The Interdisciplinary Center, Herzliya
Efi Arazi School of Computer Science

Design and Implementation of a Data Plane for the OpenBox Framework

M.Sc. final project submitted in partial fulfillment of the requirements

towards the M.Sc. degree in computer science

by

Pavel Lazar

This work was carried out under the supervision of

Prof. Anat Bremler-Bar

March 2016
Acknowledgments

I would like to thank my advisor, Prof. Anat Bremler-Bar for her support throughout my final project. I would also like to thank Yotam Harchol for his assistance and advices in the different aspects of the project. The help of Prof. Anat and Yotam was crucial to the success of this project, and for it I am very grateful.

This research was supported by the European Research Council under the European Unions Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no259085.
Abstract
The OpenBox Framework is framework that effectively decouples the control plane of NFs from their data plane. Similarly to SDN solutions that address only the networks forwarding plane (e.g., switching, routing), OpenBox provides a framework for network-wide deployment and management of NFs. The OpenBox framework is composed of three logic components: OpenBox Application, OpenBox Controller and OpenBox Instances used as the data plane.

This project presents a design of a general Open Box Instance that can be used as the data plane of the OpenBox Framework. The suggested architecture is modular in nature and allows the easy replacement of its packet processing engine. This feature allows a lot of improvement and innovation in the way packets are processed with an OBI and between them.

We also present a reference implementation of the suggested architecture which shows its useability as an OpenBox Instance and integrates it inside a working OpenBox Framework. Our reference implementation uses Click as its packet processing engine and explains how it can be easily replaced.
CONTENTS

1 Introduction

2 Background
  2.1 Processing Graph
  2.1.1 Processing Block
  2.2 OpenBox Framework Architecture
     2.2.1 OpenBox Applications
     2.2.2 Control Plane - OpenBox Controller
     2.2.3 OpenBox Protocol
     2.2.4 Data Plane - OpenBox Service Instance
     2.2.5 Execution Engine - Click

3 Related Work

4 Design Considerations

5 Architecture Overview
  5.1 Generic Manager
     5.1.1 Communication with the controller
     5.1.2 Running and Controlling the Execution Engine
     5.1.3 Execution Engine Push Message Handling
     5.1.4 OpenBox to Execution Engine Translation
     5.1.5 Managing the OBI
  5.2 Execution Engine

6 Implementation
  6.1 Generic Manager
     6.1.1 Execution Engine Runner Process
     6.1.2 Execution Engine Control Process
     6.1.3 Manager Process
  6.2 Execution Engine
6.2.1 Push Messages Elements ........................................ 47
6.2.2 DPI Elements ...................................................... 47
6.2.3 Packet Modification Elements .................................. 48
6.2.4 Miscellaneous Elements ......................................... 49

6.3 Code Flow ............................................................. 49
  6.3.1 OBI Initialization ................................................. 50
  6.3.2 Request Message Handling ..................................... 57
  6.3.3 Module Addition ................................................... 58
  6.3.4 Set Processing Graph ........................................... 60
  6.3.5 Read/Write Handlers ............................................ 62

7 Conclusions and Future Work ....................................... 63
References ........................................................................ 64

A Additional Protocol Messages Specification ......................... 66
List of Figures

1. Processing graph of an IPS ........................................... 4
2. Processing graph of a firewall ....................................... 4
3. The general architecture of the OpenBox framework .......... 6
4. OpenBox Protocol Connection setup between OBI and OBC 10
5. An example processing block definition for the RegexClassifier 12
6. A simple Click router configuration with 3 elements .......... 15
7. High level design of an OpenBox Service Instance .......... 21
8. Internal design of OBI’s generic manager ................. 22
9. Internal design of an Execution Engine ................. 28
10. Runner initialization procedure ................................. 51
11. Control initialization procedure ................................. 53
12. Push Message Receiver initialization procedure ........... 54
13. Configuration Builder initialization procedure ............. 55
14. Message Router initialization procedure ..................... 56
15. Send Hello and KeepAlive messages procedure ............. 57
16. Message handling procedure ....................................... 58
17. New module addition procedure ................................. 60
18. Setting new configuration procedure ........................... 61
19. Handling a read command ........................................... 62
List of Tables

1  Partial list of basic processing blocks . . . . . . . . . 11
2  Runner’s REST server endpoints . . . . . . . . . . . . . . 32
3  Control’s REST server endpoints . . . . . . . . . . . . . . 35
1 Introduction

Network functions (NFs), or middleboxes, play a major role in today's datacenter, operator, and enterprise networks. NFs are appliances deployed in the network's data plane. The call for network function virtualization (NFV) [8] aims to reduce the cost of ownership and management of NFs by making NFs virtual appliances, running on top of a hypervisor or in a container. While NFV improves on demand scaling and provisioning, it does not solve other problems such as the limited and separate management of each NF and the lack of multi-tenancy support. Network traffic nowadays usually traverses a sequence of NFs (a.k.a. service chain).

The OpenBox Framework [6, 7] makes network functions software-defined from a logically centralized control. The OpenBox Framework essentially decouples the control plane of a NF from its data plane. While the data plane of many NFs is similar, their logic may be different. OpenBox defines general-purpose (yet flexible and programmable) data plane entities called OpenBox Instances (OBIs), and a clever, logically-centralized control plane, which is called the OpenBox Controller (OBC). NFs are now written as OpenBox Applications on top of the OBC, which is in charge of deploying their corresponding logic in the data plane and to realize their intended behavior in the data path.

This work describes the design and implementation of the OpenBox Instance entity and its integration with the rest of the OpenBox architecture. We have created an OBI architecture design which addresses a wide set of constraints imposed by the nature of the OpenBox Framework and the problem domain it operates in. The design is highly detailed but its modular architecture leaves enough freedom for innovation and improvement from an OBI implementer.

We have implemented a reference OBI [14] to show the feasibility of the design and to be used within the OpenBox Framework. The design divides the OBI into two main parts: A Generic Manager and an Execution
1 INTRODUCTION

Engine. The Generic Manager, implemented in Python, is responsible for the communication with the OBC through the OpenBox Protocol [15] and translating its commands to the Execution Engine. The Execution Engine, implemented as an extension package for Click [12] the modular router, is the chosen packet processing engine of our OBI. The implementation can be easily deployed within a Mininet [13] network.

The structure of the remaining of this work is as follows: In section 2 we introduce the background for this work, which includes a brief overview of the OpenBox Framework and the Click modular router. In section 3 we survey related works, that can be used as replacement data plane for OpenBox. In section 4 we describe the design considerations that guided us through this project. Section 5 gives an overview of the architecture we have designed for the OBI. Section 6 shows the details of our reference implementation. Finally, we present our conclusions in section 7.
2 Background

2.1 Processing Graph

A survey of a wide range of common NFs shows that most of them can be viewed as an ordered list of processing stages. The survey shows that most common NFs use a very similar set of processing steps. For example, most NFs do some sort of header-based classification. Then, some of them would do some packet modification (e.g., translators, load balancers). Others, such as intrusion prevention systems (IPSs) and data leakage prevention systems (DLP), would further classify packets based on the content of the payload (a process usually referred to as deep packet inspection (DPI)). Some NFs would use active queue management before transmitting packets. Others (such as firewalls and IPSs) would drop some of the packets, or raise an alert to system administrator.

The OpenBox Framework abstracts packet processing as a processing graph [7], which is a directed acyclic graph of processing blocks. Each processing block represents a single, encapsulated logic unit to be performed on packets, such as header field classification, or header field modification. Each block has a single input port (except for a few special blocks) and zero or more output ports. When handling a packet, a block may push it forward to one or more of its output ports. Each output port is connected to an input port of another block using a connector.

Figure 1 and figure 2 shows two sample processing graphs for two common NF, a Firewall and an IPS, respectively. The firewall, for example, reads packets, classifies them based on their header fields values, and then either, drops the packets, sends an alert to system administrator and outputs them, or outputs them without any additional action. Each packet will traverse a single path of this graph.
2.1 Processing Graph

A processing block is a logical unit that performs some action on a packet. Each processing block has its configuration parameters that customize its behavior, as well as sets of read and write handlers, which allow querying and setting properties from and to the block. Each block may have zero or more outputs. The configuration defines which output will be used for each packet.

Some processing blocks represent a very simple operation on packets, such as dropping all packets. Others may have complex logic, such as matching the packet’s payload against a set of regular expressions and output the packet to the port that corresponds to the first matching regex, or decompressing gzip-compressed HTTP packets.

The notion of processing blocks is similar to Click’s notion of elements (see 2.2.5). However, while Click elements represent very basic packet operations
(e.g., a packet counter), OpenBox processing blocks may be quite complex (e.g., a regex classifier that matches packets against multiple regular expressions and counts the packets, bytes, and rates for each regex). In our implementation, described in Section 6, we use Click as our data plane execution engine. We map each OpenBox processing block to a compound set of Click elements, or to a new element we implemented, if no click Element was suitable.

2.2 OpenBox Framework Architecture

The OpenBox Framework is composed out of several layers, as presented in figure 3. The OpenBox Applications at the top which communicates with an OpenBox Controller (OBC) through a Northbound API. The OBC controls a set of OpenBox Service Instances, which act as the data plane, through the OpenBox Protocol.

The OBC provides an abstraction layer that allows developers to create network-function applications by specifying their logic as processing graphs. We use the notion of segments to describe regions in the network that can be configured with different policies and run different network function applications. Segments are hierarchical, so a segment can contain sub-segments. Each OBI belongs to a specific segment (which can, in turn, belong to a wider segment). Applications declare their logic by setting processing graphs to segments, or to specific OBIs. This approach allows having very flexible policies in the network with regard to security, monitoring, and other NF tasks, and by definition, supports the trend of micro-segmentation that reduces the size of network segments to allow highly customized network policies.
2.2 OpenBox Framework Architecture

Figure 3: The general architecture of the OpenBox framework.

2.2.1 OpenBox Applications

An OpenBox Application defines a single network function (NF) by statements declarations. Each statement consists of a location specifier, which specifies a *network segment* or a specific OBI, and a processing graph associated with this location.

Applications are event-driven, where upstream events arrive at the application through the OBC. Such events may cause applications to change their state and may trigger downstream reconfiguration messages to the data plane. For example, an IPS can detect an attack when alerts are sent to it from the data plane, and then change its policies in order to respond to the attack; these policies changes correspond to re-configuration messages in the data plane (e.g., block specific network segments, block other suspicious traffic, or block outgoing traffic to prevent data leakage). Another example is a request for load information from a specific OBI. This request is sent from the application through the OBC to the OBI as a downstream message, which will later trigger an event (sent upstream) with the data.

The OpenBox architecture allows multiple network tenants to deploy their NFs through the same OBC. The OBC is responsible for the correct deployment in the data plane, including preserving the applications’ priority.
and their ordering. It is also the responsibility of the OBC to merge any processing graph before committing it to the data plane. Since applications are not exposed to the merged processing graphs, and only receive event notifications for their own relevant events, they are effectively isolated from each other, implying that tenants cannot even know about other tenants that share data plane resources with them. Moreover, sharing the data plane among multiple tenants helps reduce cost of ownership and operating expenditure as OBIs in the data plane may have much higher utilization.

### 2.2.2 Control Plane - OpenBox Controller

The OpenBox controller (OBC) is a logically centralized software server that is responsible to manage the OBIs in all aspects: Setting processing logic, and controlling provisioning and scaling of instances. In an SDN network, the OBC can be attached to a traffic steering application to control chaining of instances and packet forwarding between them. OBC and OBIs communicate through a dual REST channel over HTTPS, and the protocol messages are encoded with JSON. Upon connection of an OBI, OBC determines the processing graphs that apply to this OBI based on its location in the segment hierarchy. Then, it merges the graphs to a single graph and sends this merged processing graph to the instance. The controller can request system information, such as CPU load and memory usage, from OBIs. Using this information it can scale and provision additional service instances, or merge the tasks of multiple underutilized instances and take some of them down. Applications can also be aware of this information and, for example, reduce the complexity of their processing when the system is under heavy load (to avoid packet loss or to preserve SLAs).

### 2.2.3 OpenBox Protocol

The OpenBox communication protocol [15] is used by OBIs and the controller (OBC) to communicate with each other via a bidirectional REST
interface. The protocol defines a set of messages for this communication and a wide set of processing blocks that can be used to build network function applications. Appendix A show the extension of the protocol for this work’s implementation.

Rest API
A REST server should be running at OBC and OBI. The default TCP port for OpenBox REST servers (on controller and on OBIs) is 3636. Messages are pushed to the REST server of the other side using POST method, where the URL is the /message/ directory followed by the name of the pushed message. For example, the Hello message will be pushed from OBI to OBC by sending a REST request as follows:

POST /message/Hello

With content type “application/json”, where the message object is sent as the payload of the request. The object should include the type field with its value equals to the name of the message. The connection may use standard HTTP compression and/or encryption, if supported and configured by both sides.

Transactions
Each message sent from OBC to OBI is a transaction, in the sense that it is expected to succeed in a whole, or, in case of a failure, rollback to the state that was in the OBI before the message began processing.

Each such message includes a unique (numeric) xid argument. This number identifies the transaction. When an asynchronous message is sent, it is expected that the receiving side acknowledge the receiving of a message with a 200 OK response, if the message is valid. It may delay the execution of the message, for example due to some transient load. Once the message is executed/processed, the result (if such exists) is sent back to the other side as a new message with the original xid value on the other REST channel. The receiver of the result should return a 200 OK response if the result
message is valid. In case of an error during the execution of a message, an Error message with the original *xid* value should be sent to the other side.

**KeepAlive**

Each OBI is expected to send *KeepAlive* messages every defined interval. If not received by controller, it can infer that the OBI has gone down. Controller may set the interval for each OBI using the *SetParametersRequest* message. However, upon startup of an OBI, it should immediately start sending *KeepAlive* messages. The default *KeepAlive* interval time is 10 seconds.

**Error**

Each error message has an error *type* and *subtype*. Several types and subtypes are defined in the protocol. Additional types and subtypes can be defined by OBI or OBC vendors, but these should be supported in both sides of the channel in any case. Each error message may also have a textual *error message* and some *extended message* (such as stack trace or other debug information).

**Connection Setup**

Figure 4 presents the connection setup process, as defined by the protocol. When an OBI starts, it sends to the OBC a *Hello* message that contains information on the instance such as its *identification* and supported *capabilities*. OBC then sets the required configuration to the OBI. This may include setting parameters such as *KeepAlive* message interval, or addresses for *log* and *storage* servers. It can also include injecting a custom module to OBI, if this capability is supported by the OBI. Eventually, the OBC sends a *processing graph* to the OBI and by that it sets the packet processing logic for the OBI. A *BarrierRequest* is then sent to ensure that the OBI finishes configuration before processing any further message from the OBC.
Processing Blocks

The OpenBox Protocol defines over 40 types of abstract processing blocks. Table 1 presents a partial list of some of the fundamental abstract processing blocks defined by the protocol.
### 2.2 OpenBox Framework Architecture

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>FromDevice</td>
<td>Read packets from interface</td>
</tr>
<tr>
<td>ToDevice</td>
<td>Write packets to interface</td>
</tr>
<tr>
<td>Discard</td>
<td>Drop packets</td>
</tr>
<tr>
<td>HeaderClassifier</td>
<td>Classify on header fields</td>
</tr>
<tr>
<td>RegexClassifier</td>
<td>Classify using regex match</td>
</tr>
<tr>
<td>HeaderPayloadClassifier</td>
<td>Classify on header and payload</td>
</tr>
<tr>
<td>NetworkHeaderFieldRewriter</td>
<td>Rewrite fields in header</td>
</tr>
<tr>
<td>Alert</td>
<td>Send an alert to controller</td>
</tr>
<tr>
<td>Log</td>
<td>Log a packet</td>
</tr>
<tr>
<td>BpsShaper</td>
<td>Limit data rate</td>
</tr>
<tr>
<td>VlanEncapsulate</td>
<td>Push a VLAN tag</td>
</tr>
<tr>
<td>VlanDecapsulate</td>
<td>Pop a VLAN tag</td>
</tr>
</tbody>
</table>

Table 1: Partial list of basic processing blocks

**Block Definition**

Each processing block has a strict definition, the strict definition allows the automatic parsing and verification of blocks. The processing block is defined by the following fields:

- **Name** (required, string) - A unique name for the block
- **Description** (optional, string) - A description of the block.
- **Configuration** (required, list) - A list of arguments to configure the block. Each argument is defined with the following fields:
  - Name - The name of the field.
  - Required - Is the field is mandatory or optional
  - Type - The type of the field.
  - Description - The purpose of the field and any side effects.
- **Read Handlers** (required, list) - A list of Handlers that can be queried
- **Write Handlers** (required, list) - A list of handlers that can be changed.

Each Handler is defined with the following fields:

- Name - The name of the handler
- Type - The type of the field.
- Description - The purpose of the field and any side effects.
Figure 5 shows an example of processing block definition for the RegexClassifier.

**RegexClassifier**
Classify a packet using a regex match on its content. The Classifier has N outputs, each associated with the corresponding pattern. If a packet doesn’t match any rule it will be discarded.

**Configuration**

<table>
<thead>
<tr>
<th>Name</th>
<th>Required</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>true</td>
<td>array(string)</td>
<td>A list of patterns to match against. Each pattern must be a legal regex (with no backtracking).</td>
</tr>
<tr>
<td>payload_only</td>
<td>false</td>
<td>bool</td>
<td>If true, the match will be only on the payload part of the packet. The payload is determined by the set of network layers. Default is false.</td>
</tr>
</tbody>
</table>

**Read Handles**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count_i</td>
<td>number</td>
<td>Returns the number of packets matched pattern_i</td>
</tr>
<tr>
<td>byte_count_i</td>
<td>number</td>
<td>Returns the number of bytes matched pattern_i</td>
</tr>
<tr>
<td>rate_i</td>
<td>number</td>
<td>Returns the matching rate for pattern_i, measured by exponential weighted moving average, in packets per second.</td>
</tr>
<tr>
<td>byte_rate_i</td>
<td>number</td>
<td>Returns the matching rate for pattern_i, measured by exponential weighted moving average, in bytes per second.</td>
</tr>
<tr>
<td>payload_only</td>
<td>bool</td>
<td>Read the 'payload_only' value.</td>
</tr>
</tbody>
</table>

**Write Handles**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload_only</td>
<td>bool</td>
<td>Set the 'payload_only' value.</td>
</tr>
<tr>
<td>match_all</td>
<td>bool</td>
<td>Set the 'match_all' value.</td>
</tr>
<tr>
<td>reset_counts</td>
<td>null</td>
<td>Reset all counts and rates to zero.</td>
</tr>
</tbody>
</table>
2.2 OpenBox Framework Architecture

2.2.4 Data Plane - OpenBox Service Instance

The OpenBox data plane consists of OpenBox service instances (OBIs), which are low-level packet processors. An OBI receives a processing graph from the controller. The OBI applies the graph it was assigned on packets that go through it. It can also answer queries from the controller and report its load and system information.

OBIs can be implemented in software or hardware. Software implementations can run in a VM and be provisioned and scaled on demand. An OBI provides implementations for the abstract processing blocks it supports, and declares its implementation block types and their corresponding abstract block in the Hello message sent to OBC. The controller may use a specific implementation in the processing graph it sends to the OBI, or use the abstract block name, leaving the choice of exact implementation to OBI. An OBI may be in charge of only part of a whole processing graph. In this case, one or more additional OBIs should be used to provide the remaining processing logic. A packet would go through a service chain of all corresponding OBIs, where each OBI attaches metadata to the packet before sending it to the next OBI. Upon receiving of a packet from a previous OBI, the current OBI decodes the attached metadata and acts according to it.

Our OBI implementation is divided into two: A generic manager and an execution engine. The generic manager is written in Python in about 5500 lines of code. It handles communication with the controller (via a local REST server), storage and log servers, and translates protocol directives to the specific underlying execution engine.

The execution engine in our implementation is the Click modular router, along with an additional layer of communication with the manager and storage server, and several additional Click elements that are used to provide the processing blocks defined in the protocol (a single OpenBox block is usually implemented using multiple Click blocks). All our code for the execution engine is written as a Click package, without any modification to
the core code of Click. The code of this module is written in C++ and is about 2400 lines of code.

The OBI is the main topic of this work and will be explained in later chapters.

### 2.2.5 Execution Engine - Click

Click routers [12] are built from fine-grained software components called elements. To build a router configuration, the user chooses a collection of elements and connects them into a directed graph. The graph’s edges represent possible paths for packet transfer. This layerless design was motivated by the peer-to-peer nature of packet processing. It also makes packet motion explicit and clear: packets move through the packet processor along the edges of the graph. Each router’s forwarding path is implemented by a sequence of elements; this supports fine-grained extensions throughout, since the elements can be rearranged. To implement an extension, the user can write new elements or compose existing elements in new ways, much as UNIX allows one to build complex applications directly or by composing simpler applications with pipes.

The Click architecture is centered on the element. Each element is a software component representing a unit of router processing. Elements perform conceptually simple computations, such as decrementing an IP packet’s time-to-live field, rather than large, complex computations, such as IP routing. They generally examine or modify packets in some way; packets, naturally, are the particles of network data that routers exist to process. At run time, elements pass packets to one another over links called connections. Each connection represents a possible path for packet transfer. Click router configurations are directed graphs of elements with connections as the edges. Router configurations, in turn, run in the context of some driver, either at user level or in the Linux kernel. Figure 6 shows some elements connected together into a simple router configuration. Elements appear as boxes; con-
Connections appear as arrows connecting the boxes together. Packets pass from element to element along the arrows (connections). This router’s elements read packets from the network (FromDevice(eth0)), count them (Counter), and finally throw them away (Discard).

![Diagram of Click router configuration with 3 elements](image)

Figure 6: A simple Click router configuration with 3 elements

**Elements**

The element is the most important user-visible abstraction in Click. Every property of a router configuration is specified either through the choice of elements or through their arrangement. Device handling, routing table lookups, queueing, counting, and so forth are all implemented by elements. Inside a running router, each element is a C++ object that may maintain private state. Elements have five important properties: element class, ports, configuration strings, method interfaces, and handlers.

- **Element class.** An element’s class specifies that element’s data layout and behavior.
- **Ports.** Each element can have any number of input and output ports. Every connection links an output port on one element to an input port on another.
- **Configuration string.** The optional configuration string contains additional arguments passed to the element at router initialization time.
- **Method interfaces.** Each element exports methods that other elements may access. This set of methods is grouped into method inter-


2.2 OpenBox Framework Architecture

- **Handlers.** Handlers are methods that are exported to the user, rather than to other elements in the router configuration. They support simple, text based read/write semantics, as opposed to fully general method call semantics.

**Connections**

A connection passes from an output port on one element to an input port on another. Connections are the main mechanism used for linking elements together; each connection represents a possible path for packet transfer between elements. In a running router, connections are represented as pointers to element objects, and passing a packet along a connection is implemented by a single virtual function call.

Router configurations may be seen as directed graphs with elements as vertices. However, connections link ports, not elements, and each element may have many ports. A more complete model treats router configurations as directed graphs with ports as vertices. Port graphs such as this have two kinds of directed edges, ordinary connections and internal edges. Internal edges show how packets may flow from input ports to output ports within a single element; an internal edge from an element’s input port \( i \) to its output port \( o \) means that a packet that arrived on input port \( i \) might be emitted on output port \( o \).
3 Related Work

The emphasis of this project is the implementation of a data plane for the proposed OpenBox Framework. When comparing this project to other work previously done we need to look for projects which can accomplish similar tasks as the OpenBox Framework as a whole or can be used as an alternative data plane. We could also consider previous work that are aimed at bringing software-defined networking to the middlebox domain [9, 10, 17] as long as they can be modified to be used with the OpenBox Framework.

ComB [17] proposes to consolidate multiple middleboxes into a single location. Based on the observation that not all middleboxes get to a load peak at the same time, they enjoy the multiplexing of a hypervisor to implement multiple virtual middleboxes on the same physical machine. In a similar way, they use Click as the base for their data plane.

xOMB [2] is a platform to create middleboxes using a general purpose server that provides a programmable general packet processing pipeline. It allows to program simple middleboxes such as load balancing switch, and NAT. It is a programmable framework in nature and therefore is less suitable to be used as the data plane for the OpenBox Framework.

Slick [3] presents a framework with centralized control that lets NF applications be programmed on top of it, and uses Slick machines in data plane to realize the logic of these applications. The work mainly focuses on the placement problem of data plane entities, and the problem of steering traffic between them. The Slick framework is inspired by the Click modular router and used some of its main concepts. Future work can use Slick as a replacement Execution Engine instead or aside the Click framework. We prefered to use Click because of its maturity and its wide adoption in the academic community.

Another related work in this context is the P4 programmable packet processor language [5]. The P4 language aims to define the match-action table of a general purpose packet processor, such that it is not coupled with a
specific protocol or specification (e.g., OpenFlow of a specific version). A P4 switch can be used as part of the OpenBox data plane, by translating the corresponding protocol directives to the P4 language.
4 Design Considerations

The OpenBox Service Instance (OBI) is the Data Plane implementation of the OpenBox Framework as described in 2.2. The modular nature of the framework and the flexibility it demands from the data plane introduces some design constraints and implementation challenges that needs to be taken into consideration:

1. The OpenBox Framework demands that the Data Plane packet processing engine can be implemented in either pure software, pure hardware or a mixture of both.
2. The OpenBox processing graph logic and the OpenBox Protocol are ignorant to the implementation or any change in the underlying packet processing engine.
3. The replacement of a packet processing engine within the OBI needs to be done with minimum code modification.
4. The flow of control messages is asynchronous and should not interfere with the packet processing.
5. The addition or removal of OpenBox processing blocks should be made easy and without the recompilation of the entire framework.

Items 1-4, are addressed in OBI’s architecture and high level design as described in chapter 5. Items 4 (NOTE: Item 4 affects both the architecture and the implementation.) and 5 are addressed in the specific implementation of the OBI as presented in chapter 6.
5 Architecture Overview

In order to achieve high modularity and a clear separation between the packet processing engine and the rest of the OBI features the OBI is composed of two main parts. A Generic Manager and an Execution Engine (EE). The Execution Engine is responsible for packet processing and except a minimal generic control interface is not aware of anything related to the OBI or the OpenBox Framework. The Generic Manager implements the OpenBox Protocol and is responsible for the interaction with the OpenBox Controller and with the Execution Engine. It is also has to translate all of the OBC’s instruction to the Execution Engine which he controls and monitors. Figure 7 depicts the high level design of the OBI, with emphasis on external facing components. This clear separation allows any future implementation to replace only the Execution Engine with any other software or hardware engine. This replacement can be done with a minimal set of changes to the rest of the OBI code and without any changes to the rest of the framework. Also, in case of multiple Execution Engine implementation, it allows different instances to use a different Execution Engine.
5.1 Generic Manager

The Generic Manager has several responsibilities:

1. Communicate with the OBC using the OpenBox Protocol.
2. Start and monitor the Execution Engine. The Manager’s implementation should allow the support of different Execution Engines with minimal modifications.
3. Translate controller’s Processing Graph to an Execution Engine configuration and translate read/write handler operation to an Execution Engine specific operations.
4. Receive and handle appropriately any push messages from the Execution Engine.

Figure 8, shows the internal structure of the generic manager with all the main components involved.
5.1 Generic Manager

Figure 8: Internal design of OBI’s generic manager

5.1.1 Communication with the controller

The communication with the OBC is asynchronous in nature and based on the OpenBox Protocol as described in 2.2.3. In order to support this communication we define 3 components:

1. **REST Server** - The REST server is responsible for receiving raw OpenBox Protocol messages from the OBC, deserializing them into internal objects and passing them for further processing to the **Message Router**. The REST server will only handle message parsing errors and defer any other error handling to other components (specifically the **Manager**).

2. **Message Router** - Receives message objects from the **REST server**
and passes them to relevant part of the system. This component may seem redundant but it allows to handle the OpenBox Protocol messages in an asynchronous way and apply any queueing strategies needed. Also, it decouples the REST server from any processing logic and allows easy extension of the protocol.

3. **Message Sender** - This component has two related responsibilities. Receiving message objects from the **Manager**, serializing them to OpenBox Protocol messages and sending them to the OBC. And sending log messages received from the Execution Engine to a Log Server (if configured).

### 5.1.2 Running and Controlling the Execution Engine

One of the manager’s main tasks is to start the Execution Engine and monitor its resource utilization (e.g., amount of memory or CPU used), control it (e.g., set its configuration processing chain or get specific counters) and receive any push messages from the Execution Engine (e.g., log messages) and handle them appropriately. It accomplishes this tasks with the help of the following components:

1. **Execution Engine Runner** - Allows the OBI to start, pause or stop the Execution Engine and setting the allowed resources to be used by it. It enables the OBI to query the running state of the Execution Engine and monitor its memory and CPU usage. In order to allow the easy replacement or addition of Execution Engine, the Execution Engine Runner is composed of two logical elements:

   1. An Execution Engine invariant element exposing a generic API to be used by the OBI to run and monitor the Execution Engine. This element will need to be implemented only once without any knowledge of the internal interworking of the execution engine. The element will be able to register and use a specific Execution Engine and will expose an appropriate API to the OBI to chose
the Execution Engine it wants to use.

2. An Execution Engine specific client to communicate with a specific Execution Engine. The client exposes a predefined API to be used by the OBI facing element of the Runner. Each Execution Engine will need to implement this client and implement any necessary supporting communication with the Execution Engine itself.

2. **Execution Engine Control** - Enables the OBI to control individual elements of a running Execution Engine or change its entire running configuration. The control over the Execution Engine is done via the concept of Read or Write handlers. Each handler is associated with either a specific element of the running configuration or the configuration as a whole. As in the Execution Engine Runner’s case, in order to enable the easy replacement or addition of Execution Engine, the Execution Engine control is composed of three logical elements:

3. An Execution Engine invariant element exposing a generic API to be used by the OBI to issue Read or Write commands for any element of the Execution Engine’s running configuration. This element will need to be implemented only once without any knowledge of the internal interworking of the execution engine.

4. An Execution Engine specific client to communicate with a specific Execution Engine. The client exposes a predefined API to be used by the OBI facing element of the Control. Each Execution Engine will need to implement this client and implement any necessary supporting communication with the Execution Engine itself.

It is important to note, that the Execution Engine Control knows nothing about the OpenBox processing graph or its translation to the Execution Engine specific processing graph representation. It is the responsibility of other parts in the OBI’s architecture to translate
control requests from the OpenBox processing graph to an Execution Engine specific processing graph representation.

3. **Watchdog** - It is imperative for the inner working of the OBI to make sure that the Execution Engine Runner and Execution Engine control continue to function properly and to communicate with the Execution Engine. Both components are composed of an Execution Engine specific client and therefore can have implementation issues that will affect the stability of the component they reside inside. The watchdog’s rule is to make sure both the Execution Engine Runner and Execution Engine Control are working properly and if not take any needed action to restart them or report to the rest of the OBI system if it is not possible.

### 5.1.3 Execution Engine Push Message Handling

The OpenBox Framework allows the network administrator to define processing graph the will generate alert or log messages as a response to a network event. In order to support this feature the Execution Engine needs to allow the generation of asynchronous message as a response to network event and the generic manager need to have the ability to receive and handle them appropriately. The manager has a specific component, Push Message Receiver, which is responsible to connect to the Execution Engine and receive its messages. The transport channel for the communication of messages will be defined by the implementation, and any Execution Engine will need to support it. The serialization format of messages between the Execution Engine and the Push Message Receiver will be a JSON format, this will allow any Execution Engine to support it easily. If any Execution Engine implementation would like to support a different transport channel or a different serialization format it will need to provide an appropriate proxy between the two.

The push message receiver will supply an API to allow the registration of
Message Handlers for any type of push message. The default behaviour for a message without a message handler will be to discard the message. The OBI will usually create two specific Handlers. One handler that will parse messages to OBI objects and pass them to be handled by main OBI logic. The other will just send any Execution Engine generated push message directly to a Log server without involving OBI’s main logic.

5.1.4 OpenBox to Execution Engine Translation

The OpenBox Framework defines a set of OpenBox Blocks and a way to connect them to form an OpenBox Processing Graph. The Execution Engine is capable of processing network packets based on a configuration in an Execution Engine specific representation. The main rule of the OBI is to translate the OpenBox Processing Graph into an Execution Engine configuration. The OBI is also responsible for issuing any Read/Write handlers operation from the controller. In order to do so, it must translate an OpenBox Read/Write operation on an OpenBox Block to a set of Read/Write operation on the Execution Engine.

The Configuration Builder is the component responsible for doing the above mentioned translation. As mentioned before, a major requirement from the OBI design is to support the replacement and addition of Execution Engines as easily as possible. To hold this requirement the Configuration Builder is composed of two parts:

1. An “external” facing part which is responsible of taking an OpenBox Configuration in its JSON format and transform it to an internal representation. This part is not related to any Execution Engine and it is specific only to the OBI representation.

2. An “internal” facing part which is specific to the Execution Engine in use. It is responsible of taking an OpenBox Configuration in the OBI specific format and translate it into an Execution Engine configuration. It takes each OpenBox Block object and translates it to a group of
5.2 Execution Engine

Execution Engine specific elements and the appropriate connections between them. It also, needs to create the connections between the correct elements of adjacent blocks.

In addition to the above mentioned requirements, the OpenBox Framework allows the addition of extra OpenBox Blocks. The Configuration Builder needs to be able to receive a declarative representation of new OpenBox Blocks and their appropriate translation to an Execution Engine representation. The Configuration Builder will be able to receive a serialized translation definition, parse it and extend its ability to support the new Blocks. This imposes a constraint on the implementation of the Configuration Builder to be of a dynamic nature and not to “hand build” blocks.

5.1.5 Managing the OBI

As any complex system, the OBI must have a central component responsible for managing the state machine and main logic. The Manager is the central component of the OBI, which manages the lifecycle of the Execution Engine and keeps track of controller’s requests and sends appropriate responses. The Manager uses all the other components to perform its tasks.

5.2 Execution Engine

The Execution Engine is the packet processing part of the system and therefore needs to be fast, reliable and to enable the expansion of the OpenBox framework need to be versatile and flexible. Figure 9 depicts the high level design of an Execution Engine.
5.2 Execution Engine

ARCHITECTURE OVERVIEW

Figure 9: Internal design of an Execution Engine

As can be seen in the figure, there are not many architectural constraints from the Execution Engine and any implementation can be used as an Execution Engine as long as it supplies the following logical components:

1. **Packet Processing** - The main rule of the Execution Engine is to handle network packets. Any implementation need to do it as efficiently as possible. As part of the implementation, the Execution Engine must provide the appropriate translation component. This component is responsible of translating OpenBox Processing Graph to an Execution Engine specific configuration.

2. **Run/Monitor** - This component allows the manager to start/pause/stop the Execution Engine. It also, allows to monitor the Execution Engine’s resource usage. The implementation needs to provide the necessary Runner Client as explained in section 6.1.1.

3. **Control** - The control component is the interface exposed by the Execution Engine to the manager which allows it to be controlled. In the Wrappers point of view, the control is done via the concept of Read/Write handlers, but the Execution Engine is to free to chose any
internal implementation it finds suitable. The implementation needs to provide the necessary Controller Client as explained in section 6.1.2.

4. **Push Message Sender** - This component is attached to the internal implementation of the Execution Engine and enables the ability to send asynchronous messages to the manager as a response to network events. The channel to be used will be determined by manager implementation. The push messages mechanism should be configurable and depends on the specific processing graph requested by the Controller. The push messages should impose minimal interference to the packet processing functionality of the Execution Engine.

5. **Storage Connector** - Certain OpenBox Blocks will require the storage of packet or session specific data and its retrieval later in the processing chain. The Execution Engine may chose to implement this kind of storage as a dedicated component connecting to an external storage server. The OBI architecture and design encourages the implementation of alternative Execution Engines, therefore it will not impose the use of an external storage and it can be implemented internally without any external facing component.
6 Implementation

The OBI implementation follows closely the architecture presented in section 5. The Generic manager is implemented in Python, and the Execution Engine implementation uses the Click framework as the packet processing framework and extends it with the necessary Click elements to make it compatible with the OBI architecture.

6.1 Generic Manager

The manager is implemented in Python. In order to make the OBI asynchronous the manager uses the Tornado Framework [19] which also facilitate in the implementation of the REST server needed for the OpenBox Protocol. The manager is composed of three python processes: The Execution Engine Runner process which is responsible for starting Click and monitoring its CPU and memory resources, the Execution Engine Control which opens a TCP based control channel for setting Click’s configuration and issuing read/write control commands and the Manager process which is the main entry point of the OBI and implements its entire logic.

6.1.1 Execution Engine Runner Process

The Execution Engine Runner is a separate process which is responsible for starting the Execution Engine (Execution Engine) and monitoring its running state. It exposes a REST server which allows the OBI Manager to start the Execution Engine and query its state and memory/CPU usage. The Runner REST server is implemented on top of the Tornado Framework and registers several endpoints that can be used by the Manager to query the state of the Execution Engine or issue a limited set of supported commands. Table 2 depicts the set of endpoints and their format.

<table>
<thead>
<tr>
<th>Endpoint URI</th>
<th>Action</th>
<th>Input</th>
<th>Return format</th>
<th>Meaning</th>
</tr>
</thead>
</table>

30
### 6.1 Generic Manager

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Method</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/runner/engines</td>
<td>GET</td>
<td>None</td>
<td>array(string) Get all supported execution engines.</td>
</tr>
<tr>
<td>/runner/engines</td>
<td>POST</td>
<td>Execution Engine name</td>
<td>200 for success or 400 if not a legal Execution Engine name. Set the Execution Engine to use.</td>
</tr>
<tr>
<td>/runner/start</td>
<td>POST</td>
<td>A JSON with parameters for the Execution Engine</td>
<td>200 for success or 400 if Execution Engine not set or parameters are wrong. Start the Execution Engine with a set of the same parameters as used by the Execution Engine client.</td>
</tr>
<tr>
<td>/runner/suspend</td>
<td>POST</td>
<td>None</td>
<td>200 for success or 400 if Execution Engine not set and not started. 5XX for error. Suspend the Execution Engine.</td>
</tr>
<tr>
<td>/runner/resume</td>
<td>POST</td>
<td>None</td>
<td>200 if resumed successfully or wasn’t suspended. 400 if Execution Engine not set and not started. 5XX for error. Resume the Execution Engine.</td>
</tr>
<tr>
<td>/runner/stop</td>
<td>POST</td>
<td>None</td>
<td>200 if stopped successfully or wasn’t running. 400 if Execution Engine not set. 5XX for error. Stop the Execution Engine.</td>
</tr>
<tr>
<td>/runner/running</td>
<td>GET</td>
<td>None</td>
<td>200 with boolean string. Check if the Execution Engine is still running.</td>
</tr>
<tr>
<td>/runner/memory</td>
<td>GET</td>
<td>None</td>
<td>200 with JSON info: (RSS, VMS, percent). 400 if Execution Engine not set and not started. 204 if Execution Engine stopped running. Get the Execution Engines process memory info.</td>
</tr>
</tbody>
</table>
The REST server uses an Execution Engine specific client to execute the requests. The client is a python class which must implement a predefined interface which the REST server requires. In the current implementation the Runner (and therefore the client it uses) reside on the same machine as Click. The client start/pauses/stops Click by issuing command line commands. To reflect the resources Click uses, the client uses basic linux API wrapped by the psutil module.

The above mentioned implementation has several advantages:

1. The Execution Engine can be run on a separate machine from the Manager.
2. Any blocking or long lasting operations do not interfere with the operation of the Manager. It can continue to receive OpenBox messages from the controller.
3. The Runner client is the only part that needs to be replaced for a
6.1.2 Execution Engine Control Process

The Execution Engine Control is a separate process which is responsible for enabling control over the individual elements of a running Execution Engine (Execution Engine). It exposes a REST server which allows the OBI Manager to control the behavior of specific elements in a running Execution Engine configuration as well as reading their state. The control over the Execution Engine configuration is done via the concept of a Read/Write Handlers. Each handler is associated with either a specific element or the entire Execution Engine. The Execution Engine REST server uses a specific Execution Engine Control Client to read or write handles. The Control REST server is implemented on top of the Tornado Framework and registers several endpoints which the OBI Manager can use to issue control commands to the Execution Engine. Table 3 depicts the set of endpoints and their format.

<table>
<thead>
<tr>
<th>Endpoint URI</th>
<th>Action</th>
<th>Input</th>
<th>Return format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/control/engines</td>
<td>GET</td>
<td>None</td>
<td>array(string)</td>
<td>Get all supported execution engines.</td>
</tr>
<tr>
<td>/control/engines</td>
<td>POST</td>
<td>Execution Engine name</td>
<td>200 for success or 400 if not a legal Execution Engine name.</td>
<td>Set the Execution Engine to use.</td>
</tr>
<tr>
<td>/control/connect</td>
<td>POST</td>
<td>A json with connection informa-</td>
<td>200 for success. 400 if engine not set or format error.500 for connection error.</td>
<td>Connect to Execution Engines control server.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tion: type (TCP/UNIX), address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/control/close</td>
<td>POST</td>
<td>None</td>
<td>200 for success or 400 if no Execution Engine is set.</td>
<td>Close the connection with the Execution Engines control server.</td>
</tr>
</tbody>
</table>
### 6.1 Generic Manager

<table>
<thead>
<tr>
<th>Path</th>
<th>Method</th>
<th>Request Body</th>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/control/engine_version</td>
<td>GET</td>
<td>None</td>
<td>200</td>
<td>200 with a string with Execution Engines control server protocol version. 400 if Execution Engine not set or not connected.</td>
</tr>
<tr>
<td>/control/loaded_packages</td>
<td>GET</td>
<td>None</td>
<td>200</td>
<td>200 with a list of loaded packages. 400 if Execution Engine not set or not connected.</td>
</tr>
<tr>
<td>/control/loaded_packages</td>
<td>POST</td>
<td>Package name</td>
<td>200 for success</td>
<td>Reload the running config to use the new package</td>
</tr>
<tr>
<td>/control/support_elements</td>
<td>GET</td>
<td>None</td>
<td>200</td>
<td>200 with a list of elements (local to Execution Engine). 400 if Execution Engine not set or not connected.</td>
</tr>
<tr>
<td>/control/config</td>
<td>GET</td>
<td>None</td>
<td>200</td>
<td>200 with config on success. 400 if Execution Engine not set or not connected to it. Get the current running configuration. In an Execution Engine specific format.</td>
</tr>
<tr>
<td>/control/config</td>
<td>POST</td>
<td>string with new configuration.</td>
<td>200 on success. 400 if Execution Engine not set or not connected to it. 5XX on Execution Engine failure.</td>
<td>Set the current running config of the Execution Engine.</td>
</tr>
<tr>
<td>/control/elements</td>
<td>GET</td>
<td>None</td>
<td>list(string)</td>
<td>Get a list of all elements in the running config.</td>
</tr>
<tr>
<td>/control/elements/&lt;element_name&gt;</td>
<td>GET</td>
<td>None</td>
<td>list(string)</td>
<td>Get a list of handlers for &lt;element_name&gt;</td>
</tr>
<tr>
<td>/control/elements/&lt;element_name&gt;/is_read</td>
<td>GET</td>
<td>None</td>
<td>True or False</td>
<td>Check if &lt;handler_name&gt;of &lt;element_name&gt;is readable</td>
</tr>
</tbody>
</table>
Table 3: Control’s REST server endpoints

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/control/elements/&lt;element_name&gt;/is_write</td>
<td>GET</td>
<td>None</td>
<td>Check if &lt;handler_name&gt; of &lt;element_name&gt; is writeable</td>
</tr>
<tr>
<td>/control/elements/&lt;element_name&gt;//&lt;handler_name&gt;</td>
<td>GET</td>
<td>params (string)</td>
<td>Get the data of read handler.</td>
</tr>
<tr>
<td>/control/elements/&lt;element_name&gt;</td>
<td>POST</td>
<td>string</td>
<td>Write data to a write handler.</td>
</tr>
<tr>
<td>/control/elements/sequence</td>
<td>POST</td>
<td>list of operations, each operation is a tuple of (type, element t_name, handler r_name, params)</td>
<td>Return a dict of results where the keys are &lt;element_name&gt;.&lt;handler_name&gt; and the value is the result.</td>
</tr>
</tbody>
</table>

The REST server uses an Execution Engine client to perform the requested control commands. The client is a Python class which implements a predefined interface. The communication between the client and Click is over a TCP socket. The client implements Click’s Control Protocol which is a basic request-response line based protocol. The client translates the server’s commands into an appropriate Click command(s). The client will only translate the commands themselves into Click’s commands. It is the responsibility of the OBI Manager, using the Configuration Builder to translate operation on OpenBox Block to operation on Click’s elements.

The above mentioned implementation has several advantages:

4. The Execution Engine can be run on a separate machine from the Manager, and the control can run on a separate machine from the Execution Engine.

5. Any blocking or long lasting operations do not interfere with the operation of the Manager. It can continue to receive OpenBox messages.
from the controller.

6. The Control client is the only part that needs to be replaced for a different Execution Engine

### 6.1.3 Manager Process

The Manager process is the main process of the OBI and its entry point. It is composed of several components that use a message passing design pattern to handle requests from the controller or process push messages from the Execution Engine. It is implemented in Python on top of the Tornado Framework, and uses its asynchronous facilities to achieve its tasks. The Manager uses other components to implement the OBI logic, and it delegates any actual work to them.

### REST Server

The OpenBox Protocol dictates that the messages between the OBI and the controller will be serialized to JSON and sent to an appropriate endpoint exposed by a REST server implemented on each side. The OBI’s REST server registers two endpoints:

1. `/obsi/runner_alert` - This URL is registered in the Execution Engine Runner as the URL to report when the Execution Engine stops working.

2. `/message/(.*)` - This endpoint handles all of the OpenBox Protocol Messages. It parses the message from a JSON representation to an internal Python object, sends a 200 OK response (or a 500 Error if the parsing fails) and puts the message in the Message Router for further processing.

The REST Server and the Message Sender are the only components in the system which handle raw OpenBox Messages. This will allow to replace or modify the wire representation of messages with minimal changes to the OBI itself. Actually, in case of modification, the only changes will be in
the messages module in the Message to_* (e.g., tojson) and from_* (e.g., fromjson) methods.

**Message Router**
This component holds messages received from the controller and allows the Manager to pick them up for processing when it is ready. The handling of messages is done via message handlers. The Message Router allows the Manager to register a different callable object (a function) to handle different types of messages. If needed, the Manager can change any registered message handler to accomplish easy implementation of a state machine. The Message Router also catches any uncaught exception from a message handler and translates it automatically into an Error Message that it sends to the Controller via the Message Sender.

**Message Sender**
This is a simple component which receives messages objects to send to the Controller, serializes them into the JSON format and sends them as the appropriate HTTP POST message to the Controller. The Message Sender allows other components to put the messages in a non-blocking manner and without care to the actual transport channel.

**Execution Engine Push Messages Receiver**
The Execution Engine Push Messages receiver component is responsible of receiving push messages originated from the execution engine. The component connects to a TCP/UNIX socket server where it is greeted with the engine’s name and protocol version. Then the engine’s message server pushes all messages to the client. Each message sent from the Execution Engine is in JSON format. The Receiver is oblivious of the inner structure of each message, and cares only about the type field which it uses to decide how to handle the message.

The Receiver exposes a public API function which registers a message type
to a handle function. When a message is received by the Receiver it will use the type field to call the relevant handler with the message’s content as argument. The Push Message Receiver will discard all messages unless there are handlers registered for the message.

The receiver has an asynchronous design which allows him to live in the same process as other components without blocking them. Due to its asynchronous design it is advised that each specific message type handler will be as lightweight as possible and avoid or defer all blocking tasks.

Currently, the Manager registers two message handlers: one that sends all Log messages to a Log Server (if one is supplied by the controller), and another that sends all Alert messages to the controller.

**Watchdog**

The Execution Engine Runner and Execution Engine Control are implemented as separate process. The two processes are crucial to the correct execution of the OBI and therefore there is a watchdog which monitors the state of the processes and notifies the manager if any of them stops working.

**Configuration Builder**

The Configuration Builder is one of the most important components of the OBI. It is responsible for building an Execution Engine’s configuration from an OpenBox configuration. The Configuration Builder has 2 parts:

1. External part which is not related to any Execution Engine. This part is responsible for taking an OpenBox JSON configuration and creates from it the necessary python objects. For this purpose it uses the OpenBox Block Definition (see below).

2. Internal part which is an Execution Engine specific. This part is responsible for taking each Block object and translating it to Execution Engine specific elements and their connections. It also, responsible for creating the connection between the correct elements of adjacent blocks. In the current implementation, the Configuration Builder
translates the OpenBox configuration into a Click configuration. For this purpose each OpenBox block needs to define (see Click Block definition below).

1. What Click elements it is composed from.
2. How to translate Block’s configuration fields into elements’ configuration fields.
3. How to connect elements to each other.
4. Define input and output mapping to enable connecting blocks.
5. A mapping between an OpenBox Block Read/Write handler and Click’s element (or elements) Read/Write handler. It is also need to define a transformation function to apply before/after the handler.

**OpenBox Block Definition (OBBD)**

Each OpenBox Block irrelevant of the underlying Execution Engine needs the following definition:

- **Type** - String - A unique name that identifies the block’s type (class).
- **Configuration** - A list of fields with the following attributes:
  - name - The name of the field
  - type - an optional attribute that states the allowed type for this field (e.g., String, IPv4 etc.)
  - Required - A boolean attribute that states
- **Read handles** - A list of handles that allows reading info of the block.
- **Write handles** - A list of handles that allows to write into a block and control its behaviour.

In order to support dynamic addition of OpenBox Blocks, and allow their automatic verification and configuration parsing the OBBD is represented in a JSON format.

**Click Block Definition (CBD)**

The Click Block Definition (CBD) is used by the ClickConfigurationBuilder
to translate the block based configuration into a click configuration. The
definition has the following parts and can be written as a Python class
(ClickBlock) or compiled to this class from a JSON object;

- **Type** - String - The same type as used in the OBBD.
- **Configuration mapping** - A mapping that translates a OBBD config-
  uration field or fields into element’s fields. The key is the name of a
  variable that can be used in element’s configuration and the value is a
  tuple: (OBBD fields, transformation function name).
- **OBBD fields** - a list of configuration names of the OBBD.
- **Transformation function** - A predefined function that need to be ap-
  plied on the OBBD fields. If None (or null in JSON notation) then
  the OBBD fields must contain a single name and its associated value
  will not be transformed.
- **elements** - List - each element is an instance of a Click element in the
  following format:
  - name - String - the name of the element (must be given and to
    be unique for this block).
  - type - String - The element type. Must be known to click.
  - configuration - mapping between the element’s configuration fields
    and their associated value. A value of $name means that the value
    must be taken from the block’s configuration using the configu-
    ration_mapping.
- **Connections** - List - Each connection is an object with the following
  fields:
  - from - String - Name of the ‘from’ element (source element). Must
    be one of the names defined above.
  - to - String - Name of the ‘to’ element (sink element). Must be
    one of the names defined above.
  - from_port - Integer - The output port of the ‘from’ element.
  - to_port - Integer - The input port of the ‘to’ element.
• **Multi Connection** - List - Each MultiConnection is an object with the following fields:
  - **Src** - The name of the source element.
  - **Dst** - The name of the destination element.
  - **Based_on** - What field in the source element to use as the connecting attribute.

• **input** - String or Mapping - If the input is directly and fully maps to one of the block’s elements then this field should hold the element’s name. If there is a mapping between the block’s input ports to different elements then there should be a mapping with the key being the input port number of the block and the value is a tuple of: . if None (or null in JSON notation) then there is no input for the block.

• **output** - String or Mapping - If the output is directly and fully maps to one of the block’s elements then this field should hold the element’s name. If there is a mapping between the block’s output ports to different elements then there should be a mapping with the key being the output port number of the block and the value is a tuple of: .

• **read_mapping** - Mapping - A mapping between the block’s read handles and the elements read handle. The key is the block read handle (can have $i suffix for a variable amount of handles) and the value is a tuple: where:
  - **element_name** - the name of the appropriate element.
  - **read_handle** - the element’s read handle name
  - **transform_function** - the name of the function to apply on the field before returning to the caller. If there is a $i in the handle name, then the transform function will take it as an optional argument. Can be None for no transformation.

• **write_mapping** - Mapping - A mapping between the block’s write handle and the elements write handle. The key is the block write handle (can have $i suffix for a variable amount of handles) and the value is
6.1 Generic Manager

a tuple: where:

- `element_name` - the name of the appropriate element.
- `read_handle` - the element’s write handle name
- `transform_function` - the name of the function to apply on the field before returning to the caller. If there is a $i$ in the handle name, then the transform function will take it as an optional argument. Can be None for no transformation.

**Initialization**

When the Configuration Builder starts, it needs to initialize itself and load the OBBD and associated CBD. The initialization steps are:

1. loads all the defined OpenBox Blocks.
2. Gets the list of Execution Engine internal elements of each block.
3. Verifies that all the elements needed by the blocks are loaded by the Execution Engine. This is done by getting the list of elements from the Execution Engine.

**Processing Graph Building Procedure**

The ConfigurationBuilder is given a JSON representation of an OpenBox Configuration and it needs to output an Execution Engine configuration (Click configuration string). It does so in a 4 phases.

1. Transform JSON configuration to Python objects configuration. See algorithm 1
2. Create Click elements to OpenBox blocks. See algorithm 2
3. Create Click connections from OpenBox blocks. See algorithm 3
4. Translate Python configuration object into Click configuration string. See algorithm 4
6.1 Generic Manager

Algorithm 1: JSON configuration to python configuration

Input: An OpenBox configuration in JSON representation

Output: A OpenBox configuration in Python representation

foreach JsonBlock do

/* Parse into python object */
Read the type of the block and get the appropriate class object
Instantiate the object with the block’s name and the configuration
fields value
if Any required fields is not given then
   raise BlockParsingError
end
if Any field value is of incorrect type then
   raise BlockParsingError
end
Append to the Blocks list
Add to the set of BlocksNames
end

foreach JsonConnection do

Parse The JSON into a Connection object
Verify that both the from and to blocks are in the set of Block names
end

foreach RequiredModule do

Add to Modules List
end

Runtime Module Addition

The OpenBox Framework supports the addition of external modules in runtime. The addition is done in two steps:

1. Installing the Execution Engine binary blob inside the Execution Engine. (Done by the Manager via the Runner).
Algorithm 2: OpenBox Blocks to Click Elements

Input: OpenBox Blocks in Python representation
Output: Click elements and connections in Python representation

1. foreach Block do
   2. Get the block’s CBD
   3. foreach element in CBD do
      4. Create an instance of the element:
         element_name = block_name@_@element_name
      5. foreach configuration field do
         6. if value is of the form $name then
            7. Read from configuration_mapping the value of the key name
            8. Apply the transformation function on the block’s config field value, and set it as the field’s value
         9. else
            10. Set the value as field’s value
         11. end
         12. Add to Elements list
      13. end
   14. end
   15. foreach connection in CBD do
      16. Create a connection object (prepend block_name@_@ for each element)
      17. Add to Connections list
   18. end
   19. foreach MultiConnection in CBD do
      20. foreach instance of the field based_on in the src element do
         21. Create a connection object between output i of src to input i of dst element
         22. Add to the Connections list
      23. end
   24. end
   25. end

44
Algorithm 3: Block’s Connections to Element’s Connections

Input: OpenBox Block’s Connections in Python representation
Output: Click connections in Python representation

1. foreach OpenBox Connection do
2.   Get the block’s CBD of the form block
3.   if output is fully mapped to an element then
4.     Set from field of the new connection to block_name@_@element_name
5.     Set from_port to the from_port of the block.
6.   else
7.     Find in the output mapping the element_name, port_number
8.     Set from to block_name@_@element_name
9.     Set from_port to port_number
10. end
11. Get the block’s CBD of the to block
12. if input is fully mapped to an element then
13.     Set to field of the new connection to block_name@_@element_name
14.     Set to_port to the to_port of the block.
15. else
16.     Find in the output mapping the element_name, port_number
17.     Set to to block_name@_@element_name
18.     Set to_port to port_number
19. end
20. Add the connection to the Connection list
21. end
Algorithm 4: Python Configuration to Click Configuration String

\textbf{Input} : Click’s Configuration in Python representation

\textbf{Output}: Click’s Configuration string representation

\begin{enumerate}
\item \textbf{foreach} \textit{Required package} do
\item \hspace{0.5cm} Append to result the Click string representation of the requirement
\item end
\item \textbf{foreach} \textit{Element} do
\item \hspace{0.5cm} Append to result the Click string representation of the element
\item end
\item \textbf{foreach} \textit{Connection} do
\item \hspace{0.5cm} Append to result the Click string representation of the connection
\item end
\end{enumerate}

2. After the module is installed inside the Execution Engine and it’s configuration is reloaded:
3. Loads all the new OpenBox Blocks
4. Gets the list of Execution Engine internal elements of each block.
5. Verifies that all the elements needed by the blocks are loaded by the Execution Engine.

6.2 Execution Engine

The Execution Engine in this project’s OBI implementation is based on the Click Framework. The reasons Click was chosen are:

1. Mature packet processing framework which is academically acceptable.
2. Rich configuration language which is similar in nature to the OpenBox specification for packet processing.
3. Easily and conveniently extendable with new elements.
4. Support for runtime hotswapping configuration.
5. Support for standalone packages addition.
6. Many projects which extend and enhance Click for faster packet handling [4,11,18].

Click, as is, could not provide all of the necessary services needed by the
OpenBox Framework or the OBI architecture therefore it was extended with some specific elements. All of the elements were grouped in a standalone Click package and (almost) no changes were made to the core code of Click.

### 6.2.1 Push Messages Elements

The OpenBox Framework allows the user to create alert and/or log message based on network events. Click as the Execution Engine of the framework needed to be extended to support this functionality. In order to do so two new elements were created. In order to create the elements we used the already existing “chatter” functionality which is used for printing debugging messages to an output channel. We also used the ChatterSocket element which enables to redirect the “chatter” messages to a TCP/UNIX socket.

1. **ChatterMessage** - The element prints a chatter MSG with a set TYPE. The format of the message is a simple JSON string: \{“type”:”TYPE”, “content”:MSG\}. This element can be used to send simple push messages.

2. **PushMessage** - The element is an extension of the ChatterMessage element, and allows to attach the packet content inside the MSG.

### 6.2.2 DPI Elements

The OpenBox Framework puts a great emphasis on Deep Packet Inspection. The Framework defines Blocks such as StringClassifier, RegexClassifier which perform some sort of classification based on a pattern inside the packet. The Click framework puts an emphasis on routing and therefore lacks the needed elements. A few general pattern matching elements were built which were used in the creation of some of the OpenBox Blocks.

1. **RegexClassifier** - This element uses Google’s RE2 [16] regex library to perform pattern matching on a set of regular expressions. The element tries to match packet content against all of the patterns simultaneously and outputs the packet to an output which corresponds to the first
matched pattern.

2. *RegexMatcher* - The element has 2 outputs, output 0 is for packets which match any of a set of regex patterns and output 1 (if connected) is for unmatched packets. The element will try to match all the pattern simultaneously using Google’s RE2 library.

3. *GroupRegexClassifier* - The element has K outputs, each associated with a group of regular expression patterns. The element will try to match the packet’s content against all of the patterns simultaneously using Google’s RE2 library. The packet will be send to an output corresponding to a group that matched all the patterns in the group.

4. *StringClassifier* - The elements classifies a packet based on an exact string match of the packet’s content. The element has N outputs, each associated with a corresponding string. The element will try to match any of the strings against the packet’s content using Aho-Corasik [1] algorithm. The packet will be sent to the first string matched. This element performs better than RegexClassifier when fixed strings are used.

5. *StringMatcher* - The element tries to match a packet to one of the strings given in the configuration using Aho-Corasik algorithm. If a match is found the packet is sent to output 1. If nothing is connected it is discarded. Packets which do not match any string are sent to output 0.

### 6.2.3 Packet Modification Elements

The Click framework has a few elements which enable the modification of packet’s content. The OpenBox Framework demanded a few more general elements which will enable a user to get better control over the fields of a packets.

1. *NetworkDirectionSwap* - The element allows to swap the direction of the packet. The user has full control over which layers to swap, and
he can chose to swap the following layers: Ethernet, IPv4, IPv6, TCP and UDP. When a certain layer is swapped the source and destination fields of the requested layer are swapped.

2. **NetworkHeaderFieldsRewriter** - The element allows to set a new value to one (or more) of the predefined fields. The supported fields are: ETH_SRC, ETH_DST, ETH_TYPE, IPV4_SRC, IPV6_DST, IPV4_PROTO, IPV4_DSCP, IPV4_TTL, IPV4_ECN, TCP_SRC, TCP_DST, UDP_SRC and UDP_DST.

### 6.2.4 Miscellaneous Elements

Our Execution Engine implementation using click had to be complemented with a few more elements that did not fall into any of the above mentioned categories:

1. **MultiCounter** - The element measures packet count and rate for multiple inputs. This element is used by any OpenBox Block which has multiple outputs. It allows to get the value of counters at the same time thus preventing bias in the measurements due to measuring in different times.

2. **AutoMarkIPHeader** - Most of the classification or modification elements which use layers boundaries need that the packet’s IP layer will be marked. In order to do so automatically we have created the AutoMarkIPHeader which automatically sets the IP header annotation of the packet. The element can handle both plain Ethernet packet or VLAN tagged packets.

### 6.3 Code Flow

In the following section we will demonstrate the communication and relationship between the different components. The code flow will be demonstrated by common tasks performed by the OBI.
6.3 Code Flow

6.3.1 OBI Initialization

The initialization of an OBI is composed of the following stages:

1. Start the Execution Engine Runner and start the Execution Engine itself (see 6.1.1)
2. Start the Execution Engine Control. (see 6.1.2)
3. Start the push message receiver.
4. Create and start the configuration builder.
5. Start the Message Router and register handlers for the supported OpenBox Protocol Messages.
6. Start the REST server
7. Send Hello message and repeatedly send Keepalive messages

Execution Engine

The order of operations to start the Execution Engine and its Runner is depicted in figure 10 and can be summarized by the following steps:

1. Start the Runner process and its REST server.
2. Set it to use the Click Execution Engine.
3. Request the Runner to start the Click process. Click cannot be started with an empty configuration, therefore the Manager supplies a default one.
4. The Manager registers itself as an alert handler. If the Runner will identify that the Click process has stopped working it will notify the Manager via the registered URL.
5. The Runner’s PID is registered to be monitored by the Watchdog.
Figure 10: Runner initialization procedure
6.3 Code Flow

Execution Engine Control

The Manager communicates with the Click Execution Engine via the Execution Engine Control. After it starts the Execution Engine, the Manager starts the Execution Engine Control via the following procedure which is also depicted in figure 11:

1. Start the Control process and its REST server.
2. Set it to use the Click Execution Engine.
3. Request the Control to connect to the Click process.
4. The Control’s PID is registered to be monitored by the Watchdog.
6.3 Code Flow

Figure 11: Control initialization procedure

Push Message Receiver
The Manager uses the Push Message Receiver to get messages from the Click Execution Engine. Figure 12 shows how the Manager creates the Receiver and sets handlers for the message types its support. After the receiver is created it connects to the Click Execution Engine and start receiving
6.3 Code Flow

messaging. Initially, Click’s default configuration sends a Keepalive message every few seconds.

Figure 12: Push Message Receiver initialization procedure

Configuration Builder

The Manager creates the Configuration Builder via a two steps process that is depicted in figure 13. It first uses the Control to query the Click Execution Engine about all the elements it supports. Then using the list of elements it starts up the Configuration Builder. The Configuration Builder initializes itself as previously explained.
Message Router and REST server

The final stage before the OBI is operational and is able to notify the controller about its existence is to start the Message Router with the handlers to process the supported OpenBox Protocol Messages (figure 14), and start the REST Server to receive them.
Send Hello and KeepAlive Messages
After all the initialization steps are finished and the OBI is set up and ready for processing it needs to notify the controller. The OpenBox Protocol defines a Hello Message that the OBI need to send to the Controller, the Message contains the set of capabilities of the OBI. Along with the Hello Message the OBI starts sending KeepAlive Messages to allow the controller to make sure it is working properly. Figure 15 depicts how the Messages are created inside the OBI and sent to the controller.
6.3.2 Request Message Handling

The OpenBox Framework defines that the OBI is managed by the Controller by a set of OpenBox Protocol Messages. All of the Messages currently defined, except Hello and KeepAlive, comes in the form of a Request Message that is send from the Controller to the OBI and a Response Message that is generated by the OBI after it finished its processing. Figure 16 show the Request and Response Messages' flow inside the OBI. On the receiving end, the REST server receives the serialized message and transforms it into a Message object. It passes it to the MessageRouter which calls the appropri-
ate handler inside the Manager when it is ready to process the new Message. After the Manager finishes to process the Request it will create a Response Message or an Error Message that will be passed to the Message Sender for serialization and sending to the controller.

![Message handling procedure](image)

**Figure 16: Message handling procedure**

### 6.3.3 Module Addition

One of the novel features of the OpenBox Framework is the ability to add new modules at runtime. In the current implementation of the OBI, addition of modules is achieved by ability to add packages to the Click Execution Engine. The process of adding a new module is done via the *AddCustom-
ModuleRequest Message. The Message holds all the necessary information for the OBI, and figure 17 depicts its handling inside the Manager (see 6.3.2 for message handling). It has four core stages:

1. Install the new Click package. This is done by putting the received binary in a predefined location that Click can load from.
2. Reload the current running configuration to include the newly created package. Click is not aware of any new package until it is explicitly loaded inside the configuration.
3. Requery Click for the list of supported elements. After the package is loaded inside the configuration, Click is aware of its elements.
4. Update Configuration Builder with the new OpenBox Block definitions and the related Click Block Definitions. The new definitions are given inside the Request Message content.
6.3 Code Flow

6.3.4 Set Processing Graph

One of the most important tasks of the OBI is to translate the OpenBox Processing Graph into an Execution Engine Configuration and set it. The procedure has 3 major steps which are also depicted in figure 18:
1. Build an Execution Engine Configuration from the Processing Graph. This is accomplished with the help of the Configuration Builder.

2. The Click Execution Engine Configuration is sent to the Control which uses the ClickControl Client to hotswap any running configuration to the new one.

3. The Manager uses the Control again to query the Click Execution Engine about its running configuration to make sure that everything is working correctly.

![Figure 18: Setting new configuration procedure](image-url)
6.3.5 Read/Write Handlers

After the Processing Graph is running the Controller may issue read or write requests. The Manager fulfills this requests in three stages which are also depicted in figure 19:

1. The Manager requests the Configuration Builder to translate the Open-Box Read Handler a Click Read Handler and a Transformer.
2. The Manager uses the Control to issue the Read handler command and get the returned value from Click.
3. The Manager uses the transformer to translate the Click value into an appropriate value to be returned back to the controller.

![Figure 19: Handling a read command](image)
7 Conclusions and Future Work

In this project we have implemented an OpenBox Service Instance to be used as the data plane of the OpenBox Framework. We have created a general architecture to address the design constraints from such a system. The design is robust and decouples the packet processing engine from its managing layer. We have implemented the OBI architecture using Python for the managing layer and Click as the packet processing engine. We have extended the Click Engine with the needed elements to support all of the features needed for the OpenBox Blocks. We have showed how the implementation of the packet processing engine can be replaced with a different implementation in an easy way without many modifications to the managing layer.

Many researches have been made on the improvement of Click’s packet processing speed. Future work can implement the best results of this researches or suggest new ones that are specific to the tasks the OpenBox framework tries to solve. The Configuration Builder which is the component responsible for translating an OpenBox Processing Graph into Click’s Configuration can be improved to do optimization on the Processing Graph to reduce processing speed or improve resource utilization.

A lot emphasis was put on the design of the OBI to allow it to easily replace or add another Execution Engine. Future work can be done on the implementation of another Execution Engine that will work with the current Click based one or will replace altogether. The new Execution Engine can be hardware or software based and accomplish all of the defined processing blocks or to be specialized to perform a single processing functionality extremely well.
References


A Additional Protocol Messages Specification

The OpenBox Protocol does not define the specific details related to the addition of a custom module. In the following section we will list the exact messages format used for this work’s implementation.

A.1 Request

type: “AddCustomModuleRequest”

xid: Number
module_name: String
module_content: String
content_type: String (“application/octet-stream”)
content_transfer_encoding: String (“base64”)
translation: ExecutionEngineTranslationObject (An EE specific object, currently only click_object)

ClickTranslationObject
open_box_blocks: Array<OpenBoxBlockDefinition>
click_elements: Array<ClickElementDefinition>
click_blocks: Array<ClickBlockDefinition>

OpenBoxBlockDefinition
name: String (the blocks name)
config_fields: Array<ConfigField>
read_handlers: Array<HandlerField>
write_handlers: Array<HandlerField>

ConfigField
name: String
required: Boolean
type: String
A.1 ADDITIONAL PROTOCOL MESSAGES SPECIFICATION

description: String

**HandlerField**
name: String
type: String
description: String

**ClickElementDefinition**
name: String (the elements class name)
list_argument: String (optional) - The base name for a variable amount of arguments of the same type
mandatory_positional: Array<String>
optional_positional: Array<String>
keywords: Array<String>
read_handlers: Array<String>
write_handlers: Array<String>

**ClickBlockDefinition**
name: String (needs to be the same name as the OpebBoxBlockDefinition name.)
config_mapping: Map<String, <list of OBBD fields, function name to operate on them>>
elements: Array<Element>
connections: Array<Connection>
multi_connections: Array<MultiConnection>
input: String
output: String
read_mapping: Map<String, <element_name, handler_name, function_name>>
write_mapping: Map<String, <element_name, handler_name, function_name>>

**Element**
name: String - Instance name

type: String - Element type

cfg: Map - passed to the Element object. Values of fields can be prefixed by a ‘$’ which then use the config_mapping to calculate the real value.

**Connection**

csrc: String - A previously declared element
ddst: String - A previously declared element

src_port: integer
ddst_port: integer

**MultiConnection**

csrc: String - A previously declared element
ddst: String - A previously declared element

based_on: String - The name of the field to use as size for the connections

**A.2 Response**

type: “AddCustomModuleResponse”
xid: Number
ה контролר

השתיתتكون מאחר מפרידה,ולההמשתנה ב顷יה, או שה[at מכיל, את שכבה הבינית (Control Plane) של פונקציות רשת (NF) משכנת המודעות (Data Plane) של התשתית. בדומה לעתיד,::-היה ניידות, SDN של פונקציות רשת ביבוא מספריים, עם OpenBox. התשתית מעבירה פגישות לשכנת רשת ביבוא כל רשתות התשתית הביניות, או הביניות, (OpenBox Application), ה Victor, (OpenBox Instance), או ששיש מחשבים המודעים. פוריה זו מהות תכונה של OpenBox הארכיטקтуורית המודעיהשהנונה(database), психופתושה מודלארית ומספקת התשתית לתוךיה. תכונה זו, התשתית מעבירה פגישות ושיפורים בדרר שבבתוכרה לשכת תוחב עבוק בחר מועש OpenBox בינייהו.

בנוסף לכר, אנוש משפחות מימָשות שונים של הארכיטקтуורית המודעיה. המיתמיות מראה כי הארכיטקтуורית היא ביבר שמימשים אתבכר וארח התשתית הביניות. המיתמיות של מיתמיות ביבר מימשים עבוק עבורי מתוחב הרשת מוסביה כל ית התוחלת.
תכונת ומתיווח של Data Plane

עבור OpenBox

עיבוד פרוייקט להמוגש בעובדה במעון במדעי כלים

עיל יידי

עיבוד עת במדעי כלים

פרופ. ענת ברמל-בר

מרץ 2016