IDC HERZLIYA

Efi Arazi School of Computer Science

The Interdisciplinary Center, Herzliya
Efi Arazi School of Computer Science

The Openbox Controller - Northbound API

M.Sc. final project submitted in partial fulfillment of the requirements
towards the M.Sc. degree in computer science
by

Dan Shmidt

This work was carried out under the supervision of
Prof. Anat Bremler-Bar

January 2017

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 guidance, assitance and advice. His wise advices were crucial for the success of this project.

The help of Anat and Yotam was an integral part of this project and I am very grateful for the opportunity to perform this project under their guidance.

This research was supported by the European Research Council under the European Unions Seventh Framework Programme (FP7/2007 2013)/ERC Grant agreement no 259085.

Abstract

Most modern networks nowadays contain a massive amount of appliances, each appliance typically executes one network function (NF) (eg. Firewall). Each such appliance is bought, configured and administered separately. Most NFs perform some kind of Deep Packet Inspection (DPI).

OpenBox provides a framework for network-wide deployment and management of NFs which decouples the NFs control plane from NFs data plane.

OpenBox consists of three logic components. First, user-defined OpenBox Applications that provide NF specifications. Second, a logically-centralized OpenBox Controller (OBC) which serves as the control plane. Finally, OpenBox Instances (OBI) constitute OpenBox's data plane.

This work presents a design and implementation [2] for the user facing interface of the OpenBox Controller, which allows network administrators to efficiently create and manage their NFs. The implementation supplies users with a framework from which they can build and experiment with NFs, as well as a functioning OpenBox Controller which loads NFs and manages the OpenBox control plane. The design is extensible and allows OpenBox future developers to quickly add more functionality and retrieve more data from the control plane.

Contents

1 Introduction 6

2 Background 8

2.1 Network Function - Middlebox 8

2.2 Abstracting Packet Inspection 8

2.3 OpenBox 9

2.3.1 Components 10

3 Northbound API Architecture 12

3.1 Workflows 13

3.1.1 Application Loading Workflow 13

3.1.2 Read/Write Workflow 14

3.1.3 Event/Alert Workflow 15

3.1.4 Custom Module Addition Workflow 16

4 Northbound API Implementation 17

4.1 OpenBox Abstraction Layer (OBAL) 17

4.2 OpenBox Application 17

4.3 Processing Blocks 19

4.4 Event Handlers and Event Manager 20

4.5 Read/Write Handles 21

4.6 Topology Manager 21

4.7 Application Registry 22

4.8 Application Aggregator 22

4.8.1 Processing Blocks Merge 23

4.8.2 Recovering Original Block IDs 23

4.8.3 Instance Processing Graph Inference 24

5 OpenBox Application Implementation Examples 25

5.1 OpenBox Firewall 25

5.1.1 Configuration Parameters 25

5.1.2 Processing Graph 26

5.2 OpenBox IDS 27

5.2.1 Configuration Parameters 27

5.2.2 Processing Graph 27

6 Experimental results 28

6.1 Resource Utilization 29

6.1.1 Memory Utilization 29

6.1.2 CPU Utilization 30

6.2 Message Latency 30

7 Conclusion and Future Work 31

List of Figures

1 OpenBox Architecture . . . 9
2 Application Loading . . . 14
3 Read/Write Handles . . . 15
4 Alerts . . . 15
5 Firewall Processing Graph . . . 26
6 IDS Processing Graph . . . 28

1 Introduction

OpenBox [7] is a software defined framwork for network-wide development, deployment and management of network functions (NF). It is centrally controlled by a controller called the *OpenBox Controller* which exposes an interface for NF development and communicates with the OpenBox data plane using the OpenBox protocol.

The OpenBox Controller is logically-centralized software server. Its main responsibility is to manage the OBIs on the data plane, it will send them their packet processing logic once they are ready and it will report their status back to the user. The OBC communicates with OBIs with a RESTful server and a RESTful client, these server and client comprise the "Southbound API" of the controller.

This report presents our work on the design and implementation of the "Northbound API" of the OpenBox Controller. The "Northbound API" is a set of components synergized together in order to allow users to manage and deploy their NFs to the data plane. Once the logic is deployed it's the Northbound API's responsibility to reflect the data, events and alerts which are gathered on the data plane. The user in turn will have the ability to react to such events.

As an API which is expoed to highly skilled users, the design should be simple and robust on one hand and on the other it should have enough flexibility to allow users to perform many common network development tasks such as: generic application programming, application loading and managment, hardware abstraction by exposing network topology.

Another responsibility of the Northbound API is logic aggregation. The Northbound API has the opportunity to merge several NFs together in order to save resources and shorten the different packet service chains over the network.

We implemented an OpenBox controller named *Moonlight* and tested it thoroughly. The Moonlight controller is written in java in about 7500 lines of code. The implementation as well as an easy installation script are availeble in [2].

The structure of the remaining of this work is as follows: In section 2 we introduce the background for this work, which includes an OpenBox overview and a review on it's key components. In section 3 we describe the architecture of the Northbound API in high level. Section 4 described the various components of the Northbound API, and explain their logic. In section 5 several examples for OpenBox Applications are presented. We provide experimental results and performance measurments in section 6 and in particular we show that the Moonlight Controller can handle reasonable and high load of messages with a small latency in response time.

2 Background

2.1 Network Function - Middlebox

Network functions, or middleboxes, play a major role in todays enterprise networks. NFs are appliances deployed in the networks data plane. They can be either physical or virtual, and they provide an additional layer of advanced packet processing. Network traffic nowadays usually traverses a sequence of NFs (a.k.a. service chain) [6]. For example, a packet may go through a firewall, then through an Intrusion Prevention System (IPS), and then through a load balancer, before reaching its destination. A closer look into these NFs shows that many of them process the packets using very similar processing steps. For example, most NFs parse packet headers and then perform classification based on the headers, while some NFs modify header fields, other's may classify packets based on Layer 7 payload content. Nonetheless, each NF has its own logic for these common steps. Moreover, each NF has its own management interface. It might be that each one is managed by a different administrator, and that these administrators do not know, or should not know, about the existence and the logic of the other NFs. OpenBox has a centralized controller and therefore can allow system administrators to manage NFs separately on one hand and on the other merge them together and apply them to the data plane.

2.2 Abstracting Packet Inspection

OpenBox [7] defines a Processing Graph as a directed acyclic graph that describes the flow of a packet inside a NF. Each node in the graph is a processing block that describes a single processing stage of the packet (e.g header classification, payload classification, read packets from a device). Once the NFs functionality is abstracted with a processing a graph. The OpenBox framework can now merge graphs of several NFs.

2.3 OpenBox

OpenBoxs main purpose is to decouple NFs control plane from the NFs data plane. The network administrator can create NFs and decide on which network segments every NF should be applied. While doing that, the administrator can set aside worries such as which hardware will run the NF and how to install the NF. Another gain of the OpenBox framework is the fact that there is a centralized view of all NFs in the network. Every NF can be defined separately but on the data plane most of their functionality might be executed only once, depending on the specific logic of each NF. Figure 1 shows the main components and interactions of the OpenBox framework.

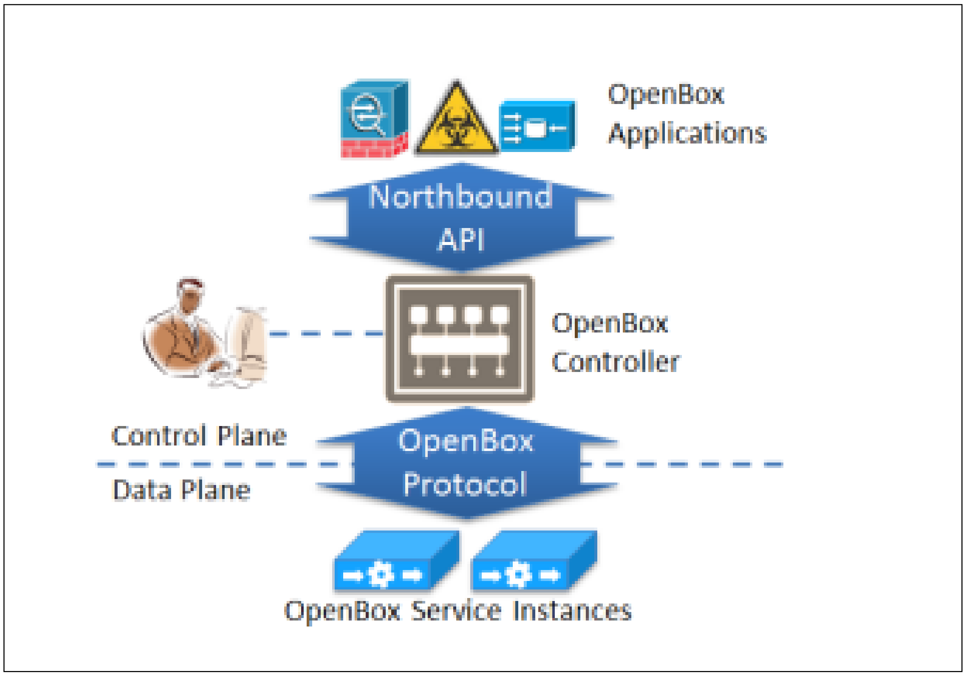


Figure 1: OpenBox Architecture

2.3.1 Components

2.3.1.1 OpenBox Northbound API

The OpenBox northbound API is a set of classes and commands that allow network administrators to craft their own NFs [4]. A network administrator is able to create NFs using a set of small generic pieces. Once an application is created the user can interact with the application and the data plane through exposed interfaces of the Northbound API.

2.3.1.2 OpenBox Application (OBA)

The OpenBox application is user defined logic for any NF, described in terms of the Northbound API.

An application is described by the following parameters:

- Application Priority
- A list of pairs $(LocationSpecifier, ProcessingGraph)$ which describe for each network location the processing graph which should be executed on that location
- Event and Alert handlers to handle various information and activities that arise in the network

Once the OBA logic is passed to the data plane, the OBA becomes event driven. Events from the data plane arrive from the OBIs to the OBA (through the OBC) so it can react. For example, when a load balancer is deployed, the various network instances in the data plane can expose their current load, and the application can change its behavior due to extreme load in one of its routes.

2.3.1.3 OpenBox Controller (OBC)

The OpenBox Controller is the centralized control of the OpenBox Framework. On one hand the OBC is facing the user and is responsible of passing information from the network administrator to the data plane (e.g. installing a new OBA), it will also trigger the right event/alert handlers on the OBA once it is needed.

On the other hand the OBC is facing the data plane and all OpenBox data plane components (OBIs) are communicating with it.

Every OBA defined is loaded on the controller. The controller decides how to apply the OBA logic on the data plane for example, it will try to use the merging algorithm to merge as much of the processing graphs without changing their logic.

2.3.1.4 OpenBox Instance (OBI)

The OpenBox Instance is the component responsible of executing the NFs logic. The OpenBox data plane consists of one or more OBIs which receive processing graphs which in turn are executed over the incoming traffic to the OBI. OBIs are running a REST server to support incoming requests (e.g receive processing graph) as well as output requests (e.g load reporting, read requests, etc). The OBIs can be implemented in any possible way (Software, Hardware, Hybrid) as long as they support the OpenBox protocol and their declared processing blocks.

2.3.1.5 OpenBox Southbound API

The OpenBox Southbound API is the communication mechanism between OBIs and the OBC. The Southbound API supports all communication needs of the controller. Whether it is control plane messages such as new OBIs which join the network (Hello Message), or it is the OBA retrieving information from the data plane. Every OpenBox message whether it is initiated on the OBC or on an OBI carries a transaction id (xid). Since the communication between OBI and OBC is asynchronous, the identifier serves as a correlation id between the different messages.

3 Northbound API Architecture

This work focuses on the design and implementation of OpenBox's Northbound API.

We describe all the functionalities of the Northbound API as workflows. Each workflow describes a role of the Northbound API. In section 3.1 we describe all Northbound API workflows.

To support all current workflows as well as future workflows, the OBC is designed in a manner which decouples OBA implementation from the OBC implementation. The Northbound API supplies a set of abstract classes (OBAL) to create homogenous API through which OBC and OBA are communicating. Further, to standardize the packet processing pipeline, Northbound API supplies a set of immutable classes which model all processing blocks and connections which allow the users to build and design their own processing graphs. To support event workflows the OBC exposes several interfaces which allow the OBA to receive notifications about events and respond back to the OBC, a similar mechanism is used to support Read/Write handles.

Other key components of the Northbound API are the "Application Registry" and "Aggregator". "Application Registry" allows applications to register with the controler and holds all the information about the OBA that might be required by the OBC. "Aggregator" is responsible for the proccesing graph merging mechanism.

In every design decision there is a constant tension between giving the user total freedom and between protecting it from making mistakes. The guidelines we used were, expose the functionality that is required for the normal use of the various applications. Although there are many functionalities which are internal to the OBC which might interest some expert users (e.g merged processing graph), those functionalities are not exposed.

3.1 Workflows

The OpenBox framework supports many workflows, some are used for self maintenance (e.g Setup Communication) and some of the workflows are features exposed to the framework users. We describe here several important workflows which are mainly the concern of the northbound API. The full description of all workflows can be found in the OpenBox spec [4].

3.1.1 Application Loading Workflow

The application loading workflow, is the process describing how a new OpenBox application is installed on the OBC and then flows down to the OBIs. After an OBA is described in the terms of the Northbound API. It is loaded through the Application Registry (Described further in section 3). The registry passes the application to the aggregation process which merges the processing graphs of all loaded applications. Once aggregated the OBC fires the ApplicationStarted event back to the application. When an OBI is started (sends a Hello message) a corresponding SetProcessingGraph request is sent to it.

Once an application receives the ApplicationStarted event it can wait for other OpenBox lifetime events (e.g InstanceUp, InstanceDown, HandleError) and can initiate the Read/Write workflows to query the data plane for information described below. Figure 2 shows the active components in this workflow.

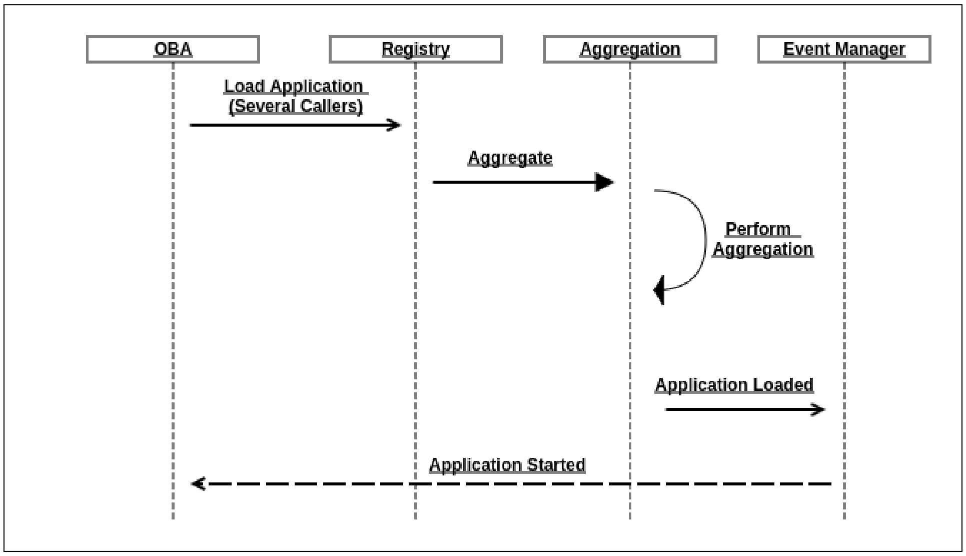


Figure 2: Application Loading

3.1.2 Read/Write Workflow

Different processing blocks expose different Read/Write handles. Application can access them through the OBC. For example, the Drop processing block exposes a “count” read handle which denotes the number of packets it has dropped. The application has access to an object called “HandleClient” and by specifying the OBI, Processing Block and Handle it can Read or Write to that handle. Figure 3 shows the active components in this workflow.

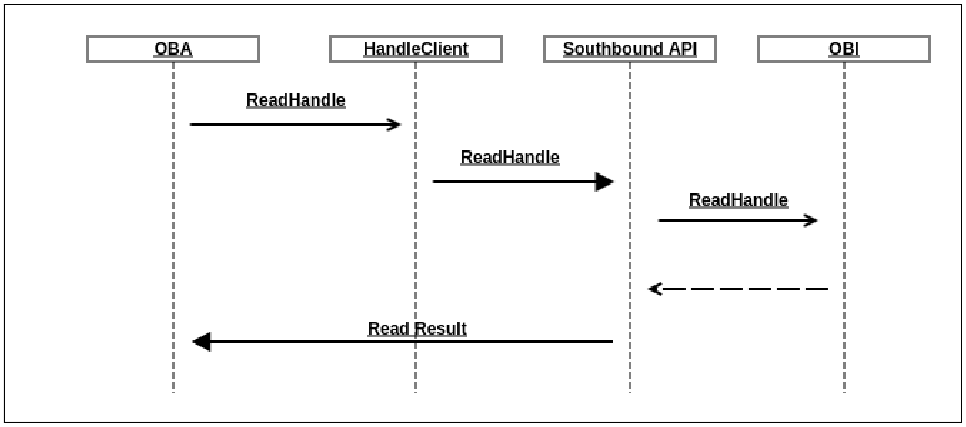


Figure 3: Read/Write Handles

3.1.3 Event/Alert Workflow

Events are the mechanism to pass information from the control plane to the application. The OBA can choose to implement handlers for the various OpenBox events (Full list in [4]). Once an event is triggered, the application can react to it. For example, in case an OBI crashes, the application might want to modify the processing graph. Figure 4 shows the workflow for Alerts, the workflow for events is the same.

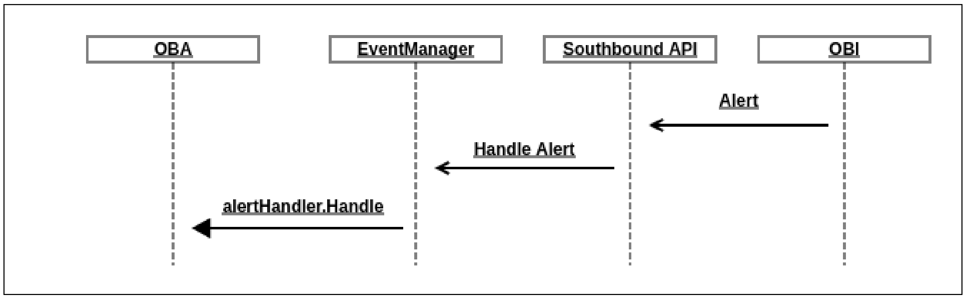


Figure 4: Alerts

3.1.4 Custom Module Addition Workflow

Custom modules are a way of extending OBIs with more functionality. The OpenBox framework allows users to define their own custom blocks and use them in their applications. Custom modules are namely binaries and translation objects created by the user and executed by the OBI. The Northbound API supports that, by allowing users to pass custom modules to the OBI. The Northbound API alleviates the fact that it is mostly agnostic to the logic of the actual blocks. Northbound API exports the *CustomBlock* abstract class. By inheriting the class, populating it's *modulename*, *binarypath*, and *translations* fields users are able to use their inheriting class just as any other block in their application.
When the application is loaded, and "Application Loading Workflow" is executed the OBC will check whether the OBI needs to receive a new custom block and will use the OpenBox protocol (via Southbound API) to pass the block to the OBI.

4 Northbound API Implementation

4.1 OpenBox Abstraction Layer (OBAL)

The "OpenBox Abstraction Layer" is a layer of abstract classes and interfaces with which the OBA creator can build NFs. It supplies the building blocks with which OpenBox applications are built as well as hooks to communicate with the OBC while it is running. The packages which compose the OBAL are *org.moonlightcontroller.bal* and *org.moonlightcontroller.blocks*.

4.2 OpenBox Application

An OpenBox application is formally defined as a tuple *(App Priority, Logic, Event Handlers)*. The BoxApplication class is the implementation for this definition. It has a priority field, a list of statements which define the logic of the OBA and hooks for read/write handles, events and error handling. It is the starting point of every OpenBox application, an application builder should inherit this class and populate it according to his desires.

1. Priority Field

   This field is used when merging several applications together. In case several applications have conflicting requirements. For example, in case packet is handled by two applications a Load Balancer and an IPS. The Load Balancer might find packet suited to proceed in specific network route however the IPS might in the same time recognize that the packet is malicious and should be dropped. When priorities are set correctly the merging algorithm will take them into account. The possible values are: *VeryLow*, *Low*, *Medium*, *High*, *VeryHigh*, *Critical*

2. Statements

   Statements are the actual processing graph. A statement is defined as the 2-Tuple *(LocationSpecifier, Processing Graph)*.

LocationSpecifier is an identifier for either an OBI *(InstanceLocationSpecifier)* in the network or a network segment (*Segment*). When a specific OBI is given the processing graph will be set in that OBI only, when a Segment specifier is given the processing graph will be set in all OBIs of that segment. The different locations specifiers are defined in 4.6.

When the user creates the statements he defines them in terms of processing blocks, connectors and a root processing block. The blocks are processing units which perform some kind of processing to packets. Each block has a single input port and several output ports. Connectors connect blocks into a graph, they connect output ports of once processing block to anothers input port. Root processing block is the first processing block in the statement. The processing blocks are described further in 4.3.

3. Application Lifecycle Events

   The BoxApplication class exposes events which are part of the application lifecycle as abstract methods which are overridable by inheriting classes. These methods are called whenever an important event has happened. These methods are: *handleAppStart(IApplicationTopology, IHandleClient)*, *handleAppStop*(), *handleError*(). The choice of public, overridable methods was made due to the assumption that each application would probably use these hooks and the most convenient way to connect to this mechanism will be to simply override the methods.

   The handleAppStart method passes some important parameters from the OBC back to the OBA. The *IApplicatoinTopology* interface can be used by the application to find LocationSpecifiers in the network and the *IHandleClient* interface can be used to read / write to processing block handles, further described in Section 4.4.

4. Instance Up/Down Events and Alerts

   Some applications might require to react when an OBI is started or stopped. These are exposed by BoxApplication in the shape of two interfaces that can be set when the application is constructed. *IInstanceUpListener* and *IInstanceDownListener*, both have a single method *Handle*() which is called when the corresponding event happens. The *LocationSpecifier* of the instance is passed as an argument to this method. Applications can, but do not have to implement these interfaces, this is the main reason for them being implemented in a way which requires the user to explicitly define and implement these interfaces. More events can be implemented in this way when the need arises. This is an important extensibility point of the Northbound API.

4.3 Processing Blocks

Processing Blocks are the building blocks of the packet processing chain. Every NF can be described as a graph of processing blocks called the processing graph.

Every processing block has a single input port from which it receives packets and 0 or more output ports to which the block outputs packets. Output ports of one processing block are connecting to the input port of another, creating the processing graph. In addition to ports, every processing block has its own configuration, which determines the behavior of that processing block. For example, *HeaderClassifier* processing block's purpose is to allow users to perform classification of packets according to header fields. To support that, the *HeaderClassifier* class has a list of rules, each rule has a priority and a destination output port to which packets are transferred.

Processing blocks also maintain read/write handles which allow to query processing blocks for information as well as remotely setting their value. For example, the *Discard* processing block will have a read handle which counts the number of dropped packets and the application will be able to read it as well as reset it when required.

The OpenBox protocol describes 22 processing blocks which the Northbound API implements with their inner configuration. Each of these blocks is an immutable class that uses the builder pattern, so that users and the inner implementation need not be concerned about concurrency and consistency of the class data. A full list of all processing blocks and configurations can be found in OpenBox protocol spec [4]. The implementation of all blocks can be found *org.moonlight.controller.blocks*.

4.4 Event Handlers and Event Manager

Event Manager is the the main crossroad of data that is passed from the control plane to the applications. It supports the workflows of events, alerts and application lifecycle events. It is an inner implementation of the OBC and OBAs are not exposed to it directly. When an application is loaded it is registered with the *EventManager*. In the registry process the application is inspected, and according to its implementation it will be notified of alerts and events, see section 4.2 for more details. Applications must implement the application lifecycle events, however alerts and instance up/down events are elective implementation. Thus if an application has implemented the corresponding interfaces *IInstanceUpListener*, *IInstanceDownListener* and *IAlertListener*, these interfaces will be used when one of the events occur. Instance Up event occurs when an OBI sends a Hello message. Instance Down event occurs when an OBI has not sent keep alive for a long period of time [4, 8]. Alerts are sent directly from the OBI when they are running the Alert block. EventManager implementation can be found in the *org.moonlightcontroller.events* package.

4.5 Read/Write Handles

Read/Write handles expose statistics from the different processing blocks. Each processing block defines which read/write handles it exposes. The application in turn, can read them or write to them. For this end the Northbound API exposes some of the Southbound APIs functionality. The Southbound API maintains a connection pool to all OBIs. Once a connection is obtained every message from the OpenBox protocol can be sent over the connection. The Northbound API uses this functionality of the Southbound API and creates an interface *IHandleClient* which is passed to the OBA when the application starts.

This interface has two methods *readHandle* and *writeHandle*, which read and write from the handles. When using this API, the application needs to specify the *InstanceSpecifier* from which it would like to read, from which block it would like to read and from which handle. When writing to a handle, the new value of the handle should be passed as well.

Since the entire system is asynchronous, the application uses another interface *IRequestSender* to receive the response. This interface is passed by the application when issuing a request and when the result is returned from the OBI one of the following two methods is called: *OnSucess* or *OnFailure* so the user can understand whether his operation succeeded or failed.

4.6 Topology Manager

Topology allows OpenBox application writers to refer to specific OBIs and segments in the control plane. Topology is defined as tree in which the leaves are OBIs and inner nodes are Segments. Specific OBIs are represented with the *InstanceLocationSpecifier* class and segments with the Segment class. Both implementing the *ILocationSpecifier* interface. When a user refers to a specific OBI it will retrieve or send information only to that OBI. When it refers to segment, all the OBIs under that segment are addressed.

The input for $TopologyManager$ is a json file (format described in the OpenBox specification [4]). When the OBC starts it reads the json file and uses the information to reach all OBIs and segments.
To expose a partial topology functionality to applications another interface is used $IApplicationTopology$. The application can use this interface to get $ILocationSpecifier$s according to their ids. This is the correct and safe way to retrieve a location specifier and then send that location a message such as a read request. The implementation for $TopologyManager$ can be found in *org.omoonlightcontroller.topology*.

4.7 Application Registry

The Application Registry is the component responsible on loading application onto the controller. As applications are actual running java code, this requires loading jar files dynamically.
When the OBC starts it will look in *./apps* directory for all jar files. Within all jar files it will look for the classes that inherit *BoxApplication*. All found classes will be instantiated and loaded into the controller. This functionality uses *java.util.ServiceLoader* which is used for plugin management. The implementation of the Application Registry can be found in *org.moonlightcontroller.registry*.

4.8 Application Aggregator

Aggregation is a key component of the OBC and the entire OpenBox framework. Aggregation is possible due to the fact that NFs tend to have similar processing blocks and thus in the actual data plane different NFs can share computation and packet flow. The algorithm for application aggregation is described fully in [7] . Here we describe how the Northbound API supports the aggregation algorithm.

4.8.1 Processing Blocks Merge

The aggregation process tries to merge processing blocks whenever possible. Every processing block has a different logic for merging. This logic is described within every block as implementation of the IStaticProcessingBlock or IClassifierProcessingBlock interface. Each of these interfaces contain two methods.

1. *canMergeWith(other)*: Which determines whether the current block can be merged with the other candidate.
2. *mergeWith(other)*: Which actually performs the merge logic of two blocks.

4.8.2 Recovering Original Block IDs

The aggregation process actually creates a new processing graph which is combined of all the loaded NFs processing graphs. Thus, the actual processing graph which is transferred to the data plane is different than the processing graph the users have created. OBIs are not aware of the merging process and they only hold the aggregated processing graph. When data flows over the control plane, from the the OBIs back to the OBA, it is described in terms of the aggregated processing graph. For example, in case we have two different OBAs a_1, a_2. with two different corresponding Alert blocks n_1, n_2 which send alerts on different packets. On the data plane there will be only a single aggregated alert block n_{12}. When n_{12} sends an alert it should be routed back to a_1 or a_2 according to the actual alert. When merging the *ApplicationAggregator* creates a mapping between the original blocks and their corresponding applications. Blocks that send messages over the control plane and need this functionality should expose the *origin_block* which triggered the message. In our example every Alert message holds the original block identifier which triggered the alert. When an Alert arrives the mapping is examined and the corresponding application is being notified about the alert. By implementing the original block id recovery in a declarative way (blocks send messages that carry the original application id), we actually gain application sanboxing. Each application can only see it's own alerts and messages. Internally there is only one aggregated application but as far as applications understand each one of them is separate and cannot access data from neighboring applications.

4.8.3 Instance Processing Graph Inference

As described in 4.2, users can define statements over either specific OBIs in the network or on complete segments in the network. When a user chooses to use a segment, all OBIs under that segment should receive the given processing graph. This presents a problem, since segments can contain other segements as well as OBIs. The behavior when a segment defines one processing graph and a lower sub-segement defines a different processing graph is as follows. The processing graph that is sent to an OBI is the processing graph which is set to its closest ancestor in the topology tree. To achieve this we perform a processing graph inference process, which is done prior to aggregation. In this process the *ApplicationAggregator* traverses the topology tree in a BFS scan and sets the processing graph for all OBIs under the current examined segment. In this way as the scan goes further down the topology tree the processing graph for each OBI is overwritten by a closer processing graph. Algorithm 1 shows psuedo code that flattens the processing graph.

Algorithm 1 Instance Processing Graph Inference

```
procedure InferStatments($box\_app$)
    $infered \leftarrow newBoxApp()$
    for $loc$ in $topology.Bfs()$ do
        $st \leftarrow box\_app.GetStatementForLocation(loc)$
        if $st$ is not $null$ then
            for $subloc$ in $topology.SubLocations(loc)$ do
                $infered.SetStatementForLocation(subloc, st)$
            end for
        end if
    end for
end procedure
```

5 OpenBox Application Implementation Examples

As part of the implementation we supply two examples for OpenBox applications. OpenBox firewall [3] and a Snort-like IDS example [5]. Both examples, are short but yet fully functiona, and they show the benefits of OpenBox.

Both NFs have several configurations which are helpful when experimenting with the application. Each example is a separate project and configuration is done with *example.properties* file. The presented examples were used to support the experimental results presented in [7].

5.1 OpenBox Firewall

5.1.1 Configuration Parameters

- *segment*: On which topology segemt is the firewall installed. Should be a valid id from the topology file.
- in_use_ifc: A boolean which indicates whether the firewall should read packets from a network interface.

- in_dump: A path to a pcap file, this should be used with $in\_use\_iface = false$.
- in_ifc: The name of the network interface to use, this should be used with $in\_use\_iface = true$.
- $rule\_file$: A path to a file containing the firewall rules.

5.1.2 Processing Graph

Our firewall example, processes packets either from a dump file or from a real network interface. For every rule defined in $rule\_file$ the firewall application creates a service chain of action blocks. An action block may be "Alert", "Log", "Drop" or "Output to Device". Output blocks are shared for all service chains. Figure 5 shows the processing graph created by the application according to the rules file.

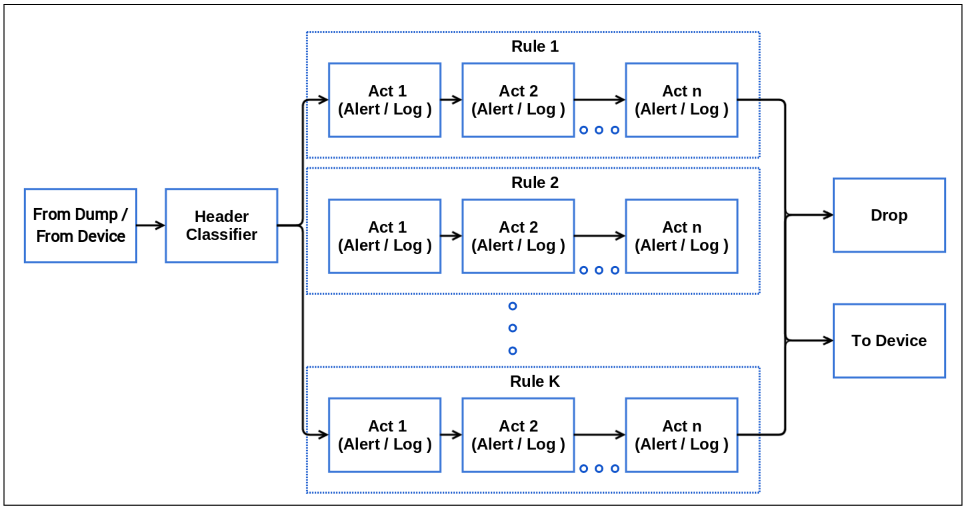


Figure 5: Firewall Processing Graph

5.2 OpenBox IDS

5.2.1 Configuration Parameters

- *segment*: On which topology segemt is the ids installed. Should be a valid id from the topology file.
- in_use_ifc: A boolean which indicates whether the ids should read packets from a network interface.
- *in_dump*: A path to a pcap file, this should be used with $in\_use\_iface = false$.
- in_ifc: The name of the network interface to use for input packets, this should be used with $in\_use\_iface = true$.
- out_use_ifc: A boolean which indicates whether the IDS should write innocent packets to a network interface.
- *out_dump*: A path to a pcap file, this should be used with $out\_use\_iface = false$.
- out_ifc: The name of the network interface to use for output packets, this should be used with $out\_use\_iface = true$.
- $rule\_file$: A path to a file containing the IDS rules.
- *alert*: A boolean indicating whether to alert on malicious packets (by default the IDS just drops them).

5.2.2 Processing Graph

Our IDS example, processes packets either from a dump file or from a real network interface. The first classification performed is by headers. Packets with tcp port set to 80 are being processed first and have a special service chain, other packets continue straight to the output block. Once a packet has been indeitified with tcp port 80, it passes through a regular expression (regex) classification. The Regex rules are defined in an exteral file $rule\_file$. In case a regex rule applies to the packet it will continue to the Alert or Alert+Discard blocks. Figure 6 shows the processing graph created by the application.

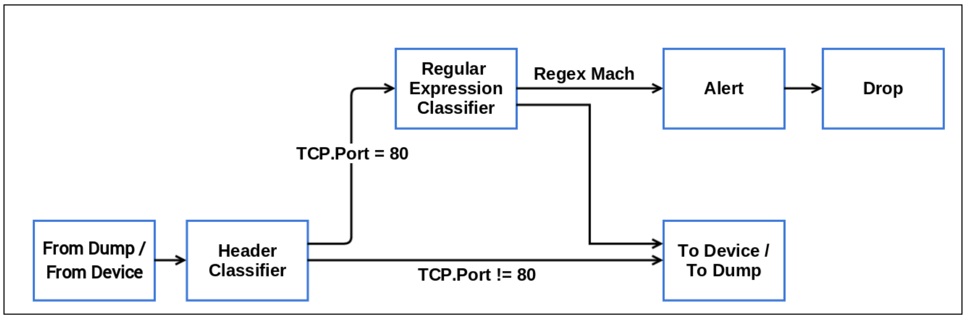


Figure 6: IDS Processing Graph

6 Experimental results

As part of the implementation, experiments were performed in order to verify that the implemntation can hold real-time traffic. Northbound API experiments, test how well the controller handles messages from the data plane.

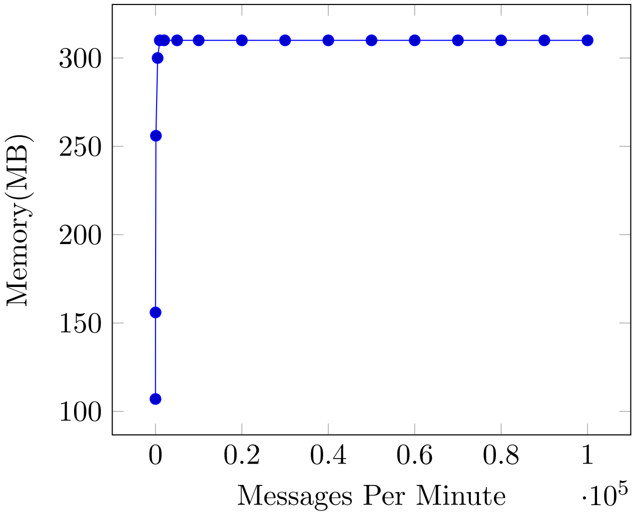
We define a metric Mpm (Messages Per Minute) which describes the amount of messages the controller receives in a minute. We show two sorts of experiments. 6.1 shows how CPU and Memory are being used under different loads of Mpm. 6.2 shows the latency of messages as Mpm load increases.

All experiments have the same machine setup. The machine runs a fully installed Moonlight Controller. The controller is loaded with a single alert block application. To trigger the alerts we use a java implemntation of a mock OBI. The mock OBI starts and performs the OpenBox handshake with the controller (Hello and SetProcessingGraph). Once the processing graph is set, the mock OBI sends alerts at a specified Mpm rate. All experiments were performed on a test bed with an Intel Xeon E3-1270 v3 CPU and 32GB RAM.

6.1 Resource Utilization

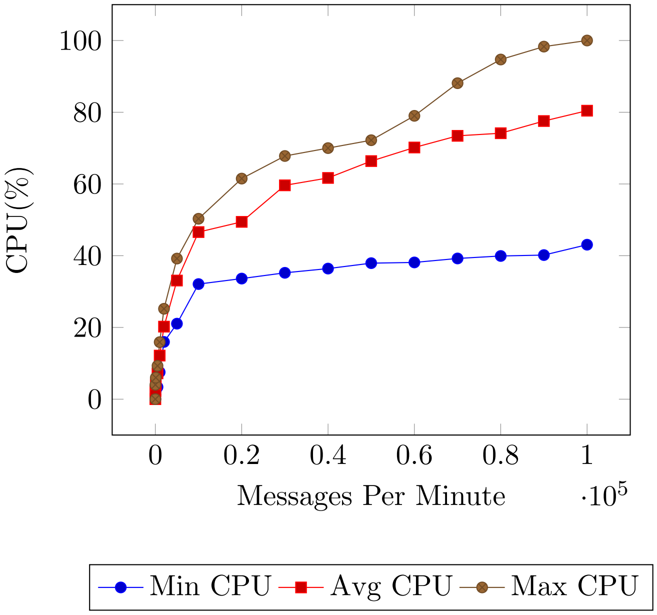
6.1.1 Memory Utilization

The memory measurments show that as Mpm increases the memory consumption is increased. However, the memory consumption in the tested scenario is always bound. Once memory consumption reaches 310MB there are no more allocations for new memory. This behavior is expected, once all of the controller's caches are fully allocated there is no more need for further allocations. Garabge collection cleans the stale memory in the controller, thus 310MB is enough memory to handle any load of Mpm.



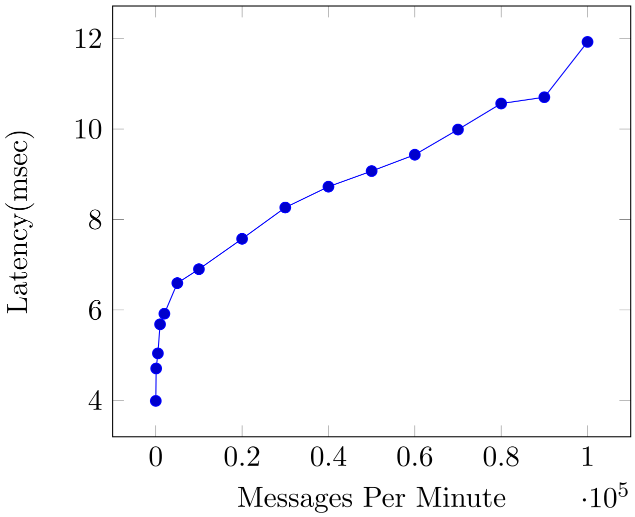
6.1.2 CPU Utilization

CPU measurements show the expected increase in CPU consumption as Mpm increases. We can also spot that the difference between minimal and maximal CPU consumption is increasing. The explanation for that behavior is that as load increases the amount of threads on the controller is increased. When threads are created / killed they have more influence on the overall CPU usage.



6.2 Message Latency

In the latency experiment we mesaure how much time passed since the OBI has sent the alert to the time it was handled by the application. Since we log the alert time inside the alert object the application can easily compare the times. Here we can see that latency increases as load increases, the graph shows almost a linear increase. Overall, latency is almost non existant. on 1000K Mpm we get 12msec of latency.



7 Conclusion and Future Work

In this project we have presented a full design and implementation of the Northbound API for the OpenBox Controller. The design addresses the requirements in the OpenBox specification and supports workflows which allow network administrators and other framework users to create, deploy, query and manage NFs. As part of the project we created the Moonlight controller, a java implementation for the OpenBox controller. We tested the implementation in our lab and showed that it can manage real live scenarios as well as extneded load on the Northbound API. The design is extendable and allows future work to be added easily. Future work can add more workflows to support end-user requests as well as new functionalities to the Northbound API such as, reloading applications while the controller is running, a native dashboard API that shows all information from the data plane and even a graphical tool that can generate NFs.

References

[1] Mininet. http://mininet.org/.

[2] Openbox controller implementation. https://github.com/OpenBoxProject/moonlight.

[3] Openbox firewall implementation. https://github.com/OpenBoxProject/MoonlightFirewall.

[4] Openbox protocol specification. http://www.deepness-lab.org/pubs/OpenBoxSpecification1.1.0.pdf.

[5] Openbox snort ids implementation. https://github.com/OpenBoxProject/MoonlightSnort.

[6] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: Enabling innovation in middlebox applications. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 67–72. ACM, 2015.

[7] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. ACM, SIGCOMM, 2016.

[8] Pavel Lazar. Design and implementation of a data plane for the openbox framework, 2016.

תקציר

מרבית רשתות התקשורת כיום מכילות כמות גדולה של מכשירים. כל מכשיר באופן טיפוסי מבצע פונקצית רשת (Network Function) יחידה (כגון Firewall) כל מכשיר מנוהל ומוגדר בנפרד.

OpenBox הינה מערכת שמאפשרת פריסה רחבה וניהול של פונקציות הרשת ע"י הפרדה בין שכבת הבקרה (Control Plane) לשכבת המידע (Data Plane) של רכיבים אלו.

ל -OpenBox שלושה מרכיבים עיקריים: אפליקציה (OpenBox Application) שמתארת את הפונקציה הרשתית אותה מבקש המשתמש לטעון, בקר (OpenBox Controller) המהווה הלכה למעשה את שכבת הניהול והבקרה של OpenBox ומופעי רשת (OpenBox Instance) שנמצאים על גבי שכבת המידע ומבצעים בפועל את הפונקציה הרשתית.

פרוייקט זה מציג את התכנון והמימוש בפועל של הממשק הצפוני (Northbound API) של הבקר. ממשק זה מאפשר למנהלי הרשת ליצור, לנהל ולוודא את תקינות פונקציות הרשת השונות שלהם.המימוש מאפשר למשתמשיו תשתית ממנה הם יכולים לבנות ולהתנסות בפונקציות רשת שונות וכמו כן מאפשר טעינה של פונקציות הרשת לתוך הבקר והעברתם אל שכבת המידע. התכנון הינו גמיש ומאפשר את הרחבתו לשימושים עתידיים נוספים.

הבינתחומי
הרצליה

בית ספר אפי ארזי
למדעי המחשב

המרכז הבינתחומי בהרצליה
בית-ספר אפי ארזי למדעי המחשב
התכנית לתואר שני (M.Sc.)

תכנון ומימוש ממשק צפוני עבור מערכת OpenBox

עבודה זו בוצעה בהנחיית **ענת ברמלר-בר**

מאת
דן שמידט

פרויקט גמר, מוגש כחלק מהדרישות לשם קבלת תואר מוסמך M.Sc.,
בית ספר אפי ארזי למדעי המחשב, המרכז הבינתחומי הרצליה

ינואר 2017