The Beacon OpenFlow Controller

David Erickson
Stanford University
Stanford, CA, USA
daviderickson@cs.stanford.edu

ABSTRACT

Beacon is a Java-based open source OpenFlow controller created in 2010. It has been widely used for teaching, research, and as the basis of Floodlight. This paper describes the architectural decisions and implementation that achieves three of Beacon's goals: to improve developer productivity, to provide the runtime ability to start and stop existing and new applications, and to be high performance.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*; C.2.3 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Measurement, Performance

Keywords

Beacon, OpenFlow, controller, Java

1. INTRODUCTION

The first open source control platform available for early OpenFlow adopters was NOX [12]. NOX allowed developers to choose whether to build network applications with a developer-friendly language (using Python), or high performance applications (using C++). As a long time user of NOX, this tradeoff led to the following question:

> Could an OpenFlow controller be both easy to develop applications for and also high performance?

The programming language and development environment have a significant impact on the productivity of developers, and can also be a limiting factor in application performance. Many developer-friendly languages exist, but their performance when used in an OpenFlow controller was unknown.

HotSDN'13, August 16, 2013, Hong Kong, China.

ACM 978-1-4503-2178-5/13/08 ...$15.00.

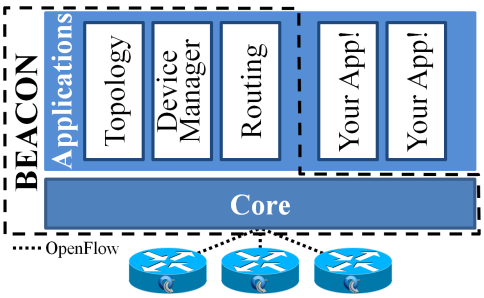


Figure 1: Overview of Beacon

A second question occurred while considering the role of an OpenFlow controller:

> If an OpenFlow controller is similar to a Network Operating System, should it be capable of starting and stopping applications at runtime?

This capability would make a controller more OS-like, and also enable new use cases. This paper explores both of these questions, and describes the Beacon OpenFlow controller. Beacon, as shown in Figure 1, provides a framework for controlling network devices using the OpenFlow protocol, and a set of built-in applications that provide commonly needed control plane functionality. This paper's contributions include:

Developer Productivity. An exploration of design and architectural choices with the goal of enabling developers to maximize their time spent productively developing applications.

Runtime Modularity. An implementation supporting starting and stopping both existing and new applications from a running Beacon instance.

Performance. Designs considered for the read and write paths of Beacon, resulting in a a multithreaded implementation with linear performance scaling.

In what follows I first present related work in §2, then discuss in detail the three main contributions: developer productivity in §3, runtime modularity in §4, and performance in §5. §6 presents comparative performance benchmarks with other controllers, §7 discusses deployment experience, and §8 concludes.

2. RELATED WORK

The first OpenFlow controller platform, NOX, was released in early 2008. NOX originally used cooperative threading to process events in a single threaded manner. In 2011 a version of NOX was released [20] with a multithreaded learning switch application.

Many other OpenFlow controllers have been released since NOX. They broadly fall into two main categories: open source single-

Language	Managed Memory	Cross Platform	High Performance
C#	✓	[1]	?
Java	✓	✓	?
Python	✓	✓	

Table 1: Candidate languages and their attributes.

instance controllers, and (so far) commercial closed-source distributed controllers.

The open source controllers are, generally speaking, used for research and development; therefore they tend to be single instance, with varying APIs presented to applications built on their platforms. A major distinction amongst them is the language they are written in.

- *C*: Trema [8] (also Ruby) and MUL [4].
- *Haskell*: Nettle [21], McNettle [22], and NetCore [16].
- *Java*: Maestro [18] and Floodlight [2], a fork of Beacon's early 2011 code using the Apache license.
- *OCaml*: Mirage [19] and Frenetic [13].
- *Python*: POX [5], Pyretic [17], and RYU [7].

Distributed controllers are able to distribute their state across multiple running instances for fault tolerance. Some of the public controllers in this space include Onix [15] from Nicira Networks, Big Network Controller [1] from Big Switch Networks, and ProgrammableFlow [6] from NEC. Onix has the additional capability of scaling performance by adding additional instances.

3. DEVELOPER PRODUCTIVITY

This section discusses design choices for Beacon intended to improve developer productivity. Design choices are presented in the following areas: programming language, libraries, and API.

3.1 Programming Language

C and C++ were the primary programming languages of OpenFlow controllers prior to Beacon. These languages can be used to produce very high performance applications, but they also came with a significant developer burden. Common problems included: long (>10 minute) full compilation times, compiler errors obfuscating the actual cause, and manual memory management programming errors leading to segmentation faults, memory leaks, etc.

Approaches have sought to minimize some such problems: incremental compilation of peripheral components to decrease compilation time and strict use of techniques such as smart pointers to ameliorate memory errors. These approaches are imperfect, as are the developers applying them. Choosing C/C++ for some environments is the right decision; however, this paper explores whether a more developer friendly language could be used to create a high performance OpenFlow controller intended to run on commodity hardware, where CPU and RAM are easily (and cost effectively) scaled.

Three language attributes were identified as desirable for a candidate programming language: managed memory, cross platform, and high performance.

Automatic memory management (also called garbage collection) can eliminate most memory-related programming errors. Languages with this ability also usually have no, or minimal compilation, eliminating time wasted waiting for the program to compile. Such languages also provide error reporting indicating the exact line(s) that failed to compile/run. The candidate languages I considered when designing Beacon are shown in Table 1 and included: C#, Java, and Python. There are a variety of other potential languages, but these were the ones I was most familiar with at the time.

The expected operating system that Beacon would run on was Linux. But, it would be convenient to also run on other platforms such as Mac OSX and Windows without a significant porting effort. The effort involved in porting had prevented prior controllers from running on non-Linux platforms. All candidate languages have some ability to be executed on all of these platforms. However, the Common Language Runtime (CLR), the official interpreter for C#, lacked support for operating systems other than Windows. Meanwhile the remaining languages had official support on all three operating systems. Therefore, C# was ruled out.

High performance is a subjective term. In this context one component of its definition is the ability to scale performance with processing cores. The lack of true multi-threading in the official interpreter for Python eliminated this language as a candidate. The remaining primary candidate, Java, still had an unknown absolute performance when used in an OpenFlow controller. Other programs written in Java such as Hadoop and Tomcat exhibited high performance, so it seemed that Java would be a good choice. Section 6 examines the performance of Beacon - which turned out to be surprisingly high.

3.2 Libraries

Beacon leverages multiple off the shelf libraries in an attempt to maximize code reuse and to ease the development burden of both the controller itself, and applications using it. The most significant library is Spring.

Two main components of Spring are used in Beacon: the Inversion of Control (IoC) container, and the Web framework. A common task in Java is to create instances of objects and then to "wire" them together by assigning one as a property of the other. IoC frameworks allow developers to list in an XML file or as Java annotations, which objects to create, how they are wired together, and then provide methods to retrieve the resulting objects. Using an IoC framework can save significant developer time versus the common alternative of building many purpose-built factory classes. Beacon uses Spring's IoC framework for wiring within and between applications, and is explained further in §4. Spring's Web framework is used to map Web and REST requests to simple method calls, and performs auto conversion of request and response data types to and from Java objects.

3.3 API

The Application Programming Interface (API) for Beacon is designed to be simple and to impose effectively no restrictions, in that developers are free to use any available Java constructs, such as threads, timers, sockets, etc. The API for interacting with OpenFlow switches is event based. Beacon uses the observer pattern where listeners register to receive events of interest.

Beacon includes the OpenFlowJ library for working with OpenFlow messages. OpenFlowJ is an object-oriented Java implementation of the OpenFlow 1.0 specification. OpenFlowJ contains code to deserialize messages coming off the wire into objects, and to serialize and write message objects to the wire.

Interacting with OpenFlow switches occurs via the *IBeaconProvider* interface. Listeners register to be notified when switches are added or removed (*IOFSwitchListener*), to perform switch initialization (*IOFInitializerListener*), and to receive specific OpenFlow message types (*IOFMessageListener*).

Beacon also includes reference applications that build upon the core, adding additional API:

[1]No official support from Microsoft for platforms other than Windows.

Device Manager. Tracks devices seen in the network including their: addresses (Ethernet and IP), last seen date, and the switch and port last seen on. Device Manager provides an interface (*IDeviceManager*) to search for known devices, and the ability to register to receive events when new devices are added, updated, or removed.

Topology. Discovers links between connected OpenFlow switches. Its interface (*ITopology*) enables the retrieval of a list of such links, and event registration to be notified when links are added or removed.

Routing. Provides shortest path layer two routing between devices in the network. This application exports the *IRoutingEngine* interface, allowing interchangeable routing engine implementations. The included implementation uses the all-pairs shortest path [9] computation method. Routing depends on both Topology and Device Manager.

Web. Provides a Web UI for Beacon. The Web application provides the *IWebManageable* interface, enabling implementers of the interface to add their own UI elements.

4. RUNTIME MODULARITY

Most OpenFlow controllers have the ability to select which applications to build (compile time modularity) and which applications to launch when the controller starts (start time modularity). Beacon has the additional capability to not only start and stop applications while it is running, but to also add and remove them (runtime modularity), without shutting down the Beacon process.

This enables new ways for developers to interact with deployed Beacon instances. Possible use cases include: creating and installing an application to temporarily improve debug information gathering; rolling out bug fixes or enhancements to already running applications; installing new applications at runtime from an "app store"; or quarantining and disabling misbehaving applications.

To enable this functionality, Beacon uses an implementation of the OSGi specification, Equinox. OSGi defines bundles which are JAR (archive) files containing classes and/or other file resources, and mandatory metadata. Bundle metadata specifies the id, version, dependencies on other bundles or code packages, and code packages exported for other bundles to consume. Modularity is based on the bundle unit. A list of bundles controls what is used at start time, and at run time the bundle is the unit that can be started, stopped, added, and removed.

Developers determine how their application is modularized. For example, a bundle may contain multiple applications, just one application, or a single application may be spread across multiple bundles. These decisions will usually be made based on the degree of modularity an application has. For example, if a piece of the application could be replaced at start or runtime, such as the routing engine being used by Beacon's Routing application.

A component of the OSGi specification is the service registry. It acts as a broker for services to register themselves, and consumers to retrieve an instance of a particular service meeting their needs. Beacon heavily uses this model, where service providers export the service interfaces mentioned in §3.3, and consumers request and receive implementations of such services. The service lifecycle is dynamic: services can come and go based on what bundles are currently installed and running.

5. PERFORMANCE

Performance of an OpenFlow controller is typically measured as the number of *Packet In* events a controller can process and respond to per second and the average time the controller takes to process each event. This section describes Beacon's architecture for processing OpenFlow messages.

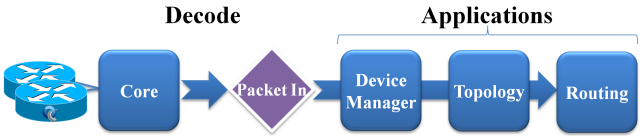


Figure 2: Beacon *IOFMessageListener* Single Thread Pipeline

5.1 Event Handling

Applications implementing the *IOFMessageListener* interface register with the *IBeaconProvider* service to receive specific OpenFlow messages type(s) arriving from OpenFlow switches. Registered listeners form a serial processing pipeline for each OpenFlow message type. The ordering of listeners within each pipeline is configurable, and the listener can choose whether to propagate each event or not. An example pipeline can be seen in Figure 2. This pipeline has three applications registered to receive *Packet In* messages: Device Manager, Topology, and Routing.

5.2 Reading OpenFlow Messages

To achieve high performance, Beacon and all of its applications are fully multithreaded. Two of the multithreaded designs considered for reading and processing OpenFlow messages are presented next.

5.2.1 Shared Queue

Figure 3a shows the Shared Queue design which contains two sets of threads. The first set of threads, I/O threads, read and deserialize OpenFlow messages from switches, then queue read messages into a shared queue. Each switch is assigned to a single I/O thread, and multiple switches may be assigned to the same thread. The I/O thread also writes outgoing OpenFlow messages to its switches.

The second set of threads, pipeline threads, dequeue OpenFlow messages from the shared queue, running each through the *IOFMessageListener* pipeline corresponding to the type of message. This is efficient for pipeline threads; whenever there are queued messages, all pipeline threads can be busy processing them. However, this design also necessitates a lock on the shared queue, and the corresponding lock contention between both sets of threads.

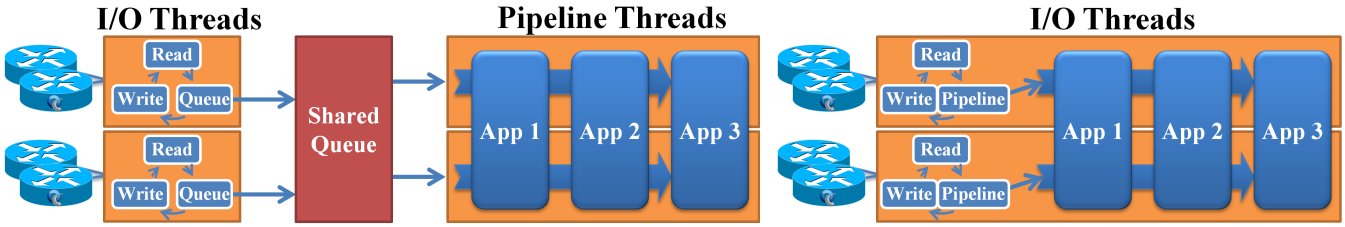
Variants of this design could have multiple queues, perhaps one per switch, or even more than one per switch, each with different priorities. This could improve fairness of message processing between switches, via round-robin servicing of the queues.

5.2.2 Run-to-completion

Figure 3b shows the run-to-completion design, a simplified version of the shared queue, having just a single pool of I/O threads. Each thread is similar to the I/O threads in the shared queue design, however, instead of queueing the message to be processed by a pipeline thread, it directly runs each read message through the *IOFMessageListener* pipeline.

This design does not require locks anywhere on the read path (excluding locks within applications), because the thread that deserializes the message also runs it through the pipeline. It further provides the following guarantee to *IOFMessageListeners*: for each switch, only one thread will ever be processing messages from that switch at a time. This enables lock-free switch-local data structures, such as those in Beacon's Learning Switch application.

This design does have the limitation that busy switches may be statically assigned to a subset of threads, leaving other threads idle.



(a) Shared Queue (b) Run-to-completion

Figure 3: I/O Designs

Algorithm 1 Initial Switch Methods

```
1: function WRITE(OFMessage msg)
2:     buf.append(msg)
3:     flush()
4: end function
5: function FLUSH
6:     socket.write(buf)
7: end function
```

Algorithm 2 Revised Switch Flush Method

```
1: function FLUSH
2:     if !written && !needSelect then
3:         socket.write(buf)
4:         written ← true
5:         if buf.remaining() > 0 then
6:             needSelect ← true
7:         end if
8:     end if
9: end function
```

Periodic rebalancing of switches to threads could help prevent this situation.

Beacon uses the run-to-completion design with a configurable number of I/O threads. OpenFlow switches upon connection are assigned in a round-robin fashion to I/O threads, and remain statically assigned to the thread until they disconnect. Each I/O thread takes the simple approach of processing all available data from each switch in turn.

5.3 Writing OpenFlow Messages

The way that messages are written to switches also affects performance. Beacon is multithreaded; therefore writes can occur from multiple threads simultaneously and methods must be synchronized to prevent race conditions. [2]

Algorithm 1 contains the immediate write design for sending messages to OpenFlow switches. Applications call the write method, which serializes and appends the message to a switch-specific buffer, then immediately writes it to the socket.

Performance with this design was slow. Tracing showed that the Java JVM was issuing a system kernel write for each socket write, with no intermediate batching. In a busy system, the time spent in user-kernel transitions was significant.

A revised batching design fixed this problem. The first part of this design is shown in Algorithm 2, a modified flush method containing two boolean flags: *written* and *needSelect*. The *written* flag is set to true when a socket write occurs, preventing subsequent writes until the flag is set back to false. The *needSelect* flag is set when there are unwritten data after a socket write, indicating the outgoing TCP buffer was full, and the remaining data needs to be written in the future when buffer space becomes available.

[2]Synchronization constructs are omitted from the pseudocode.

Algorithm 3 Revised I/O Loop

```
1: function IOLOOP
2:     while true do
3:         for all Switch sw : switches do
4:             sw.written ← false
5:             sw.flush()
6:             if sw.needSelect then
7:                 sw.selectKey.addOp(WRITE)
8:             end if
9:         end for
10:        readySwitches = select(switches)
11:        for all Switch sw : readySwitches do
12:            if sw.selectKey.readable then
13:                readAndProcessMessages(sw)
14:            end if
15:            if sw.selectKey.writeable then
16:                sw.needSelect ← false
17:                sw.flush()
18:            end if
19:        end for
20:    end while
21: end function
```

The second part of the design are modifications to the I/O loop shown in Algorithm 3. Lines 3–9, and 15–18 have been added to support this design. Lines 3–9 ensure that each switch will write any outgoing data once per I/O loop when the outgoing TCP buffers are not full. Lines 15–18 ensure that writes only occur when there is available outgoing buffer space detected using the system select function call.

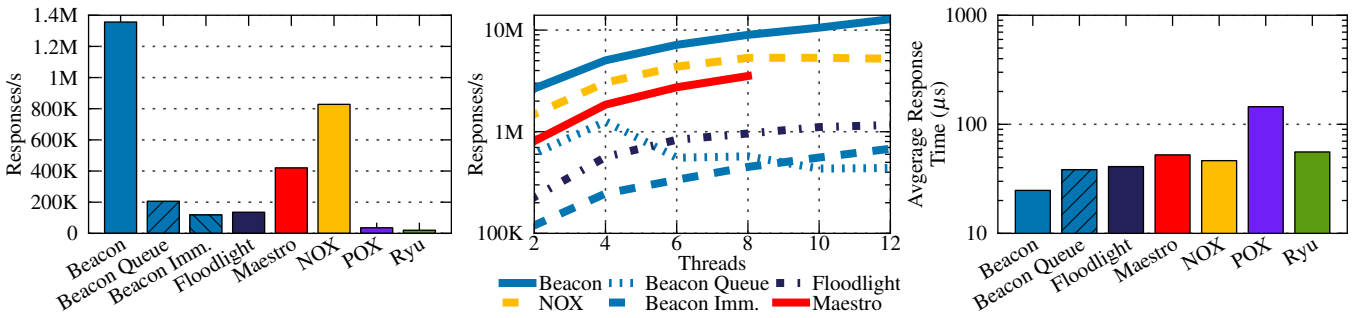
The result is that messages are either written immediately or once per I/O loop. For systems under heavy load a natural batching effect occurs between I/O loops, decreasing the write calls, and user-kernel overhead.

6. EVALUATION

The current standard for evaluating OpenFlow controller performance is Cbench. Cbench simulates OpenFlow switches, each sending *Packet In* messages to the controller under test. In the following benchmarks Cbench is configured to run 13 tests per controller/thread combination, each lasting 10 seconds. Each test's total responses received are averaged to produce a responses-per-second result, then the last 10 tests' results[3] are averaged to produce the final result.

Tests were run on Amazon's Elastic Computer Cloud using a Cluster Compute Eight Extra Large instance, containing 16 physical cores from 2 x Intel Xeon E5-2670 processors, 60.5GB of RAM, using a 64-bit Ubuntu 11.10 VM image (ami-4583572c). Cbench connected to each controller over loopback, and was given dedicated CPU core(s). Running locally was chosen because the

[3]The first three tests are considered warmups to provide time for the VM-based languages to perform any adaptive optimization and caches to warm up, and their results are not counted.



(a) Throughput: Single thread (b) Throughput: Multithreaded (c) Latency: Single thread

Figure 4: Cbench Tests

		Read Design	
		Queue	Run-to-completion
Write Design	Immediate		*Beacon Immediate*
	Batched	*Beacon Queue*	*Beacon*

Table 2: Beacon variations differing by chosen read and write design.

Cbench to controller traffic exceeded the capacity of the instance's 10Gb NIC.

As many open source controllers are included in these tests as possible, except where the controller did not work correctly with Cbench, and communication with the author did not result in a solution. Where available, the controllers were launched using the author's recommended settings. The following controllers are evaluated: Beacon, Floodlight, Maestro, NOX, POX, and Ryu. Three total versions of Beacon are evaluated: "Beacon", "Beacon Queue", and "Beacon Immediate". Each version differs based on the choice of read and write design. These choices are summarized in Table 2. Beacon uses the run-to-completion reading design and the batched message writing design, Beacon Queue uses the shared queue reading design and the batched writing design, and Beacon Immediate uses the run-to-completion reading design and the immediate writing design.

Cbench operates in either throughput or latency mode. In throughput mode, each of 64 emulated switches constantly sends as many *Packet In* messages as possible to the controller, ensuring that the controller always has messages to process. Figure 4a shows Cbench throughput mode results using controllers with a single thread. In this test Cbench and the controller are each bound to a distinct physical core on the same processor. Beacon shows the highest throughput at 1.35 million responses per second, followed by NOX with 828,000, Maestro with 420,000, Beacon Queue with 206,000, Floodlight with 135,000, and Beacon Immediate with 118,000. Beacon Queue performs much slower than Beacon on one CPU core because the OS must schedule both the I/O and pipeline threads using just one CPU core. Beacon Immediate attempts to write every outgoing message immediately to the TCP socket, requiring a kernel call for each, reducing performance below Beacon Queue. Both Python-based controllers run significantly slower, POX serving 35,000 responses per second and Ryu with 20,000.

Beacon's performance differs from Floodlight because Floodlight is based on older Beacon code that had not had its custom I/O design optimized for performance. Floodlight subsequently switched to using the Netty framework to handle its I/O. Also, Beacon's Learning Switch application uses a custom Hopscotch Hashing [14] hash map, which is slightly slower, but significantly more memory efficient than Java's built-in HashMap.

Figure 4b evaluates the scaling properties of the multi-threaded controllers. In this test, one instance of Cbench is used for tests with four or fewer threads, and a second instance of Cbench is launched for tests with six or more threads because a single instance is unable to saturate Beacon when running six or more threads. Beacon scales linearly from two to 12 threads, processing 12.8 million *Packet In* messages per second with 12 threads. NOX scales linearly from two to eight threads, handling 5.3 million *Packet In* messages per second at eight threads. However, after eight threads, performance decreases, because the process now spans two CPU sockets, and cache coherency protocols (and their associated overhead) are needed to maintain synchronization primitives used by NOX. In this benchmark, threads one through seven are on the first physical socket, and eight through 12 are on the second socket. Maestro scales linearly to its maximum of 8 threads at 3.5M *Packet In* messages/s. Beacon Queue has slower absolute performance than Beacon, scaling well to four cores, then decreases due to the same overheads that NOX experiences. Beacon Queue introduces lock contention both at the queue between the I/O and pipeline threads, and in the per-switch MAC table. Floodlight scales at the same rate from two through six threads, then slows down from eight through 12; however, it still slowly gains performance. Beacon Immediate shows the slowest initial absolute performance, but manages to scale linearly, beating Beacon Queue in performance when using 10 and 12 threads because it does not have the locking overhead that Beacon Queue does.

The latency test uses Cbench to emulate one switch which sends a single packet to the controller, waits for a reply, then repeats this process as quickly as possible. The total number of responses received at the end of the time period can be used to compute the average time it took the controller to process each. Results are shown in figure 4c. Beacon has the lowest average latency at 24.7 μs, Beacon Queue at 38.7 μs, followed by Floodlight, Maestro, NOX, and Ryu, each between 40 and 60 μs. POX is the outlier in this test taking 145 μs on average. The extra scheduling time needed when shuffling messages between the I/O thread, queue, and pipeline thread is apparent in the latency difference between Beacon and Beacon Queue. Beacon Immediate is omitted because Beacon's write behavior and performance is equivalent to Beacon Immediate when there is at most a single outstanding message per switch.

7. DEPLOYMENT EXPERIENCE

Use of Beacon by developers has provided valuable feedback for Beacon's design decisions.

As part of other research [10, 11], Beacon has run the network for the last two and a half years of an 80 server, 320 virtual machine cluster containing 80 virtual switches (one per server), and 20 physical switches wired as a k=4 fat tree. Beacon's modularity enabled a custom routing engine to extend the default, and

the creation of a custom Web UI for tracking experiments. Applications were inserted into the front of the *Packet In* pipeline to convert broadcast traffic to unicast (ARP and DHCP) to prevent flooding. Runtime modularity was primarily used locally during development to reload modified bundles without restarting a running Beacon instance.

Another user reported using Beacon for a distributed OpenFlow controller [23], a multicast media network, and an access controlled Wi-Fi network. He commented, "the simple yet effective design and self-explanatory codebase are the major drives for our preference of Beacon," and, "...(although the) OSGi bundling scheme has its own learning curve, it greatly simplifies the modularization of the whole framework. Further, in the context of our distributed controller implementation, OSGi was the main enabler for us to restart the necessary components of the controller to provide fault-tolerance."

The author of FlowScale [3] liked that Beacon's modular approach enabled him to remove unneeded parts of Beacon, while easily extending the code to add his own REST interface and statistic gathering applications. He liked OSGi for updating code and restarting buggy bundles, but did not like OSGi's steep and time consuming learning curve.

In retrospect, most of Beacon's original design decisions have been validated, with the exception of OSGi and runtime modularity which has had mixed results. It provided utility and enabled new use cases, but at the expense of some ease of use.

8. CONCLUSION

Beacon explored new areas of the OpenFlow controller design space, with a focus on being developer friendly, high performance, and having the ability to start and stop existing and new applications at runtime. Beacon showed surprisingly high performance, and was able to scale linearly with processing cores, handling 12.8 million *Packet In* messages per second with 12 cores, while being built using Java.

9. ACKNOWLEDGEMENTS

This work was supported by a Microsoft PhD Fellowship. Thanks to Ali Khalfan and Volkan Yazici for details of their experiences using Beacon. Thanks also to my advisor Professor Nick McKeown and the members of the McKeown Group for their feedback and reviews of this work.

10. REFERENCES

[1] Big network controller. http://www.bigswitch.com/products/SDN-Controller.
[2] Floodlight. http://floodlight.openflowhub.org/.
[3] FlowScale. http://www.openflowhub.org/display/FlowScale/FlowScale+Home.
[4] Mul. http://sourceforge.net/projects/mul/.
[5] POX. http://www.noxrepo.org/pox/about-pox/.
[6] Programmableflow controller. http://www.necam.com/SDN/doc.cfm?t=PFlowController.
[7] Ryu. http://osrg.github.com/ryu/.
[8] Trema. http://trema.github.com/trema/.
[9] Camil Demetrescu *et al*. A new approach to dynamic all pairs shortest paths. *JACM*, 51(6), 2004.
[10] David Erickson. *Using Network Knowledge to Improve Workload Performance in Virtualized Data Centers*. PhD thesis, Stanford University, May 2013.
[11] David Erickson *et al*. Optimizing a virtualized data center. In *SIGCOMM*, 2011.
[12] Natasha Gude *et al*. NOX: towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.
[13] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, 2013.
[14] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. *Distributed Computing*, 2008.
[15] Teemu Koponen *et al*. Onix: A distributed control platform for large-scale production networks. *OSDI*, 2010.
[16] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.
[17] Christopher Monsanto, Joshua Reich, Nate Foster, Jen Rexford, and David Walker. Composing software defined networks. In *NSDI*, 2013.
[18] Eugene Ng. Maestro: A system for scalable openflow control.
[19] Charalampos Rotsos, Richard Mortier, Anil Madhavapeddy, Balraj Singh, and Andrew W Moore. Cost, performance & flexibility in openflow: Pick three. In *ICC*, 2012.
[20] Amin Tootoonchian *et al*. On controller performance in software-defined networks. In *HotICE*, 2012.
[21] Andreas Voellmy *et al*. Nettle: Taking the sting out of programming network routers. *PADL*, 2011.
[22] Andreas Voellmy *et al*. Scalable software defined network controllers. In *SIGCOMM*, 2012.
[23] Volkan Yazıcı *et al*. Controlling a Software-Defined Network via Distributed Controllers. In *NEM Summit*, 2012.

APPENDIX

The evaluation used the following language runtimes: Oracle x64 Java 1.6 Update 37, Python 2.7.2, and PyPy 1.9.0. Controller specifics are listed below:

- **Beacon**. Version 1.0.2.
- **Floodlight**. Version 0.90.
- **Maestro**. Version 0.2.1.
- **NOX**. Verity branch, commit f75d2f31d934185fe0ce7c2482d0f8ae950b34f9 (9/6/12).
- **POX**. Betta branch, commit 52712bcaf0c230ba6d3915f56ab32b281a0d8de3 (12/2/12).
- **Ryu**. Version 1.5.

Cbench was used as the test harness for controller evaluations, the Git commit used was f848965336c1275c7847149c0089c16645ad8d32. Cbench's throughput mode tests were launched either with one or two instances of Cbench, based on the number of threads the controller under test was running. The following command launched a single Cbench throughput test:

```
taskset -c 0 cbench -c localhost -p 6633 -m 10000 \\
    -l 13 -w 3 -M 100000 -t -i 50 -I 5 -s 64
```

The following two commands launched two Cbench instances for throughput tests:

```
taskset -c 0 cbench -c localhost -p 6633 -m 10000 \\
    -l 13 -w 3 -M 100000 -t -i 50 -I 5 -s 32
taskset -c 8 cbench -c localhost -p 6633 -m 10000 \\
    -l 13 -w 3 -M 100000 -t -i 50 -I 5 -s 32 -o 33
```

And latency mode tests were launched with the following command:

```
taskset -c 0 cbench -c localhost -p 6633 -m 10000 \\
    -l 13 -w 3 -M 100000 -i 50 -I 5
```