



FINAL REPORT

05/07/2010

Andrew Arminio

Christopher Austin

James McCauley

1 TABLE OF CONTENTS

2	Introduction	3
3	Problem Statement.....	3
4	Process overview and Team Organization.....	5
4.1	Project management	5
5	Requirements.....	6
5.1	Acquisition	6
5.2	Functional Requirements.....	6
5.3	Constraints	8
6	Solution	8
6.1	Decision Making Architecture	8
6.2	User Interface Architecture	9
7	Future Work.....	11
8	Conclusion.....	11

2 INTRODUCTION

While home and small office/home office networks have provided any number of benefits - internet access, media sharing, shared printers, etc. -- they seldom live up to their potential. Many such networks are subject to security problems, some of which allow attackers access, and some which negatively impact usability by providing too much or the wrong security. These networks also often simply do not provide for as good of a user experience as possible, for example because bandwidth provisioning in such networks is simple-minded.

While our client, Nicira Networks, focuses on large networks such as enterprise and datacenter networks, the technologies they use and develop could serve to iron out the rough spots of home/SOHO networks, and allow for whole new classes of home network capabilities. It has been the task of Team NOX and the NOX At Home project to help make this a reality.

3 PROBLEM STATEMENT

In 1981, the seminal paper "End-to-End Arguments in System Design" by Saltzer et al. famously articulated a design principle that had already been key to the design of communication networks for quite some time. The case was presented that many network features needed to extend all the way from one endpoint to the other -- from a sending application to the receiving application -- and that given that requirement, the correct placement of those features was in the endpoints and not in the underlying communication system. In essence, this espouses a model that is often summed up as "dumb network; smart hosts." There is no debate that the arguments of the paper are valid and that the principle is a solid one. However, not all design elements are best served by this principle and this principle alone. Although Saltzer and his coauthors address this in their paper, the principle itself and the "dumb network; smart hosts" model essentially became network design dogma.

A dumb network is not without its problems, and there are bona fide reasons for wishing to build smarter networks -- to put functionality into the network itself. Corporate networks, for example, are often subject to strict reliability and security constraints which do not fit well with the IP's "best effort" approach. To resolve this, there has been considerable effort to make corporate networks more manageable, essentially by "bolting on" features (firewalls, VLAN control, audit logs, etc.). While these may provide tools for management, one might summarize these developments as "working harder, not smarter." One reason for the lack of smarter networks is simply that adding any real "brains" to a network has posed at least one question that has been difficult to answer: where do these brains *go*? There would seem to be two basic approaches -- a distributed one or a centralized one -- and neither of these has been incredibly appealing. The distributed approach is difficult at best simply due to the inherent difficulty of distributed solutions for the complex and

variable nature of network management. In the case of internetworks, this approach would often be entirely impossible because administrative boundaries between the owners of different networks. On the other hand, centralized approaches have long been decried in the networking world. There are (justified) concerns about having a single point of failure and about scalability.

Challenges aside, there were real problems to be solved for which adding functionality to the network seemed to be key to the solution, and this spurred a string of research at Stanford University. The first of this research was SANE; a clean-slate approach that strived to make enterprise networks more secure and to make that security easier to manage. This was followed by Ethane, a major refinement that, significantly, was not a clean-slate approach; it was incrementally deployable in existing networks. Finally, this brings us to NOX and OpenFlow. NOX and OpenFlow correspond to the two major components of Ethane, but where Ethane was academic and aimed at one specific goal (security in enterprise level networks), NOX and OpenFlow are real solutions that have been generalized for a multitude of purposes, including ones that are, as yet, undefined. NOX and OpenFlow work hand in hand with the aim of separating low-level packet forwarding from high-level decision making. Said another way, they make the network itself programmable, and this enables smart networks.

OpenFlow is the part of this system that manages the low-level packet forwarding, and is an open standard governed by the OpenFlow Switch Consortium. This standard defines an interface to control an abstract *flow table* in a network switch. Through manipulation of this flow table, one can inspect network traffic, control how packets are forwarded (or not forwarded!), and alter the traffic in other ways (such as packet header rewriting and QoS management). Hardware routers implementing OpenFlow exist today, and there are also software implementations that transform a Linux machine into an OpenFlow switch. OpenFlow, which received the Best Demo award at SIGCOMM 2008, reached version 1.0 at the end of 2009. This release was a major milestone, representing the first version of the standard intended for general availability in commercial products.

By itself, an OpenFlow switch does nothing -- it requires a controller to make high-level decisions. By far, the most predominant of these is NOX. NOX is an open source "operating system" for networks that provides a centralized programming interface to the entire network via its ability to control OpenFlow switches. It should be noted that while NOX is *logically* centralized, it need not actually be a completely centralized system. The actual machines running NOX can be replicated, implement failover, be load-balanced, or utilize other modern datacenter designs aimed at improving reliability and scalability (which did not, for the large part, exist twenty years ago). Further enabling scalability is that much of the "grunt work" is handled by the switches. While performance will obviously be impacted by the specific functionality implemented on top of NOX, Ethane (using the same basic approach and architecture as NOX) was able to manage over 5,000 hosts from a controller running on a single commodity PC.

Our client, Nicira Networks, is the primary developer of NOX, as well as the primary developer of Open vSwitch -- a virtual network switch that, among other things, implements OpenFlow. While Nicira's focus is on applying NOX and OpenFlow technologies to large networks (campus, corporate, enterprise, datacenter, and virtual networks), they are curious as to how their technologies might also be applied to improving small networks, such as home networks and small office/home office networks. Specific examples of how NOX and OpenFlow might enhance these networks include things such as:

- Traffic prioritization
- Isolation of guest users
- Better parental control
- Centralized network diagnostics / improved network transparency
- Password protected file sharing that works

Network control software for the home/small office on the level that one could implement using NOX and OpenFlow breaks new ground. As we brainstormed possibilities, we were all convinced that there are a fairly large number of interesting ideas in addition to the ones above, and there are probably a number of "killer apps" in this space that we have not yet discovered. It is clear that there is room here for a community of developers to explore and fill this new realm of possibilities. Our project was to facilitate this -- to build a platform on which many independent applications for home and small office networks could be built.

4 PROCESS OVERVIEW AND TEAM ORGANIZATION

At one of the first meetings of our team, we sat down and discussed our various strengths and weaknesses and how we might organize our group. The roles we assigned (outside of coding, which was to be shared amongst all of us) were that Murphy would be the technical lead, Chris would keep track of scheduling and deliverables, and Andy would take care of making sure that everything from meeting and face-to-face communication ended up in electronic form on our wiki or messageboard. Notably, however, none of us were keen to take on a management leadership position. Our naïve assumption was that getting things done would be the natural result of the fact that all of us were interested in the project and wanted to succeed in the class. In hindsight, this was probably a very poor decision, and a strong *project* leader would have been extremely beneficial.

4.1 PROJECT MANAGEMENT

Our initial project management plan for this project was to follow a modified Lean development process. Lean is an iterative process initially developed by Toyota for its manufacturing plants. The principles developed by Toyota were then adapted by the Agile development community for software development. Lean seemed to fit well with some of our goals. Specifically, some of the things that initially drew us to Lean were:

- It was an iterative and Agile process
- Its principle of “Decide Late” would help prevent us from locking ourselves into specific implementation details until those details were needed.
- The Kanban tool used by Lean development would allow us to easily track progress both for our own benefit and for reporting.

Our vision for the process was that we would begin each milestone by dividing our goals into chunks and then running each chunk through three phases: research, development, and testing. The progress of each portion of the milestone would be tracked through an online Kanban tool. At the end of each milestone we would review the process and identify bottlenecks before moving on to the next milestone.

Ultimately our process ended up being simpler (due in part to issues discussed in the team structure section). Instead of the Kanban we primarily used a team forum and wiki to facilitate keeping each other abreast of our progress. When Chris moved from a coding role to a primarily documentation role, Murphy and Andy switched primarily to informal status meetings once or twice a week in person or over IM. These informal status meetings were used to discuss work completed and direction for the next week. This setup worked because of the small size of the development team and the fact that the two developers were working on orthogonal portions of the project; Murphy was working on the NOX side of the platform while Andy was working on the Qooxdoo side.

5 REQUIREMENTS

5.1 ACQUISITION

This project began while Murphy was an intern at Nicria, and Murphy has been involved in it from close to its conception. As such, Murphy was present at many of the early discussions of this venture, well before it became a capstone project. Many of these decisions were along the lines of brainstorming uses for NOX in the home, and our group continued these brainstorming sessions. Given a handful of resultant ideas, forming a list of requirements for the platform was relatively simple: find the common ground -- the features that all (or many) such uses would require.

5.2 FUNCTIONAL REQUIREMENTS

The motivations for and principles driving our project translate to a number of specific requirements and an associated functional specification.

Give policy application writers access to high level names for policy configuration

Policy writers must have access to mappings that associate low level network addresses to high level names for users. To enable this, our platform needs a module to facilitate:

- Adding and removing users
- Associating users with low-level network addresses
- Adding attributes to users for use by policy applications

Identify users on a network using passive user identification

NOX At Home is meant to be used largely, as the name would suggest, at home. For this reason, we do not think a corporate-style login is acceptable. We believe home users would find it intrusive and unpleasant. For this reason, we wish to focus on *passive user identification*. Specifically, it must be possible to write a module that will:

- Intercept any network traffic with specific criteria.
- Look at intercepted network traffic for evidence that it belongs to a specific user.
- Identify where (IP, MAC, previously identified machine) a user is active.
- Alert the rest of the system that a user is known to be or suspected to be active. In cases where a user is only suspected to be active, information about the likelihood must be provided.
- Identify new evidence sources and incorporate them into the system.

Maintain a view of the current network state

The system should be able to aggregate the messages from various evidence sources and make a decision about their meaning. These decisions should be recorded as a coherent and comprehensive view of what users are active on the network at any given time. To support this, the software must provide the following functionality:

- Aggregate and store messages from evidence applications.
- Analyze such evidence messages and make a decision on where and if a user is active on the network.
- Store decisions about active users on the network as the network state.
- Provide network state information to policy applications.
- Provide network state information to administrators using a web interface.

Integrate arbitrary network policies

The system needs to provide a means for independently developed network policy applications to run. They should have the ability to access network state decisions from the network view and the ability to be configured from a unified interface. Our system must provide the following functionality to support such policy applications:

- Identify new policy applications and incorporate them into the system.
- Have a means for such applications to retrieve information from the network view.
- Have a means for such applications to know what users are set up on the system.

Provide for integrated user interfaces

The system needs to provide a means for developers of network policy and evidence gathering applications to build user interfaces that can be accessed through a web browser. The importance of the user interface will likely vary widely depending on the specific developer and their application, and this must be confronted. Specifically, we must:

- Provide a means for creating integrated/homogeneous UIs
- Provide an “easy path” for developers who are uninterested in building UIs, such that creation of a basic UI with minimal effort is possible
- Provide developers who are interested in creating a rich UI for their application a means of doing so
- Streamline the communication between UIs and NOX applications

5.3 CONSTRAINTS

The constraints on our design were relatively straightforward. First and foremost -- our design needed to exist as a NOX application or applications. This effectively limited our choice of implementation languages to C++ and Python, effectively limited our choice of web server for the UI to Twisted Web, etc. We also needed to design around the eventual target hardware platform -- a PC Engines alix2c3 or alix2d3. As these are small, single-board computers that are not particularly heavy in either CPU power or RAM (and a virtual memory page file is impractical due to the storage available), we needed to avoid particularly CPU-intensive and memory-intensive designs. Lastly, as NOX At Home is eventually to be open source, all outside components were required to use compatible licenses.

6 SOLUTION

6.1 DECISION MAKING ARCHITECTURE

The decision making architecture for NOX At Home consists largely of independent or semi-independent NOX applications in one of two categories: Evidence Sources or Policies (or Decision Making Applications). We facilitated our goal of flexible “plug-in” style extensibility in large part by simply working within the standard NOX model -- peices are implemented as independent NOX Component subclasses which communicate largely through the NOX event system instead of direct calls. There are also a number of lesser support classes (such as the User Manager, a boilerplate-reduced version of the NOX Component baseclass, and a packet classifier capable of finding and prioritizing multiple matches) which are not enumerated here.

Evidence sources are NOX applications that are responsible for enabling passive user identification as previously outlined. The evidence about users that these modules provide

is to be interpreted and used to enable policies to make decisions based on user identity. Our current implementation is able to track users logging in and out of Facebook (the latter not being particularly significant, as users seldom explicitly log out of Facebook), as well as tracking Windows / MSN Messenger sessions. This is done by installing flow rules to monitor traffic. While the MSN Messenger evidence source is relatively straightforward, the Facebook evidence source is somewhat more interesting. For example, DNS queries are also monitored in order to support Facebook. When a machine looks up “login.facebook.com”, the Facebook evidence source adds the resulting IP address to the list of IPs it is monitoring. This is necessary because login.facebook.com resolves to a relatively large pool of addresses. Additionally, Facebook returns compressed HTTP responses.

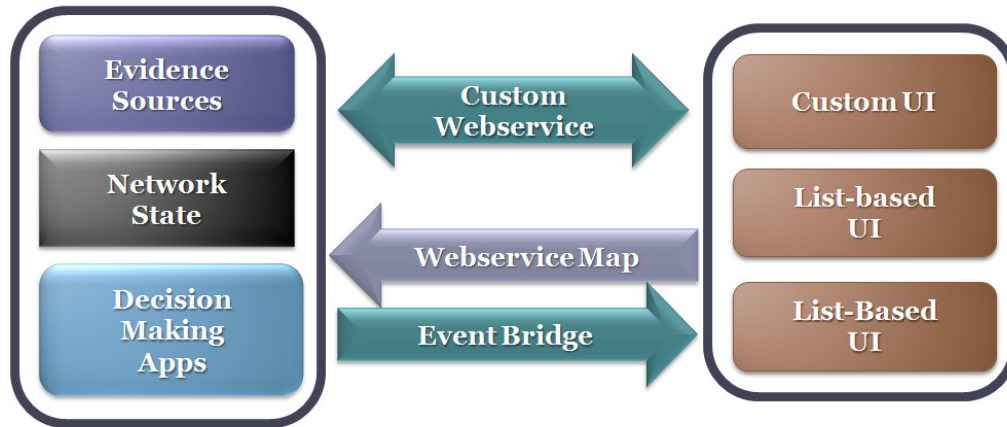
Because our target platform is not particularly powerful, we did not wish to decompress these. Instead, we monitor all outgoing HTTP requests to Facebook. If we are potentially interested in Facebook’s response (because it may contain the user’s identification, for example), we alter the HTTP request by removing gzip from the “Accept-Encoding” header, forcing Facebook to return uncompressed results.

Policies, on the other hand, are where real features are to be implemented -- things that are truly meant to improve users experience. Some of the possibilities for these were listed above: traffic prioritization (throttling long transfers while allowing streaming media and interactive web browsing sessions more than their “fair” share of bandwidth), isolation of guest users (keeping visiting friends or clients from accessing your shared folders when you allow them onto your network to check their email), better parental controls (which cannot be subverted by logging in to a different user account, for example), etc. Unfortunately, we only have prototype implementations of throttling and guest user isolation implemented at this point.

6.2 USER INTERFACE ARCHITECTURE

Our system required browser-based user interfaces, and the flexibility to support both rich user interfaces for developers to whom this was a priority, but also an “easy path” to allow developers with less interest in user interfaces to still produce some sort of usable GUI. We also required the ability to integrate user interfaces from multiple independently developed applications into a single “container” UI. We decided to use the Qooxdoo web framework to facilitate this, as it has a fairly clean API, a rich widget set, and a liberal license.

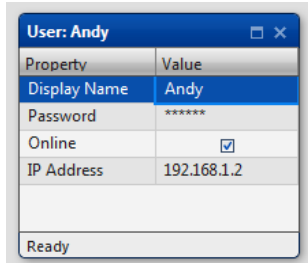
To further our requirement that development of a user interface not be an undue burden, we developed three main components. Of these, the Event Bridge and WebService Mapping are concerned with reducing the complexity of communicating between NOX applications and their respective browser-based user interfaces. The third piece allows for the simple generation of list-based user interfaces.



The Event Bridge provides a system by which the browser can “listen” directly to NOX events, by registering a handler in much the same way as one can do from within NOX itself. The event objects are packaged up by the Event Bridge, and exposed through a web service to a component running in the browser which retrieves and then dispatches them. There are two caveats here. Firstly, this functionality only works automatically with Python-based events. C++-based events do not have the run time type information available to let us automatically convert them into a form usable by JavaScript. However, C++-based events can be handled specially and converted to a JavaScript-friendly form manually in much the same manner that some of NOX’s built-in events are converted into a Python-friendly form. It should be noted, however, that all of the NOX At Home code is Python to begin with, so NOX At Home-specific events can be translated automatically. The second caveat is that events on the browser are advisory-only. Whereas an event handler running within NOX can stop further processing of the event or alter the data associated with the event, events passed through the Event Bridge can do neither of these. Nevertheless, the Event Bridge allows a user interface to monitor and respond to events without the need to explicitly record, translate, and expose them through a web service, and then having to retrieve and dispatch them in the browser. It is a single mechanism that handles all of this.

The WebService Mapping is another NOX component which radically reduces the amount of code required to build user interfaces. Essentially, the WebService Mapping uses Python’s rich run time type information to automatically turn arbitrary Python objects (indeed, entire Python object hierarchies) into web services that are then exposed through NOX’s web service component. The mapping is generally quite obvious. By way of example, if some object `foo` has some field `bar`, and `bar` is an object with a method `baz`, where `baz` takes a string parameter and returns the same string but in uppercase, one might call this from Python using code such as: `foo.bar.baz("helloworld")`. If the `foo` object were added to the WebService Mapping, this would expose that method as a web service, allowing you to access it through a GET request to a URL along the lines of `http://noxbox/foo/bar/baz/helloworld` (the result of which would be a response containing “HELLOWORLD”). The URLs allowed are quite flexible, allowing for a number of

method call and parameter passing schemes. While the results may not provide the elegance of a well-designed REST interface, a real strength of this system is that Python objects used in the implementation of a policy or evidence source can often be exposed to their user interface without writing any special code at all aside from simply adding them to the WebService Mapping's root.



Property	Value
Display Name	Andy
Password	*****
Online	<input checked="" type="checkbox"/>
IP Address	192.168.1.2

The final of our three user interface related components leverages the fact that we think many user interfaces will consist mainly of lists: lists of users, lists of policies, lists of configuration options, etc. With this in mind, we made the building of such list-based user interfaces extremely simple. By simply creating the structure of the list using Python lists and dictionary data types, a user interface can be rendered in the browser without writing a single

line of HTML or JavaScript. Our JavaScript code in the browser parses these lists and dictionaries (having been translated to JSON within NOX), and uses these to generate the specified user interface out of Qooxdoo widgets.

7 FUTURE WORK

At no point was our group under the impression that we could implement even a fraction of the functionality we discussed and brainstormed about -- the idea that there was significant future work to be done has been with us from the beginning. While we have provided some useful pieces of the puzzle, the real work for NOX At Home is hopefully still to come, with the implementation of a multitude of different policy applications to make users' lives easier. To facilitate this, a number of platform-related items remain:

- More evidence sources should be implemented. The more evidence sources there are, the better the system should work.
- Combination of evidence should be handled using the rules established by Hooper or Bayes.
- Building on the strength of the List Based UI, we could leverage Qooxdoo's Forms API to build a Form Based UI component, allowing developers to easily create more varied data entry forms.
- A simple distribution of the system as a flash-card image should be made available.
- A website should be started to foster a community.

8 CONCLUSION

We do not think any of us were entirely satisfied with our performance or entirely pleased with the amount of progress we made (in large part, we think this stemmed from the lack of a team member who was willing to take on a project management role). Despite this, we do think our team produced some useful code, that moves us a large step closer to seeing NOX in the home, and we believe our sponsor agrees.