2 On-going Work and Future Directions

There are many possible extensions for Odin. We are currently implementing support for WPA2 Enterprise and guest authentication. We are also investigating policy based network management using Odin, where a policy can be expressed using a domain specific language (DSL), which can be translated into a set of Odin applications. We are also exploring other applications that can be built using Odin, integration with DHCP, dynamic channel reconfigurations, and physical layer virtualisation by slicing the air interface. Another extension would be to integrate Odin agents with different measurement tools, so as to facilitate techniques like hidden terminal mitigation. Lastly, we are also considering Odin within the context of home network management.

Bibliography

[1] IEEE 802.11, wireless local area networks, http://www.ieee802.org/11/, January 2011.

[2] Aruba networks enterprise WLAN solutions. http://www.arubanetworks.com/pdf/solutions/AB_ENT.pdf, June 2012.

[3] Ruckus wireless. http://www.ruckuswireless.com/enterprises/enterprise-wlans, June 2012.

[4] Motorolla wireless LAN solutions. http://www.motorola.com/Business/US-EN/Business+Product+and+Services/Wireless+LAN, June 2012.

[5] Meru networks. http://www.merunetworks.com/index.html, June 2012.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[7] 802.11-2007. Technical report, 2007.

[8] hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS authenticator. http://w1.fi/hostapd/, January 2012.

[9] Lightweight access point protocol. http://tools.ietf.org/html/rfc5412, March 2012.

[10] Amin Tootoonchian and Yashar Ganjali. HyperFlow: a distributed control plane for OpenFlow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, page 33, Berkeley, CA, USA, 2010. USENIX Association.

[11] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martn Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105—110, July 2008.

[12] Cai Z, Cox A. L., and Ng. T. S. E. Maestro: A system for scalable OpenFlow control. tech. rep. TR10-08. Technical report, Rice University, 2010.

[13] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. *IN PROC. OSDI*, 2010.

[14] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: a scalable fault tolerant network manager. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, page 77, Berkeley, CA, USA, 2011. USENIX Association.

[15] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[16] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. page 351. ACM Press, 2010.

[17] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254265, August 2011.

[18] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, page 279291, New York, NY, USA, 2011. ACM.

[19] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. page 217. ACM Press, 2012.

[20] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets '11, page 7:17:6, New York, NY, USA, 2011. ACM.

[21] Andreas Voellmy and Paul Hudak. Nettle: taking the sting out of programming network routers. pages 235–249. Springer-Verlag, January 2011.

[22] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, page 110, New York, NY, USA, 2009. ACM.

[23] Lavanya Jose, Minlan Yu, and Jennifer Rexford. Online measurement of large traffic aggregates on commodity switches. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, page 1313, Berkeley, CA, USA, 2011. USENIX Association.

[24] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: dynamic access control for enterprise networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, page 1118, New York, NY, USA, 2009. ACM.

[25] Jeffrey R. Ballard, Ian Rae, and Aditya Akella. Extensible and scalable network monitoring using OpenSAFE. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, page 88, Berkeley, CA, USA, 2010. USENIX Association.

[26] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, page 1919, Berkeley, CA, USA, 2010. USENIX Association.

[27] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, page 1212, Berkeley, CA, USA, 2011. USENIX Association.

[28] Peter Dely, Andreas Kassler, and Nico Bayer. OpenFlow for wireless mesh networks. In *Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, page 16. IEEE, July 2011.

[29] Kok-Kiong Yap, Masayoshi Kobayashi, David Underhill, Srinivasan Seetharaman, Peyman Kazemian, and Nick McKeown. The stanford OpenRoads deployment. In *Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, WINTECH '09, page 5966, New York, NY, USA, 2009. ACM.

[30] OpenRadio project. http://snsg.stanford.edu/projects/openradio/, June 2012.

[31] Vivek Shrivastava, Nabeel Ahmed, Shravan Rayanchu, Suman Banerjee, Srinivasan Keshav, Konstantina Papagiannaki, and Arunesh Mishra. CENTAUR: realizing the full potential of centralized wlans through a hybrid data path. In *Proceedings of the 15th annual international conference on Mobile computing and networking*, MobiCom '09, page 297308, New York, NY, USA, 2009. ACM.

[32] Rohan Murty, Jitendra Padhye, Alec Wolman, and Matt Welsh. Dyson: an architecture for extensible wireless LANs. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, page 1515, Berkeley, CA, USA, 2010. USENIX Association.

[33] Rohan Murty, Jitendra Padhye, Ranveer Chandra, Alec Wolman, and Brian Zill. Designing high performance enterprise Wi-Fi networks. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 7388, Berkeley, CA, USA, 2008. USENIX Association.

[34] Paramvir Bahl, Ranveer Chandra, Jitendra Padhye, Lenin Ravindranath, Manpreet Singh, Alec Wolman, and Brian Zill. Enhancing the security of corporate Wi-Fi networks using DAIR. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, MobiSys '06, page 114, New York, NY, USA, 2006. ACM.

[35] E. Rozner, Y. Mehta, A. Akella, and Lili Qiu. Traffic-Aware channel assignment in enterprise wireless LANs. In *IEEE International Conference on Network Protocols, 2007. ICNP 2007*, pages 133–143. IEEE, October 2007.

[36] R. Chandra and P. Bahl. MultiNet: connecting to multiple IEEE 802.11 networks using a single wireless card. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 882– 893 vol.2. IEEE, March 2004.

[37] Hamed Soroush, Peter Gilbert, Nilanjan Banerjee, Brian Neil Levine, Mark Corner, and Landon Cox. Concurrent Wi-Fi for mobile users: analysis and measurements. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, page 4:14:12, New York, NY, USA, 2011. ACM.

[38] Gregory Smith, Anmol Chaturvedi, Arunesh Mishra, and Suman Banerjee. Wireless virtualization on commodity 802.11 hardware. In *Proceedings of the second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, WinTECH '07, page 7582, New York, NY, USA, 2007. ACM.

[39] Y. Al-Hazmi and H. de Meer. Virtualization of 802.11 interfaces for wireless mesh networks. In *2011 Eighth International Conference on Wireless On-Demand Network Systems and Services (WONS)*, pages 44–51. IEEE, January 2011.

[40] Arunesh Mishra, Minho Shin, and William Arbaugh. An empirical analysis of the IEEE 802.11 MAC layer handoff process. *ACM SIGCOMM Computer Communication Review*, 33(2):93, April 2003.

[41] H. Velayos and G. Karlsson. Techniques to reduce the IEEE 802.11b handoff time. In *2004 IEEE International Conference on Communications*, volume 7, pages 3844– 3848 Vol.7. IEEE, June 2004.

[42] Jon-olov Vatn. An experimental study of IEEE 802.11b handover performance and its effect on voice traffic. 2003.

[43] Ramya Raghavendra, Elizabeth M. Belding, Konstantina Papagiannaki, and Kevin C. Almeroth. Understanding handoffs in large ieee 802.11 wireless networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, page 333338, New York, NY, USA, 2007. ACM.

[44] Y. Bejerano, I. Cidon, and J. Naor. Efficient handoff rerouting algorithms: a competitive on-line algorithmic approach. *IEEE/ACM Transactions on Networking*, 10(6):749– 760, December 2002.

[45] C. -F Chiasserini and R. Lo Cigno. Handovers in wireless ATM networks: in-band signaling protocols and performance analysis. *IEEE Transactions on Wireless Communications*, 1(1):87–100, January 2002.

[46] I. Papanikos and M. Logothetis. A study on dynamic load balance for IEEE 802.11b wireless LAN. In *Proc. COMCON*, 2001.

[47] I. Ramani and S. Savage. SyncScan: practical fast handoff for 802.11 infrastructure networks. In *Proceedings IEEE INFOCOM 2005. 24th Annual Joint Conference of the IEEE*

Computer and Communications Societies, volume 1, pages 675– 684 vol. 1. IEEE, March 2005.

[48] Yair Amir, Claudiu Danilov, Michael Hilsdale, Raluca Musloiu-Elefteri, and Nilo Rivera. Fast handoff for seamless wireless mesh networks. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, MobiSys '06, page 8395, New York, NY, USA, 2006. ACM.

[49] Pedro Estrela. *Transparent and Efficient IP mobility.* PhD thesis, Universidade Tecnica de Lisboa, Lisbon, December 2007.

[50] IEEE 802.1X, http://standards.ieee.org/getieee802/download/802.1X-2010.pdf, January 24th, 2011.

[51] IEEE 802.21, Media Independent Handover, http://www.ieee802.org/21/, January 13th, 2011.

[52] Cisco Systems Inc. Data sheet for cisco aironet 1200 series, 2004.

[53] Y. Bejerano and Seung-Jae Han. Cell breathing techniques for load balancing in wireless LANs. *IEEE Transactions on Mobile Computing*, 8(6):735–749, June 2009.

[54] 802.11k-2008 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 amendment 1: Radio resource measurement of wireless lans, March 2012.

[55] 802.11r-2008, 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 amendment 2: Fast basic service set (BSS), March 2012.

[56] CAPWAP protocol binding for IEEE 802.11. http://www.ietf.org/rfc/rfc5416.txt, March 2012.

[57] Floodlight. http://floodlight.openflowhub.org/, March 2012.

[58] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263297, August 2000.

[59] Radiotap. http://www.radiotap.org/, August 2011.

[60] Open vSwitch. http://openvswitch.org/, June 2012.

[61] ath9k linux wireless driver. http://linuxwireless.org/en/users/Drivers/ath9k, February 2012.

[62] OpenWRT. https://openwrt.org/, February 2012.

[63] Debugfs. http://lwn.net/Articles/334546/, May 2012.

[64] Linux HA project. http://www.linux-ha.org/wiki/Main_Page, May 2012.

[65] Kok-Kiong Yap, Yiannis Yiakoumis, Masayoshi Kobayashi, Sachin Katti, Guru Parulkar, and Nick McKeown. Separating authentication, access and accounting: A case study with OpenWiFi. July 2011.

[66] OpenID authentication 2.0 specification. http://openid.net/specs/openid\-authentication\-2_0.html, March 2012.

[67] D. Giustiniano, D. Malone, D.J. Leith, and K. Papagiannaki. Measuring transmission opportunities in 802.11 links. *Networking, IEEE/ACM Transactions on*, 18(5):1516 –1529, October 2010.

[68] Sangho Shin, G. Forte, Anshuman Singh Rawat, and Henning Schulzrinne. Reducing MAC layer handoff latency in IEEE 802.11 wireless LANs. page 19. ACM Press, 2004.

[69] Antonin Guttman. R-trees. *ACM SIGMOD Record*, 14(2):47, June 1984.

[70] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

Appendix

A.1 Example Reactive Mobility Manager

```
public class OdinMobilityManager extends OdinApplication {

  private ConcurrentMap<MACAddress, Long> clientCurrentSignalMap = new
                                    ConcurrentHashMap<MACAddress, Long> ();

  ...
  ...
  /* register subscription ["*", "signal", >, 200] */
  ...
  ...

  private void handler (OdinEventSubscription oes,
                        NotificationCallbackContext cntx) {

    /* Check to see if this is an associated client */
    OdinClient client = odinMaster.getClients().get(cntx.staHwAddress);
    if (client != null) {

        /* The agent that triggered this handler is the one
         * hosting the client's LVAP */
        if (client.getOdinAgent() != null
            && client.getOdinAgent().getIpAddress()
                == cntx.agent.getIpAddress()) {

            clientCurrentSignalMap.put(cntx.staHwAddress, cntx.value);
        }
        /* The subscription was triggered at an agent other than the
           client's LVAP. If the signal strength is 10 units higher
           at the new agent than at the client's current one, then
           perform an LVAP handoff */
        else if ((client.getOdinAgent() == null)
                   || (client.getOdinAgent().getIpAddress()
                       != cntx.agent.getIpAddress()
                       && clientCurrentSignalMap.get(cntx.staHwAddress)
                           + 10 < cntx.value)) {

            odinMaster.handoffClientToAp(cntx.staHwAddress,
                                         cntx.agent.getIpAddress());
            clientCurrentSignalMap.put(cntx.staHwAddress, cntx.value);
        }
    }
}
```

Figure A.1: Java code for a simple reactive mobility manager. The application LVAP-handoffs the client to the AP where the client has the best received signal strength.