***Title**: Open interface for BGP management*

***Volume**: 1/1*

***Student**: Eduard Serra Miralles*

***Director**: René Serral Gracià*

***Department**: Arquitectura de Computadors*

***Date**: January 30, 2013*

Table of Contents

Contents

1. Introduction

Nowadays, the business logic behind Internet relationships among carriers is mainly managed by a well-known routing protocol known as Border Gateway Protocol or BGP. This EGP alike protocol is in charge of almost all decision-making processes for traffic forwarding among different Autonomous Systems all over the network. BGP has been on the Internet since its first major release which was at 1994, now it still runs on all large and not-so-large carriers as BGP version 4, which after going through more than 20 drafts, was finally codified in RFC-4271 in January 2006 and was brought to a much more near industry level of deployment.

BGP was designed to cope with the routing decisions much more like a reachability protocol than an actual routing protocol. To this extent, the routing process decisions do take into account routing policies, rule-sets, and route-maps which are not the traditional metrics used at least since then, or in most IGP routing protocols.

BGP has been implemented and offered in most likely any hardware that was targeted for actual routing in any carrier level routing environment. Still, BGP implementations (open or not) have had low level interaction, and not much innovation and further development; configuration and maintenance of this protocol have been much likely seen under CLI oriented implementations and, on top of all, last concerns regarding BGP were all about security, to prevent prefix hijacking, path hijacking, and other BGP security-related issues.

We can introduce our project by saying that we want to design and implement the first steps of a new management plane that would not only allow us to reach and configure BGP in a more

automated and visual way, but also dynamically introduce an extensive interface or open API for anyone who may need to interact with the BGP process and its information.

1.1 Motivation

The aspects that we want to take care of regarding current BGP implementations are mainly management and security related issues that will be discussed throughout the whole project. For each of them, we will focus on what options will lead to a better implementation of the protocol, as well as evaluating how much of an improvement our solution would represent.

One of our goals for this project is to open up internal BGP features to external actors in form of a clear and easy to use Open API, in order to achieve a new level of interaction between the BGP process and any entity that would want to interact with it, thus seamlessly increasing BGP functionalities while keeping BGP decision process and protocol unchanged.

1.2 Objectives

We will aim to construct an API system on top of BGP; one which should be dynamic enough to offer several capabilities in terms of management, usage and even security. We will see that the proposed design, will not only offer the foretold functionalities but will also be integrated in a non-disruptive invasive way, preserving whole process functionalities of any BGP process flow.

The term of Open API refers a set of resources that usually work under web based technologies and provide in a user-friendly manner a set of services that ease the intercommunication between

possible clients of the service and the service itself. In our specific case, the services will be offered through a RESTFUL interface.

From a management point of view, so far all alternatives dealing with BGP management involve manual CLI interaction, hindering the apparition of BGP automation processes, or even allowing BGP to adapt its behaviour to external factors. Hence, this work will aim to develop and deploy a service-like API that allows the interaction between BGP's internal process with external applications directly from and to the router, thus allowing a new level of automation between the BGP process and any service oriented design.

We will also show how through the new managed API it is straight forward to enhance BGP functionalities by managing BGP update messages, and even add standard security measures specified in many BGP security proposals, such as using ROA-RPKI[1] infrastructure.

The rest of the document is structured as follows, in the next section we describe the relevant related work to this project in the form of State of the Art, and afterwards we will specify the problems and solutions proposed to them in the Specification section. Design section then will include the software engineering decisions that will be taken in order to get the solution constructible. Then we will continue to explain how we bring to live an implementation of the specified solutions in the Implementation section.

Finally we will provide the Conclusions with a Planning and estimated Economic Analysis section, and we will end up with Future development, where we will discuss what near future could bring to our proposals.

2. State of the art

During this chapter, we will introduce, explain and briefly evaluate all the elements and subjects that interact within the scenario in which this project takes place.

2.1 Internet

2.1.1 Network Hierarchy

The Internet is a huge collection of networks maintained by many Internet Service Providers (ISPs). ISPs can be classified into different tiers based on the sizes of their networks. Tier-1 ISPs usually have national-wide backbone networks, tier-2 ISPs may have a state-wide network; and tier-3 ISPs may have an even smaller network and usually provide Internet access to end users.

Because of the size of the Internet lately, no other than a hierarchical routing approach would have seemed logical. Internet is then, distributed through a large number of AS, each of which consists of a number of routers under the same technical administration (e.g., using the same routing policy). An AS is identified by a 16-bit integer (this may have been extended to 32 bits due to its limit (65536) already, see figure 1) and usually belongs to a single ISP, although one ISP may own multiple AS numbers.

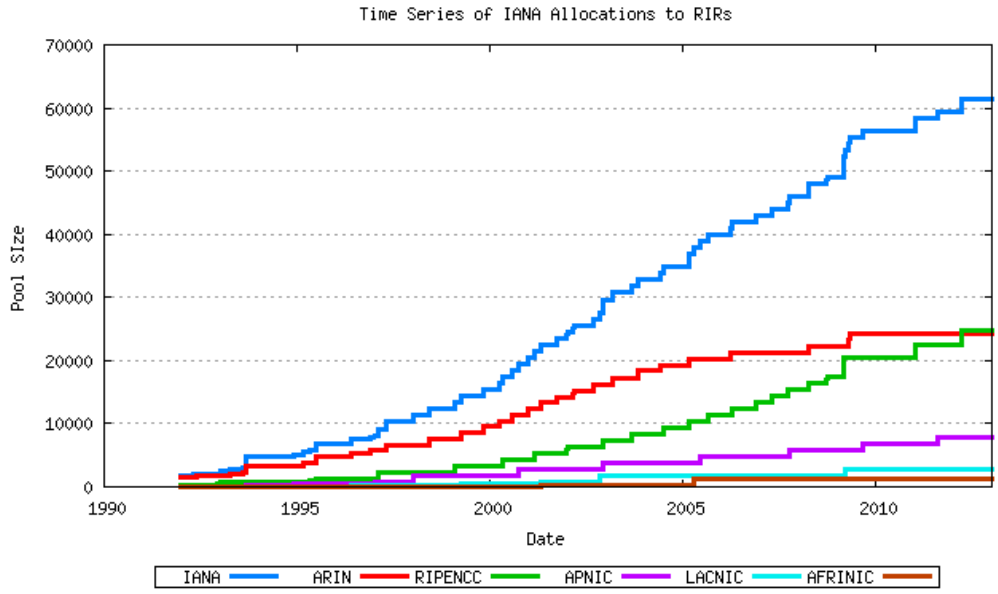

Figure 1 - Cumulative IANA AS block allocation per RIR, almost no more direction space available (Source: IANA)

The Internet Assigned Number Authority (IANA) is the central authority of the whole IP address space and AS number assignment. When the Internet was small, any organization could apply directly to IANA for a block of IP address space (or IP prefix) as well as AS numbers. As the Internet grew, it became obvious that a single authority could not handle the extremely large number of requests. As a result, a hierarchical structure was developed for IP address and AS numbers allocation, which implied the creation of the RIRs.

2.1.2 Routing Overview

On the AS level, the Internet can be drawn as a graph, where a vertex is an AS and an edge is a routing-protocol session between two ASs. Nowadays, BGP is the most commonly, if not almost the only, inter-domain routing protocol used on the Internet for exchanging routing/reachability information between ASs.

Within an AS infrastructure, Intra-domain routing protocols (e.g., RIP, OSPF, IS-IS) are used for exchanging reachability information within an AS. For example, a network in an AS to communicate with another non-directly connected network located in the same AS, an intra-domain routing protocol is usually used to discover the path between them two. Such a path usually consists of several hops through a number of routers inside the AS.

2.1.3 Routing protocols

There are two popular main approaches and one variant used by routing protocols. The two main approaches are distance vector and link state.

In a distance vector routing protocol, each node maintains a routing table consisting of a number of vectors. Each vector represents a route for a particular destination in the network, and is usually measured by some distance metric (e.g., number of hops) to that destination. Each node periodically advertises its routing tables to its direct neighbours, and updates its own routing table based on the advertisements received from others. A fair example of a distance vector protocol would be RIP [2].

In a link state routing protocol, each node advertises its link states to every other node in the network by flooding Link State Advertisements (LSAs). An LSA usually consists of a link identifier

(e.g., a subset attached to a link), state of the link, cost of the link, and neighbours of the link. Every node receives the LSAs from every other node in the network, and builds the same link state database (which is a weighted graph as each edge is associated with a cost). Each node then runs a metric-evaluation algorithm (e.g. Djikstra's shortest path algorithm) to calculate the path to the destination. OSPF[3] or IS-IS[4] are two popular link state routing protocols.

Last but not least, there is a variant of distance vector protocols that has gained strength through the years to the point to be considered a third main approach, which is path vector protocol. This kind of approach maintains the path information that gets updated dynamically over time and changes. They do, for example, store the destination, the path to reach the destination and the next element (*nexthop*) to jump to. An example of this kind of protocols is BGP.

2.2 Border Gateway Protocol (BGP)

2.2.1 Introduction

Border Gateway Protocol (BGP)[6] is the routing protocol used to exchange routing information across the Internet. It makes it possible for ISPs to connect to each other and for end-users to connect to more than one ISP. BGP is designed to scale up to current internet's topological sizes, as well as allowing nodes to have multiple connections to unrelated routing domains.

BGP first became an Internet standard in 1989 and was originally defined in RFC 1105. The current version, BGP4 was adopted in 1995 and is defined in RFC 4271.

BGP has proven to be scalable, stable and provides the mechanisms needed to support complex routing policies. Formerly speaking, we will commonly talk about "BGP", although we will actually mean BGP version 4 (BGP4). There is no need to specify the -4 version number because earlier versions have been deprecated, and very few vendors even still support them.

The Border Gateway Protocol is an inter-Autonomous System routing protocol. The primary function of a BGP speaking system is to exchange network reachability information with other BGP systems. This network reachability information includes information on the list of Autonomous Systems (AS) that reachability information traverses. This information is sufficient to construct a graph of AS connectivity from which routing loops may be pruned and some policy decisions at the AS level may be enforced.

BGP provides a set of mechanisms for supporting Classless Inter-Domain Routing[7]. These mechanisms include support for advertising a set of destinations as an IP prefix and eliminating the concept of network "class" within BGP and the whole classful addressing space

system. BGP version 4 also introduces mechanisms which allow aggregation of routes, including aggregation of AS paths.

Routing information exchanged via BGP supports only the destination-based forwarding paradigm, which assumes that a router forwards a packet based solely on the destination address carried in the IP header of the packet. This, in turn, reflects the set of policy decisions that can (and cannot) be enforced using BGP. BGP can support only the policies conforming to the destination-based forwarding paradigm.

A unique AS number (ASN) is allocated to each AS for use in BGP routing. As explained in the section above, the numbers are assigned by IANA and the Regional Internet Registries (RIR), the same authorities that allocate IP addresses.

Several years after BGP strengthened his representation on the network, some studies expected an ASN shortage in the 16-bit range around the year 2011-2012 (Figure 1). In Oct 2009, a new RFC defined a new type of a BGP extended community which carries the 4-octet Autonomous System number, increasing to 32-bit AS numbers, thus solving the problem for the time being.

2.2.2 BGP Protocol Overview

A BGP router first establishes connections with the other BGP routers in which he is meant to communicate. After a handshake that involves going through several states in their connection finite state machine, they start the exchanging route paths that fulfil through update messages.

If the router decides to update its own routing tables because a new announced path is better, then it will subsequently propagate this information to all of the other neighbouring BGP routers, and each one of them will then repeat this same procedure.

Each BGP router contains a Routing Information Base (RIB) which consists of the whole routing information maintained by that BGP node. The RIB contains three types of information:

BGP routers exchange information using four types of messages:

- Open: Open messages are used to start a BGP session by requesting that a BGP session be opened over an existing TCP/IP session.
- Update: This message type contains the actual route updates. The route updates are composed of the following (Table 1):
    - Unreachable Routes: Withdrawn routes are routes that are down or no longer reachable.
    - AS-Path Attributes: are used to provide route metrics. Along with the NLRI information are path attributes. Path Attributes allow BGP to make determinations of what is the best path. There are several categories that a Path Attribute can fall into.
    - Network Layer Reachability Information (NLRI): An NLRI is composed of a LENGTH and a PREFIX. The length is a network mask in CIDR notation (eg. /25) specifying the number of network bits, and the prefix is the Network address for that subnet.

Sections	Frame Format	
Unreachable Routes	Unfeasible Routes Length (2 bytes)	
	Whithdrawn Routes (variable)	
Path Attributes	Total Path Attribute Length (2 bytes)	
	Path Attributes (variable length)	
NLRI	Length (1 byte)	Prefix (variable)
	Length (1 byte)	Prefix (variable)
	Additional length/prefix pairs…	

Table 1 - BGP update message format

Updates received are placed in the Routing Information Base (RIB). If a route in an Update message is better than all other routes in the RIB, then that route is placed in the Forwarding Information Base (FIB).

- Keepalive: This is the packet used to keep the session running when there are no updates. *Keepalives* are sent between BGP speakers to let each other know they are still there. When a BGP router fails to hear a *Keepalive* message, it removes all routes heard from that peer from its forwarding information base (FIB).
- Notification. Used to indicate errors, such as an incorrect or unreadable message received, and are followed by an immediate close of the connection with the neighbouring router.

2.2.3 BGP Workflow

The BGP processes the incoming and outgoing updates through several modular stages that can be identified. The following diagram (Figure 2) describes the process flow.

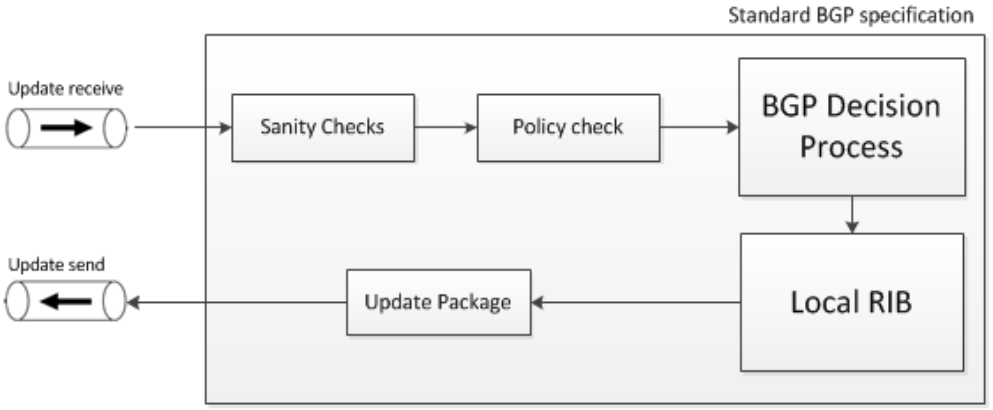

Figure 2 - BGP update management specification

1. When an update message is received, first step is to check that the message is properly formatted and contains no protocol errors. If an error is found, a BGP notification type message is sent to the sender to warn about the error found.
2. Policy check refers to particular filter implementations. Some implementations may have different filtering capabilities than other ones. Assuming there is no policy to withdraw the update message, the update would henceforth be passed to the BGP decision process.
3. The BGP decision process is the brainpower of the BGP protocol. It has the mechanisms to decide whether a route is better than another one, based solely on the attributes that are to be received with the update message and the current routes (with their attributes as well). The BGP decision process, once a route is evaluated, schedules (if needed) the update to the Local Route Information Base (RIB), which contains the whole state representation of the BGP network.

4. The Local RIB, the very same algorithm that updates its information is able at the same time to evaluate the changes and schedule an update messaging process for other network nodes that may be affected by these changes.

2.2.4 BGP Finite State Machine (FSM)

Border Gateway Protocol as defined in RFC 4271 defines what is called a "finite state model" which describes BGP's behavior at routing engine startup and during the establishment of BGP neighbor sessions. The finite-state-machine is a description of what actions should be taken by the BGP routing engine and when during the connection stage. There are six states in the model, and there are specific conditions under which each BGP state will transition to the next during the process of establishing first a TCP connection, and then a BGP session. Each step indicates a different state in the BGP session.

The following flow diagram, as in Figure 3, illustrates the BGP connection establishment:

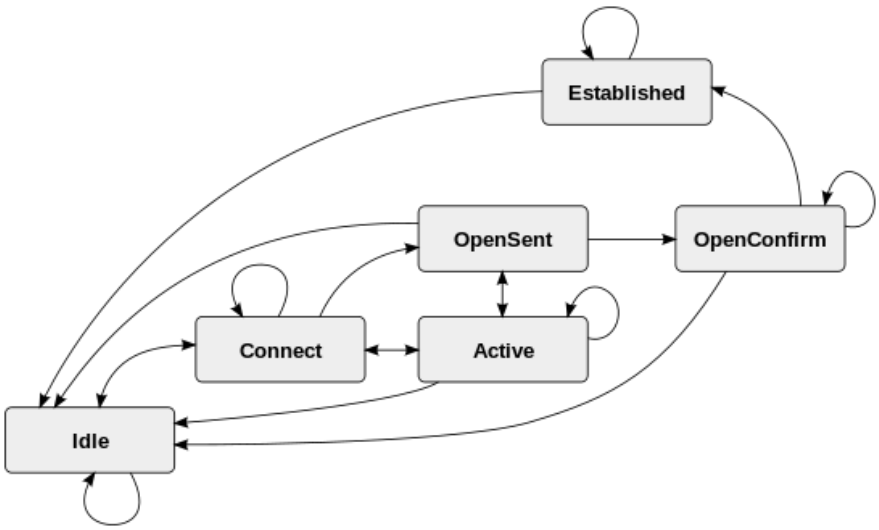

Figure 3 - BGP Finite State Machine

In this following section, we will briefly describe what every state means to the BGP process:

- *Idle:* is the initial state of the BGP Finite State Machine on startup. A BGP speaking router in the IDLE state is awaiting a session it sits in the IDLE state for an start-type event. Whenever a BGP peering session is shut down because of an error, it returns to the IDLE state.
- *Connect:* Once the BGP software and it's environment have been initialized, BGP initiates a TCP connection to the remote neighbor IP address. The CONNECT state indicates the router has awaiting the completion of a TCP connection between itself and another BGP speaking peer. The BGP Finite State Machine remains in CONNECT until the TCP three-way handshake completes.
- *Active:* The router has started the first phase of establishing a BGP session by initializing a new TCP three-way handshake to the remote router (peer) because the initial connect failed. Typically, you only see this state if you failed the initial connect. From the ACTIVE state, BGP will attempt to send another OPEN message to negotiate a BGP session. If the second attempt fails, the state falls back to CONNECT.
- *Open Sent:* At this stage, a TCP connection should be open (TCP three-way handshake completed) and an OPEN message successfully transmitted by both routers.
- *Open Confirm:* BGP confirms that the OPEN message was received, a KEEPALIVE message is transmitted and the BGP state transitions to ESTABLISHED.
- *Established: A*fter the BGP session parameter negotiation is completed, the routers begin exchanging BGP routes. It is the only state in which BGP routers can actually exchange protocol data.

2.2.5 BGP Security

The Internet is vulnerable from attacks through its routing protocols and BGP is no exception. Faulty or deliberately malicious sources can disrupt overall Internet behaviour by injecting bogus routing information into the BGP-distributed routing database. What could be considered most disturbing is that those sources of bogus information can be either outsiders or true BGP peers, so enabling a full-protection policy against anything but trusted peers may still not be enough to solve BGP security issues.

Cryptographic authentication of peer-peer communication is not an integral part of BGP. As a TCP/IP protocol, BGP is subject to all TCP/IP attacks, e.g., IP spoofing, session stealing, etc. Any outsider can inject believable BGP messages into the communication between BGP peers, and thereby intoxicate the peer-to-peer protocol session. Any break in the peer-to-peer communication may have a direct effect on routing that can be widespread considering BGP convergence context.

Moreover, tendency these last years has proved that the most crucial blind-spot of BGP in a security context is that it completely relies on and believes in the routing information sent from peers, thus authentication mechanisms which provide route announcement authenticity are being strongly persecuted by all the organizations that are directly affected by the lack of security of BGP.

Though lower layer issues threaten BGP security as much as any other, we won't address them in this work as will focus on the actual BGP operational protocol weaknesses, and discuss both how much of a threat can they be and what solutions are to be provided to solve them.

Due to the lack of security in its design and the critical *innocence* that BGP implies, there are several situations where BGP can be exploited with fatal consequences. There are two major weaknesses, which are explained subsequently.

2.2.5.1 Route and Origin authentication

BGP lacks a way or mechanism to verify if a peer announcing a path in a BGP message is in fact an authentic and legitimate path to the destination, or if the peer is even able to announce that path. The lack of route or path authentication gave birth to network attacks commonly referred as *path or route hijacking*, where an AS injects BGP messages in the BGP ecosystem in order to modify the route which was being used to get to from an AS to another, and include himself at some point in this route's path. This can be achieved, for example, removing several nodes from a large AS path to make other AS BGP nodes think the faking AS has a better, shorter path to the destination.

Origin or prefix hijacking is fairly the same tactic, but this time the AS fakes the destination (the prefix address space destination, not the path), announcing in the BGP messages that it manages the network prefixes being announced.

The results of these attacks are to be feared, since they can represent from a *naive* Denial of Service due to some misconfiguration from an announcing AS, to a malicious Man-in-the-Middle attack to steal, read, or modify the data that is to be forwarded all through the faking AS.

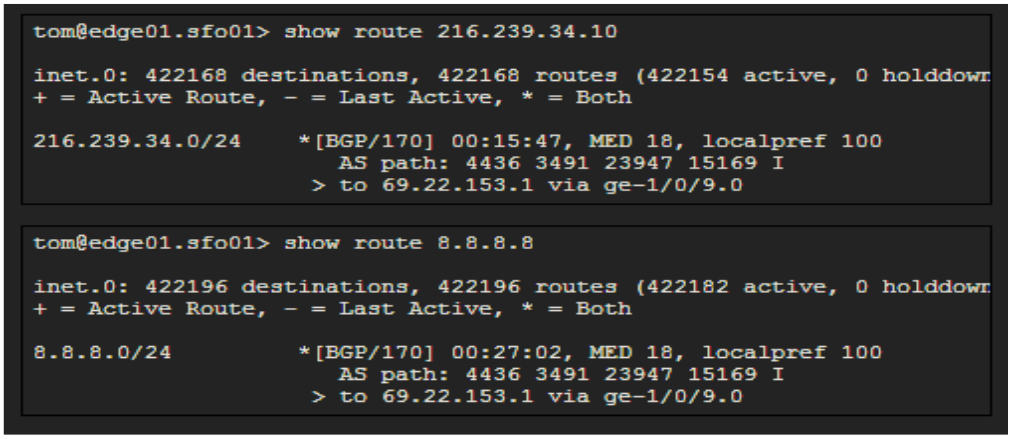

Figure 4 - "Show route" command executed on 5th of November shows prefix hijacking for Google owned prefixes

There have been quite a few examples lately, one of them recent from 5th of November, when the Indonesian telecommunications company Moratel tried to block several Google services inside Indonesia; BGP was supposed to announce the Google networks inside Indonesia using a Null route (route that in fact would lead communication nowhere) instead the de-facto one, but at the same time when doing so, a “supposed non-malicious” misconfiguration failure made the route propagate outside of Indonesia, which in the case became a route hijack for all Google service on all AS that where to get the announcement provider-wise. Consequently, Google was excommunicated from network traffic for more than 30 minutes. Figure 4 shows what a “show route” on a CloudFlare's networking company router on November 5th, the path to one of Google’s network space IP (216.239.34.10) and to their public DNS (8.8.8.8), both of them announced with an AS path that goes through ASN 23947, which is in fact Asia Pacific Network Information Centre (Moratel).

2.2.5.2 Resource Public Key Infrastructure (RPKI)

Resource Public Key Infrastructure (RPKI), also known as Resource Certification, is mainly a specialized public key infrastructure framework designed to firmly secure the Internet's routing infrastructure.

RPKI provides a way to connect Internet number resource information (such as Autonomous System numbers and IP Addresses) to a trust anchor. The certificate structure mirrors the way in which Internet number resources are distributed. That is, resources are initially distributed by the IANA to the Regional Internet Registries, who in turn distribute them to Local Internet Registries (LIRs), who then distribute the resources to their customers. RPKI can be used by the legitimate holders of the resources to control the operation of Internet routing protocols to prevent route or prefix hijacking among others. In particular, RPKI is used to secure the Border Gateway Protocol (BGP).

RPKI uses X.509 PKI Certificates[5] with Extensions for IP Addresses and AS Identifiers. It allows the members of Regional Internet Registries, known as Local Internet Registries (LIRs), to obtain a resource certificate listing the Internet number resources they hold. This offers them valid-proof of possession, though it should be noted that the certificate does not contain identity information. Using the resource certificate, LIRs can create cryptographic attestations about the route announcements they authorise to be made with the prefixes they hold. These attestations are called Route Origination Authorizations (ROAs).

Work on standardizing RPKI is currently (late 2011) ongoing at the IETF in the SIDR working group[8], based on a threat analysis which was documented in RFC 4593. The standards cover BGP origin validation, while work on path validation is underway.

2.2.5.3 RPKI/Router protocol

The RPKI/Router protocol is an IETF standardized client protocol meant to define the actual data management and communication to interact RPKI Origin Authentication mechanisms.

The Origin Authentication mechanism proposed by the IETF standards is a trust-anchor oriented architecture that would serve ROA information to local network caches in order for their local routers verify the authenticity and legitimacy of the BGP updates received.

A Route Origin Authorization (ROA) is a cryptographically signed object that states which Autonomous System (AS) is authorized to originate a certain prefix. They are

A ROA do usually contain information about a given address space such as:

- The AS Number that is authorized
- The prefix that may be originated from the AS
- The Maximum Length of the prefix

Maximum Length specifies the length of the most specific IP prefix that the AS is authorized to advertise. When it is not set, the AS is only authorized to advertise exactly the prefix specified. Any more specific announcement of the prefix will be considered unauthorized. This is a way to enforce aggregation and prevent hijacking through the announcement of a more specific prefix.

2.2.6 New BGP protocol specifications

We will proceed to introduce several different propositions to BGP that would solve, or at least, mitigate the security holes present in the protocol.

2.2.6.1 Secure BGP (sBGP)

Secure BGP design begun in 1996 [9], where a group of experts started to plan how to secure the authenticity of the transitive information that goes through BGP. sBGP then represents one of the major contributions to the study of inter-domain routing security, and offers a relatively complete approach to securing the BGP protocol by placing digital signatures over the address and AS Path information contained in routing advertisements and defining an associated PKI for validation of these signatures

In particular, sBGP architecture employs three security mechanisms:

- First, a Public Key Infrastructure is used to support the authentication of ownership of IP address blocks, ownership of Autonomous System (AS) numbers, an AS's identity, and a BGP router's identity and its authorization to represent an AS. This PKI parallels the IP address and AS number assignment system and takes advantage of the existing infrastructure (Internet registries, etc.)
- Second, a new, optional, BGP transitive path attribute is employed to carry digital signatures covering the routing information in a BGP UPDATE. These signatures along with certificates from the sBGP PKI enable the receiver of a BGP routing UPDATE to verify the address prefixes and path information that it contains.
- Third, IPsec is used to provide data and partial sequence integrity, and to enable BGP routers to authenticate each other for exchanges of BGP control traffic.

Though it’s been quite a while since sBGP came out, it hasn’t been not fully implemented nor deployed in any real carrier-grade scenario. This is due to the fact that sBGP introduces an unavoidably high processing load and memory-exhaustive consumption for every update message, which is the several signature checks for every attestation that is held into the message. Also, the

update message length is larger for every message since it has to carry a signature attestation per AS it traverses.

It is widely spread that, although sBGP does indeed solve the security holes in the BGP environment, it is resource-consuming enough for the experts to have a second thought about it.

2.2.6.2 Secure Origin BGP (soBGP)

Secure Origin BGP [10] is a response to some of the significant issues that have been raised with the sBGP approach, particularly relating to the update processing load when validating the chain of router attestations and the potential overhead of signing every advertised UPDATE with a locally generated router attestation. The validation questions posed by soBGP also includes the notion of an explicit authorization from the address holder to the originating AS to advertise the prefix into the routing system.

The AS path validation is quite different from sBGP however, in that soBGP attempts to validate that the AS path, as presented in the UPDATE message, represents a feasible inter-AS path from the BGP speaker to the destination AS. This feasibility test is a weaker validation condition than validating that the UPDATE message actually traversed the AS path described in the message. soBGP targets the need to verify the validity of an advertised prefix. It verifies a peer which is advertising a prefix that has at least one valid path to the destination.

The best feature of soBGP is that it is incrementally deployable and allows deployment flexibility in its working; BGP verifies the route originator and its authorization. New BGP message is used to carry security information and it has fixed additional scalability requirements. It uses web of trust model to validate certificate.

2.2.6.3 Pretty Secure BGP (psBGP)

Pretty Secure BGP [11] puts forward the proposition that the proposals relating to the authentication of the use of an address in a routing context must either rely on the use of signed attestations that need to be validated in the context of a PKI, or rely on the authenticity of information contained in the RIRs.

The weakness of the reliance on RIRs is that they lack accuracy for the current authenticity of the information that is represented in a route registry object. The information may have been accurate at the time the information was entered into the registry, but this may no longer be the case at the time the information is accessed by a relying party.

Moreover, soBGP states that a PKI can't be constructed in a hierarchical deterministic manner because of the indeterminate nature of some forms of address allocations. This is much a contradiction lately when the very same protocol specification proposes and assumes to use the same PKI infrastructure to map AS numbers hierarchically in a PKI context, thus assuming AS numbers can have this hierarchical PKI design path but IP prefixes do not.

2.2.6.4 Security models comparison

The proposal having most of the support from the community is the sBGP architecture, which employs three security mechanisms, Public Key Infrastructure (PKI) to support the authentication of ownership (secure origin), Digital signatures covering the routing information (AS path validation), IPsec to provide data and partial sequence integrity. In sBGP & soBGP a public key certificate is issued to each BGP speaker whereas psBGP employs common public key certificate for all speakers within one AS resulting requirement of fewer BGP speaker certificates.

Proposal	Trust Model	Topology Auth.	Path Auth.	Origin Auth.	Deployed
sBGP	Centralized	Strong	Strong	Strong	No
soBGP	Web-of-Trust	Strong	None	Strong	No
psBGP	Centralized	Weak	Strong	Weak	No

Table 2 - Comparison based on several key security aspects

As we can see in Table 2, origin authentication is much solved in sBGP & soBGP, whereas path authentication is resolved in sBGP and psBGP. Although psBGP uses centralized trust model but it is weaker solution than sBGP. Still, sBGP present several deployment issues that are yet to be solved.

2.2.7 BGP environments and implementations

BGP protocol can be brought in many different ways although it is most likely to be seen is as part of a bundle in a router's firmware. However, as for developing, researching and/or testing purposes, there are several BGP open-source implementations that can be compiled and installed in different operating systems. As our final goal will focus on implementing new functionalities in BGP, we will enumerate several of these implementations as well as briefly describe their environment.

2.2.7.1 Quagga Routing Suite

Quagga [12] is a routing software suite, providing implementations of OSPFv2, OSPFv3, RIP v1 and v2, RIPng and BGP-4 for Unix platforms, particularly FreeBSD, Linux, Solaris and NetBSD. Quagga is a fork of GNU Zebra which was developed by Kunihiro Ishiguro. The Quagga tree aims to build a more involved community around Quagga than the current centralised model of GNU Zebra.

Additionally, the Quagga architecture has a rich development library to facilitate the implementation of protocol/client daemons, coherent in configuration and administrative behaviour.

Quagga daemons are each configurable via a network accessible CLI (called a 'vty'). The CLI follows a style similar to that of other routing software. There is an additional tool included with Quagga called 'vtysh', which acts as a single cohesive front-end to all the daemons, allowing one to administer nearly all aspects of the various Quagga daemons in one place.

2.2.7.2 BGP-SRx

Since Quagga is open-source, once can grab its code and go for a different implementation; BGP-SRx[13] is a fair example of this. This modified version of Quagga is sBGP enabled and supports update packet signatures through X.509 certificates, as well as support for the whole PKI identification structure as sBGP specifies.

2.2.7.3 OpenBGPD

OpenBGPD[14] is a free implementation of the BGP protocol. It allows ordinary machines to be used as routers exchanging routes with other systems speaking the BGP protocol.

Started out of dissatisfaction with other implementations, OpenBGPD nowadays is a fairly complete BGP implementation, powering many sites. Users often praise its ease of use and high performance, as well as its reliability.

OpenBGPD is primarily developed by Henning Brauer and Claudio Jeker; OpenOSPFD also by Esben Nørby. Both are part of the OpenBSD Project. The software is freely usable and re-usable by everyone under a BSD license.

2.2.7.4 BIRD internet routing daemon

BIRD [15] is an open source implementation of a TCP/IP routing daemon for UNIX like systems. It was developed as a school project at the Faculty of Mathematics and Physics, Charles University in Prague, with major contributions from developers Martin Mares, Pavel Machek and Ondrej Filip. It is distributed under the GNU General Public License.

BIRD supports both IPv4 and IPv6, multiple routing tables, BGP, RIP and OSPF routing protocols, as well as statically defined routes. Its design differs significantly from the better known routing daemons, GNU Zebra and Quagga. Currently BIRD is included in many popular Linux distributions such as Debian or Ubuntu.

BIRD is used in several internet exchanges (for example LINX, LONAP, DE-CIX and MSK-IX) as a route server, where it replaced Quagga because of its scalability issues. According to 2012 Euro-IX survey, BIRD is most used route server amongst European internet exchanges.

In 2010, CZ.NIC (as the current sponsor of BIRD development) received LINX Conspicuous Contribution Award for contribution of BIRD to the advancement in route server technology.

2.2.7.5 XORP

XORP [16] is an open source Internet Protocol routing software suite originally designed at the International Computer Science Institute in Berkeley, California. The name is derived from extensible open router platform.

The product is designed from principles of software modularity and extensibility and aims at exhibiting stability and providing feature requirements for production use while also supporting networking research. The development project was founded by Mark Handley in 2000. Receiving funding from Intel, Microsoft, and the National Science Foundation, it released its first production software in July 2004.

In July 2008, the International Computer Science Institute transferred the XORP technology to a new entity, XORP Inc., a commercial *startup* founded by the leaders of the open-source project team and backed by Onset Ventures and Highland Capital Partners. In February 2010, XORP Inc.

was wound up, a victim of the recession. However the open source project continued, with the servers based at University College London. In March 2011, Ben Greear became the project maintainer and the www.xorp.org server is now hosted by Candela Technologies.

The XORP codebase consists of around 670,000 lines of C++ and is developed primarily on Linux, but supported on FreeBSD, OpenBSD, DragonFlyBSD, NetBSD. Support for XORP on Microsoft Windows was recently re-added to the development tree. XORP is available for download as a Live CD or as source code via the project's homepage.

The software suite was selected commercially as the routing platform for the Vyatta line of products in its early releases, but later has been replaced with Quagga.

2.2.7.6 Vyatta

Vyatta [17] manufactures software-based virtual router, virtual firewall and VPN products for Internet Protocol networks (IPv4 and IPv6). A free download of Vyatta has been available since March 2006. The system is a specialized Debian-based Linux distribution with networking applications such as Quagga, OpenVPN, and many others. A standardized management console, similar to Juniper JUNOS or Cisco IOS, in addition to a web-based GUI and traditional Linux system commands, provides configuration of the system and applications. In recent versions of Vyatta, web-based management interface is supplied only in the subscription edition, however, all functionality is available through KVM, serial console or SSH/telnet protocols. The software runs on standard x86-64 servers.

The Vyatta system is intended as a replacement for Cisco IOS 1800 through ASR 1000 series Integrated Services Routers (ISR) and ASA 5500 security appliances, with a strong emphasis on the

cost and flexibility inherent in an open source, Linux-based system running on commodity x86 hardware or in Xen or VMware virtual environments. Vyatta also provides a Cisco Replacement Guide on its website which shows various Cisco products and the comparable Vyatta/x86 solutions.

2.3 Network Management

We are about to present a brief study on the management services' tendency that is taking part as an ever-evolving solution to network management.

2.3.1 Network OS (NOS)

We refer as a Network Operating System [18] the software/firmware that controls most of the operative hardware resources inside a network element, in order to give user-level functionalities on top of it. The term of Network Operating System has gained strength and importance during this last years due to the fact that network management has been an evolving issue as the network itself has been, specially this last decade, where the Internet size and its complexity has increased overwhelmingly.

Consequently enough, requirements of internet carriers or ISP have grown as the same Internet's importance has. The flow of traffic these days is such that any carrier-level critical error can shut down hundreds of thousands of services in few seconds; this is in fact one strong justification why networking management has become so important. There have been plenty of management tools, CLI environments and management protocols that made network administration simpler, but as the networks grow, the need of better, scalable network management tools are felt somehow needed, and the software which once controlled a network element has remarkably foregrounded to the extent of becoming a whole OS to care about.

2.3.2 Network Management Systems (NMS)

There are different approaches when talking about network management systems. However, the tendency for most of them these days is to seek a centralized model where one or several administrators are able to operate and orchestrate the whole network environment visually, instantly, and with relative ease.

To achieve this, vendors started researching and developing towards this idea but, as expected, every vendor focused their own efforts on management frameworks for their own manufactured devices, which consequently, even if some of these network management frameworks do have support for non self-vendor devices, to have full support and access to all management services of a particular device one is forced to use the proper vendor-specific network management system.

Still, there are several management frameworks that do try to give an alternative to vendor-oriented specific management systems, most of the times relying on generic, universal management protocol specifications, such as SNMP. However, generic protocols sometimes do not fit all particularities of the network management needs (especially when networks require different routing protocols), thus comparing one generic NMS to a vendor-oriented one, usually the generic one lacks from many supported aspects from the specific one.

2.3.3 Information Propagation Interfaces

To configure, propagate and evaluate the information from a network could be the final objective of any NMS. Thing is, the NMS usually is a separate entity and needs to obtain network status data through some sort of mechanisms, thus this mechanisms must offer a way to obtain the data from the outside.

There are several network management protocols and variants that provide mechanisms that can be accessed through many programming languages. A protocol defines the conversation language that two elements must agree with if they want to talk. The most referred network management protocol is most possible the Simple Network Management Protocol.

If we zoom out one step backwards, instead of following the guidelines of a standardized protocol like SNMP to perform the communication, one is able to adopt a communication architecture model that allows deciding the internal protocol itself, so in terms of effectiveness one could actually choose to invent his own protocol to fit his needs.

When a network system wants to expose functionalities outside itself for others users to use them, it is commonly referred as the system provides an Application Programming Interface (API). An API could be understood as entry points that an outsider may refer to perform certain actions without the implicit knowledge nor resources that are needed to perform them.

An API may include specifications for routines, data structures, object classes, and variables and so on. API specifications take many forms, from International Standards, to particular vendor standards, or even libraries from programming languages, among others. There are many API's written in different languages that allow interaction with, for example, SNMP-enabled devices.

2.3.3.1 Simple Network Management Protocol (SNMP)

The SNMP [19] allows any external system to query and trap almost any kind of data from the router as long as the router is enabled to do so, and share it through the protocol. SNMP forms a huge network standard that has already gone through 3 versions and hundreds of RFC reviews. What's most interesting of this protocol is that it allows an extensive design in terms of what information is able to be shared through the SNMP protocol, which means any vendor can add its own particular information base that can be managed inside the network element and therefor queried and obtained through the SNMP protocol. This is done through the standard SNMP MIB definition process.

SNMP also enables support for network configuration, but as every vendor has its own methods, it has been put aside and many management teams do not even consider it as a serious configuration solution.

Still, SNMP is just a way to obtain the data directly from the router, which for large networks this is still not a feasible approach to have a one management system that has to cope with everything: the management software not only should offer an output representation of the data, and configuration options, but it also should be the one in control and store all of the data gathered throughout the network elements. The other thing is, the gathered data and evaluations may not be the final product to be shown at an administration level, but also might be the input for another application which can perform further calculations and simulations on that same data, for example.

Having said this, the conclusion is that in some cases (when felt needed, e.g. due to the network size in question this problem wouldn't escalate properly) a third layer would be put in the middle between the data gathering process and the output process to act as a managing node that would

be the one in charge of all the direct connections and information held in the real network devices: this is often referred as a proxy network element, commonly seen in real big SNMP scenarios. This has its own advantages, from avoiding several applications to connect directly to the nodes themselves, which in turn improves security, to a remarkable modularization upgrade, which directly impacts design and deployment of other modules depend on the data on top of the data-management layer.

2.3.4 Web Services and REST

Lately, there has been the apparition of a large number of different network management bundles [20] that do offer the whole management through a web-based interface (be them secured with HTTPS or not), and even offer web-based technologies APIs to perform administration network-wise; not the tendency itself is important but what it is, is that they do rely on Web-based technologies instead of any other to perform the output.

It's not a secret that web 2.0 and its applications have proven to prevail in the scenes at the moment. The fact that any internet-enabled device allows the very same data representation and management has pushed web technologies to be preferred for many programming environments, not to mention that they were always on par with Software Engineering background advances.

One of the most common API providers for Web services is the REST software architecture for distributed systems. REST-style architectures consist of clients and servers: Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources, and a resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

Key goals or aspects that REST include are:

- Scalability of component interactions
- Generality of interfaces
- Independent deployment of components
- Intermediary components to reduce latency, enforce security and encapsulate legacy systems

2.3.4.1 Extensible Markup Language (XML)

Extensible Markup Language (XML) [21] is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is defined in the XML 1.0 Specification produced by the W3C, and several other related specifications, all gratis open standards.

The design goals of XML emphasize simplicity, generality, and usability over the Internet. It is a textual data format with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web services.

Many application programming interfaces (APIs) have been developed to aid software developers with processing XML data, and several schema systems exist to aid in the definition of XML-based languages.

As of 2009, hundreds of XML-based languages have been developed, including RSS, Atom, SOAP, and XHTML. XML-based formats have become the default for many office-productivity tools, including Microsoft Office (Office Open XML), OpenOffice.org and LibreOffice (OpenDocument), and Apple's iWork.

3. Specification

During this chapter we will introduce the current situations' problems and our proposals to solve them, as well as detailing why will they be an improvement in management, security and maintaining BGP protocol transparency.

3.1 Problem statement

3.1.1 Management plane

It is clear that the efforts on networking management systems are being focused on the idea of automation and orchestration from a single resource management service, which at the same time is able to configure all system's elements in a more visual and intuitive manner.

Vendor specific management systems are being brought to live, such as Juniper NSM [22] or Cisco Prime NMSs [23], and they do develop towards the idea of not offering direct solutions, but rather providing high customization capabilities in form of API. However, such solutions are based on proprietary mechanisms that usually hinder interoperability among different vendors.

On the other hand, during our analysis of the BGP protocol implementations, we have observed that there's a prominent lack of data management capabilities from the very internals of BGP. Not only that, but also we could state that the current configuration methods rely mainly in CLI environments, and moreover, the ones that do not fall in this category often provide their

management API in an old fashioned way or even using technologies that do not fit the needs of BGP common data (such as configurable SNMPv2).

3.1.2 BGP update messages

As stated before, BGP always relies on and trusts the updates received from other peers. One important thing about the protocol is that only supports filtering for stateless kind-of data, like the AS we receive the message from and so on. It also does not offer a way to manage and see the updates that are being received in real time, which forces an administrator to infer what was received and from whom just by the state of the BGP daemon and its limited information about the updates.

In addition, even if the protocol itself allows protocol extension through headers, BGP seems internals are rigid enough to trouble the newer security implementations which most times are compelled to deal with backwards compatibility issues.

3.1.3 Security and efficiency

We have already discussed about some alternatives to secure the BGP protocol. The most complete one, sBGP, does solve almost all the current security holes in BGP, but it hasn't been deployed because the signing process per message is too much resource-consuming and suffers from scalability issues for actual carrier-grade routers.

Nowadays sBGP is still under some heavy design due to the fact that any new protocol proposal has to be BGP4 backwards compatible since the transition can't be instantaneous, which adds even more complication the problem. Also applying psBGP or soBGP might not happen since solution partial deployment would prove ineffective.

3.2 Proposed Solution

3.2.1 Web-based technologies for management

In order to bring new BGP management functionalities for external management platforms, we have been considering different alternatives. After some research we concluded that nowadays using the broadly adopted Web-based technologies is the best logical approach.

Particularly, REST is a ubiquitous technology designed to work in a client-server scenario, providing stateless querying invocation and, most importantly, simplicity of use and deployment, fitting significantly to our design path. Also, given that REST is a Web-based technology, many independent programing platforms will benefit from such API, most (if not all) of them do already implement Web related protocol support.

So therefore, our proposal falls for an implementation of a RESTful interface that will provide and expose meaningful, technology-independent, BGP routing functionalities out of the doors as web services.

We'll list the functionalities that shall be included during implementation:

1. Get RIB table from the BGP router.
2. Get FIB table from a BGP router for a specified domain address.
3. Get number of update messages being treated.
4. Get AS Path, community attributes local preference, median, and other path-related attributes for a given route.

There are several calls that shall be added later when we explain the specification for other stated problems.

3.2.2 BGP Update Message Inspection

The Watcher update messaging treatment will be one of our goals for our project. We will aim to provide a programming framework that will externally give the opportunity to treat every single update message in a fully customizable and programmable way. We will use this architecture approach to implement several managing capabilities that BGP does not currently offer, such as update filtering based on update attributes, update providers, and most importantly being able to consider external information, such as AS origination validity databases, when processing the update messages.

Considering the above features ,the Watcher shall be able to:

1. Attach to a BGP router in order to intercept the BGP update message processing logic.
2. Receive the BGP update messages that are going inwards and outwards the BGP router, and read them through the very same protocol definition.
3. Offer for each update direction (in and out) a programmable workspace where anyone willing shall be able to develop on top to it. In our case we will develop update filtering options for BGP update messages and security improvements.
4. Offer a programmable workspace to manage any kind of messages that shall be received; we will use this programmable space to treat managing and orchestration aspects for the watcher itself.

Last but not least, we want to include the managing aspects we previously cared about in BGP ecosystem, therefore the Watcher will appropriately include the ideas of open management through REST interfaces we have visited before.

The Watcher shall include configuration use cases through REST interfaces:

1. Get update messaging statics (we will move the responsibility from BGP to the watcher for this)
2. Get status (check whether it is connected and fully working or standby)
3. Turn on/off filtering options.
4. Add/remove update filters.

In addition, we will include 2 more use case calls for the former BGP REST interface that will be:

1. Turn the Watcher update messaging management on, passing the proper Watcher IP station where it runs.
2. Turn the Watcher update messaging management off.

3.2.3 ROA security improvement

Being it our last goal and since we have introduced a new managing platform that will greatly improve update messaging management, As a proof of concept we will implement a ROA mechanism on top of the watcher architecture, which will benefit from its simplicity hence reducing burden within the BGP router by offloading all the origin authentication workload.

This new addition will also be managed through the REST interface, so two new use cases will be added on the former Watcher part:

1. Turn on ROA, passing as argument the ROA cache repository IP which will provide the certified IP prefixes and AS numbers able to announce them.
2. Turn off ROA.

4. Design

The following chapter will contain, discuss and explain all the considerations that were taken into account throughout the design stage. The discussion will be focused on the description of the main parts composing the system, and most importantly on the decisions taken to make a clear and versatile implementation afterwards.

4.1 API Design

As we have discussed before, for management purposes we aim at developing an information interface subsystem able to distribute the routing. The communication architecture best fitting for our case is positively the client-server paradigm, where an administrator client running some agent will query the different routers on the system through the management node for specific internal data. Thus, making the router node an active part within the network health assessment ecosystem, opposed to legacy systems, where routers are mere packet forwarders, which only report basic status information such as link status or forwarded packets.

Figure 5 – Network scenario with a single client-server paradigm applied

This client/server model, shown at Figure 5, proposes that the interface is directly built on top of the routers resources, which assumes two premises:

1. The router must offer the some API on top of itself, using its own resources and learning to talk (thus support) the language used by this new communication interface.
2. The router must have a mechanism to manage the queries/connections that are being opened, what also implies access management.

This model would work for one node but it's far from optimal in a real network scenario:

- We have already seen that routers in general do not offer support natively for such API. Therefore, it would be needed for a router to implement such support which implies additional burden on the router processing, which in general, is.
- Any interface call made to a router node will forcefully have to be handled by it, which implies an added workload for management, also undesirable but not so easy to evade as the protocol issue might be.
- Support for multiple administration systems is still viable since our intention is to use stateless querying, but then again, administration access and its security becomes a local, node-wise issue which for larger networks is yet again undesirable.

The next alternative is an evolution of the previous one that will solve, or at least mitigate, the impact of the stated issues we had when they were extrapolated for larger networks.

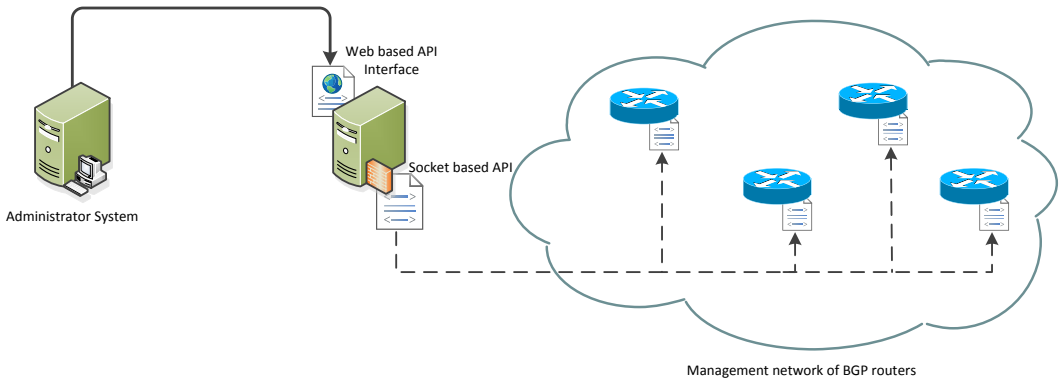

Figure 6 - Network scenario with proxy-like architecture

In the previous scenario, see

Figure 6, we exemplify a more realistic network with several routers. In this case, we will make use of a proxy-like architecture to handle the managed connections to the routers. The proxy architecture works as an element or application that acts as an intermediary for requests from clients seeking resources from other servers, providing this way resource caching, access control, service gateway, among many other advantages. In our particular case, some of the issues we had would be solved, for instance:

- Routers do no longer need to learn from new web protocol technologies that most probably they aren't supported by legacy systems. Nevertheless, a new protocol is needed to traverse the information from the router nodes to the proxy application, as we discuss later on.
- Since we have an intermediary proxy in the communication, there are several aspects that can substantially benefit:
    - *Efficiency*: caching mechanisms could be applied. Caching allows a lower ratio of calls to the end routers if they are not absolutely needed. Hence, reducing workload and increasing procedure's speed.

- *Data translation*: from the BGP's internal representation to a REST-enabled language can be moved from the router nodes to the intermediary one; this will reduce the workload on the routers as well as the requirement conditions, which in the end is an overall benefit to this model.
- *Access Control:* it would greatly enhance and ease the task of access security control and its management since it would work as a unified access service.

After this, we can safely say that this is the best approach to use in our design. We will also see in further chapters that other design and implementation aspects will also benefit from it.

4.1.1 Web-based REST interface

The REST interface will be used to request data from a management node to the element that will manage the call, see Figure 7. The managing element will then evaluate the call, see what it has to get and from which router/s, and return the proper response depending on what was the conclusion.

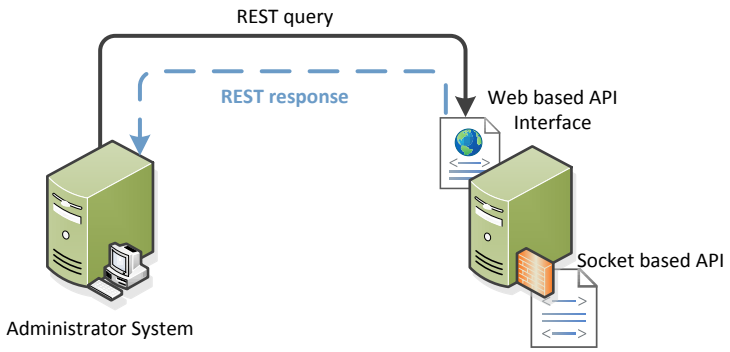

Figure 7 - REST API call procedure

Differently from the requirements that will be asked for the lower socket-based interface, this high-level interface will be implemented using a REST framework that will ease and automate the implementation task. We have no special functional or non-functional requirements, but we can observe that there will be a dependence to match the technology of the socket-based API, since the intermediate node will be the one to *translate* the REST calls to lower layer procedures.

The REST architecture defines the working aspects of the communication mechanism that will be used, but not the protocol itself. Additionally, it uses virtually the same identification rules and parameters that the HTTP[24] protocol does. Such rules always consist of several parameters split by slashes, in front of a destination address or hostname that is the one to be reached for the query evaluation.

We will proceed to define how our REST calls will be structured to be considered valid calls, what arguments should be taken and what return types shall be sent back.

4.1.1.1 REST interface protocol

First off, out of the 5 most well-known HTTP methods that are in use, we will only use 2 of them:

- GET method: We will use the GET method to indicate that the current query is asking for specific data in return, thus the response will carry payload data formatted in a proper way.
- POST method: We will use the POST method to indicate we are also passing some additional information with the query. For example, requesting the Forward Information Base out of a router, we need to pass the IP of the target.

Notice that we are not using the payload space for the requests, not even in the POST form; REST only uses the header address as the query itself, so using the POST method is just an standard REST form to indicate that we are piggybacking some additional information within the query.

Then after, we will use the following formatting style for calls:

http:// <*target*> / <*service name*> / <*operation*> / {*additional parameters*} / …

- The *http* tag and the slashes are just part of the standardized addressing of the HTTP protocol.
- *Target* will be the host in particular who this query is addressed to; in our case shall be the application node working as intermediate. Can be a hostname or the conjunction of an IP address and port.
- *Service name* shall identify the protocol we will use. This way we shall have the possibility to have more than one protocol if we get to have different types of nodes.
- *Operation* will identify the specific operation we want to execute.
- *Additional parameters* will be according to the operation rules we stated during the specification stage. There will be operations that will require them, others won't.

As for the return, the HTTP interface does already take care of the response codes for the response messages. As long as the implementation succeeds in programing correctly all the functions and operation types, the HTTP communication return types will be correct.

As for the internal protocol return types, all responses coming from the API request will come formatted properly in the payload body inside the response. The responses will come formatted using the XML markup language; As a matter of fact, most if not all the internet HTTP traffic is by *de facto* formatted in XML, which is why we'll be using it too.

We will have three types of return:

- SUCCESS: Request operations that succeed in the specified time will come back with a success label identifier.
- TIMEOUT: Meaning the request was understood properly but the layer below did not return the estimated results in the specified time-out time.
- ERROR: Will be returned if the intermediate layer is able to catch a handled error, or a non-crashing behaviour is detected during a call.

4.1.2 Socket-based interface (SBI)

The socket-based interface (SBI) represents a communication interface that will allow two-way interactive communication between two network nodes. We want it to be a light, simple, reusable network communication interface that will use the actual mechanisms of the routing environment we choose to work on, to perform the communication process.

The reason behind this decision may be part of some non-functional requirement, which would be to maintain as much as possible the routing atmosphere, preventing any unnecessary, additional workloads or added burdens for the routing environment in overall terms.

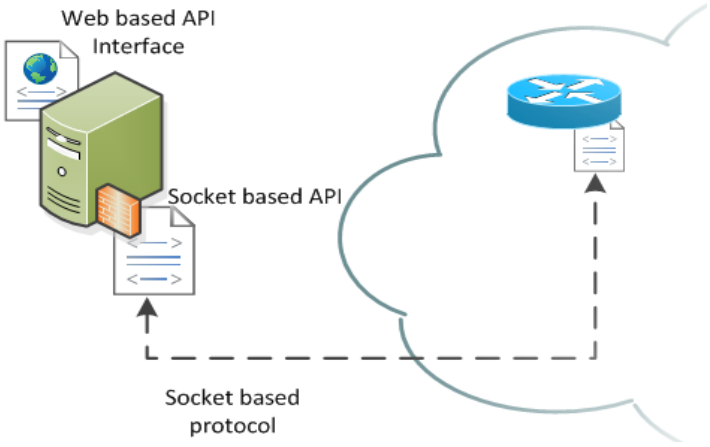

Figure 8 – Socket based API communication procedure

The aforementioned interface, see Figure 8, will work as an extensible protocol handler for further protocols that may be developed. The final idea behind its implementation is to provide an external library using the very same daemon communication mechanisms in order for other projects to include its functionalities and allow ease of development towards the idea of Software Defined Networks (SDN), without having to meddle with low level details of the router.

This Socket-based interface (SBI) will conform the communication architecture but not the protocol itself (just like REST did), we will now proceed to the discussion of the design aspects that will apply for the protocol.

4.1.2.1 SBI protocol definition

In this section, we will describe how the protocol that will operate through the SBI will handle the data types and resources that traverse through it.

As for what data formatting refers, we won't be using any standard formatting since we don't expect a router implementation's internals support a recent, web-oriented format language, such

as XML. Anyhow, even if we are yet to know the implementation's specific data structures, we will use the following nomenclature to pass protocol data messages:

[*total_size*] [*operation*][*size_of_element*] [*element*] [*size_of_element*] [*element*] …

- *Total size*: shall be the total size of the message past as parameter in bytes.
- *Operation*: shall be an identifier for the protocol to identify what kind of operation is it about, be it query or response.
- *Size of element*: shall be the size of the following element in bytes. The elements passed depend completely on the protocol definition.
- *Element*: shall be the serialization [25] of the data that is represented by the element depending on the language and object we are handling.

Return types will use the same parameter passing structure, and will use additional tags or identifiers to generate the answer. The return tag types may be:

- SUCCESS: Request operations that succeed in the specified time will come back with a success label identifier.
- TIMEOUT: Meaning the request was understood properly but the layer below did not return the estimated results in the specified time-out time.
- ERROR: Will be returned if the intermediate layer is able to catch a handled error, or a non-crashing behaviour is detected during a call.

4.2 Update Inspection Mechanism Design (Watcher)

As we already discussed previously, the main focus of this work is to evaluate and examine BGP update packets to increase Legacy BGP features, neither modifying nor extending BGP itself. All this processing will be performed by a Update Inspection Mechanism application, which we hereby name as Watcher.

The Watcher will consist on a separate application able to establish and handle a connection, through a SBI, to one or several routers. The router(s) will start forwarding the update messages that they receive or send to the offloaded inspection mechanism. For every update message the system will evaluate, as previously configured by the administrator, the validity of the aforementioned update message. Then, the Watcher will respond to the routing daemon stating the action to perform. Currently we support the following actions.

Additionally, apart from the SBI that will manage the update inspection, the Watcher will have a second SBI added for management purposes. Such SBI will use its own protocol calls that we will define later on. This second SBI will be identical to the one offered in the router nodes in terms of design, as we show in Figure 9, where we can observe an example of the final architecture.

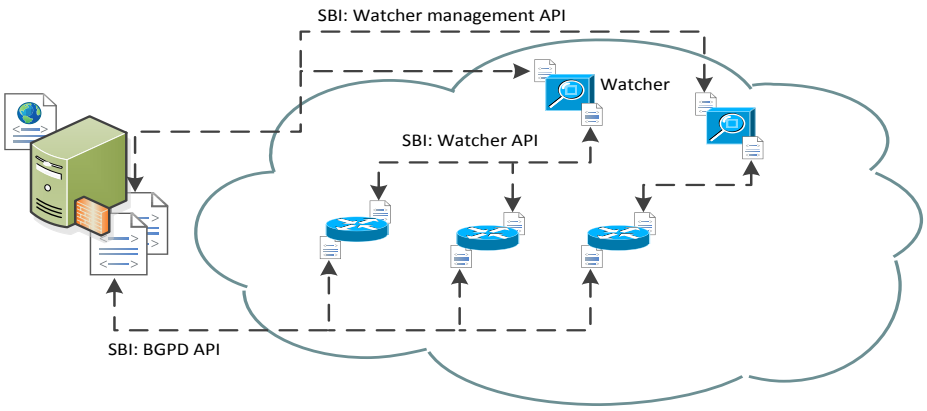

Figure 9 - Example of network using all SBI for all available nodes

The essential idea behind the Watcher is to provide a programing environment where anyone willing will be able to develop new update control mechanisms for BGP without the necessity to examine all the internal implementation of the BGP daemon. The Watcher shall be considered as a programming framework that removes all the related complication and gives an abstraction independent of the particular implementations allowing to seamlessly develop new BGP protocol functions and features.

In this project we will use this idea of *Watcher framework* and add several functionalities on top of it, like update content filtering, and finally, even a ROA service, without any modification to the BGP decision process.

Of course, adding the Watcher to the whole picture requires slight adaptations from the BGP implementation, that's why there are several changes that will have to be made on the BGP update message management. It is worth mentioning that the changes are limited to the data management but not to the protocol itself.

4.2.1 BGP internal changes

The design plans to hook the space before the update message enters the BGP decision process, and send it to the watcher through the SBI. A Packet Manager module will keep track of the updates that are sent to the watcher, as well as the arrival of a Watcher response, that in its case will wake up and check for the corresponding updates and act accordingly.

Regarding the BGP workflow model, a switching mechanism within the BGP daemon will be implemented to turn on or off the packet inspection mechanism, this way ensuring that in case there is no Watcher active, the daemon will be able to work normally. The following Figure 10 illustrates how this mechanism will be placed in the BGP process flow.

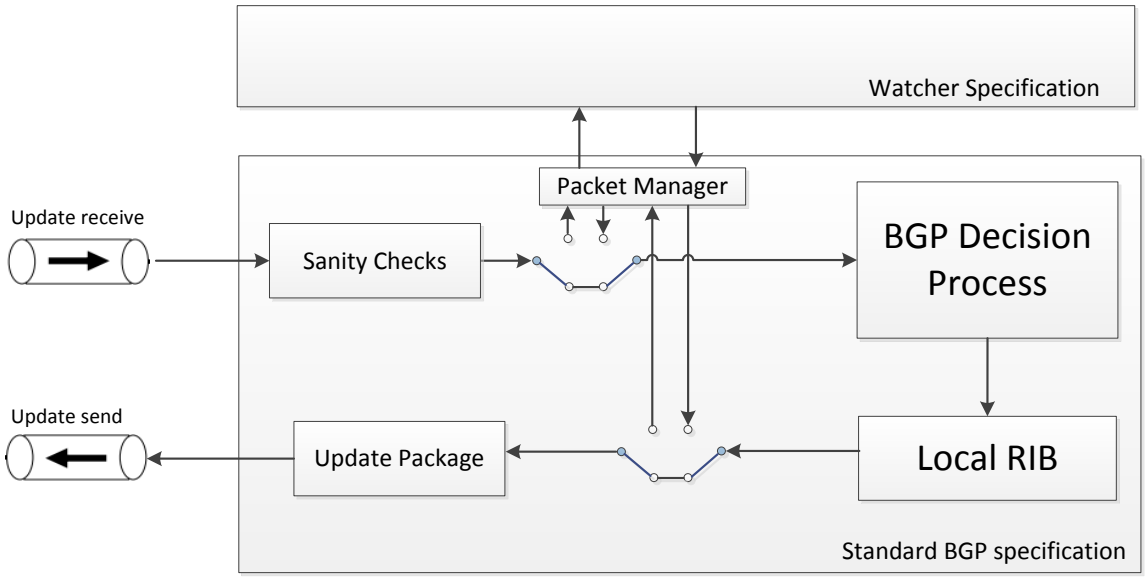

Figure 10 - Watcher switching mechanism in BGP process flow

This approach gets a full programmable and transparent interface on top of BGP, decoupled from the main routing process, and without touching its core intelligence, which is the BGP Decision process.

In fact, there are several important reasons why we should not touch the decision process:

- Compatibility: it ensures backward compatibility to other BGP specifications, and even cross compatibility to other new specifications since the layer of intelligence is added out of the workflow.
- Modularity: achieving our objectives without touching BGP core intelligence makes our approach more decoupled than other specifications, which at the same time makes it easier to re-implement them on a different approach.

The following flowchart diagram (Figure 11) further details this design decision:

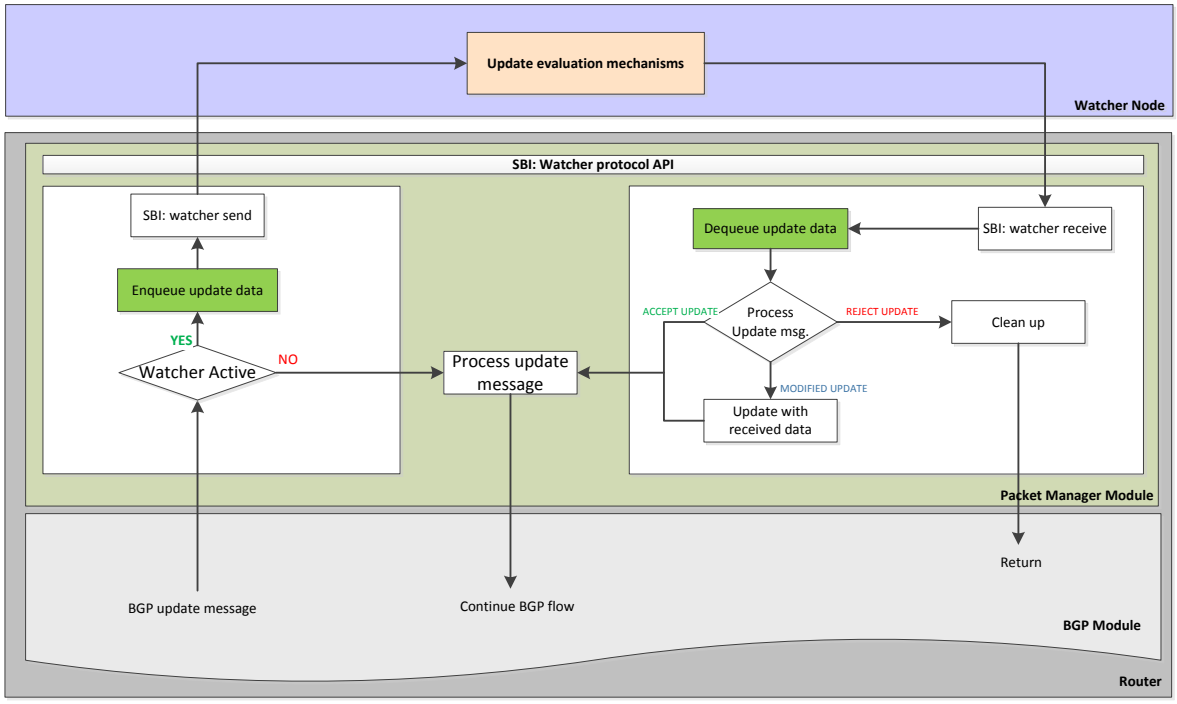

Figure 11 - Proposed design for BGP-Watcher interaction

Watcher will support three types of return: Accept update, Deny update, and Modified update. Each and every one of them will be further specified in the next chapters.

4.2.2 Watcher protocol design

The Watcher will have one SBI interface with two different purposes: first one will correspond to the API to fulfil the BGP update message inspection interaction, and the other one, will provide a stateless API for message querying for management, just like the one we want for the BGP implementation.

This protocol API will be the one in charge of receiving the connections from the former BGP routers, manage the update messages that may arrive traversing it, and finally sending back the result to the router again.

Regarding the protocol formatting, we will follow the aforementioned SBI specifics, so we will have two types of message operation codes from the BGP router:

1. *Update receive*: will indicate that the update was received from another router.
2. *Update send*: will indicate that the update was to be sent to other routers.

For both of them, the full serialized update will be expected to be passed right after the operation code with its size.

Our model proposes three types of response from the watcher API once an update package has been evaluated.

1. *Accept update:* indicates that the update content passed the Watcher policies so it should be also accepted in the BGP daemon. Notice that this doesn't mean we take for granted the content of the update, nor modify the tables directly depending on the content; this just states that the update shall pass to the BGP decision process.
2. *Rejected update:* indicates that the update content did not pass the Watcher policies and therefore, should be taken out from the BGP process flow.

3. *Modified update:* indicates that the update was somehow modified by the Watcher, so the Watcher is expected to have sent the modified version together with the response. This means the update is to be read again, and afterwards treat as it was an *Accepted update*.

As for the management API, we will take the same standards as the BGP approach has.

4.3 Route Origin Authentication on Watcher Design

To conclude the design, we will focus our efforts on a Route Origin Authentication system that is to be implemented on top of the Watcher itself. As the Watcher provides a plain programmable interface, the design will be straightforward and rather dependant on the implementation decision we make at its time.

In our specific case, we will be deploying a RPKI-Router protocol client to validate the BGP update legitimacy on top of the Watcher.

For every update message the watcher receives and ROA is on, we will have to perform a query to ask the validity of the prefix announced by the AS originator of the announcement. Return types for RPKI/Router are:

- VALID : when the route announcement is covered by at least one ROA
- INVALID: two cases:
    - When the prefix is announced from an unauthorized AS. This could either mean that there is a ROA for this prefix for another AS, but no ROA authorizing this AS; or this could be a hijacking attempt.
    - When the announcement is more specific than is allowed by the maximum length set in a ROA that matches the prefix.
- UNKNOWN: The prefix in this announcement is not covered (or only partially covered) by an existing ROA.

5. Implementation

5.1 Router Implementation

Quagga will be the BGP-enabled routing daemon implementation we will choose to construct our solution model onto. Quagga is an open-source implementation of the GNU Zebra project under GNU Licence, what consequently makes our implementation distributable under GNU Licence if we use it. Quagga includes several modularized protocol implementations, such as IS-IS, OSPF, and of course BGP.

Additionally, Quagga's architecture has a rich development library to facilitate the implementation of protocol and client software with consistent configuration and administrative behaviour, which is fitting as there are design parts in our model that depend directly on the implementation's provided internal support.

Quagga is programmed in C language hence any changes made into its implementation shall be programmed with its language.

5.1.1 Quagga-based communication library (LIBQO)

As the design planned, we are going to use the internal Quagga communication functionalities to develop an exportable, easy-to-use, communication library to develop client and server side communication protocols through.

This library will focus on the interaction model that BGP counts on. As we commented, BGP communication works through a Finite State Machine (FSM) standardized protocol which takes care of the data transmission aspects, which includes link state and implementation version control.

Our idea is to export the FSM implementation far enough to allow another application easily include its functionalities and communicate to each other.

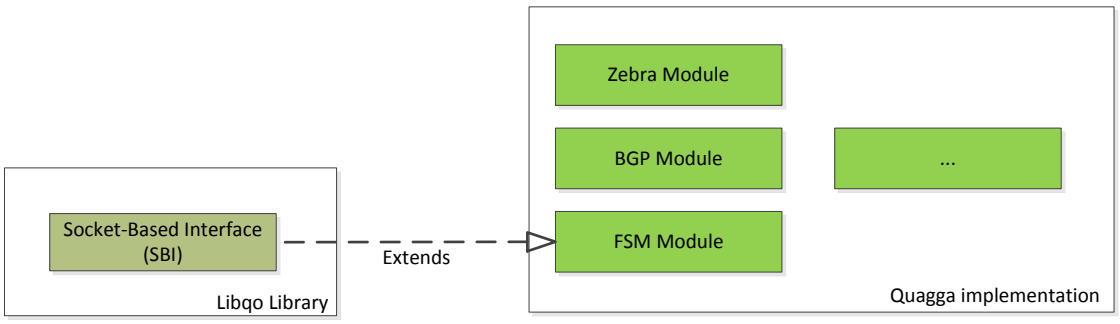

Figure 12 - LIBQO communication library

In Figure 12, we show how the software packages will be modelled in the beginning. This new library will allow the implementation of standalone applications able to interact with other ones using the same mechanics.

5.1.2 The BGPD Interface (Client and Server)

Once we have our own Quagga-based communication library, we will proceed to implement the BGP management interface. We will use the external library to create the new communication layer able to manage the socket-based requests thrown to Quagga.

At implementation time we decided it was better to implement the whole protocol (client and server side API's) into the external library for non-functional reasons, like implementation time,

reusability, or even further development purposes. The following module diagram, as in Figure 13, shows the module implementation:

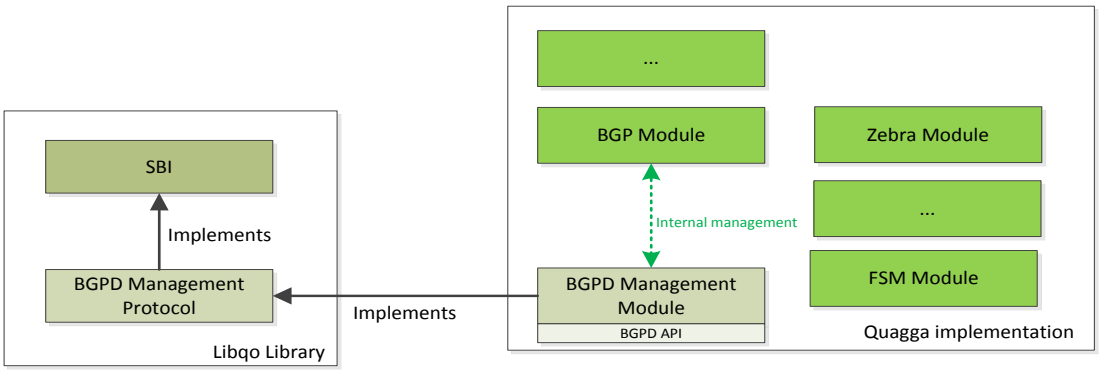

Figure 13 - BGPD Management Module and API

The BGPD Management Module implements the server side API of the BGPD Management Protocol. Within the BGPD Management Module, there are implemented all the management functions: the ones that retrieve data from the BGP Module, and the ones that configure different BGP aspects. All the management functions implemented use the available BGP Module resources in a neither invasive nor disruptive way, therefore securing the stability of the Quagga daemon.

As for what the BGPD client-side interface is, we will implement concrete functions in the library that will allow direct calls to a BGPD API, just by invoking them. The result is to be one of the specified for SBI based calls. The additional data that might be returned as a result of a BGPD call will be put under outgoing parameterized address space, and will come properly formatted using data representation classes, such as *routeSets* or *routes*, that are to be included in the library.

Some of the calls included in the client-side API are:

result_ret ***getRIB****(destination, timeout, return_space):* if returns success within the specified *timeout* time, will leave in *return_space* the complete list of routes returned by the router *destiation*.

result_ret ***getFIB****(FIB_target, destination, timeout, return_space):* if returns success within the specified *timeout* time; will leave in *return_space* the forwarding table for the router *destination*.

result_ret ***getStatistics****(destination, timeout, return_space):* if returns success within the specified *timeout* time; will several statistic counters referring to this router state, such as uptime, state, RIB size, among others.

result_ret ***setWatcher****(watcher_address, destination, timeout):* success if within the specified *timeout* time, otherwise error or timeout. Sets the connection to the watcher's address up in ortder to enable the update message instpection mechanism.

result_ret ***unsetWatcher****(destination, timeout):* success within the specified *timeout* time, otherwise error or timeout. Shuts down any possible active watcher mechanism.

5.1.3 The Watcher Client interface

The Watcher interface will be implemented in a similar way as the BGPD Management API was. We will first implement the protocol module, client and server side API's, in the LIBQO library using the SBI current implementation, and then implement this last one in one separate module in Quagga.

Differently from the BGPD Management Module, the Packet Manager Module is client side; therefore, there is no available SBI API for it. The decision to connect to a Watcher is then made at the BGPD module and not the other way around.

The final implementation design will be as in Figure 14:

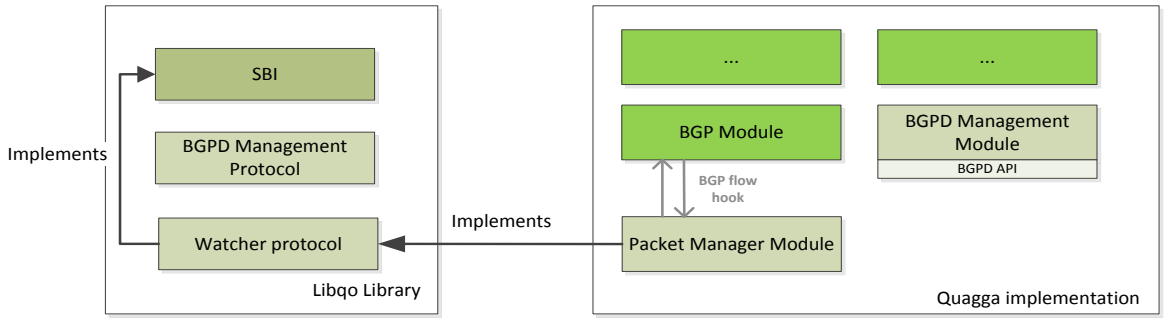

Figure 14 - Watcher Packet Management Module

The changes discussed in the Design stage for the BGP process in order to be able to divert the update messages' path were implemented in the most non-disruptive way possible. The implementation included as much as several lines of code in the already existing BGP Module within Quagga which later, if Watcher enabled condition remains true, the Update message workflow jumps to the Packet Manager workflow which will be in charge of the update transaction process and its return.

5.2 Watcher Implementation

5.2.1 Watcher Server Interface

The Watcher will be a standalone application and will be implemented in the same C language in order to support the recent implemented functionalities we added in the external LIBQO library.

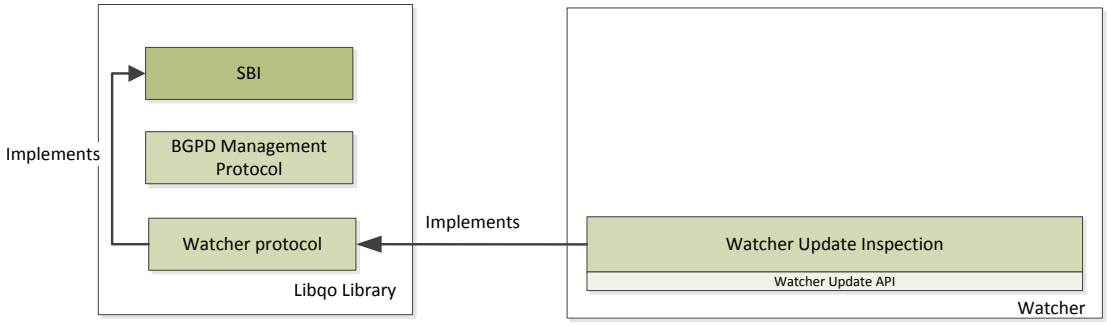

Figure 15 – Module Implementation of Watcher server-side

The Watcher Protocol module in the LIBQO library is designed to offer an implementable server-side API that will allow the deployment of a whole Watcher server application instantly. Its implementation is meant to offer an event-driven interface that will provide a function hook to examine concurrently each and every update message received and return its evaluation value.

Additionally, the data types included are easy, uncomplicated, but without renouncing to their full signification, thanks to the several packages inside the module that are meant to help treat the data in a more straightforward manner.

The flowchart graph in Figure 16 represents the final implementation representation for the former Watcher server-side API:

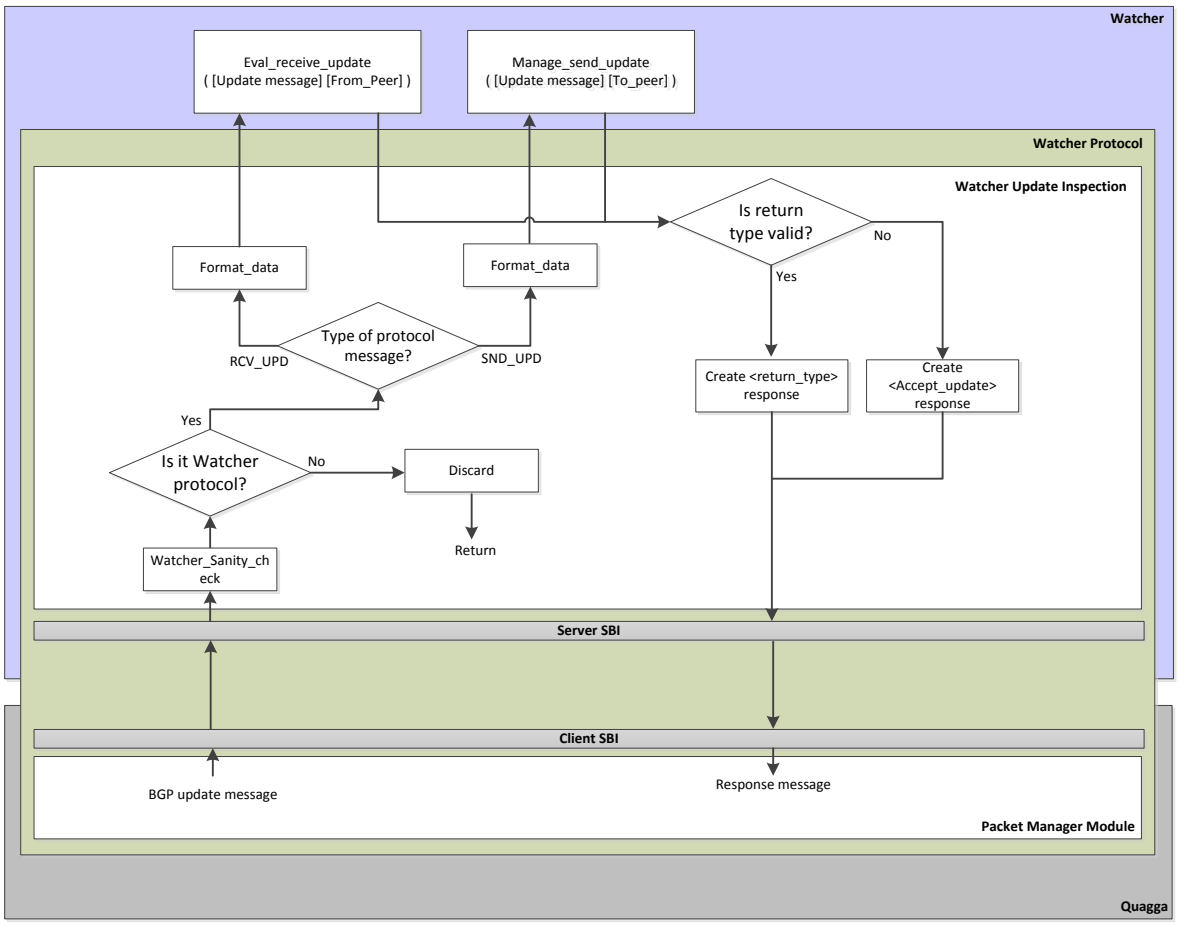

Figure 16 - Implementation flowchart for Watcher Server

As can be observed, the actual evaluation functions for the updates are totally decoupled with the Watcher model, and due to this, it makes it fitting and compatible with any other technology that could be applied in its context.

5.2.2 Watcher Management Interface

Since the Watcher Management Interface must be server-side, we'll use the same interface space that is already deployed within the Watcher protocol to allow a third type of protocol message, which in the case will be the *management type* message.

After such message is received, the Watcher will get the rest of the message parameters and pass them without formatting to the previously configured administration function hook.

The following implementation diagram in Figure 17 presents the activity flow for a management query. It is important to note that the other two process paths have been removed for clear view of the management query process flow.

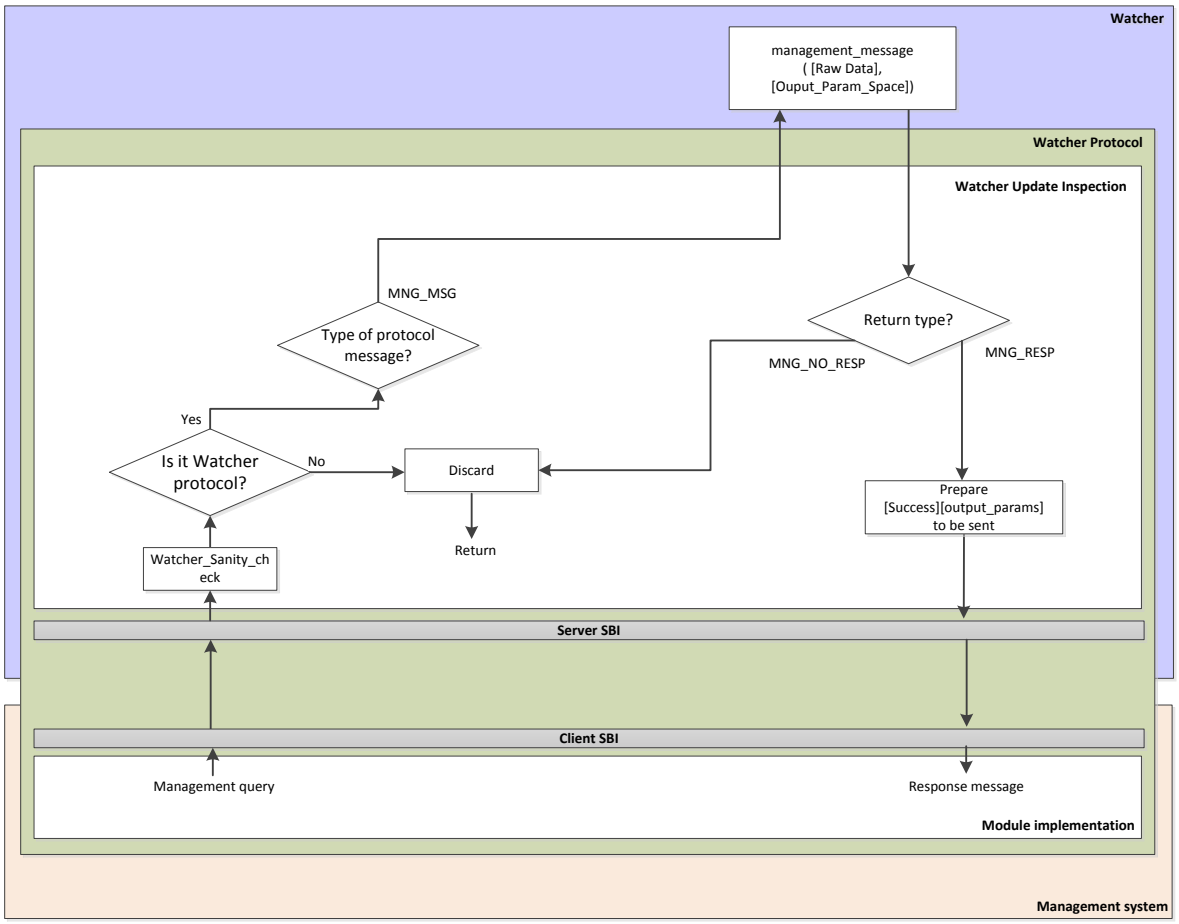

Figure 17 - Implementation flowchart for Watcher Server (Management interface)

As can be appreciated, for the former part of the Watcher management API, we won't have such specific calls for in its layer, such as a *get* function to get some data, but instead we provide the means to do so. Therefore, we will only have one Watcher management API call.

result_ret ***sendManagementPacket****(destination, timeout, mng_message_space, return_space):* returns *success* if the call was received and evaluated at the Watcher, *timeout* or *error* otherwise. Sends *mng_message_space* to the Watcher, and if succeeds, return value shall be stored upon *return_space*.

Yet, and as an example, we will implement several additional calls on top of the management interface of our watcher, using the very same API specifications and return types:

- *Get*-type API requests*:*

result_ret ***getStatistics****(destination, timeout, return_space):* success if within the specified *timeout* time, otherwise error or timeout. Return will be the update counters for the watcher, being them divided in two categories, *Incoming* and O*utgoing* updates, and for each one of them *Accepted*, *Rejected*, and *Modified* update counters.

result_ret ***getStatus****(destination, timeout, return_space):* success if within the specified *timeout* time, otherwise error or timeout. Return will indicate the status of the Watcher (if is in use or not by any element).

- Watcher filter management*:*
    - Filter functions will consist in a very simple implementation of a filtering module that will allow filtering for two directions (incoming and outgoing directions) and several types (*ASN, Withdrawal prefix, Nexthop, and NLRI prefix*).

result_ret ***setFiltersOn****(destination, timeout):* success if within the specified *timeout* time, otherwise error or timeout. Turn on custom Watcher filtering capabilities.

result_ret ***setFiltersOff****(destination, timeout):* success if within the specified *timeout* time, otherwise error or timeout. Turn off custom Watcher filtering capabilities.

result_ret ***newFilter****(destination, timeout, filter_type, filter_direction, filter_element):* success if within the specified *timeout* time, otherwise error or timeout. Tries to create a new filter with the specified filtering options. Nothing happens if it already existed.

result_ret ***removeFilter****(destination, timeout, filter_type, filter_direction, filter_element):* success if within the specified *timeout* time, otherwise error or timeout. Tries to erase filter with the given filtering options. Nothing happens if didn't exist such filter.

- RPKI/Router (ROA) management:

result_ret ***setROAon****(RTR_local_server, destination, timeout):* success if within the specified *timeout* time, otherwise error or timeout. Turns on Route Origin Authentication through the RPKI/Route mechanism, getting the ROA entries at the *RTR_local_server* specified.

result_ret ***setROAoff****(RTR_local_server, destination, timeout):* success if within the specified *timeout* time, otherwise error or timeout. Turns off Route Origin Authentication through the RPKI/Route mechanism.

5.3 Unified Service Provider Implementation

The network node will act as a service gateway for REST queries asking for network-wise routing information, and henceforth will make the decisions where to get the data from and how to serve it. To get the data however, the intermediate server will have in use the LIBQO library and its managed API's to access the routing environment information through our new socket-based information transport.

However, we must take into account that it also has to be able to serve the REST requests from the layer above, which means the implementation of this application will be two sided.

Since LIBQO is C written, we have studied several C frameworks that allow the creation of REST-oriented services, and we finally have decided for the Apache Axis2/C implementation.

Apache Axis2/C is a Web services engine implemented in the C programming language, used to provide and consume WebServices. It has been implemented with portability and ability to embed in mind, hence could be used with ease as a Web services enabler in other software. Additionally, Axis2/C provides XML support, which will be fitting for the result values of the REST interface

The Axis implementation module will be constructed as follows in Figure 18:

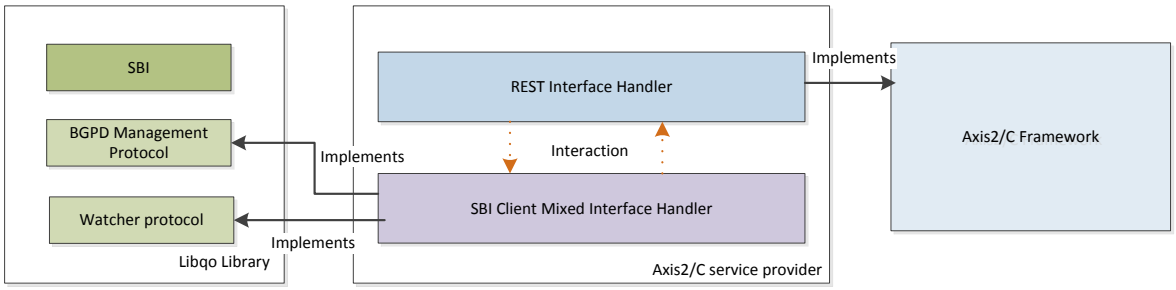

Figure 18 - Module Diagram of the Unified Service Provider Module

5.4 RPKI/Router ROA implementation

To implement the ROA mechanism on top of the Watcher application, we will use an open library that offers the RPKI/Router standard, namely the RPKI/RTR protocol.

Introducing the RTRlib, is a lightweight, C written library. It implements the RPKI/RTR protocol for the router end and the proposed prefix origin validation scheme. The RTRlib provides functions to establish a connection to a single or multiple trusted caches and to determine the validation state of a prefix/origin AS mapping.

The Watcher will provide the API management functions to turn on or off, and even specify from which trusted cache it may learn the ROA entries from. Figure 19 shows how the implementation module diagram would end up to:

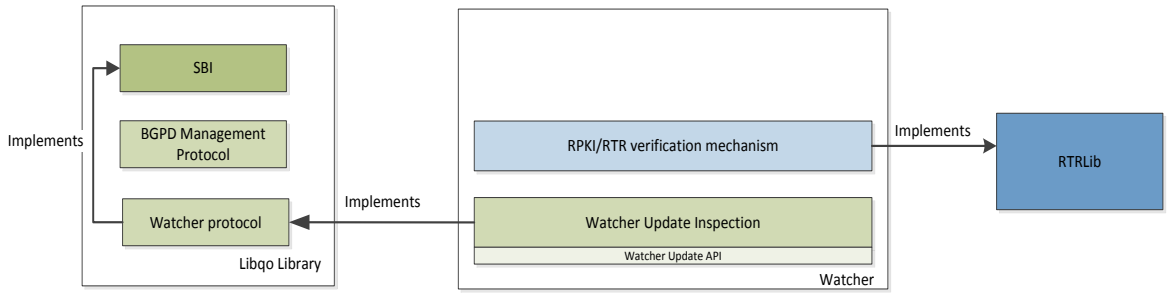

Figure 19 - Watcher Module Implementation Overview with RTRlib

Additionally, we will add a proprietary and configurable decision process for the RPKI/RTR mechanism that will allow to decide what to do for every return type, which means that a RTR valid verification for a given prefix does not necessarily have to return an *Accept Update* Watcher return message to BGP but it will be configurable, and so does apply for the Invalid and Not Found RTR entries.

6. Test and Results

To test our solution model we will arrange a previously-defined environment and test its functionalities to its fullest.

6.1 Testbed preparation

We will make use of virtualization to prepare several BGP-enabled Quagga implementations running on a computer. This way we will simulate a small BGP network to work upon, since for example, the Update Inspection mechanism needs BGP network traffic to actually be tested.

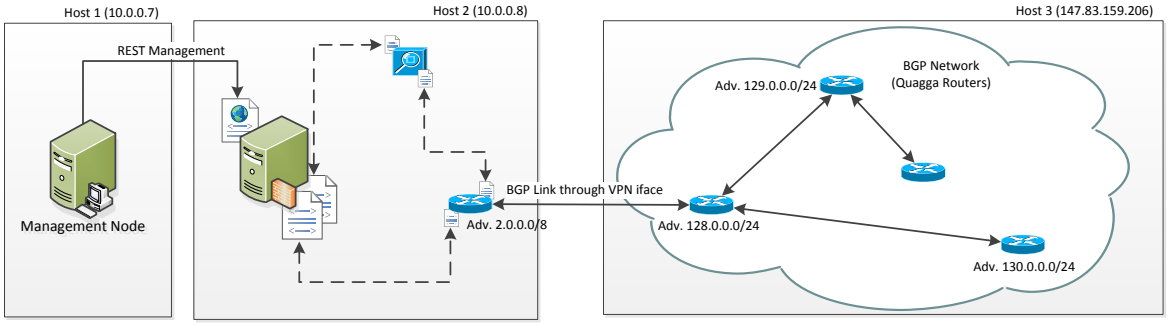

Figure 20 - Testbed preparation for first scenario

Figure 20 shows the preparation of the testbed we are going to use. We will put under test 2 action flows that will be explained and evaluated in the following sections.

6.2 Management through REST interface

We will first show how a whole Open management system of our project can be managed through REST interfaces. The BGP peer relationship will be all configured from the start, but the BGP link status from the Quagga router in Host 2 to the BGP network in Host 3 will be shut down in the beginning for testing purposes. Once we have finished setting up the Watcher and configuring the filtering options, we will turn on BGP Link to eavesdrop the Updates messages in the watcher interface, and after ask for its update statistics.

The flow will go as follows:

1. BGP in Host 2 Is started. Since it has no connection to no BGP network, it will remain Idle, stating that the BGP neighbours configured are unreachable.
2. We will start the Watcher daemon and the intermediate service unifier (Axis 2/C Service provider), they both will remain Idle.
3. We will then query a getRIB message to the Host 2 Router. To test the commands we will make use of a Web-based REST client to have a better view of the output. The command we will issue will have this parameters based on the network configuration we have:

   ```
   http:// 10.0.0.8:9090 / services / quagga_openapi / quagga / getRIB ? dst=127.0.0.1
   ```

   This command will issue a *getRIB* operation command to the *services/Quagga_openappi/quagga* REST service interface; additional parameters are to define the target of the getRIB command, which for now since it's running on the same host will be the localhost address.

The response we get is what is seen in Figure 21:

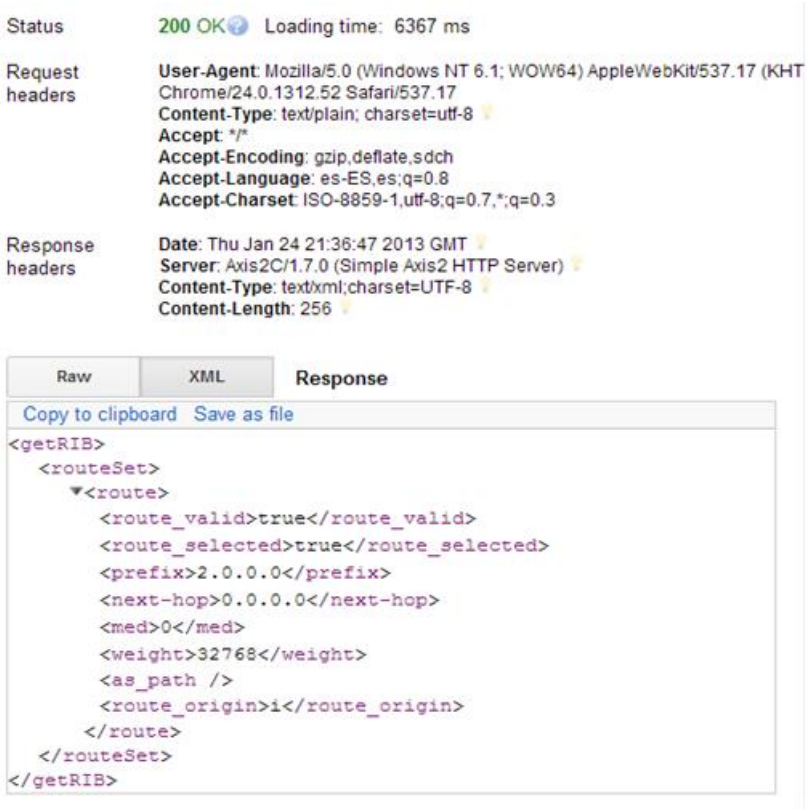

Figure 21 - Result from REST call getRIB

During the time's call, the Axis Server contacted the router in Host 2, established a SBI based connection, and finally asked for the full RIB table held in Quagga. Then, BGPD module within Quagga accessed the BGP legacy module and read its internal RIB table, to send it back to the AXIS server. Finally the Axis server translated the data and sent it back XML-formatted to the calling REST client.

As it can be observed, the whole specific data is returned in XML language, so there is no data loss due to the protocol specification because it was design to be able to do so.

- Next step is to configure the Watcher daemon. To do this we will issue the command:

http:// 10.0.0.8:9090 / services / quagga_openapi / quagga / setWatcher / 127.0.0.1?dst=127.0.0.1

The address next to setWatcher is the Watcher target, which indicates the address of the Watcher daemon. As in our case resides in the same host, the target address is the same one as the destination of the query. The return is as follows on Figure 22:

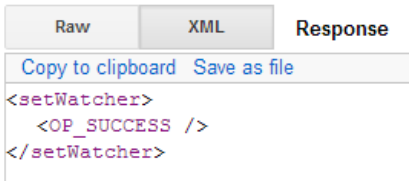

Figure 22 - Return from setWatcher

Still, it doesen't mean they are connected already. To verify this, we will issue a command to the Watcher Management API to check if it is already running accordingly:

http:// 10.0.0.8:9090 / services / quagga_openapi / watcher / getStatus?dst=127.0.0.1

This time the service is *watcher* and not *quagga* since, of course, we are actually talking to a Watcher. The return of this call will return a 0 or 1 depending if it is running or not. In this case the return will be one, hence the Watcher is connected to BGP and ready to catch the update messages.

- The RPKI/Router mechanism and Watcher-side filters are by default turned off, so we will proceed to turn on the VPN between the Host 2 and Host 3 and enable the BGP link.
- The watcher is prepared to throw onscreen the updates that are being treated at the moment. We will see that, for every update, it will print out all its available data and then, return the *Accept Update* message to BGP. The following figure shows the output for both daemons:

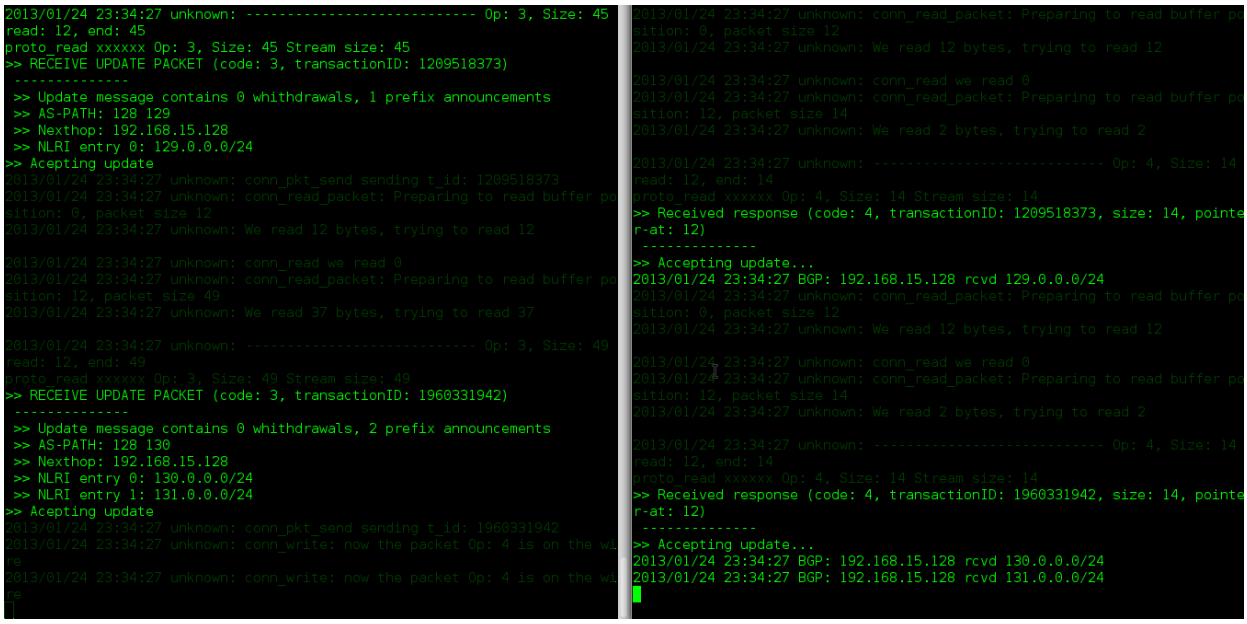

Figure 23 - Output for Watcher and BGP daemon respectively

For clearness purposes, we have obfuscated unnecessary SBI protocol output in Figure 23. The output window on the left represents update messages being received at the watcher. Notice that once we have finished reading the update attributes, we send the *Accept Update* return.

On the other side, the messages are being received (looking at the transaction ID, they are the same for both daemons), and afterwards they are accepted, sending them to their legacy BGP process flow.

- Once this is done, the BGP network will have converged in no time since it is a small network. We will use the Watcher Management API to request its update statistics.

```
http:// 10.0.0.8:9090 / services / quagga_openapi / watcher / getStatistics?dst=127.0.0.1
```

The result is as follows in Figure 24:

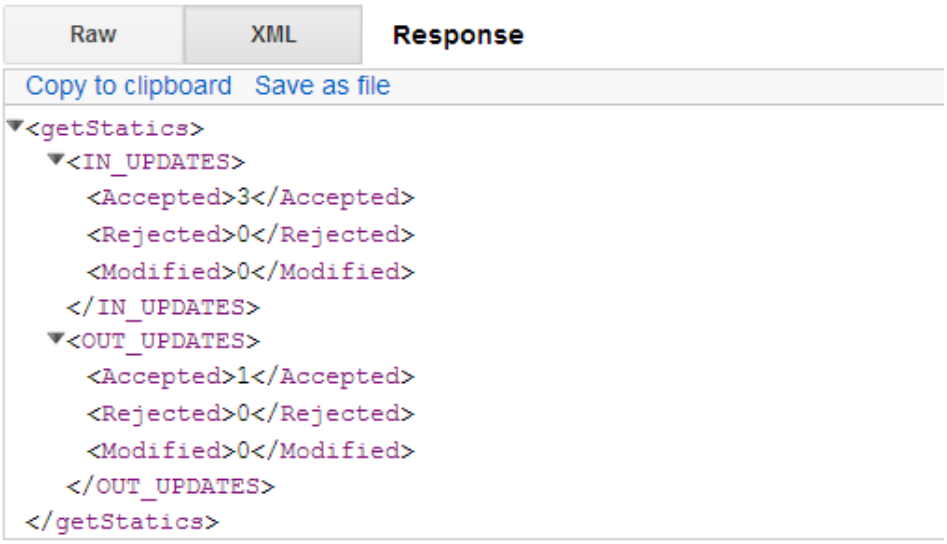

Figure 24 - Get Statistics from Watcher REST call

We can conclude that the Watcher has successfully evaluated all the updates incoming and outgoing from the BGP daemon and the rest interface is working as expected.

We will proceed to a more interesting scenario where we will apply the RPKI/Router algorithm to prevent Route Origination Hijack.

6.3 Preventing an Origin Hijack attempt

From this point, we will assume that we got to the point where we were about to turn on the VPN to enable the BGP link. We will do this as we do not want to repeat ourselves through the configuration stage.

The scenario this time will be slightly different, as shown in Figure 25:

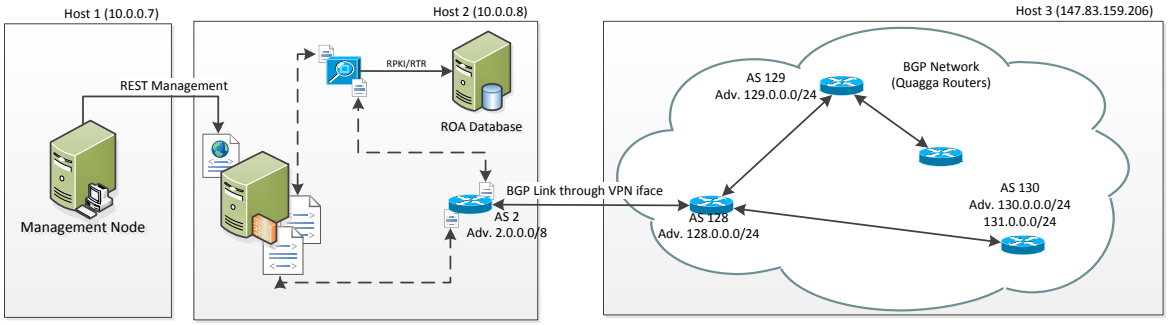

Figure 25 - New security-enabled scenario to test RPKI/RTR

We will deploy a little RPKI database from RIPE NCC that will allow the creation of a local cache of ROA entries, in them we will manually put the ones we are using in out prepared environment.

- Before turning on the link, we will first issue a Watcher command in other to anchor him to our newly deployed ROA cache through the RPKI/RTR protocol. To do this, we will use the REST call:

  ```
  http:// 10.0.0.8:9090 / services / quagga_openapi / watcher / setROAon / 127.0.0.1
  ```

  Again, the address refers to *localhost* since the ROA application runs in parallel inside Host 2. The return of this call shall be *Success*, but more interestingly, Watcher will output the routes learnt from the RPKI/RTR server just as follows in Figure 26:

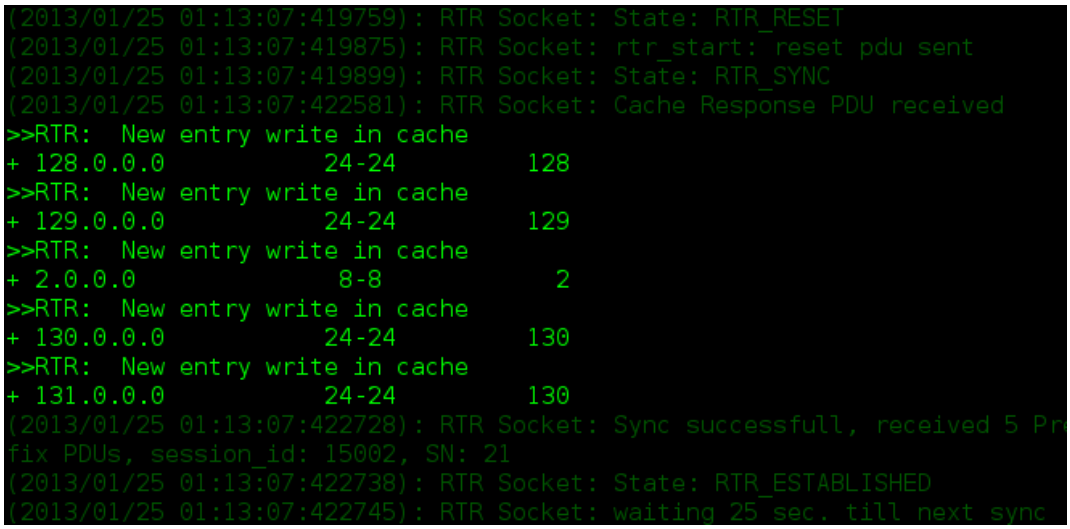

Figure 26 - RPKI/RTR protocol in action, learning ROA entries available

Watcher is now ready to evaluate calls using the ROA mechanism. We will now connect the VPN to establish the network and see how it works.

- In next step, we open the VPN link. The Watcher reports activity for several update messages seen in Figure 27:

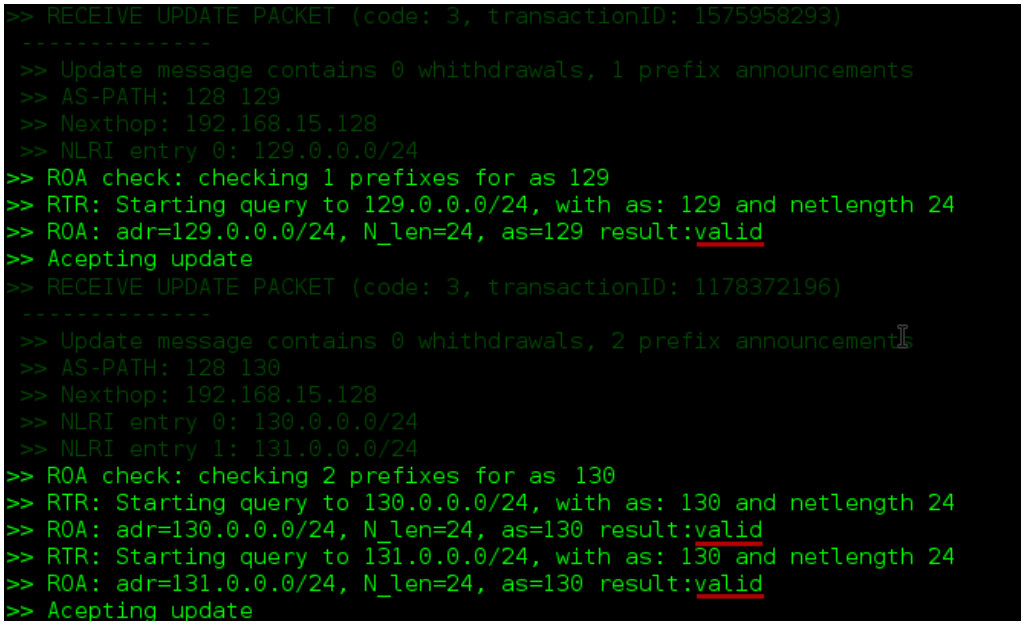

Figure 27 - RPKI/RTR in action, validating NLRI entries from update messages

- As it can be observed, the local cache is able to validate the origination legitimacy of the prefixes being announced in the updates. The default action for *Valid* update origin NLRI prefixes is to accept the message.

  If we query the Watcher for the update message statistics, we will see all of them where validated hence they were all accepted, therefore the results will be the same as the first scenario.

We will now introduce in router representing AS 129 a hijack behaviour, allowing AS 129 announce a network it should not have at all, and owned by one of the other routers in the network, as in AS 130.

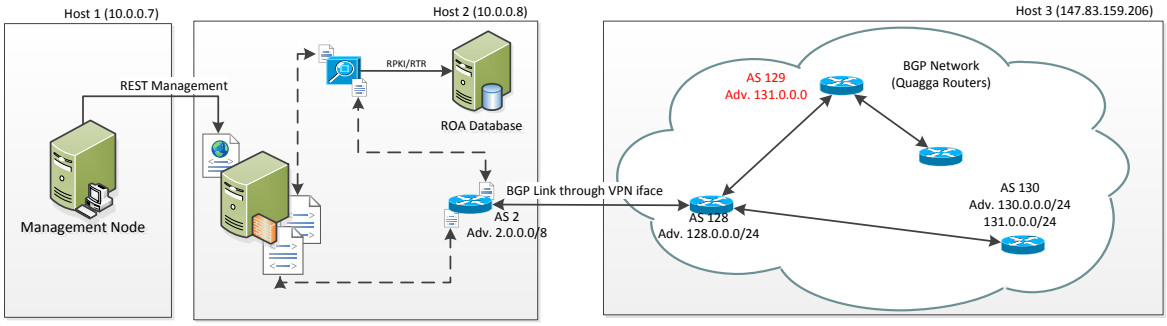

Figure 28 - AS 129 will try to announce a network not owned

- In the BGP network configuration file for router AS 129, we will introduce the prefix 131.0.0.0/24 which is being currently announced by AS 130, and then restart the daemon (we could have done it through CLI as well). Once we have this node ready, we turn on the VPN link, and await for the watcher response, which in seconds will show the results:

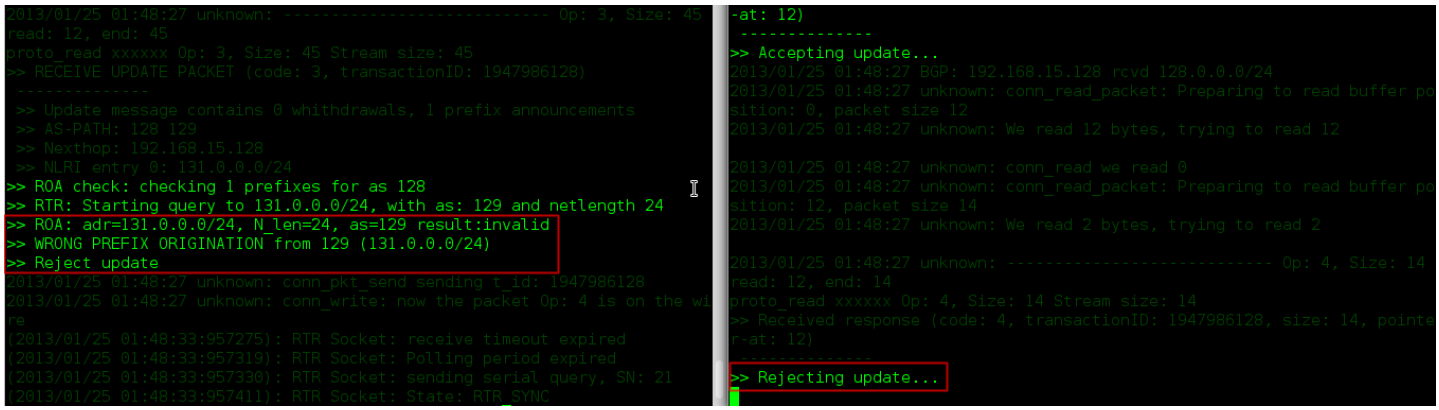

Figure 29 - ROA check-up detects the prefix hijack attempt

- As can be seen in Figure 29, the RPKI/RTR check correctly verifies that the prefix coming from AS 129 is announcing a network that it should not be announcing. Therefore, it filters the update and asks Quagga BGP daemon to do the same.
- Finally, if we check the statistics again of the update messaging on Watcher, we will clearly see that one of the updates was filtered, in this case, because it was not a legitimate announce from the AS.

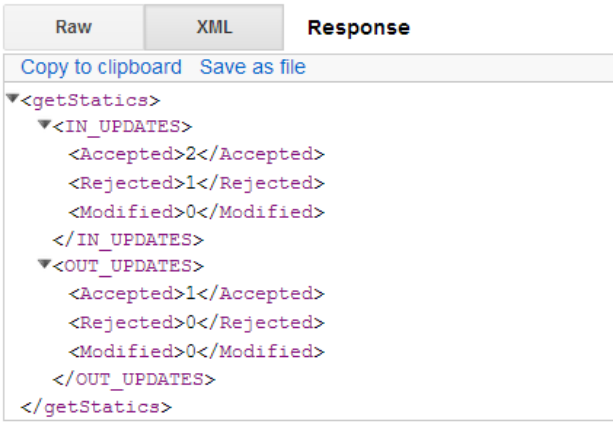

Figure 30 - Statistics shows the update was indeed rejected

7. Planning and Economic Analysis

7.1 Planning

In this part we will detail the overall planning for this project. As it will be observed, the required design was always on par with the research stage, where some of the decisions were rectified along with the design.

- February 2012:
    - Project starts
    - Definition of project scope
    - Objectives at short and long term definition
- March 2012:
    - Research on REST services operation mode, compatibility, available frameworks
    - Design starts, intermediate model is defined, soon will start some implementation tests to deploy the framework
- April 2012:
    - Research on AXIS 2/C usage
    - Design and first iteration of the REST interface
    - Research on Quagga's legacy internal design for short-term development

- May 2012:
    - REST interface development undergoes.
    - Design for a SBI protocol starts. Research focuses on Quagga's FSM connection model.
- June 2012:
    - REST interface is nearly done; the implementation target is switched to focus Quagga and SBI transfer protocol.
- July 2012:
    - The Update Inspection Mechanism is defined; design towards this idea undergoes.
    - The implementation of the SBI starts a modular approach in order to allow other protocols (ie. Watcher) use the same communication specifics.
- August 2012:
    - The SBI for BGPD is nearly done, it will advance towards the Watcher now; Quagga still needs more refinement to interact with BGP's internal data.
    - First watcher approach comes to an actual implementation in C. SBI based communication is not defined at all yet.
- September 2012:
    - BGPD calls for the managed SBI is done, and the internal calls to the BGP module do also work as intended.
    - Research and design stages are done already; efforts will now focus on finishing the implementation.
- October 2012:
    - Watcher SBI protocol implementation as well as Watcher API for its ease of deployment undergo.

  - Project report starts at this time.

- November 2012:
  - Watcher SBI protocol for update interception is nearly done. Still, the Watcher management protocol is yet to be implemented as such.
- December 2012:
  - All functions for the whole management plane are implemented. Heavy testing during second half of December to bugfix several automation issues.
- January 2013:
  - Finish project memory, test of whole case scenario, evaluate results.

As it can be observed in the following Gant diagram in Figure 31, what took most time was definitely the implementation of the socket-based protocol in conjunction of the Watcher standalone and the BGPD Module inside Quagga. Because each of these two elements' implementations were always strongly related to each other, the implementations stages of both of them were always tied as well.

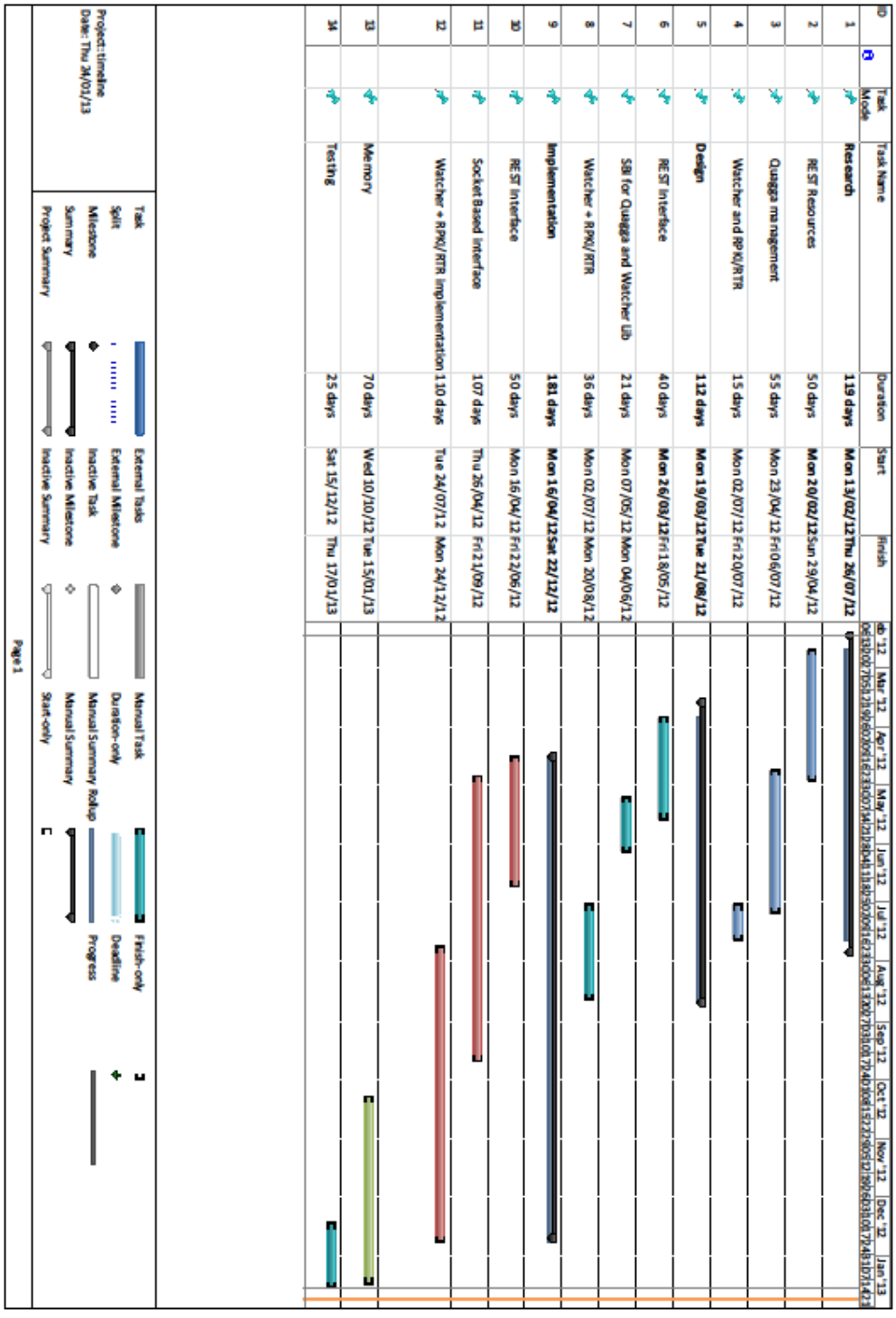

Figure 31 - Gant diagram for this project's planification

7.2 Economic analysis

We will proceed to estimate the cost of the implementation in a real-world environment. In order to do so, we will approximate the cost based on the hours dedicated to the design and implementation stages.

It must be noticed that, most of invested time in this project were shared across different tasks and was not always the same per day, which is not actually reflected in the Gant diagram. Therefore, an absolute calculation on the Gant hour tasks would render inappropriate. To do the estimation then, we will use merged work time of the design and implementation, and suppose that the design took 25% of the total time whilst the implementation the remaining 75%.

Approximate total working time: 800h

	Hours	Price per hour	Total
Design	200	50€	10.000
Implementation	600	40€	24.000
Total			34.000€

Table 3 - Estimation of price per hours dedicated in this project

8. Future Development

This section will cover the ideas and paths opened by this project's development. There are to main branches the development may undergo through, which are the Open management interfaces, and the Update Interception Mechanism development.

8.1 Open management interfaces

We have proved that open management interfaces, such as our Open REST interfaces, greatly enhances the opportunities of network management and implementations beyond. As the API is *Open*, any development process would easily be able to construct their own API entries to fulfil their needs, eliminating from the very beginning variations and divergences that can occur due to the vendor-specific development specifications.

The Open API system we propose in fact is no different from any OS system that works towards the idea of driver-based communication specification. We haven't gone that far in this project, but in future work, it would be practical to have distributed network operating systems that are able to communicate to any network element through driver-oriented default interfaces, indifferent from the vendor type or software implementations beyond the actual driver.

8.2 BGP protocol ease-of-development

Since now, any specification for a new BGP implementation meant internal changes which were strongly related to the implementation of BGP it is being developed upon. In our case, the Watcher architecture proposes a new framework were to start implementing from scratch without the need of modifying or even threatening the current BGP specification.

The Watcher framework most important aspect is that it allows a whole new level development on top of BGP completely decoupled with the whole protocol itself; it opens the door for free development for BGP additions entirely agnostic of the very same BGP implementation itself.

The future development options for the BGP framework are almost limitless, and can work towards different aspects of the protocol like security or even BGP-specific functional aspects, such as convergence time or new route propagation algorithms.

9. Conclusions

This project has proved that open interfaces would render useful for network management systems if they are open enough to allow different implementations in a system-wise environment.

- As for the REST interface, it is offered in high-level web-based interface; as long as the proper documentation is provided, almost any management system is be able develop and deploy an API system fitting its needs thus allowing the use of the REST interface to interact to the whole network scope in a decoupled and secure way.

  Since it is REST based, new web-based management systems are would not even have the need to develop such API.
- As for the SBI interface in the network nodes, due to its modularity, it solves the whole communication process, and eases the task of developing for functionalities on top of the network nodes. It is also offered the possibility to re-implement the API for deployment on other BGP enabled devices, as long as they provide its code or API's to perform the same implementation jobs.
- As for the Watcher, it offers what we think the most interesting aspects as for what future development refers. The watcher is a fully functional framework that is easily deployed and offers straightforward functions to manage its capabilities. It works as a middleware layer that offers its capabilities rightful API, without the need for a developer to meddle with the actual implementation at all.

In overall terms, we are happy with the results as they provided exactly what we were looking for from the very beginning. We hope this project will serve for future developers to test new BGP implementations and achieve better security standards than the ones we have right now.

10. References

[1] - M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph (September 2005) RFC 4158, "Internet X.509 Public Key Infrastructure: Certification Path Building", IETF

[2] - C. Hendrik (June 1988) RFC 1058, "*Routing Information Protocol* , The Internet Society"

[3] - Moy, J. (April 1998) RFC 2328, "*OSPF Version 2,* The Internet Society. OSPFv2"

[4] - D. Oran (February 1990) RFC 1142, "*IS-IS protocol specification" (IETF variant)*

[5] - C. Adams, S. Farrell (March 1999) RFC 2510*, "Internet X.509 Public Key Infrastructure: Certificate Management Protocols*"

[6] – D. Meyer, K. Patel (January 2006) RFC 4274, *"BGP-4 Protocol Analysis"*

[7] - Fuller, T. Li, J. Yu, K. Varadhan (September 1993) RFC 1519, "*Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation"*

[8] - Secure Inter-Domain Routing Working Group homepage

[9] – Stephen Kent , Charles Lynn, Karen Seo (2000): Secure Border Gateway Protocol (S-BGP) at IEEE Journal Vol. 18

[10] – Albaro Retana (2003): Secure Origin BGP (soBGP) at *North American Network Operator's Group (NANOG) and Cisco Systems*.

[11] - Tao Wan, Evangelos Krankis, P.C. Van Orshot (2004): Pretty Secure BGP (psBGP) at *Internet Society*

[12] - Quagga routing suite homepage

[13] - BGP-SRx project homepage.

[14] - OpenBGPD project homepage.

[15] - BIRD project homepage.

[16] - XORP project homepage.

[17] - Vyatta enterprise official homepage.

[18] - Further information on Network Operating Systems can be found here.

[19] - M. Fedor, M. Schoffstall, J. Davin (May 1990), RFC 1098 Simple Network Management Protocol (SNMP): historical source.

[20] - Recent examples of this could be Nagios, OpenNMS, Zabbix, OPSview, Accelops, among others.

[21] – Motorola, J. Reagle, D. Solo (March 2002), RFC 3275 "(Extensible Markup Language) XML-Signature Syntax and Processing"

[22] - Juniper NSM software specifications can be found here.

[23] - Cisco Prime is in fact a set of resources that include a NSM software model, but they offer different services, like real-time cloud computing services, mobile network management, and so on. More information can be found here.

[24] - Hypertext Transfer Protocol (HTTP): The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web. Official IETF RFC can be found here.

[25] – Serialzation specification for interfaces can be found here.