Hoolock : Seamless Mobility in OpenFlow-enabled Wireless Access Networks

Nikhil Handigol and Wei Wei
Stanford University

December 12, 2008

1 Introduction

The widespread deployment of wireless technologies such as Wi-Fi and Wi-Max in today's networks has resulted in the emergence of an increasing number of mobile hand-held devices such as mobile phones and PDAs capable of data processing and communication. It is expected that the number of such devices will only increase in the future. However, Voice-over IP (VoIP) and other real-time streaming applications pose a serious challenge in achieving true mobility in Wi-Fi networks. This is partially because of the high handoff latencies [1] and the resultant severe data loss experienced by mobile users while moving from one access point (AP) to another. Our goal in this work is to develop a system to eliminate handoff latency and data loss during handoff by exploiting the potential of multiple radios in mobile devices. We develop a novel hand-over scheme, **Hoolock**, that achieves seamless mobility without data loss.

Previous works that have proposed the idea of using two radios [2], achieve only the partial goal of zero handoff latency. However, because of a lack of complete control over flows within the network, they fail to provide a solution to eliminate the loss of downstream packets during handoff. OpenFlow [3] provides us with this capability, and we exploit it to achieve lossless hand-over.

2 System Design

Our system consists of a mobile client with two Wi-Fi radio interfaces. As Figure 1 shows, when the mobile client moves away from AP-1 towards AP-2, the quality of the wireless link between the client and AP-1 degrades and that of the link between the client and AP-2 improves.

In order to accomplish lossless hand-over, we need a way to reroute the ongoing flows as the client moves from one AP to another. Hoolock achieves this using the NOX controller.

Traditional mobile systems with a single radio interface suffer significant degradation in performance during hard handoff from one AP to another. However, the configuration with two radio interfaces provides the possibility of seamless hand-over between APs.

Hoolock operates based on a 4-stage protocol, which is described below:

- **MAKE_REQ** : The client is using one radio (*radio-1*) for transmission, while using the other radio (*radio-2*) to monitor the quality of the available wireless links (eg. Signal to Noise Ratio) to the surrounding APs. When the link quality degrades below a certain threshold, the client issues a MAKE_REQ to the NOX controller.

- **MAKE_ACK** : The NOX controller prepares itself for a *flow-in* event from the client, and acknowledges the same to the client using MAKE_ACK.

- **BREAK_REQ** : The client connects to the new AP using *radio-2*, and sends all its outgoing traffic over it. However, it continues to receive packets from the old AP over *radio-1*. As the first outgoing

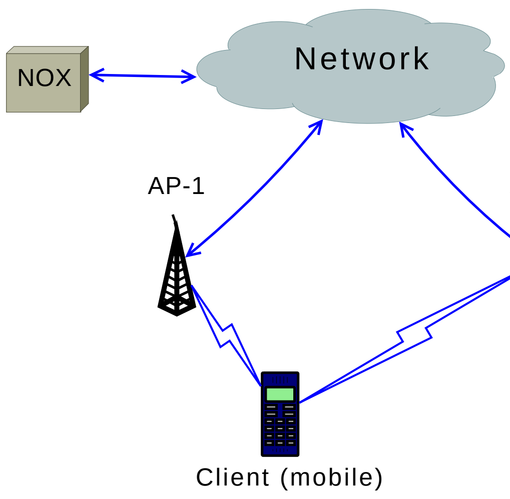

Figure 1: OpenFlow network with mobile client.

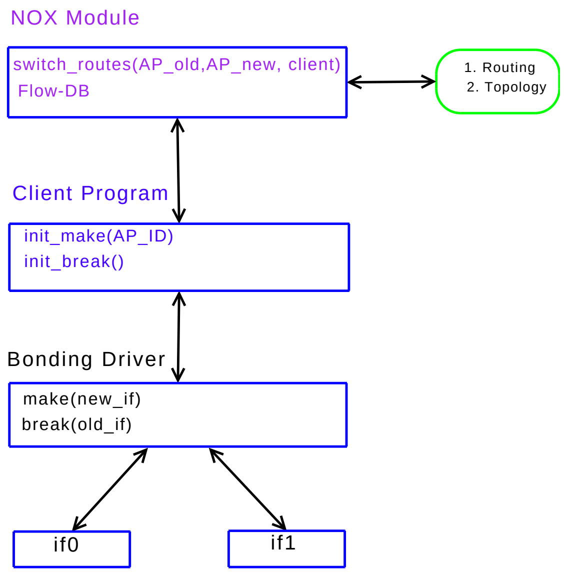

Figure 2: Hoolock system design.

packet reaches the new AP, it triggers a flow-in event at the NOX controller. The NOX controller, now, looks up its flow-database, computes the common root-switches and reroutes all the flows destined to the client to go over the new AP. Simultaneously, the client issues a BREAK_REQ to NOX.

- **BREAK_ACK** : After the controller gets a BREAK_REQ message from the client, it waits for a predefined *flush-time* (which equals RTT/2), so that all the packets remaining on the old route reach the client. It then sends the BREAK_ACK message to the client, upon the receipt of which, the client dissociates *radio-1* from the old AP and uses it for background scanning.

3 Implementation

Figure 2 shows the design of the Hoolock system. It consists of three main parts - the bonding driver, the NOX module and the client program.

The bonding driver is a Linux kernel module that provides a single virtual interface for the upper layers to send packets to and receive packets from. It transparently manages the two physical wireless interfaces (called *slaves*) to behave as required by the protocol described earlier. It allows the client program to communicate to it using ioctl function calls, and provides two major functions:

- **make()** - Start transmitting over the new slave while still listening on the old slave.
- **break()** - Stop listening on the old slave.

In addition to the above functions, the bonding driver also intelligently avoids any possible packet reordering due to rerouting of flows (especially, if the new route has a lower latency than the old one). It does so by buffering packets received on the new slave while in the make() mode and delivering them up when break() is called.

The NOX controller maintains a database of the currently active flows and their corresponding routes within the network. It uses the Topology and Routing modules to route and switch flows. Upon the receipt of a flow-in event from a client in transition, it calculates the corresponding route to the destination, and uses it to calculate a corresponding *reverse-route*. It then looks up its database for all the routes with the client as the destination and matches them with the reverse-route to calculate the common *root-switch*. It then, reroutes all the flows destined to the client from root-switch onwards to follow the reverse-route, while updating the corresponding database entries.

The client program is the core module that coordinates between the bonding driver and the NOX controller. It communicates with bonding driver via the ioctl() function and interacts with NOX over a TCP channel to execute the four-stage protocol described in Section 2.

4 Performance Evaluation

4.1 Methodology

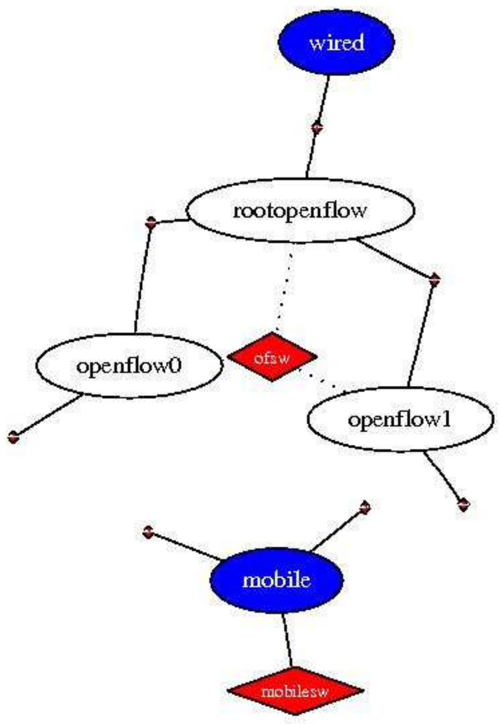

Figure 3: Simulation topology for evaluation of Hoolock.

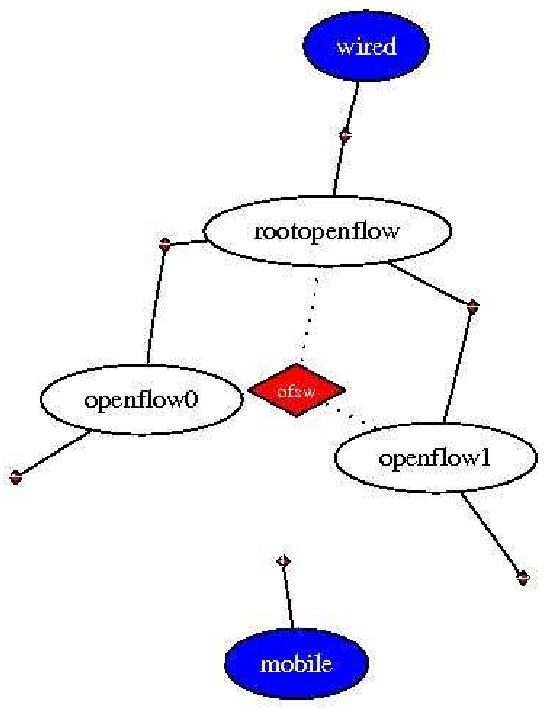

Figure 4: Simulation topology for evaluation of hard handoff.

We use Virtual Distributed Ethernet (VDE) to evaluate the performance of Hoolock, and compare it with that of hard handoff and the base-case of no handoff. We assume hard handoff requires approximately 150 ms to scan the available channels before associating with a new AP. Figures 3 and 4 show the topologies of the networks used in the simulation of Hoolock and hard-handoff respectively. The topology consists of three OpenFlow switches - $rootopenflow$, $openflow0$, and $openflow1$ and two clients - $wired$ and $mobile$.

We simulate the association and dissociation of *mobile* with the OpenFlow switches by creating and destroying pipes. We use wirefilter to simulate the link quality parameters such as delay, loss, and loss-burst.

We evaluate the performance of various handoff schemes by transferring UDP data from *wired* to *mobile* using iperf for 20 seconds, and various rates, and initiating a handoff roughly half way through the transfer. We use the loss rate reported by iperf as the performance metric.

4.2 Results

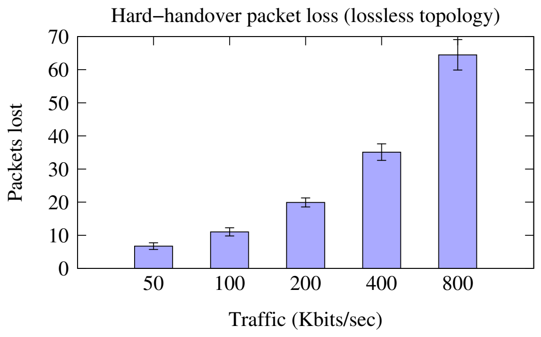

Figure 5: Hard-handover packet loss in a lossless topology.

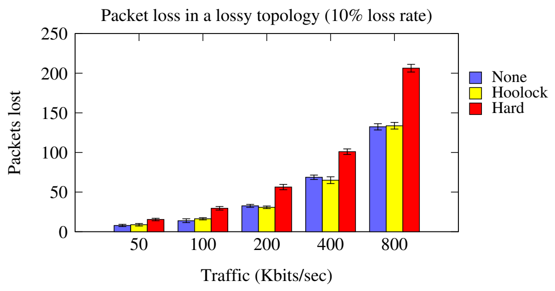

Figure 6: Packet loss in a lossy topology.

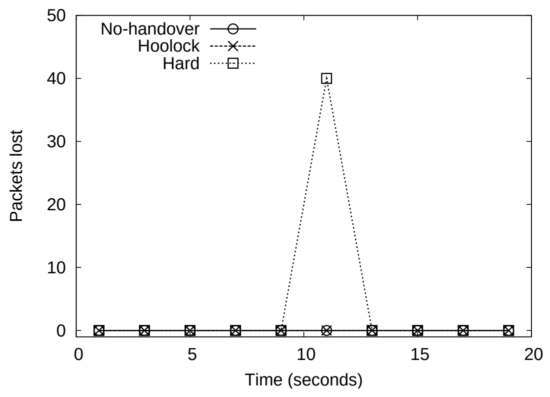

Figure 7: Packet loss in a lossless topology.

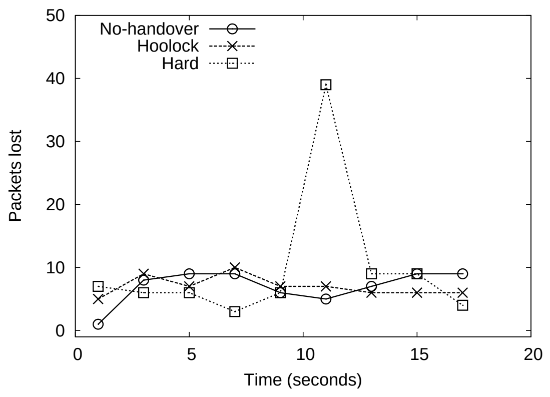

Figure 8: Packet loss in a lossy topology.

Figure 5 shows the number of packets lost during a hard handoff for various traffic rates in a lossless topology. We observe that the number of packets lost increases with increasing traffic rate. This is due to the packets that get dropped during the time period between the dissociation and association of *mobile* with an AP. The loss experienced by Hoolock was consistently zero. Hence, we do not plot those results.

Figure 7 shows a typical variation of packet loss during various 2-second intervals with time. While Hoolock and no-handoff experience zero loss consistently, hard-handoff experiences a large spike during the handoff period.

Figure 6 shows the number of packets lost by various handoff schemes in a topology where the last-hop link has a loss rate of 10% and a loss-burst of 1.5. We can observe that the loss experienced by Hoolock is statistically equivalent to that experienced when there is no handoff, which is due to the loss induced by lossy links. On the other hand, hard-handoff experiences a much larger data loss. This result can be better understood by the time-series graph in 8, which shows that Hoolock performs as well as no-handoff, while hard-handoff experiences a spike in packet loss during the handoff period.

Thus, Hoolock enables truly seamless mobility and provides data transfer equivalent to continuous transmission without handover.

5 Lessons Learned

- **Locking and memory issues in the bonding driver**
  The Linux kernel is a very complex system that needs to support multiple concurrent processes with a limited physical memory. These aspects became very clear as we developed the bonding driver code to support communication over multiple physical interfaces. There were two problems, in particular, that gave us a hard time, but provided invaluable lessons at the end of the day. One was the intricate locking mechanism used in the Linux kernel - lock well to keep your data safe. The other was the memory management - if you don't manage your memory well, you will run out of it!

- **Limitations of a virtual simulation environment**
  The VDE is a fantastic system to develop and debug a network application. However, when it comes to getting realistic results, it does not compare well with the real world. One major reason for this is the amount of processor and memory load that the single host machine takes to support all the VMs of the network. The exact order in which various processes are scheduled becomes very critical and can cause unexpected results.

- **Bugs/Hacks in NOX**
  As we developed our NOX application, we found two major bugs/hacks in NOX that were preventing us from successful implementation. One is the route installation mechanism. The NOX routing module adds flow table entries for a new flow in order along the route. This can lead to unnecessary flow-in events, especially on a slow system like VDE. A more correct solution would be to add the flow table entries in the reverse direction of the route. The second one is "route poisoning" - a mechanism in NOX which erases old flow table entries when a host joins a new switch. We had to disable route poisoning in order to achieve communication over multiple paths.

6 Conclusions and Future Work

Hoolock enables seamless mobility in Wi-Fi networks by providing a mechanism for lossless handoffs with zero latency. Hoolock accomplishes this by using two radios - scanning the available channels with one while maintaining an active connection over the other. Thus, when a hand-over is needed, it can do so immediately. It has a superior performance compared to that of hard-handoff in terms of data loss during handoff.

The performance of Hoolock has been evaluated using a simulation environment. In the real world, the wireless links are influenced by various factors, like multipath fading, Doppler shift and shadowing effects. In addition, the performance of the NOX controller, the client program and the bonding driver will be influenced by other system-wide factors. Our future work will consist of deployment of Hoolock on a real campus-wide network and evaluation of its performance.

The Open-Flow infrastructure exhibits its power in our work. It has many advantages over the rigid, traditional networks. The major difference is its capability to manage the flow table entries in each switch, which provides administrators a fine-grained control over the network traffic. The handoff application is just the tip of the iceberg - Open-Flow and its applications have a huge potential in defining the future Internet.

References

[1] A. Mishra, M. Shin, and W. Arbaugh, "An empirical analysis of the ieee 802.11 mac layer handoff process," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 93–102, 2003.

[2] V. Brik, A. Mishra, and S. Banerjee, "Eliminating handoff latencies in 802.11 wlans using multiple radios: applications, experience, and evaluation," in *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2005.

[3] "OpenFlow." Available: http://www.openflowswitch.org.