RIPE NCC

RIPE Network Coordination Centre
www.ripe.net

Hogeschool van Amsterdam
Amsterdam University of Applied Sciences

Hogeschool van Amsterdam
www.hva.nl

Replacement of RIS Route Collectors

February - July 2014

W.A. Miltenburg
RIPE NCC
GII team
Supervisor RIPE NCC : C. Petrie
Supervisor HvA: D. van der Meer

Student information:
W.A. Miltenburg
Student number: 500617246
+31 (0)6 538 08 698

School:
Hogeschool van Amsterdam
Information Technology & Computer Science
System and Network Engineering
Duivendrechtsekade 36-38
1114AD Amsterdam-Duivendrecht
The Netherlands
+31 (0)20 595 1610
Supervisor: D. van der Meer
School period: 2013-2014

Company:
RIPE NCC
Global Information Infrastructure (GII)
Singel 258
1001EB Amsterdam
The Netherlands
+31 (0)20 535 4444
Supervisor: C. Petrie

This bachelor thesis was written during my graduation project, second semester, at RIPE NCC from the beginning of February till the first week of July 2014.

Preface

This bachelor thesis is part of the graduation project of Wouter Miltenburg, which was from February till the end of June. During the graduation project I have worked for the RIPE NCC and worked on the project ‘Replacement of RIS Route Collectors’.

In this preface I want to thank my supervisor Colin and the GII manager Romeo for their support and help during this project. It is not only these two people that I want to thank for their help, but also the rest of the GII and R&D team, people who I have interviewed and the rest of the RIPE NCC. I also want to thank the community of RIPE, which uses and supports the RIS project, and makes all of this possible.

The last person I want to thank is Douwe from the Hogeschool van Amsterdam. Not only for the effort and the help that I got during this project, but also for the help during my school period.

Wouter Miltenburg

May 2014

Table of Content

Summary

This document is part of the graduation project of Wouter Miltenburg. The topic of the graduation project is replacing the current Routing Information Service (RIS) collectors that the RIPE NCC provides as a service to its community. It is a system that collects the routes, which are used in different systems for further processing. The main question that the RIPE NCC had was formulated into the following:
“How can the current implementation of the RIS Route Collector be replaced with a better alternative, aimed to process updates faster, make information easier to integrate in the RIPE NCC Hadoop storage backends (e.g. through XML, JSON, or YAML), and meet the gathered requirements?”

The current RIS collectors, based on Quagga, involve a high delay and are facing its limits with the current number of peers. It is therefore necessary to research a possible alternative system that could replace the current implementation. The research stage began with gathering all the requirements from different involved parties. These requirements are listed in a MoSCoW scheme as a prioritisation scheme and the research was based on finding a possible alternative that could meet the requirements. During the research, a possible alternative was created that was based on ExaBGP for the BGP connections. ExaBGP would create JSON formatted messages, indicating which prefixes it had received and what the state of the peering relations were. These messages would be saved in a queue were different consumers could consume the messages from. Since the old system was based on Quagga, which produced MRT RIB and update dumps, it was also necessary for the new ‘alternative implementation’ to support this feature. Different consumers should be created to support all these features.

This possible alternative was chosen as the alternative that met most of the requirements. It resulted in a development stage where different producers and consumers were created. The producer would receive messages from ExaBGP and insert them in the queueing mechanism, which was decided to be RabbitMQ. These consumers would consume messages from the queues and insert the messages in the Hadoop storage back-end. Another consumer would maintain the state and generate MRT formatted RIB and update dump files.

The third stage of the graduation project was to deploy the prototype in a test setup. This test setup would be used for testing the integrity of the whole prototype and to see if the state reflects the state of Quagga. The RIB files that would be created by both Quagga and the prototype were compared to see if the integrity of the files can be guaranteed with the prototype. The conclusion of this test was that ExaBGP can satisfy the same level of integrity as Quagga.

The prototype is therefore a possible alternative that, with some more testing, meets the requirements that were gathered during the research stage. It needs some more testing to test the throughput and scalability, but it is indeed the alternative that the RIPE NCC was looking for.

1. Introduction

This document is part of the graduation project of Wouter Miltenburg. The graduation project is part of the study Information Technology & Computer Science at the Hogeschool van Amsterdam (HvA), from February till the first week of July 2014.

This document gives an overview of how the final result has been created. This document will also give an overview of which methodologies were used during this project and the decisions that have been made during the graduation project. The stages that were involved in this project are being described and the research that has been done is added as an appendix. The research report itself will be discussed in this document, but if the reader wants to have a better understanding of the research, or want more information, it is recommended to read the whole research paper. Also, the development and deployment of the prototype will be discussed in this paper and the results that came out of this prototype.

The topic of this graduation project is the Routing Information Service (RIS) that the RIPE NCC provides as a service to its community. To be more specific, this project is about replacing the current implementation of the RIS Route Collectors that collects the routes, which are used in different systems for further processing. It is a system that was launched in 2001 and started with the idea to collect and store Internet routing data from several locations around the world. This data, which is collected and saved, can be accessed via RIPEstat and can be used for further analysis. RIPE NCC also performs analyses on the data that is collected with the RIS project and publicise the outcomes via RIPElabs. The Remote Route Collector (RRC), as its name might reveal, is the system that collects all the Internet routing data from several points around the world and allows other applications or systems to retrieve this information.

As more peers want to connect to the RRCs, it also takes more and more time to process all this data. This can cause some problems: people have to wait longer for the data, analyses have to wait because of the delay, and when IP hijacking occurs you do not know it immediately.

The first chapter will begin with some background information about the RIPE NCC and the project. A description of the current implementation is given in the second chapter, which also describes the problems that the RIPE NCC is facing with this implementation. The chapters thereafter will describe the research for possible alternatives and the research methodology that has been used during the research. After the chapters that outline the research stage, the proposal will be described and outlined. The next two chapters will outline the development and research stage. The last chapters will be the conclusion of this project and recommendations for a future project.

2. Background

This chapter gives the reader a general overview of the assignment and the reason why this project has been initiated. This chapter provides the reader with information, which might be used in subsequent chapters.

2.1. Organisation

The RIPE NCC is one of the five Regional Internet Registries (RIRs) in the world and is providing its services to Europe, the Middle East and parts of Central Asia. A RIR allocates and oversees the registration of Internet number resources in its service region. Internet number resources are IPv4 addresses, IPv6 addresses and Autonomous System (AS) numbers.

It was established in the year 1992 as a not-for-profit organisation and has grown ever since. There are current 128FTE working for the RIPE NCC. It also provides technical and administrative support to RIPE, a forum which is open to everyone who is interested in the technical development of the Internet.

RIPE NCC currently provides the following services to the community:

- Internet Governance;
- Allocation of IPv4, IPv6 and AS numbers;
- RIPE database;
- RIPE routing history;
- Operation of K-root nameservers;
- ENUM delegations;
- Collecting and publications of Internet Development and performance statistics;
- RIPE Atlas.

2.2. GII

The RIPE NCC maintains the following staff structure [1]:

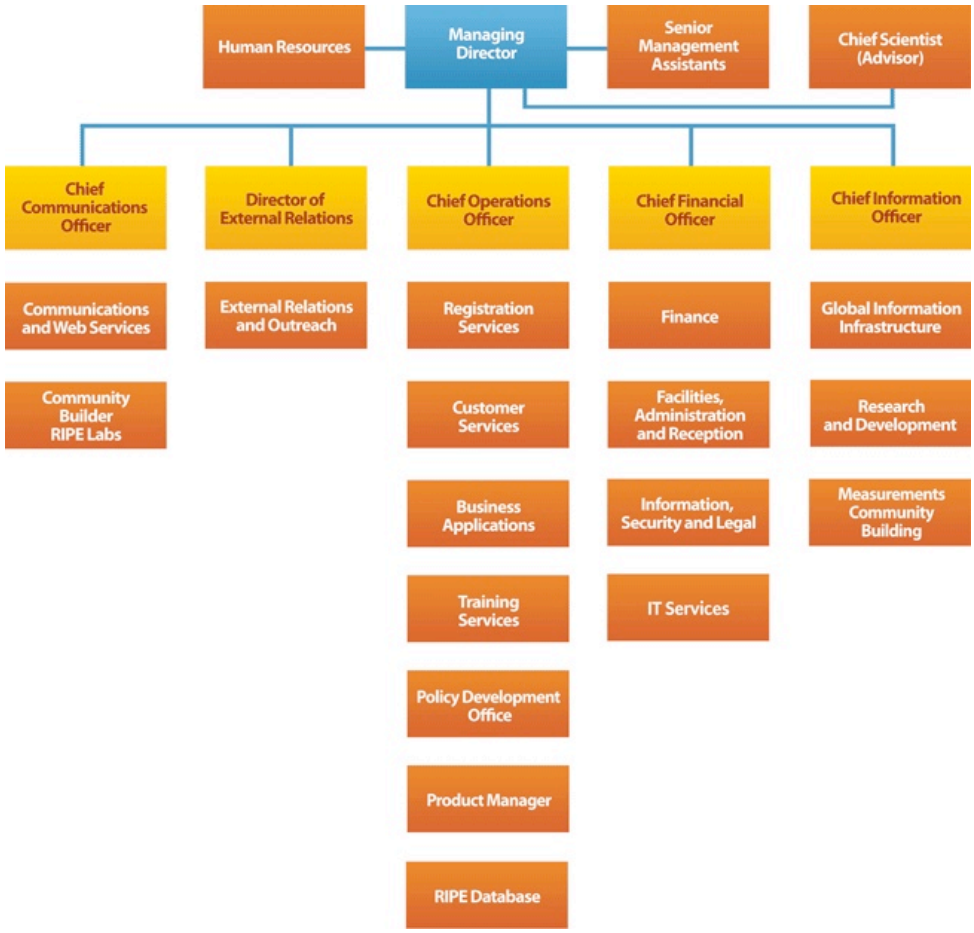

Figure 1 - Staff structure of the RIPE NCC

The research, development, and deployment of the prototype was all under the supervision of the Global Information Infrastructure (GII) team. The primary task of GII is to maintain the global information infrastructure, which involves the K-root instances. It also involves maintaining the RIS infrastructure, which is the reason that the project was under the supervision of the GII Team.

[1] Image acquired from the following webpage: http://www.ripe.net/lir-services/ncc/staff/ripe-ncc-staff-structure (RIPE NCC)

2.3. Original assignment

Before the project started, the initial assignment was phrased as:
"How can the current implementation of the RIS Route Collector process be replaced with a better alternative, aimed to process updates faster and make information easier to integrate in the RIPE NCC Hadoop storage backends (e.g. through XML, JSON, or YAML)?"

The following description of this project was included in the application form:
"The current RIS route collection mechanism, based on Linux systems running Quagga BGP daemons, was developed over 10 years ago. The RIPE NCC Science department would like to improve the current process to allow for more fine-grained data collection, instead of the current process that depends on BGP table dumps created at regular intervals.

This research project focuses on researching an alternative for the RIS route collectors and aims to develop and build a prototype. This prototype may be released to third parties and support needs to be provided to internal and potentially external users during this project."

During the first weeks, analyses of the assignment were done, which is described in the next subchapter.

2.4. Analysing the assignment

The current RIS implementation, which is used by the RIPE NCC, was released in 2001. Back then, use cases and requirements were different from the ones people have now. The project "Replacement of RIS Route Collectors" has been initiated in the year 2013 and started in the year 2014. It is aimed to research possible alternatives for the current implementation but also to develop and deploy a prototype. You could say that the prototype is a proof of concept. It is used to prove if the proposed alternative can meet all the requirements and if it is possible to create the proposed alternative.

The use cases that exist now are outlined subsequent subchapters.

2.4.1. Analyses of Internet Routing

People at the RIPE NCC and the community can download the MRT formatted files, which are provided by the current RIS implementation. These MRT formatted files hold the whole RIB state for each of the route collectors, or all the update messages that have been received by the RRC in a given time period. Users can download these files to do research on routing. For example, when someone wants to know if their prefixes are announced and are propagating through the network.

Note: This is possible with the current implementation

2.4.2. IP hijacking detection

When someone else is announcing a prefix that has been allocated to a certain entity, one wants to know this in a short amount of time. The person wants to know which AS is announcing the prefix and can take action to resolve this issue.

Note: This is not possible with the current implementation. Users have to wait for a long period before this data is available and it is only useful if this data is available in a short amount of time.

2.4.3. Capability to view the live RIB state

Users can get a live RIB state at any given moment and can see what the state of the RIB is. This can be useful for users who want to get a live overview of a RIB and want to do research with it.

Note: This is not possible with the current implementation. Users have to wait for a long period of time before this data is available.

2.5. Evolving requirements

Since the use cases have differed in the recent years, it is also an indication that the requirements have been changed. The current requirements are listed in chapter 5.

The current implementation, which will be presented in a subsequent chapter, can not satisfy all these requirements. A new system that can support such use cases and features must be first of all extensible and provide us with the information that is processed in a later stage.

People from the RIPE NCC would already benefit from the fact that if they have access to an extensible system, which only provides a marginal subset of all these features, but that allows extra features to be integrated in such a system easily.

2.6. Definitive assignment

After receiving all the requirements and analysing them, the conclusion was that the assignment differed from the original assignment. A better definition of the assignment is:

“How can the current implementation of the RIS Route Collector be replaced with a better alternative, aimed to process updates faster, make information easier to integrate in the RIPE NCC Hadoop storage backends (e.g. through XML, JSON, or YAML), and meet the gathered requirements?”

2.7. Subquestions

In this research project the traditional main- and subquestion system is not used.

Different people and different departments are involved in this project and have different ideas and requirements for this project. Therefore, a main- and subquestion system will not be really helpful to find the best alternative for the current implementation. When a MoSCoW scheme is implemented, which is filled with requirements that were proposed during interviews, better research can be done. The MoSCoW scheme will give the requirements different priorities and is, in this case, a better methodology to use during this research stage.

2.8. Personal goal

The personal goal of the author of this document was to gain more technical and professional skills in a multilingual company, like the RIPE NCC, during this graduation project. Technical skills that were necessary for this project were knowledge about BGP, RIPE NCC systems, inner workings of systems, and it was also the goal to improve these technical skills. Professional skills that the author wanted improve on were the communication skills and writing skills in English that were necessary during the graduation project.

2.9. Skills

The skills that were necessary and described in the application form of this graduation project are described below[2].

- I.An3: An existing, complex, large-scale or worldwide research on technology or methodology and analysing alternatives;
- I.Ad3: Ability to apply argumentation from a tech, business, costs/benefits, risks and legislation perspective;
- I.On3: Design a secured, multi-site, worldwide business network including possible security measures with specialistic and state-of-the-art technology;
- I.Re3: Being able to prepare a customised application for deployment and testing.

2.10. Planning

This table gives a brief overview of the high level planning that was created at the begin of the project:

Name of stage	Brief description	Time
Research stage	Researching possible alternatives	1 month
Development stage	Developing prototype	2 months
Deployment and test stage	Deploy prototype	2 months

Table 1 - Initial planning

2.11. Note on prototype

The prototype is built to see if it is possible at all to meet all the requirements. It is possible that in the end the prototype does not meet the requirements, and that the 'possible alternative' was not as perfect as we may have thought. If this is the case, it should be documented why it is not possible to create an RRC based on the proposed 'alternative implementation' and what the recommendations are for the next project.

2 The skills that are described in this paragraph are from "Bachelor of ICT domeinbeschrijving" and have been translated to English: Schagen, J.D, van der Kwaak, W., Leenstra E., Smit. W. en Vonken F. *Bachelor of ICT - domeinbeschrijving*, Amsterdam, 2009

2.12. Abbreviations

This table gives a brief explanation of the abbreviation of terms that are used throughout this paper. If there is an RFC applicable to the explanation it will be listed in the last column.

Term	Explanation	RFC
BGP	Border Gateway Protocol, an Inter-Autonomous Routing Protocol used for exchanging network reachability information with other BGP systems.	4271
FTE	Full Time Equivalent.	N/A
GIL	Python Global Interpreter Lock. Mutex that prevents multiple threads from executing Python bytecodes at once.	N/A
HBase	Hadoop Database.	N/A
HDFS	Hadoop Distributed File System, is a distributed file system and is an Apache Hadoop subproject.	N/A
INRDB	Internet Number Resource Database, holds datasets from the RIPE NCC and other entities. There are two clusters, INRDB1 and INRDB2.	N/A
MRT format	Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format, format for exporting routing protocol messages, state changes and routing information.	6396
RIB	Routing Information Base, contains routes. There are different routes per RIB. One for received routes, one for that contains routes after applying the Decision Process, and one that contains routes that will be send to other peers in the case of BGP.	N/A
RIS	Routing Information Service.	N/A
RIB state	Creating a RIB based on the messages received from a peer, which contains the routes received from a peer.	N/A
RRC	Remote Route Collector.	N/A

Table 2 - Wordlist

3. Current implementation

Several RRCs at different worldwide locations, provide a periodic RIB dump or a dump of all the received updates during five minute periods. These dump files are saved in a MRT formatted file and will be periodically synced using rsync to a server called ‘Alpaca’. A NFS share is mounted on “Alpaca”, which contains all the MRT formatted files. The next step is to save this information in a database so that it can be retrieved at a later moment.

The current implementation uses Hadoop/ HBase as its database. The MRT formatted files that are stored on the NFS-share are copied to HDFS.

From HDFS, the data will be inserted into HBase that also uses HDFS as its filesystem. Thrift is used as an API to access data that is stored in HBase and that needs to be accessible for public facing websites (i.e., RIPEstat).

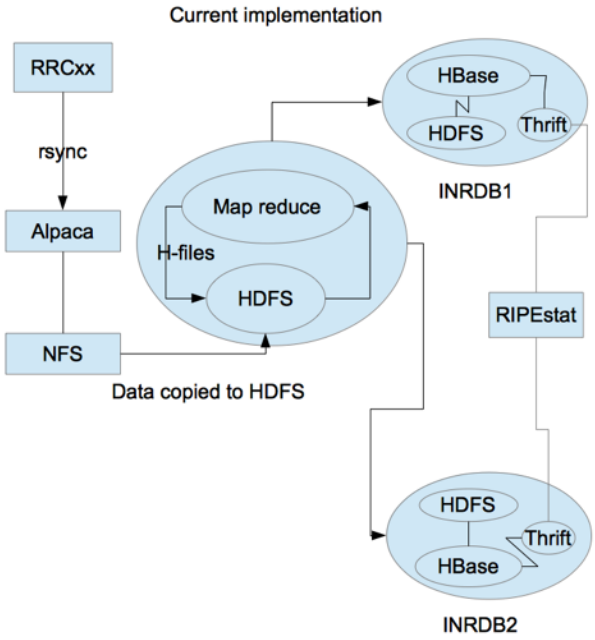

Figure 2 - High level overview of current implementation

3.1. Delay

One of the issues in the current implementation, is that there is a lot of delay involved in the processing part. The data must be copied several times and needs to flow to all different subsystems. This is in fact the main reason why this project has been initiated in order to develop a mechanism that will later allow to reduce the latency.

3.2. Quagga

Quagga is being used to receive all the BGP updates on the RRCs. It maintains BGP peering connections with multiple peers. Quagga also introduces some problems. There have been cases where Quagga missed some of the BGP updates and this caused a RIB that was not accurate. It is a single threaded application, which does not take advantage of a system with multiple cores. It also reaches the maximum utilisation of the single core that it uses and cannot handle any extra peers.

Some of the people at the RIPE NCC do not really trust the RIB dump implementation of Quagga. There were cases where the RIB table got corrupted or that Quagga missed updates during those dumps.

4. Research methodology

This chapter describes the methodology that was used during the research stage of this project.

4.1. Main question

This project has been initiated to answer the main question that has been outlined in chapter 2.6.

To answer this question, the MoSCoW scheme is used to prioritise different aspects of this project. For example, is less delay more important than an information scheme that is easier to integrate? In subchapter 4.3 the MoSCoW scheme is introduced and it explains how it was used in the research stage.

4.2. Process

During the research stage different methodologies were used to gather all the required information, process the information and research it. It involved gathering information about the project, understanding RIS, gathering all the requirements by interviewing people and setting up meetings. All these requirements were prioritised using MoSCoW and was used to prioritise the different requirements of this project. Most of the research in the research phase was done be reading documentation and RFCs as part of desk research.

4.3. MoSCoW scheme

The MoSCoW scheme will give the requirements different priorities and is, in this case, a better methodology to use during this research stage. It is not the only methodology or toolkit that was used in this project. A few examples of other methodologies that were used are desk research and interviewing the stakeholders.

The MoSCoW scheme is also used to see if a possible alternative will meet all the requirements and how much work needs to be done when a requirements is not met. It is also used to see if all these requirements can be met with another alternative.

4.4. Disclaimer

The use of this methodology has been approved by the supervisors of this project[3].

3 During the meeting on Tuesday 11th of February 2014

5. Requirements definition

Different people and different departments are involved in this project and have different ideas and requirements for this project. Therefore, a main- and subquestion system will not be really helpful to find the best alternative for the current implementation. When a MoSCoW scheme is implemented, which is filled with requirements that were proposed during interviews, better research can be performed. The MoSCoW scheme was used to prioritise the different requirements.

5.1. Description

MoSCoW, also known as MoSCoW prioritisation or MoSCoW analysis, is used to reach an understanding about the importance that all stakeholders have on a requirement.

The MoSCoW contains the following categories according to *A Guide to the Business Analysis Body of Knowledge*[4]:

***Must**: Describes a requirement that must be satisfied in the final solution for the solution to be considered a success.*

***Should**: Represents a high-priority item that should be included in the solution if it is possible. This is often a critical requirement but one which can be satisfied in other ways if strictly necessary.*

***Could**: Describes a requirement which is considered desirable but not necessary. This will be included if time and resources permit.*

***Won't**: Represents a requirement that stakeholders have agreed will not be implemented in a given release, but may be considered for the future.*

Note: Sometimes the word "Won't" is substituted for "Would".

5.2. Composition of MoSCoW scheme

The MoSCoW scheme, which is presented in the next subchapter, is composed from the requirements that were received during the research stage. The following categories exist in the MoSCoW scheme of this project: "Must, Should, Could, Would and Out of scope". The category 'out of scope' is added because some requirements are not in scope of the RIS project and are not considered to fit in the 'would' category. In a separate subchapter it will be explained why some requirements are considered to fall under the 'out of scope' category.

4 *A Guide to the Business Analysis Body of Knowledge*, International Institute of Business Analysis, Whitby, ON (Canada), (2009, page 102)

5.3. MoSCoW scheme

The following MoSCoW scheme was used during the research stage of the project.

Must:	
	Announcements of anchor/beacon IP
	MRT formatted files
	RAW data
	Metadata
	Ordering
	Less delay
	Same attributes
	eBGP multihop
	Scaling
Should:	
	Live data stream
	Integrity of data
	Extensible
Could:	
	High resolution timestamp
	Authorisation and encryption
Would:	
	None
Out of scope:	
	Spike detection
	Aggregation of data
	Correlation between atlas/anchor and RRC data
	Separate full feed from partial feed

Table 3 - MoSCoW scheme

The input for these requirements were provided during interviews and meetings. The notes that were taken during these interviews and meetings can be found in the research paper. This MoSCoW scheme was present to and approved by the staff of the RIPE NCC during a meeting about the requirements and priorities[5].

5 The meeting with the Science Division group was on Thursday 27th of February 2014

5.4. Out of scope

Out of scope means that the requirements are not considered in scope for this or future projects.

The following requirements are considered out of scope:
- Aggregation of data;
- Correlation between atlas/anchor and RRC data;
- Separate full feed from partial feed[6].

The reason that the ‘aggregation of data’ requirement is considered to be out of scope for this project. It is considered to be more a post-analysing item. The job of the RRCs is to receive the BGP updates and do further processing. A separate tool or project is needed to provide an aggregation of data between the RRCs.

The correlation between the Atlas probes, Anchor, and RRC data is also considered out of scope. This is also considered a post-analysing item which is not the RRC’s job. It can cause more overhead and the RRC’s job should be clear and should not take extra overhead to provide such features. There is a considerable chance that because of this overhead extra delay could be involved.

The ‘separate full feed from partial feed’ requirement is considered to be out of scope of this project. It can not really be determined on a RRC if a peer is sending a full feed or a partial feed and there is no standardised way of determining if a peer is sending its full feed to the RRC. Therefore, this needs to be determined after the routes have been collected, which is done in the post-processing state of the whole RIS system.

[6] When a peer sends its full feed the advertised routes are not filtered by the neighbour. With partial feeds the neighbour filters the routes that are send to the RRC

6. Research phase

This chapter will provide the reader with an overview of which projects or programs were considered in scope and that were researched during the research stage of this project. All this information is outlined in the research paper, which gives readers further explanation about the different projects and contains the original references to statements that have been made in this chapter.

6.1. Conditions that must be met

A program, or project, is considered in scope when it meets one or several of the requirements that were gathered during meetings and interviews.

6.2. In scope projects

The following projects or applications have been considered in scope and can be listed as a ‘possible alternative’:

- Custom solution with ExaBGP;
- BGPmon;
- OSR Quagga;
- BIRD Internet Routing Daemon;
- OpenBGPD;
- XORP;
- Vyatta;
- PyRT;
- BMP (BGP Monitoring Protocol);
- Ryu.

All of the projects that are listed above where found during the research stage of the project. They were used in other programs or were referenced on several websites. The following paragraphs will give a brief description of each ‘possible alternative’. All the information that is presented in the next paragraphs is based on the information from the research paper. If the reader wants to get a better understanding of the following possible alternatives, it is advised to read the research paper.

Custom solution with ExaBGP

A custom solution relying on ExaBGP as its core. ExaBGP will receive all BGP packets and passes them on to another application. This application needs to be developed and will push the BGP updates it receives from ExaBGP in a queue. It is possible to receive raw messages from the application’s point of view, which the application can later on store in a queue system. The queueing systems will be discussed in this chapter as well.

BGPmon
BGPmon, not to be confused with the protocol BMP[7] or the service BGPmon, is created with the same philosophy as this project. It is a project that is maintained by the Network Security Group of Colostate University, which is still actively maintained by this project group. BGPmon supports MRT files as input, but cannot dump RIBs in a MRT formatted file. It can connect with multiple peers and the BGP messages that it receives will be outputted in a XML format. This data can be retrieved from two default ports, 50001 and 50002, which separates the BGP updates form the RIB tables. Raw data is available in the output that BGPmon produces. It is not possible to announce routes, which is a design decision of the BGPmon developers, and therefore it is not possible to announce the anchor or beacon IPs[8].

OSR Quagga
The Open Source Routing is supported by the Internet Systems Consortium (ISC) and aims to support the community in releasing a mainstream, and stable routing code to enable network innovation. They focus on Quagga and partners with the existing developers community, independent code committers, service providers and academic institutions to deliver a higher quality code base for Quagga. Code that is created by the OSR team will be pushed upstream and will be included in the mainstream releases of Quagga if they accept it. They also state on their Github account that the code should not be used in a production environment and advice to use the Quagga mainstream releases instead.

BIRD Internet Routing Daemon
The BIRD Internet Routing Daemon was developed as a school project at the Faculty of Math and Physics of the Charles University of Prague. It received contributions from Martin Mares, Pavel Machek and Ondrej Filip and is now sponsored by CZ NIC Labs. The project aims to develop a fully functional dynamic IP routing daemon primarily targeted on Linux, FreeBSD and other UNIX-like systems.

OpenBGPD
OpenBGPD is a free implementation of BGP version 4. The project started out of dissatisfaction of other implementations that existed at that time and is now a fairly complete BGP implementation. It is used by different users and advertises that users often praise its ease of use and high performance, a well as its reliability on their website[9]. The latest release went live on the 1st of November 2009, which carries the version 4.6.

---

[7] For more information about the BMP protocol, please read the following draft: http://tools.ietf.org/html/draft-ietf-grow-bmp-07 (BMP RFC, IETF, 22 October 2012)

[8] For more information, please visit the following website: http://bgpmon.netsec.colostate.edu/index.php/join-the-peering/peering-faq (BGPmon Peering FAQ)

[9] For more information please visit the following website: http://www.openbgpd.org/ (OpenBGPD)

XORP
XORP, which is an abbreviation for eXtensible Open Router Platform, supports OSPF, BGP, RIP, PIM, IGMP, OLSR. The project has been initiated to create an open source extendible routing platform where researchers could perform tests on, but still make it a solid and stable platform. It is created with security in mind, and if one of the routing instances fail it will not drag the other ones down with it. XORP creates a RIB per client, but it does not provide a built-in function to dump or save a RIB to a file. The IPC system could be used in combination with the XRLs to retrieve the information from the multiple RIB instances, which needs to be consolidated to one RIB instance. This RIB could then be used to produce a RIB dump, which is one of the requirements that must be met.

Vyatta
Vyatta provides a software-based virtual router, virtual firewall and VPN Solution for the Internet Protocol network. The network routing software engine was XORP and has been replaced by Quagga on April 2008. In the year 2012, Brocade Communications Systems acquired Vyatta and renamed it to “Vyatta, a Brocade Company”. In April 2013, Brocade renamed the Vyatta Subscription Edition (VSE) to Brocade Vyatta 5400 vRouter. Their latest commercial release of the Brocade vRouter is no longer open source based.

PyRT
Python Routing toolkit is made for the purpose of collecting route information. The project was last updated on 5th of May 2002. It only supports one peering session according to the documentation. This could be solved by running multiple instances of the program per session. It saves all information to a MRT formatted file, but this file will only contain a dump per BGP update that the program has received. In order to create a RIB table, a separate application needs to maintain the state of the RIB table and cannot miss any updates that are sent to the RRC. If this happens it will lead to an inconsistent or corrupted state of the RIB table.

BMP (BGP Monitoring Protocol)
Another possible solution could be to use the BGP Monitoring Protocol (BMP) that is supported by Cisco and Juniper. It is specifically designed to monitor the BGP updates and states of peers. All messages stored in the Adj-RIB-In from the neighbour will be sent to the monitoring station, which then sends the message to the server for further processing. The BGP update messages received by the peer will be encapsulated in Route Monitoring messages. It is still a draft and there are not that many servers or clients available who can process this data. Another application still needs to be developed which creates the RIB state from the information that is inside the Route Monitoring messages.

Ryu
Ryu is a component-based software defined networking framework that is actively maintained by a community. It is a framework that can be used to receive BGP updates and do further processing with this data. In order to support such a use case, an application must be developed that is using the Ryu framework.

6.3. Other modules/implementations

This subchapter will describe some other modules or implementations that are necessary to deliver a complete product. Some applications or possible alternatives require other software that can do the further processing. The following applications are queueing mechanisms that are needed, by some 'possible alternatives', to store the update messages. For more information about the queueing mechanisms, the reader is advised to read the research paper, which will provide the reader with more information.

Apache Kafka

Apache Kafka is a queueing system that originates from a LinkedIn project to build a more scalable queueing system, which can handle more requests than other traditional systems back then[10]. They have used the system in production and made the code open source. Apache Kafka took over the project and it is now an incubated[11] project. When a consumer wants to retrieve the message from the Kafka queue, the messages will not be deleted from the queue after retrieval. This provides the ability to re-read packets and retrieve updates even if they were consumed before.

RabbitMQ

RabbitMQ is an open source message broker software that uses the Advanced Message Queueing Protocol (AMQP). It is written in Erlang and is built on the Open Telecom Platform framework and is released under the Mozilla Public License. Rabbit Technologies Ltd. develops and provides support for RabbitMQ. Rabbit Technologies was founded in 2007 and was back then a joint venture between LShift and CohesiveFT. It was acquired in April 2010 by SpringSource, which is a division of VMware and it became part of GoPivotal in May 2013.

HornetQ

HornetQ is an open source project for building a multi-protocol, embeddable, very high performance, cluster, asynchronous messaging system from JBoss. The code base was originally developed under the name JBoss Messaging 2.0. It started with Tim Fox who was the project leader until October 2010. The current project leader is Clebert Sysconic with its core engineers being Andy Taylor, Francisco Borges, Howard Gao, and Jeef Mesnil. The project itself was released on the 24th of August 2009.

ØMQ/ZeroMQ

ØMQ, also known as ZeroMQ, is a high-performance asynchronous messaging library which can be used in scalable or concurrent applications. The library provides a message queue, but unlike message-oriented middleware, a ØMQ system does not need a dedicated message broker. The library is designed to have a familiar socket-style API.

6.4. Research

If the reader wants to have more information about the research that has been done, it is advised to read the research paper, which is included as an appendix. It outlines the features that are supported by the programs or projects, but also the features that are missing. The next chapter will describe the conclusion that has been made, which is based on the information that was gathered during the research stage.

---

[10] For more information, please look at the following presentation: http://www.slideshare.net/charmalloc/apache-kafka (Medialets, Apache Kafka presentation)

11 Apache process that could lead to a Apache Top-Level-Project

7. Conclusion of research

This chapter draws a conclusion from the information that is outlined in earlier chapters.

7.1. Total score overview

Table 4 will show a total overview of the scores that are listed in the MoSCoW scheme per ‘possible alternative’.

Possible alternative	Score ++ (+2)	Score + (+1)	Score - (-1)	Score -- (-2)	Total score
Custom solution with ExaBGP	6	6	1	1	15
BGPmon	6	2	3	3	5
OSR Quagga	7	1	1	5	4
BIRD Internet Routing Daemon	8	1	4	1	11
OpenBGPD	8	1	2	3	9
XORP	3	3	4	4	-3
Vyatta	7	1	2	4	5
PyRT	3	3	2	6	-5
BMP (BGP Monitoring Protocol)	3	3	2	6	-5
Ryu	1	5	6	2	-3

Table 4 - Total score overview of research report

The total score reflects how much work needs to be done in order to meet all the requirements. Scores which are higher involve less work that needs to be done. This is also the case when a ‘possible alternative’ supports multiple requirements.

The following points are given to get to a total score:
- ++ is worth 2 points
- + is worth 1 point
- - is worth -1 point (subtraction)
- -- is worth -2 points (subtraction)

The ‘possible alternative’ with the highest score is the custom solution that uses ExaBGP as its core. This is a criterion which can be used to evaluate what the best solution is to use as an alternative.

Another criterion is if it is possible, to develop and deploy a prototype that is based on this 'possible alternative'. To measure, if this is indeed possible, the scores in the diagram above are used. As the score also reflects how much work need to be done in order to meet all the requirements, this is a good indication if the prototype could be developed and deployed.

Be aware that the total score is not the only criteria that is used to draw a conclusion. It is really important that the 'must have' requirements can be all met, otherwise the application will miss some requirements that are the most important ones.

7.2. Conclusion

The conclusion of the above is to use the custom solution based on ExaBGP. It is a very extensible, open source, and lightweight application. It does its job, which is to give another program its output and that program should do the further processing.

Every possible alternative is compared with the requirements that are listed in the MoSCoW scheme. This is to get a good overview if all the requirements can be met and if much effort must be done to meet all the requirements. What also needs to be considered as an important criterion is, if all the 'MUST' requirements can be met.

Not all the requirements are met when ExaBGP is used as a standalone application. Another application needs to be developed to do the further processing and that meets some of the "must" requirements. To meet all the "must" requirements it also involves modifying the source code. But when you look at the overall picture, ExaBGP is more flexible and extensible than the other applications and it is possible to meet all the 'must' requirements. It allows RIPE NCC to use this implementation even if the use cases will differ in the next years to come. As a side note, the original developer of ExaBGP has shown interest to add the requested features in ExaBGP.

The conclusion of the research stage is:
"The best alternative is ExaBGP, but some modification is required, as well as some other applications that need to be developed. It is advised to develop a prototype to see if all requirements can be met and if it is the alternative as we have outlined in this research report."

8. Proposed solution

This chapter describes the proposal that follows the conclusion, which has been outlined in the previous chapter.

8.1. Custom solution using ExaBGP

As explained in the previous chapter, the best alternative is ExaBGP with a queueing mechanism. Some additional components need to be developed that rely on a queueing mechanism.

Multiple queueing solutions were evaluated during the research stage. The best queueing mechanism that could be used in the new implementation is Apache Kafka. The most important argument is that it provides the ability for a consumer to re-read packets from the queue. This can be useful in cases were multiple consumers want to read packets from a queue without destroying the data that is stored in this queueing mechanism. In our case, this would be useful if an API is provided to the community. The community can then create a tool that uses this API.

The following needs to be done to meet all the must requirements:

- State machine; a RIB state needs to be created, which is not done by ExaBGP, this information is used to create a RIB dump;
- MRT formatted files; application that retrieves information from the queue and stores it in a MRT formatted dump file;
- BGP metadata; provided by ExaBGP but must be stored somewhere, is also vital to create a RIB-state (if a connection to a peer is lost, the RIB entries from that peer must be flushed);
- Ordering; ExaBGP needs to add a sequential serial number in its output;
- Less delay; mechanism that inserts the data directly into the database;
- RAW data; must be added to the output or be able to receive this.

If all the “must” requirements must be met, it requires developing an application for putting all the ExaBGP messages in a queue system, which can be later on retrieved for further processing. Figure 3 will give the reader a total overview of the system that needs to be developed.

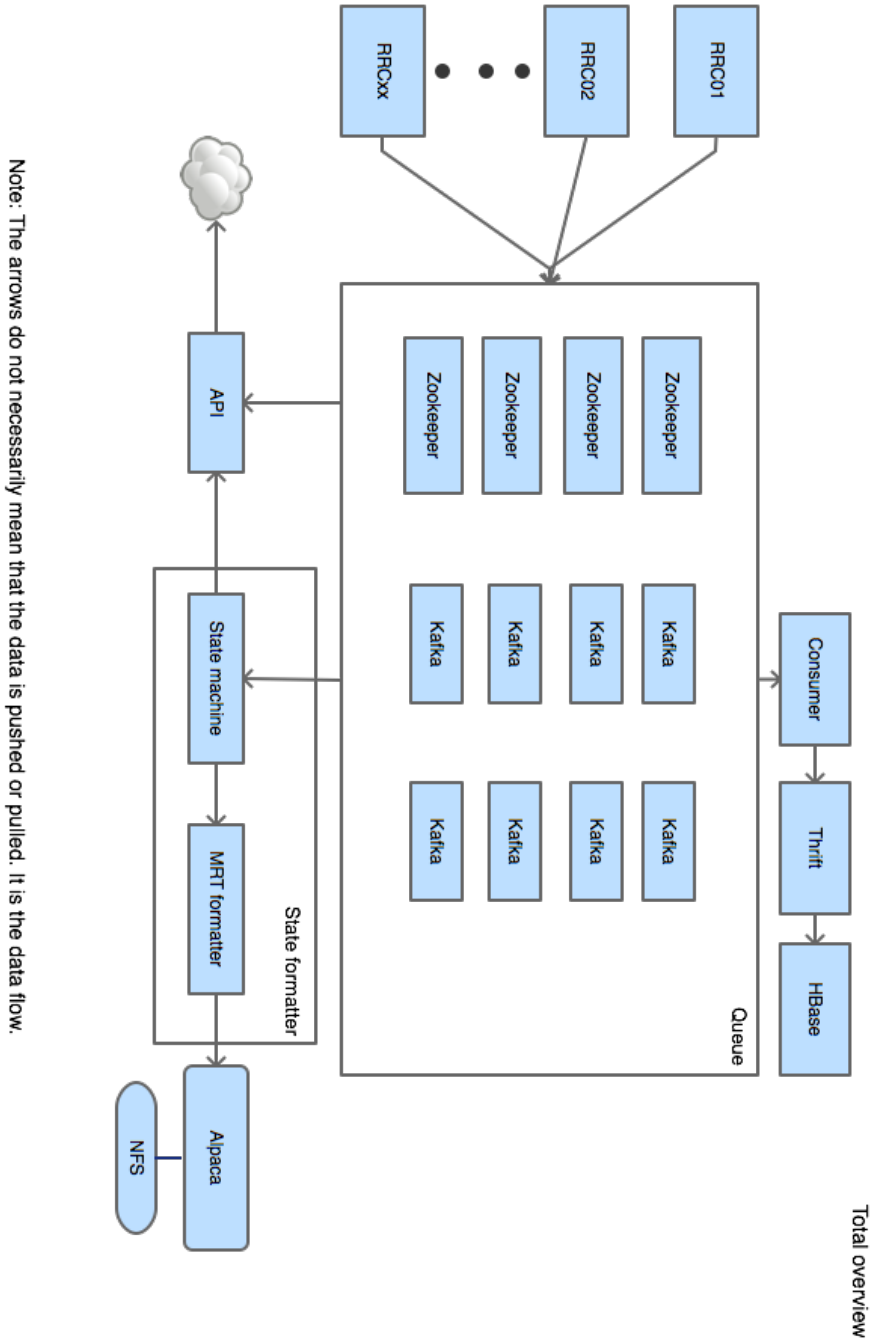

Figure 3 - High level overview of proposed prototype

Another separate application retrieves information from the queue, creates a RIB state and stores this in a MRT formatted file. The MRT formatted file contains all updates that were received during the interval. As explained earlier, there is also a separate file, which contains the RIB state which is formatted in a MRT format. This application could be configured to do this periodically, and efficiency could be gained in the way the MRT formatted file is created. This could reduce the latency, which is involved in the current implementation.

The information that is stored in the queue must also contain the BGP state messages and could be stored in a separate topic. Delay is involved when the MRT formatted files are copied from the “state formatter” to Alpaca, and later on, copied to HDFS. To overcome this delay, an application needs to be developed, which directly inserts the information into the database.

8.2. Producer

The application, which receives the information from ExaBGP on the RRC itself, must provide a mechanism to store all the information that it received from ExaBGP whenever it cannot connect to Zookeeper/Kafa. This is illustrated in figure 4.

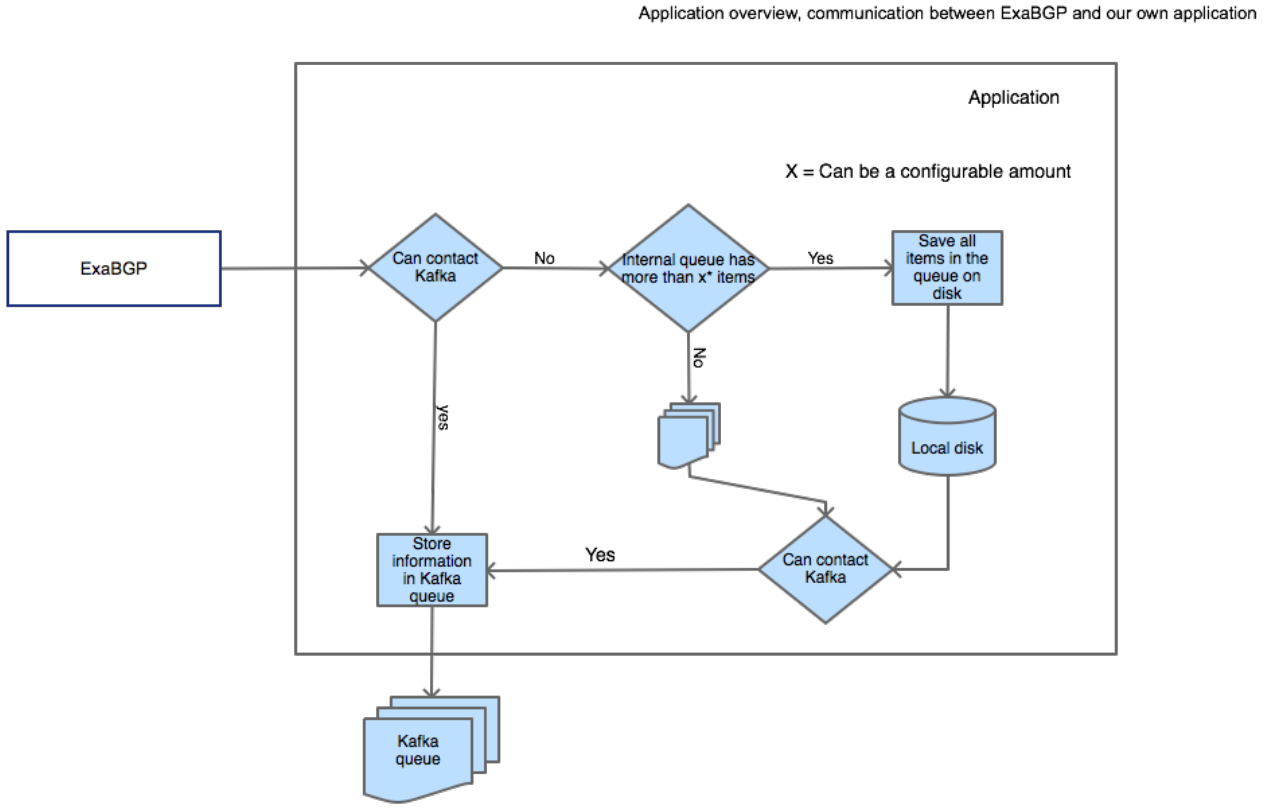

Figure 4 - High level overview of proposed producer application

The application will use multiple threads to process all information that it receives from ExaBGP, which is illustrated in figure 5.

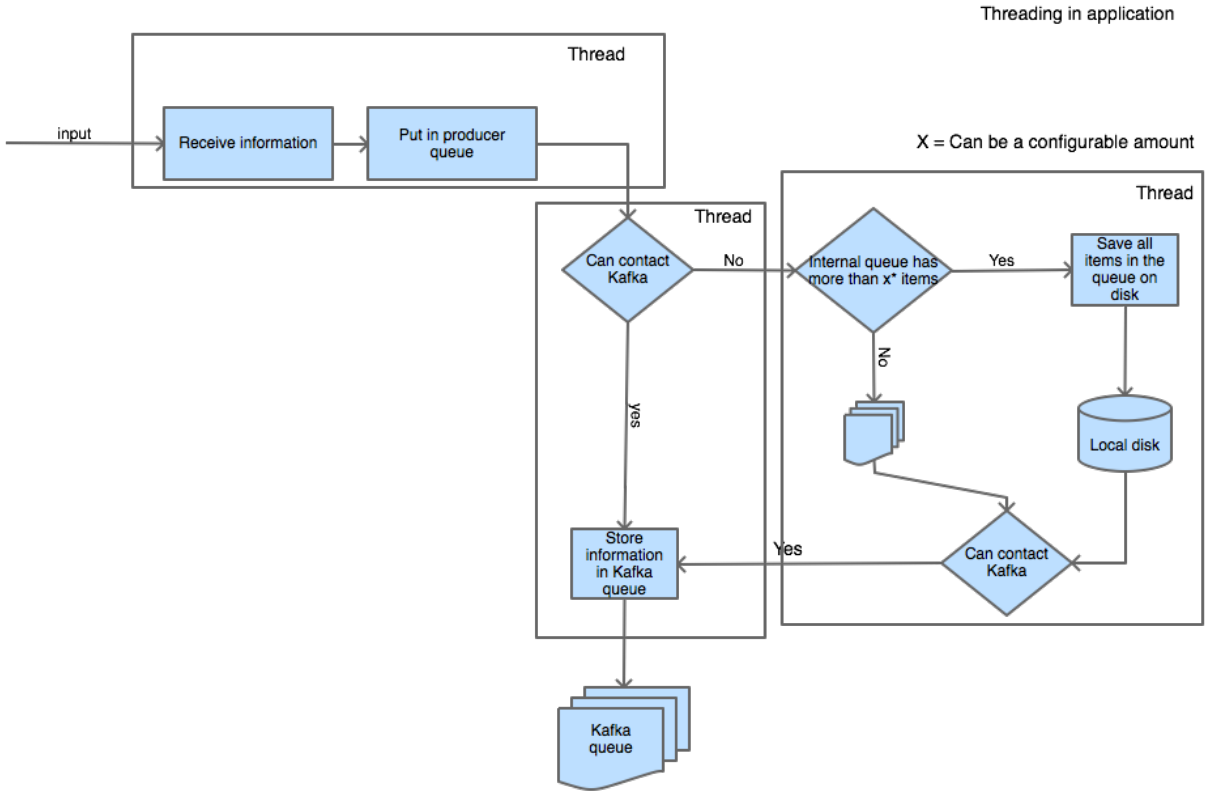

Figure 5 - High level overview of multithreading in proposed producer application

This keeps all the functions separated from each other. There are three threads that provide the following functions:

- **Receiver**: Sending the received information, from ExaBGP, to a producer queue
- **Producer**: Put the information into the queue system or pass the information to another thread if the queue cannot be reached. If the queue is reachable again it will put all the locally stored information in the queue first before it puts the new information into the queue
- **Store**: If the queue cannot be reached this thread will save all the information in another queue or store the information locally on disk.

8.3. State formatter

Another application needs to be developed that retrieves the information, from the same queue, and creates a RIB state. This is illustrated in the following image:

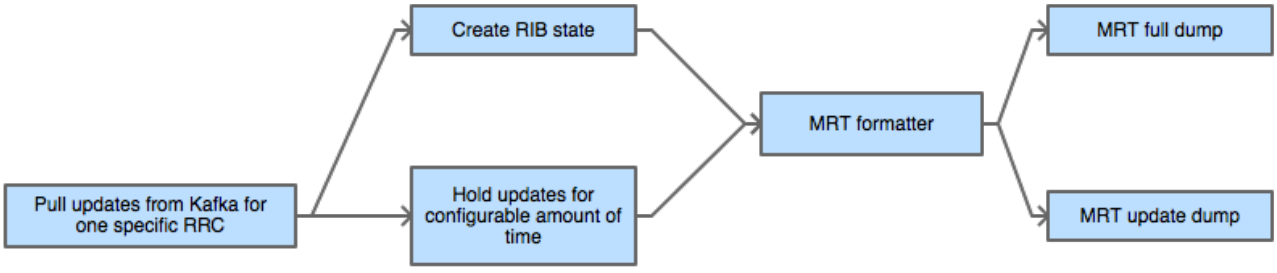

Figure 6 - High level overview of proposed state formatter application

The application will receive the information from the queue and stores it in a local queue. After a configurable amount of time it will create two MRT formatted files. One which contains all the updates that ExaBGP received and one that holds the complete RIB state. These MRT formatted files are then copied to Alpaca that stores these MRT formatted files on its NFS-share. This is done to maintain the MRT formatted files that are used by the community, but also by other programs that do further processing on them and insert it into the HBase database. It allows the currently existing mechanisms to continue working and maintains backwards compatibility.

8.4. HBase consumer

To reduce the latency even more, it would be useful to insert the updates straight into HBase, which is illustrated in figure 7.

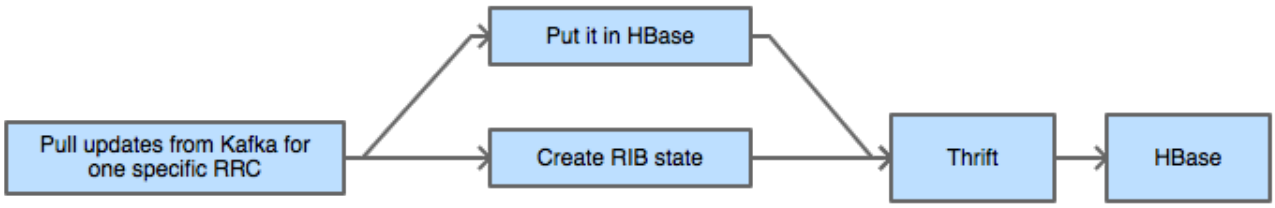

Figure 7 - High level overview of proposed HBase consumer application

The information that is retrieved from the queue is directly inserted in HBase using Thrift as its ‘gateway’. It will also create a RIB state so that it can periodically save a complete RIB state in HBase, this is also done in the current implementation.

It is basically the same functionality, besides from inserting it into HBase, as the “state machine”. The reason that this application does not rely on the data from the “state formatter” is that it could be the case that the MRT formatted files might be deprecated in future. If this is the case, the whole “state formatter” does not need to be any longer used as it does not have any dependencies. There is also the case of locking that might be involved if the HBase consumer relies on the data in the state machine. This involves complexity and could cause race conditions in the HBase consumer if it is not implemented correctly.

If there is an API that relies on the data from the state machine, it could rely on the data of this application or use the state machine in another program.

8.5. Queueing system

The queueing system is a very important part of the overall working of the prototype too, it is the central system where all information is logged and that lends itself to be used as an inter-process communication (IPC) mechanism. For example, producers write their status information to a specific topic. This information is used by the consumer to detect the RRCs that are alive and the ones that publish the information. Then the consumer can look at the 'Consumer control queue' to see if all the information that is published by a specific RRC is also processed by another queue. When a new RRC comes online it can register itself to the control queue of all RRCs, the consumers see this and a consumer can subscribe to the topics, that the specific RRC has created, and process all the information. When a new RRC is deployed, a consumer does not need to be configured to subscribe to these topics, but automatically detects it and retrieves the information from the topics.

The following image illustrates the queue mechanism.

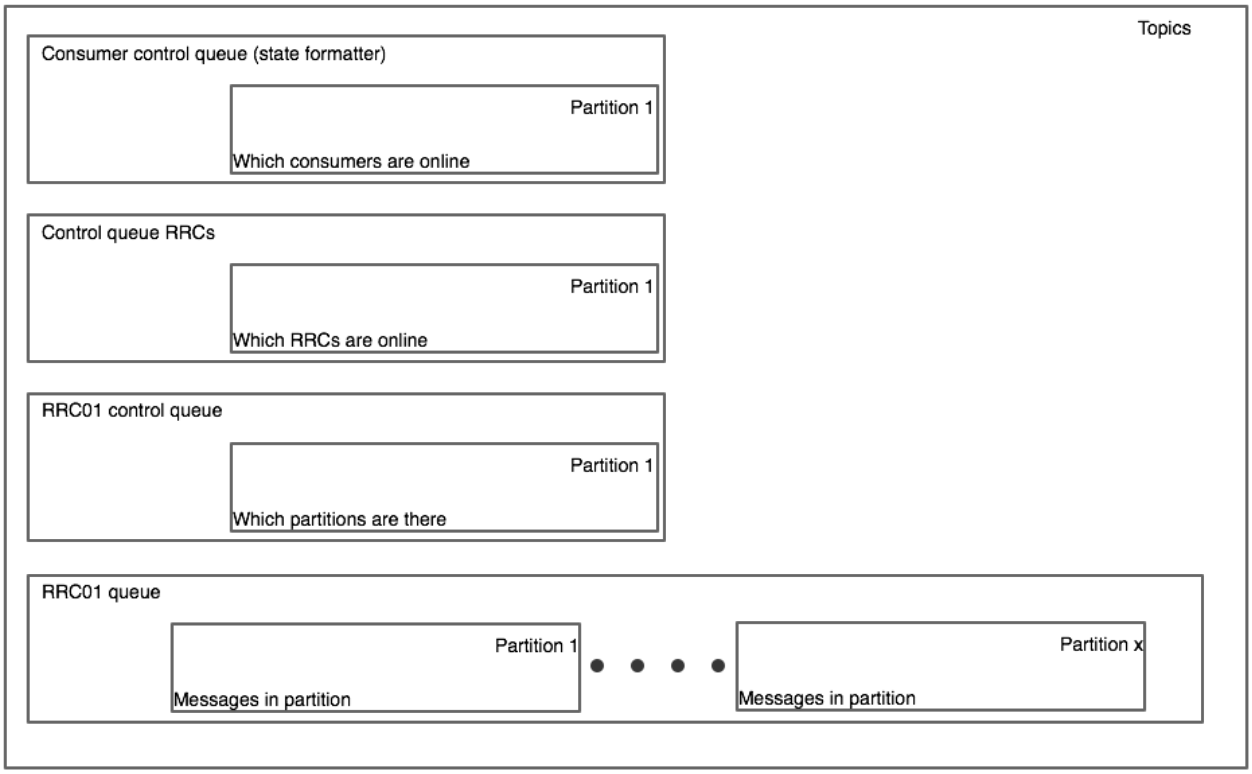

Figure 8 - Proposed queue mechanism

The auto-detection and subscription mechanism is not implemented in the prototype, but the queue system and prototype will be constructed to facilitate such a need for the future.

The queues of the RRC are divided in multiple partitions, this is for the sake of scalability. If there is only one partition per RRC the system does not provide us with the scalability that is needed in the RIS project. All these different partitions are stored in, for example the RRC01 control queue, so that the consumers know how many partitions there are and the sequence ID that is stored in a specific partition.

8.6. Developer planning

This chapter will describe the planning of the development stage.

The planning will be divided in several stages as outlined in the following table.

Stages	Description	Estimated time
Stage 1 **Preparation**	Virtual Machines: Virtual machines need to be created to use it as a development environment. The applications will be developed on these virtual machines but are also tested. Involves: Installing Virtual Machines, Zookeeper/Kafka, HBase, ExaBGP, libbgpdump.	1 week
Stage 2 **ExaBGP and Kafka**	Change ExaBGP code to meet all the “must” requirements. Put the messages in the queue system Messages must be saved locally if the connection is lost with Zookeeper/Kafka. Add keepalive messages to the output/BGP metadata.	2 weeks
Stage 3 **State formatter**	Create state machine. Create code to create MRT formatted files.	3 weeks
Stage 4 **Insert data into HBase**	Insert data straight into HBase.	2 weeks

Table 5 - Developer planning

8.7. What is necessary

Multiple virtual machines are necessary to develop and test the code. In the beginning of the development stage the virtual machines will run on a laptop, but it may be useful if some parts of the project are installed on a real server. When a real server is used and the application is tested, some requirements could be set on the required specifications of a server that must run a RRC.

It must also be possible to connect a test RRC with a test peer. This could be done to determine the scalability of the application. If the HBase consumer is ready for use, it will be tested to see if the HBase consumer can insert data in a test setup.

9. Development

During the development stage, several changes were introduced. This resulted in a prototype that differs slightly from the proposal, which is described in the previous chapter. The changes, which were introduced, were the result of several discussions about how the prototype could be developed in an efficient way and one which could be reused.

During a discussion with some team members[12], the choice was made to switch from Kafka to RabbitMQ. RIPE NCC is more familiar with the use of RabbitMQ as a queueing system, which was one of the reasons to use another queueing system. The other reasons had to do with the capabilities that Kafka supports but that were not used and the Python library for Kafka. The Python library for Kafka was not really mature and contained some bugs that are listed as issues on Github. The re-read capability of Kafka was not used in the prototype, whereas RabbitMQ has better libraries and could provide security mechanisms that could be used for securing the data transmission. If it is later on decided that another queue mechanism will be necessary, it is still possible to do so.

The inner working of the state formatter has also been changed. The first proposal stated that the state formatter will insert both the RIB and the update messages in HBase. As the RIB is already inserted into HBase, by the legacy systems which relies on the MRT formatted files, it is not really necessary to reimplement such a task in the prototype. It is decided that the HBase consumer will only receive the update messages from the queueing system and insert it into HBase. The contents of the MRT formatted files will be used to insert the RIB contents into HBase, using the existing mechanisms. This is also explained in the following image.

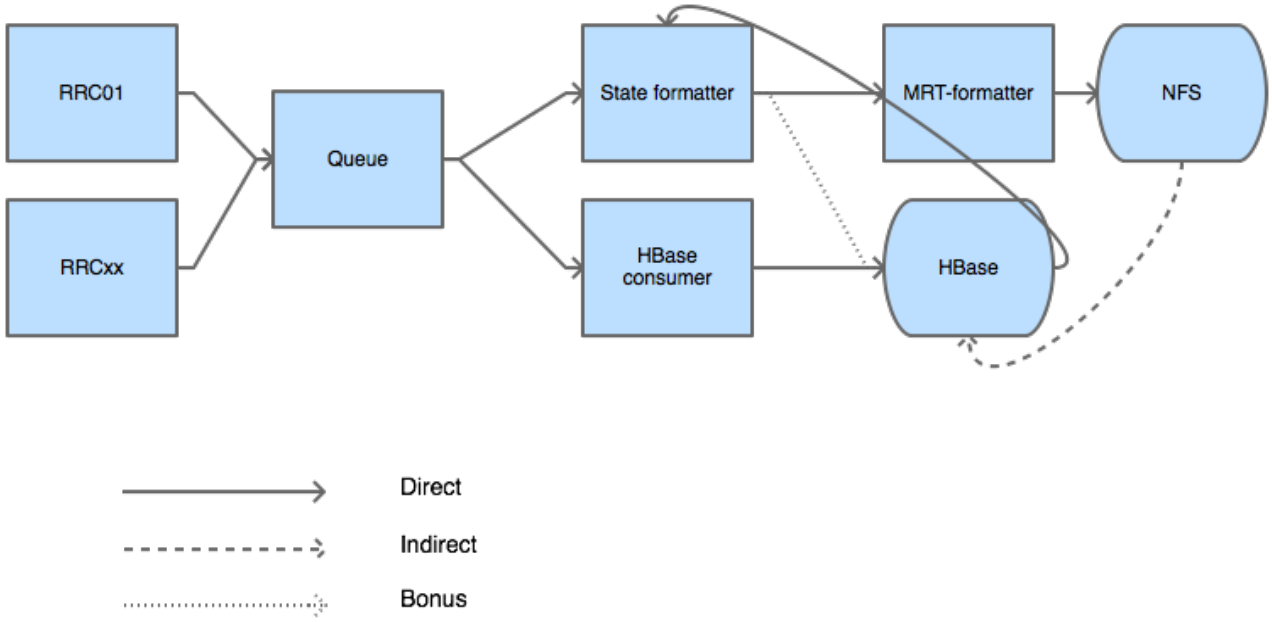

Figure 9 - High level overview of data flow in prototype

[12] During the meeting at 18th of March 2014

Because the prototype changed a bit, the planning also had to be changed according to these changes. The planning has been changed to the following.

Stages	Description	Estimated time
Stage 1 Preparation	Virtual Machines: Virtual Machines need to be created to use it as a development environment. The applications will be developed on these Virtual Machines but are also tested on these Virtual Machines. Involves: Installing Virtual Machines, RabbitMQ HBase, ExaBGP, libbgpdump.	1 week
Stage 2 ExaBGP and RabbitMQ	Change ExaBGP code to meet all the “must” requirements. Put the messages into the queue system. Messages must be saved locally if the connection is lost with RabbitMQ. Add keepalive messages to the output/BGP metadata.	2 weeks
Stage 3 Insert data into HBase	Insert data straight into HBase.	2 weeks
Stage 4 State formatter	Create state machine. Create code to create MRT formatted files.	3 weeks

Table 6 - Changed developer planning

9.1. Process

During the development stage, weekly meetings were organised to discuss the progress. It was also used to talk about how a specific application should be developed. A lot of feedback was processed and also implemented in the resulting prototype.

During the development stage a tracker system was used for creating several tasks per stage. Creating these tasks allowed for a better overview of the whole project and were smaller units of works must be finished before the other one could be started. During the weekly meetings the process that has been made was discussed and a new sprint was created, which was one week of time, in which some of the small units of work must be finished. It allowed for the supervisors to have a better overview of the state of the project. After the works was reviewed the ‘unit of work’ was accepted by the team manager.

9.2. Preparation

Several virtual machines were created and used as a development environment. This environment was used to emulate the new RIS implementation. It consisted of three virtual machines, which were used to serve the RabbitMQ services. RabbitMQ was installed in an active/activate cluster setup with the queues replicated across all servers. This setup was used to simulate a cluster setup that could be used in a real production environment. If one server is down, for maintenance or if it crashed, one of the other servers will be elected as a leader and could handle the requests. Such setups are used in production environments and were also used for testing the producer to be broker/cluster aware.

Two other servers were used as software routers running Quagga with different configurations. One of these announced thousands of routes where the other only announced several routes. Another server was used to run ExaBGP with the producer software and acted as an RRC.

There was one other server that was used to run HBase and the Thrift gateway. Thrift is used as a gateway and allows us to use a Python library to connect with the Thrift gateway and insert data into HBase.

All these virtual machines were created using VirtualBox and Vagrant for easy deployment. Vagrant can be seen as a wrapper around other virtualisation software such as VirtualBox and VMware.

9.3. ExaBGP and RabbitMQ

During the second stage of the development of the prototype, the ‘producer’ needed to be created. The ‘producer’ is a program that receives messages from ExaBGP and pushes it to the queue. Since it is important that the producer does not lose messages, otherwise it would lead to an inconsistent state, the producer has to facilitate a save mechanism. This save mechanism needs to save the messages when the connection is lost with RabbitMQ. It was also necessary to modify the source code of ExaBGP since some requirements could not be fulfilled with the current implementation of ExaBGP. The modifications in the source code in ExaBGP and the extra functions that have been implemented are described in the following paragraphs.

9.3.1. ExaBGP modifications

The source code of ExaBGP has been changed to implement new features, or to add more information in the ExaBGP messages, to meet some of the requirements. The following functions or output elements were added to ExaBGP:

- Added high resolution timestamp (done by the ExaBGP developer in a later commit);
- Raw message is also added in the update messages;
- Sequence keys have been added;
  - Variable $counter_messages introduced to hold the sequence keys;
- Keepalive messages can now be passed to the backend;
- Neighbour capability negotiation messages are parsed and can be passed to the backend;
- Added id variable. Variable that is used to hold the hostname ($hostname), parent process identifier ($ppid), and the IP address of the neighbor ($nip). Concatenated using the following format: $hostname_$ppid_$nip. Useful for later process to determine what the original RRC and its PID was during the time that the BGP message was received.
- “Type” added in the output. Specifies what kind of message it is. The following types have been added ('update','open','keepalive','state', 'raw', 'notification',‘none’).
- Introduced notify element in JSON output. ExaBGP is raising an error when the notify option is not empty. If it raises an exception, the contents of the $notify variable will be passed to the backend. The code and subcode of $notify is added in the JSON notification message.

The changes that were made in the source code of ExaBGP have been pushed to the upstream branch[13]. Since the pull request was accepted by the developer of ExaBGP, all changes or added functions are now in the main releases of ExaBGP. When a new version of ExaBGP is released it is not longer necessary to apply all the changes to the new version, since the modifications and new features are now part of the mainstream code.

9.3.2. Producer general information

When the source code is downloaded from the repository the following files will be in the producer directory[14]:

- producer (package):
  - __init__.py
  - queue_connector.py
  - store_local.py
- producer_program.py
- settings

The hierarchy and structure of these files and directory come from the different processes that exist in the producer program. The producer_program.py file is the parent process of all the other subsequent processes that are launched by the 'producer' application. The producer_program is executed by ExaBGP during the startup of ExaBGP. The producer package contains the other three files, which are used by the producer_program. The __init__.py is created so that Python treats the directory as a Python package. The other two files acts as submodules that contains code for the other two processes. The queue_connector contains the source code for pushing messages to the queue and the store_local file contains the code for storing the messages locally when the connection with a queue is lost. The settings file contains configuration parameters which are loaded in the program during the startup of the producer program.

9.3.3. Producer processes

The final version of the prototype of the producer program differs slightly from the original proposal. Instead of threads, multiple processes are used to do all the work of the producer. The main reason to switch from a multi-threaded application structure to a multi-process application structure, is because of GIL[15] in Python. Since only one thread of a process can run at the time, no real processing benefit is involved using the multi-threaded model. When multiple processes are used, by using the multiprocessing library, multiple processes can run at the same time, with an interface which is similar to the one of threading.thread. It also involves a performance benefit when a server has multiple processors. The multi-process architecture is shown in figure 10.

---

[13] https://github.com/thomas-mangin/exabgp/pull/2https://github.com/thomas-mangin/exabgp/pulls/wmiltenburg?direction=desc&page=1&sort=created&state=closed (Github pull request)

[14] Items listed in a tree structure

[15] For more information about GIL, please visit the following website: https://wiki.python.org/moin/GlobalInterpreterLock (Python GIL)

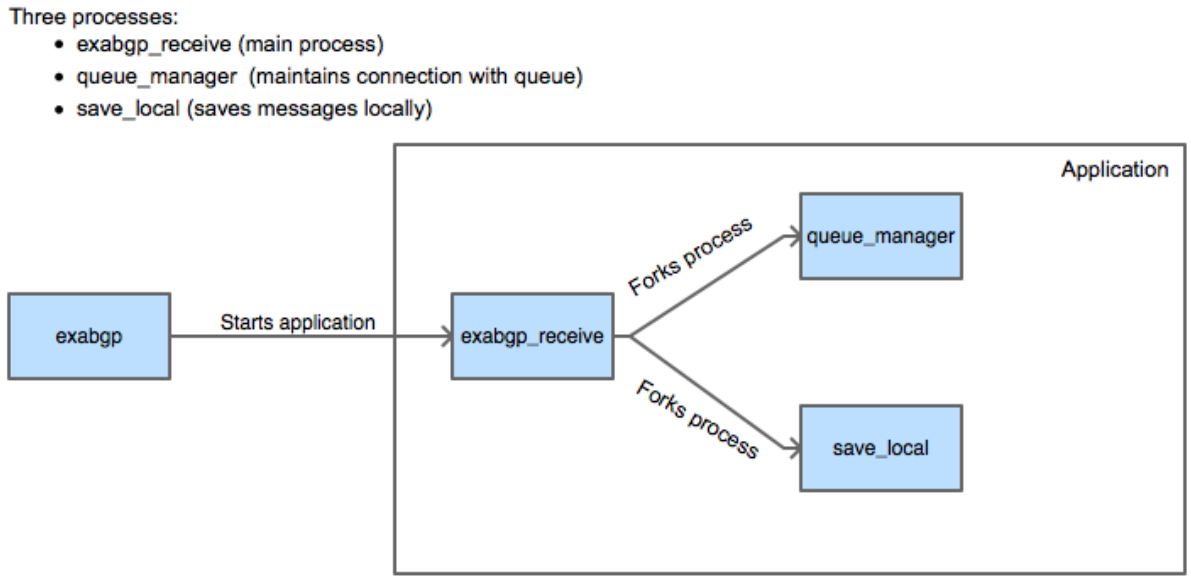

Figure 10 - Processes that exist in the producer prototype

There are three processes; exabgp_receive, queue_manager and the save_local process. The exabgp_receive process is the main process that forks all the other processes. It also creates the pipes and when all of this is done, it will only receive the messages from ExaBGP and send it to the queue_manager process. This function will also look at the contents of the message to see if there is a notification element with the value “shutdown.” If this is the case, the message will be passed to the other processes and the main process will also send a message containing a “received_shutdown” message. This message is unique in the producer application and causes the other processes to exit if they have finished their work.

9.3.4. Producer IPC

IPC, or message passing, is done using the multiprocessing Pipe class from the multiprocessing module, which returns a connection type tuple. It is possible to exchange messages using these connectors and it will raise an error when one end of the pipe is receiving and sending both at the same time. Messages bigger than 32MB may raise a ValueError depending on the OS that is used[16]. The pipe connections that exist between the processes in the application are shown in figure 11.

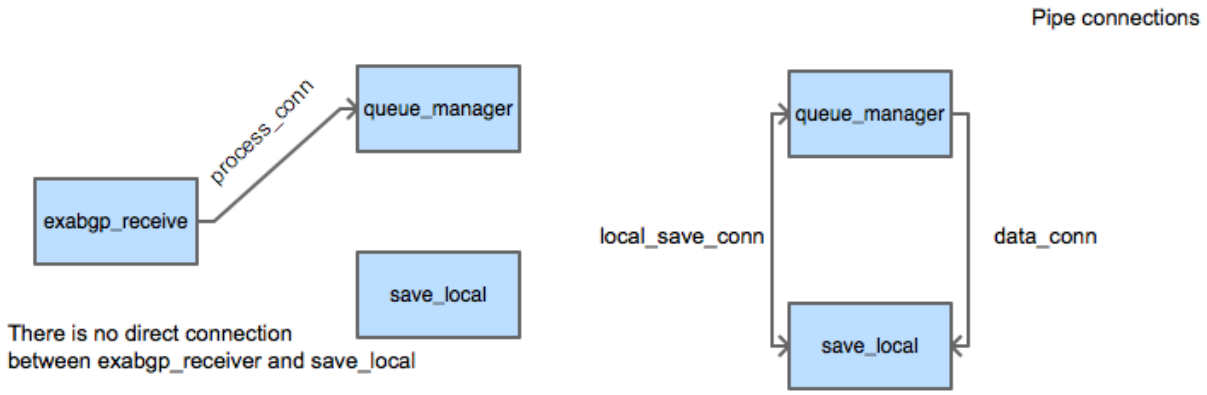

Figure 11 - IPCs that exist in the producer prototype

[16] For more information, please visit the following website: https://docs.python.org/dev/library/multiprocessing.html#multiprocessing.Connection (Python)

Connection types
exabgp_receive > queue_manager (messages are only sent from exabgp_receive to queue_manager).
save_local <> queue_manager (local_save_conn, messages are send both from and to queue_manager and save_local).
queue_manager > save_local (data_conn, messages are only send from queue_manager to save_local).

Since the save_local and queue_manager both sends messages, it was decided to use the connection pipes, since it allows a two way connection.

Description of connection pipes
process_conn: consists of process_parent_conn and process_child_conn. The exabgp_receive process receives all messages from ExaBGP and sends these messages using the process_parent_conn connection. The process_child_conn is used by the queue_manager process, which only reads from the pipe connection and does further processing on these messages.

local_save_conn: consists of local_save_parent_conn and local_save_child_conn. The queue_manager process sends the messages that need to be saved locally using the local_save_parent_conn. It also uses this connection to receive the messages that are stored locally if there is a connection with the RabbitMQ queues after a connection has closed. The local_save_child_conn is used by the save_local process to read the messages that need to be saved locally. When the connection is up again, it will send all the locally saved messages, using the pipe connection, to the other process. This other process will try to pull the messages from the queues.

data_conn: consists of data_conn_queue_manager and data_conn_save_local. The queue_manager uses the data_conn_queue_manager pipe connection to signal to the other process that the connection is up again and is willing to receive a message. The save_local process will use the data_conn_save_local pipe connection to see if the queue_manager is willing to accept messages. If it receives an ‘up’ message, it will send its locally saved messages to the queue_manager process. The queue_manager process will on its turn send the received messages to the queue. If this fails, it will not acknowledge the message, using the data_conn pipe connection. This causes to save the message locally and does not remove if from the local storage. If the message is acknowledged by the queue_manager, using the data_conn pipe connection, the message will be removed from the storage.

During the time that the locally saved messages are send to the queue, the producer will still process incoming messages. It will add the messages to the end of the local queue.

9.3.5. Producer messages

The message passing mechanism described in the previous paragraph is also illustrated in figure 12.

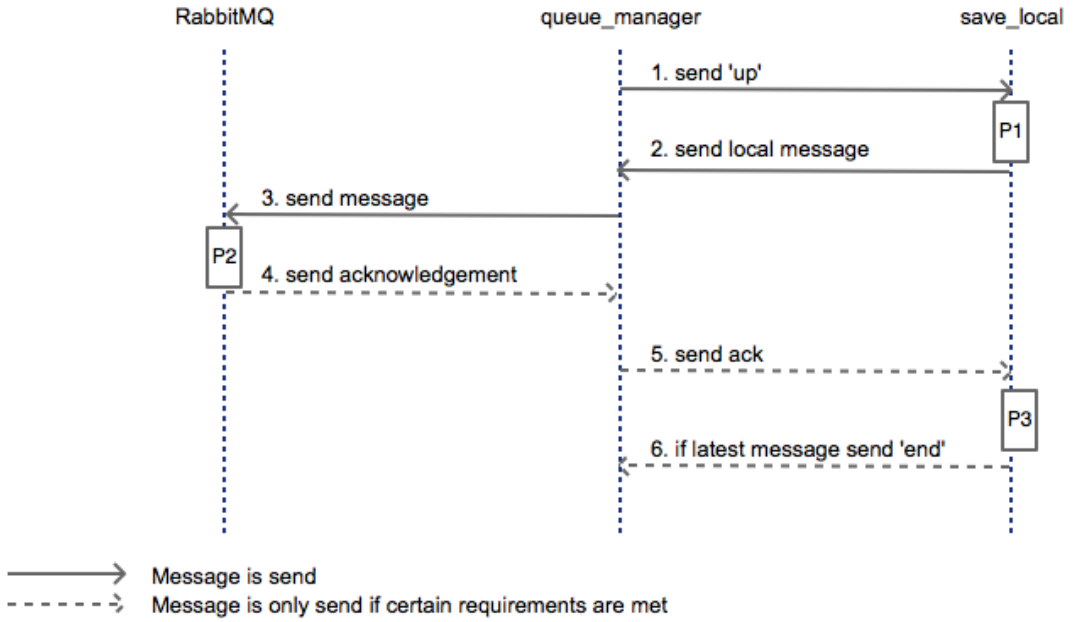

Figure 12 - Overview of messages that are exchanged between processes

9.3.6. Tests performed

The following tests have been performed to see if the producer works as it should:

- Verified that messages are saved locally and send to the RabbitMQ brokers when the connections fails multiple times during one session;
- Verified the message arrival at the RabbitMQ brokers with the announcements of 2540 and 620 different routes;
- Queue order verification, the order of messages in the update queue was verified to be reliable;
- Verified that the raw message in the JSON string is the same as the raw BGP message by comparing the raw data with the one captured by a packet capture;
- Verified that messages that are saved locally and later on inserted in the queue, when a connection is torn down, and when previous locally saved messages were sent to the RabbitMQ queues;
- Verified of that different types of messages are added to the queue;
- Shutdown verification when a messages receives an invalid JSON message that the producer will shut itself down. It has been verified that the application terminates when it receives an invalid message.

9.3.7. Unit tests

Unit tests are created for testing the producer application. There are currently two tests, one for testing the RabbitMQ connections and one for testing if messages are locally saved. When one or both of the tests are started, it will create a ‘ConnectionMessages’ object that starts the other processes and creates the connection pipes.

The two tests will use the connection pipes to send messages to the other process. With the ‘RabbitConnection’ test, a message will be send to the RabbitMQ queue and later on retrieved and compared. With the ‘SaveMessage’ test, a message will be saved locally and later on retrieved and compared with the original message.

9.4. HBase consumer

The third stage of the prototype was to build a prototype application that receives messages from the RabbitMQ queues and inserts it into HBase. HBase is already being used by RIPE NCC and the current RIS implementation. This application would connect to one of the update queues and consume messages from that queue. This subchapter will describe the HBase consumer, its internal workings, and what the final prototype can do.

9.4.1. General information HBase consumer

This program makes use of the HappyBase, pika and msgpack modules for connecting with the Thrift gateway and the RabbitMQ brokers. The Thrift gateway is used for inserting, receiving and manipulating data that reside in HBase. HappyBase on itself makes use of the Thrift Python library, which is used to connect with the Thrift gateway. The msgpack module is used for encoding the message, which is received from a queue, in such a way that multiple applications[17] can decode the data, by using msgpack, for further processing. This is already done with other data that needs to be saved in HBase, by the RIPE NCC, and it is convenient to follow this approach.

The application can only connect to one queue at a time. This is a design decision[18] to benefit from the multi-core architecture which is available on modern servers. It is possible to run multiple instances of the application at the same time, with a different configuration file, which results in multiple processes that run at the same time. These multiple processes can concurrently pull data from the RabbitMQ queues, which result in a performance benefit. If the decision was made to connect to multiple queues, and only use one process, it could lead to a bottleneck. Since one of the requirements of the project is to research a scalable application, it makes more sense to support multiple processes connecting to multiple queues. This design decision supports this, which is the reason why this is also implemented in the prototype.

The HBase consumer is created to insert all the messages in HBase. These messages can later on be used by the state formatter when it crashes. Imagine a situation where the state formatter crashes and its last dump is several minutes old. During these minutes it consumed the messages and acknowledged them, which results in a deletion of the data. The state formatter can now rely on the data stored in HBase to come to a consistent state and create the MRT formatted files. The MRT formatted files could then be used by the legacy system to populate the RIS tables. A separate table is used for storing the ExaBGP messages so that it can not interfere with the schema in current tables. The data that is stored in HBase can also be used by other applications, which use HBase to query for data.

9.4.2. Connection with RabbitMQ

The HBase consumer connects with the RabbitMQ brokers using the pika library. Due to the concept of RabbitMQ, a message is deleted from the queue when a consumer acknowledges the message. This is one of the reasons why there is an exchange ‘fanout’, which creates two queues with the same messages. Such a design creates the ability to read the messages from one queue, acknowledge it, and that another application can still receive the message from a broker using the other ‘duplicate’ queue.

---

[17] These applications can be written in another programming language if msgpack is compatible with the certain programming language.

[18] This decision was made during a weekly development meeting on 8th of April 2014

When a message is received by the application, the application will only send an acknowledgement when all the processing is completed. This creates use cases where an exception is raised, which is handled by the application, and the message is not acknowledged. Either, the connection with RabbitMQ is closed or the application sends a 'nack'[19] to RabbitMQ. Both solutions will cause the message not to be removed from the queue, so that the program can try to receive the same message again, from the queue, and insert it into HBase.

9.4.3. Connection with HBase

The HBase consumer connects with HBase using the HappyBase module, which on its turn relies on the Thrift library and the Thrift gateway. The Thrift gateway, as its name might reveal, is a gateway between an application and HBase using the Thrift library. This allows an application not to be written in Java, which is the native library programming language. Happybase is a library that simplifies the interaction with HBase and the Thrift gateway and is therefore used to simplify the code that is used to interact with Thrift.

The application encodes the message contents using the msgpack module, which results in an encoded message, and which could be decoded by other applications using different programming languages. During a meeting with developers of RIPE NCC, it was decided to use the following table structure during the prototype phase of this application:

Row	**Qualifier: Neighbour IP**	**Qualifier: Neighbour IP**	**...**	**Qualifier: Neighbour IP**
RRC hostname + timestamp + counter	ExaBGP message	ExaBGP message	...	ExaBGP message
RRC hostname + timestamp + counter	ExaBGP message	ExaBGP message	...	ExaBGP message
RRC hostname + timestamp + counter	ExaBGP message	ExaBGP message	...	ExaBGP message

Table 7 - Table structure in HBase, the amount of columns is for illustration

The counter in the row key is hundred characters long, which creates the ability to use the row key for sorting. When a 'scan' is used to receive the contents of the HBase table, it will return the messages in ascending order. Since the row keys are ordered in a lexicographical order[20], the counter property in JSON is prepended with zeroes so that it consists of one hundred characters. This is with the default settings, but it could be configured to an arbitrary amount of characters. This results in receiving the messages in ascending order if the row keys are as long as all the other row keys. In the prototype it is used to receive the messages in order and not add the complexity in the other applications that rely on ordered data that is stored in HBase. For example, the ordering complexity does not need to be added in the state formatter since it already receives the messages in order, as long as the row keys are as long as the other row keys.

---

[19] When a 'nack' is sent to RabbitMQ, a not acknowledged message, it will keep the message in the queue.

[20] For more information, please visit the following website: http://wiki.apache.org/hadoop/Hbase/DataModel#row (HBase, Apache).

When the methods are invoked, for inserting data in HBase, and when processing completes, the method will return the control to the method which originally invoked the method. If no exception was raised during the insertion in HBase, an acknowledgement will be sent to RabbitMQ. If there was an exception, the code will catch this exception and the application will later on try to reconnect with HBase and a 'not acknowledged' message will be sent to RabbitMQ.

9.4.4. HBase consumer structure

All the methods in the HBase consumer heavily rely on each other and especially when it comes up on handling exceptions that may be raised. The 'receive' method runs inside a loop and invokes the 'timeout' method when an empty message is received. This is a back-off timer for not constantly polling the server for messages and does not result in a busy-waiting model. When the control is returned back to the 'receive' method it will re-try to receive messages from the queue. If a message is received, it will invoke the 'confirm_receive' method, which checks if an empty message is received, if so the 'lock_time' and 'connect_timeout_check' variable are set. When the message is not empty, it will invoke the 'hbase_processing' module for inserting the message in HBase using HappyBase and msgpack. The 'hbase_processing' method also invokes the 'row_key_counter' for prepending the 'counter' property with zeroes, as outlined earlier. When 'row_key_counter' finished prepending the zeroes it will return the resulting row key back to the 'hbase_processing' method, which will on its turn try to insert the message in HBase using the Thrift gateway. When the message is successfully inserted in HBase it will return control back to the 'confirm_receive' method, which will send an acknowledgement message back to RabbitMQ and return control back to the 'receive' method.

The preceding explanation on the relationships between the methods are outlined figure 13.

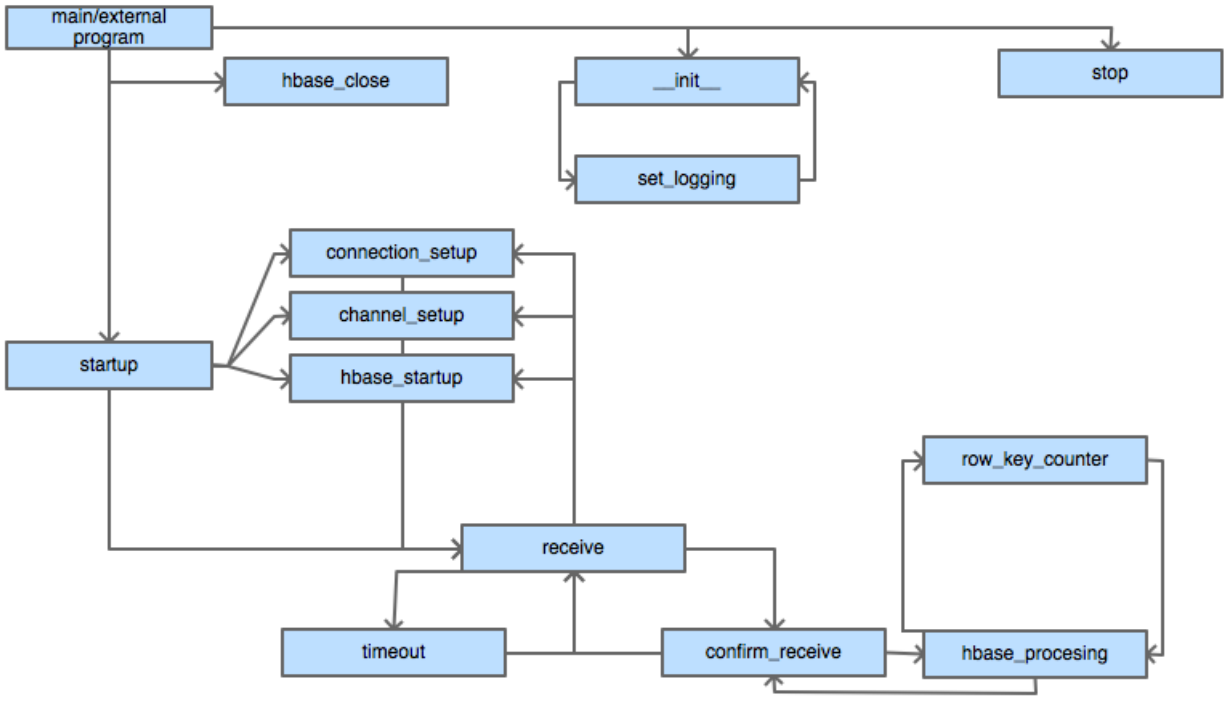

Figure 13 - Relationships between methods in the HBase consumer

If an exception occurred during processing, the exception may be caught one-step up in the hierarchy. For example, it is perfectly possible that the ‘hbase_processing’ method raises an ‘ErrorSendingData’ or ‘TTransportException’ that is caught by the ‘confirm_receive’ method, which will send a ‘not acknowledged’ message back to RabbitMQ and closes the connection with HBase.

The following image shows which methods have remote connections.

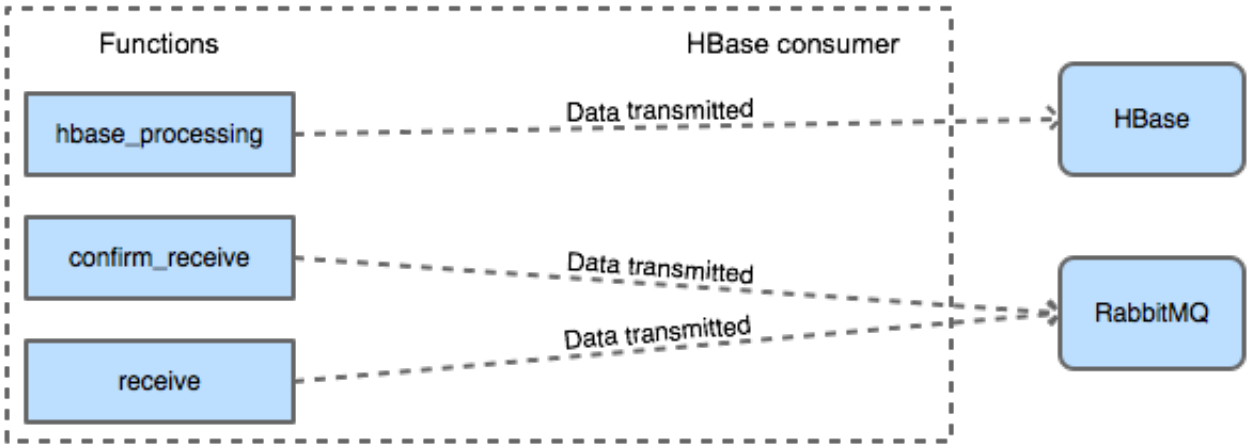

Figure 14 - High level overview of remote connections that exist in the HBase consumer

9.4.5. Unit tests

Unit tests are created for testing the HBase consumer application. Per test, it will create an hbase_consumer instance and will use the methods in the ‘HBaseConsumer’ class to receive data from and insert data in HBase. When messages are received or inserted, all results will be compared with each other. If the resulting output is the same as what is expected to be, the test will succeed.

The most important methods are tested, such as the ‘confirm_receive’ and ‘hbase_processing’ methods. The ‘timeout’ and the RabbitMQ connection are also tested. From processing the data till testing the data transmission between remote connections, it will be tested if this is done correctly.

9.4.6. Other tests

Other tests that have been performed were from some use cases’ point of view. For example, what happens when the connection with the Thrift gateway or the RabbitMQ queues are lost? Such use cases should be tested and were tested after the unit tests were developed.

A test was performed to see what happens when the connection with the Thrift gateway was lost. The HBase connection would simply be closed, more precisely the instance variables were cleared, and a 'not acknowledge' message was sent to the RabbitMQ brokers. This resulted in the message to be saved in the queue, so that it could be retrieved at a later moment. The application would periodically try to reconnect with the Thrift gateway, and if it succeeded it would pull the messages from the RabbitMQ queue and insert it into HBase. If it could not reconnect with the Thrift gateway it would set a back-off timer to try to reconnect at a later moment.

Another test was performed to see what happens when the connection was lost with one of the RabbitMQ brokers. The application will try to reconnect with another broker and if it succeeded it would pull the messages from that queue and insert it into the HBase table. If the application could not reconnect with one of the RabbitMQ brokers it would set a back-off timer. The application will then periodically try to reconnect with one of the brokers.

9.5. State machine

The fourth stage of the prototype was to build a prototype application that receives messages from multiple RabbitMQ queues and save this information in a MRT formatted file. This subchapter will describe the state machine consumer, its internal workings and what the final prototype can do.

9.5.1. General information state machine

The state machine is an application for storing the messages and a RIB state in an unified and standardised format. This format is saved locally on disk and allows other programs to read or copy this information to another application for further processing. The format used by the application, for storing this information in a unified and standardised format, is the MRT format, as specified in RFC 6396[21]. The reason for choosing this format has to do with the legacy systems, which uses the MRT formatted files as input for processing, but also because these files are used by the community. It is also a requirement to maintain the old formatted MRT files and that a new application can also create such files, since it is not possible to switch to a whole new format without maintaining the old format.

This application also creates two different files, based on the interval, for storing a RIB state and all BGP messages that ExaBGP received during this time interval. To accomplish this, the state machine connects to all the update queues of one RRC, and stores this messages locally in a SQLite database. Since SQlite is supported with a default Python installation, it was chosen to use SQLite as the local database. The SQLite database is used in this prototype and later versions of the prototype are recommended to use HBase. When a certain timestamp has been received, the application starts processing only one of the two files. If both files needs to be created, it is necessary to start the application twice with different settings in the configuration file. This distinction between 'modes' has to do with processing efficiency, one process only does one job and focusses on it. It has also to do with being able to temporarily stop consuming from a queue. For example, if one process both creates the RIB and update files, it could block one of these two when consuming from the queues is stopped. It must temporarily stop consuming from a queue, otherwise it would lead to an inconsistent state, but since it is blocking each other it is inefficient. It is therefore simpler and more efficient to have this distinction between these modes.

---

[21] Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format: https://tools.ietf.org/html/rfc6396 (IETF, October 2011)

9.5.2. Update mode

When the application runs in 'update' mode, it will pull all the messages from the RabbitMQ queues and tries to add these messages in its local database. When the application detects that a message reached a certain timestamp, it will be discarded and not acknowledged. Next, the application will remove the queue from the list of queues that the application knows, which results in a situation where the application temporarily does not pull messages from the queue. When all queues are removed from the queue, the application will try to generate a MRT formatted file.

Since the discarded messages are not acknowledged, the application will receive these messages when it tries to pull a message from the queue.

9.5.3. RIB mode

When the application runs in 'rib' mode, it will pull all the messages till a certain timestamp from the RabbitMQ queues, and tries to create a RIB state. The RIB state will be saved in a local database and some messages, which resides in the update queue, can have an effect on the RIB state. When the application detects that a message reached a certain timestamp, it will be discarded and not acknowledged. Next, the application will remove the queue from the list of queues that the application knows, which results in a situation where the application temporarily does not read the contents of a queue. When all queues are removed from the queue, the application will try to generate a MRT formatted file.

Since the discarded messages are not acknowledged, the application will receive these messages when it tries to pull a messages from the queue.

The keepalive, open and state messages could have an effect on the RIB state, which is also outlined in subchapter ‘9.5.5 State machine structure’.

9.5.4. MRT library

During this fourth development stage, it was necessary to create a MRT library for being able to create the MRT formatted update and dump files. Since there were no suitable MRT libraries that could be used, DPKT[22] is one of the few MRT serialisers and was not really working as it should, it was necessary to create such a library. The library can currently create the following MRT objects:

- TABLE_DUMP_V2
  - PEER_INDEX_TABLE
  - RIB_IPV4_UNICAST
  - RIB_IPV4_MULTICAST
  - RIB_IPV6_UNICAST
  - RIB_IPV6_MULTICAST
- BGP4MP
  - BGP4MP_STATE_CHANGE
  - BGP4MP_MESSAGE
  - BGP4MP_MESSAGE_AS4
  - BGP4MP_STATE_CHANGE_AS4

Other types or subtypes specified in the MRT RFC[23] are currently not supported by the library, since it also not used in the state machine. Since the modular structure of the library it is perfectly possible for one to create support for one of the other types or subtypes.

9.5.5. State machine structure

When the state machine is started, by executing the ‘state_formatter.py’ file, it first creates a ‘State’ object from the ‘state_rib’ or ‘state_update’ file, depending on the mode it is running in. If all the initialisation and declaration of the instance variables are done and set, it will call the ‘recv_messages’ method which runs in a while loop. This loop will constantly call the ‘receive_message’ method of the ‘Connector’ object, which is inside the ‘connector.py’ of the ‘rabbit_connector’ directory, for receiving messages from the queue. If it receives an empty message or ‘None’ object, it will temporarily not consume messages from that queue. When all queues are temporarily blocked the whole application will sleep for a configurable amount of time, so that all queues can be read from again. This is more efficient than constantly calling sleep since one empty message does not block the whole application, but only results in a situation where the application temporarily does not read from that queue.

If a valid JSON message has been received by the application, it tries to determine its type. This must be either one of the following types: “open, keepalive, update, state or notification”. When the type is determined, of the message that the application just had received, the method will be called that can do the processing for that message type. Every method that is invoked to do the processing on behalf of the application, tries to determine if the message is also in the allowed time interval.

---

[22] DPKT: https://code.google.com/p/dpkt/ (DPKT, Google Code)

[23] RFC 6396: https://tools.ietf.org/html/rfc6396 (October 2011, IETF)

When it is determined that the timestamp of the message is equal or higher than the allowed maximum timestamp, the message is discarded and not acknowledged. The ‘set_lock’ function of the ‘Connector’ object is also invoked, this invocation of this function will result in a deletion of the queue in the list of known queues. The queue will not be read till the ‘Connector’ object decides to call the ‘queue_list’ method[24], to determine all available queues. If there are not any queues left to consume from, the ‘Connector’ object will invoke the ‘file_close_and_reset_locks’ method of the ‘State’ object, the ‘State’ and ‘Connector’ object can invoke each other methods for signalling events to one another. When the ‘file_close_and_reset_locks’ method is invoked, the processing will begin. This will result in either an update file, which contains all the BGP messages that the application has received during a certain time interval, or a RIB dump file. When the processing of the file has been completed, the ‘rib_move_file’ method or ‘file_close_and_reset_locks’ will move the file to the ‘complete’ directory, where other applications can copy or read the contents of the file without raising a conflict since no write actions take place.

The rest of this subchapter will describe the effects that a certain message type could have on the resulting update file or the state of the RIB. If the reader wants to get a better understanding of all the functions and methods that exist, the reader could look at the next subchapter ‘9.5.7 Flowchart functions’ to get a better insight of the functions and methods that exist.

State messages result in an alteration of the RIB state when it is in ‘rib’ mode. If a peer goes down, all attributes of the previously announced routes are removed, but the timestamp of this removal is saved. It also results in the modification of the peer information in the database, which can be used by another application to determine if a peer is up or down. When a peer goes up it effects the state information in the database, which can also be used by another application, but also the state of the previously received prefixes that it had received. It will remove all prefixes for that certain neighbour.

When the application is in ‘update’ mode it will write out the BGP4MP_STATE_CHANGE or BGP4MP_STATE_CHANGE_AS4 message, depending if the peer supports 4 byte AS numbers. The application knows if the peer supports 4 byte AS numbers by looking at the negotiated capabilities in the open message. If it is not known whether it is supported, due to a state message that came before an open message, the application assumes that the peer supports 4 bytes AS numbers as a default.

Keepalive messages result in no alteration of the RIB state when it is in ‘rib’ mode. The timestamp of the last keepalive message is saved for a future version of the state machine. It could be used for determining if the keepalive messages are in time. If it is not in time an alert can be created for signalling that an expected keepalive messages has not been received in time. Currently the keepalive mesage is only used for determining if the timestamp is higher or equal than the maximum allowed timestamp. This is done to determine if the queue must be removed from the queue list and can also have the effect that the RIB file must be written to disk.

When the application is in ‘update’ mode it will write out the BGP4MP_MESSAGE or BGP4MP_MESSAGE_AS4 message, depending on the negotiated capabilities of the peer.

---

[24] This is also done during the initialisation and declaration of the ‘State’ object

Open messages result in no alternation of the RIB state when it is in ‘rib’ mode. It is only used to see if the peer supports the 4 byte AS number capability and is used when the RIB state is written to disk in the MRT format.

When the application is in ‘update’ mode, it will write out the BGP4MP_MESSAGE or BGP4MP_MESSAGE_AS4, depending on the negotiated capabilities.

Update messages result in an alteration of the RIB state when it is in ‘rib’ mode. The timestamp, body and neighbour IP is saved in the database. It is also used for determining the BGP ID of the neighbour, since this is only stored in an JSON update message from ExaBGP. All this information is saved in the database and is retrieved when the ‘start_processing’ method is invoked. When the ‘start_processing’ method is invoked, it will use the ‘rib_retrieve_attributes’ to receive all BGP attributes of the body message in binary representation according to the BGP RFC.

When the application is in ‘update’ mode, it will write out the BGP4MP_MESSAGE or BGP4MP_MESSAGE_AS4 message, depending on the negotiated capabilities with the peer.

Notification: Are acknowledged and all information in the database is deleted if it is a notification that tells the consumers that the RRC is offline.

9.5.6. Flowchart functions

The preceding subchapter describes the functions and methods that exist in the application. To give the reader a better insight of all these functions and methods, figure 15 gives the reader an overview of all these functions and methods that exist in the application.

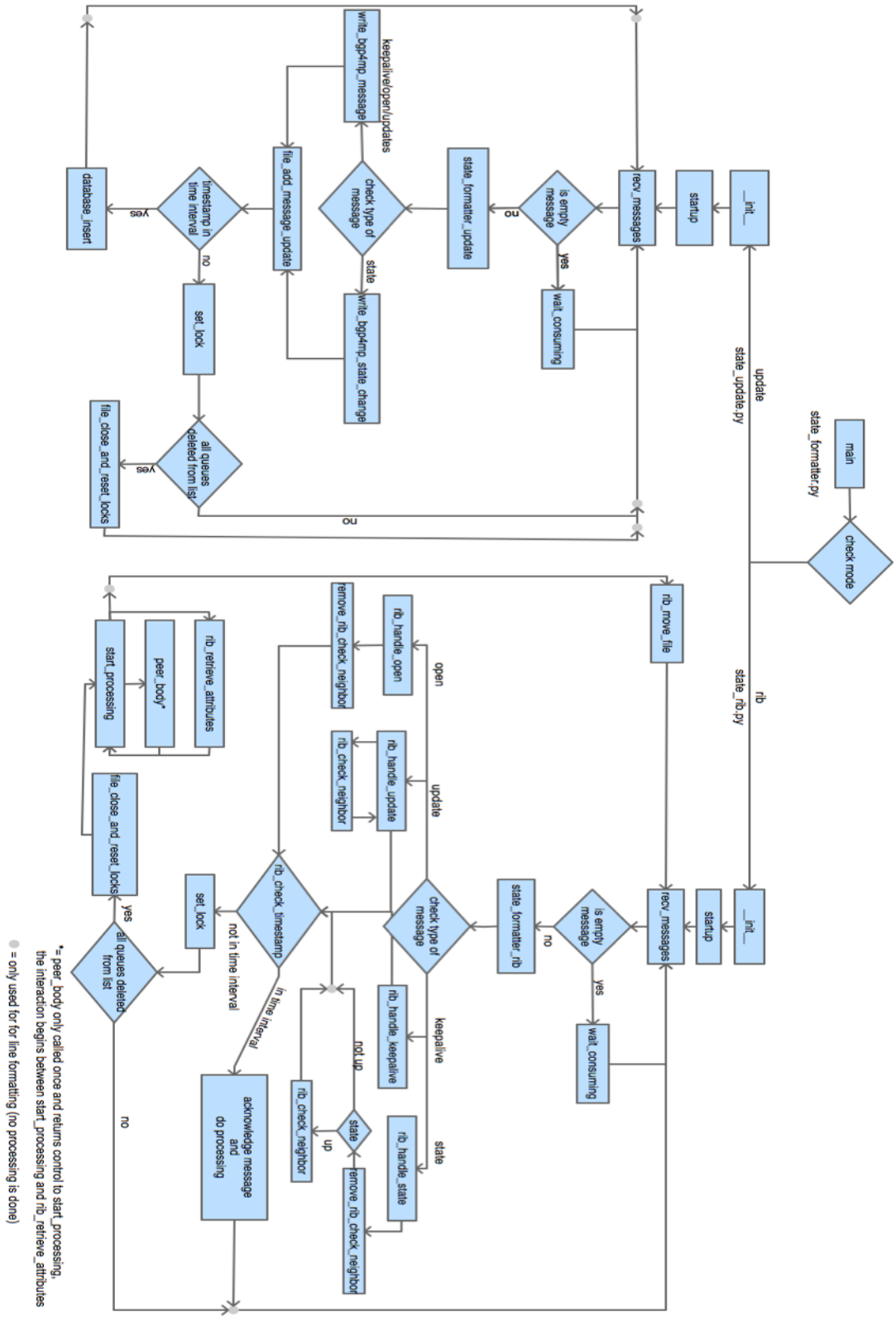

Figure 15 - Flowchart of functions that get invoked in the state machine

9.5.7. Unit tests

There are three unit tests that test the internal working of the state formatter and its connections that it has to maintain. There is a 'GeneralTestInfo' class, which is used for creating object of state_update and state_rib, which allows the unit test to have direct access to the database and the RabbitMQ queues. The following unit tests currently exist:

- DatabaseTest
- RabbitMQTest
- HandlerTest

DatabaseTest: tests if it can insert and retrieve data from the database. If the same data is retrieved, as it was originally inserted, it will pass the test. If the retrieved data differs, the test will fail.

RabbitMQTest: tests if there is a connection with one of the RabbitMQ queues, but also checks if there is data to be retrieved. If a connection could not be set-up with one of the queues, the test will fail.

HandlerTest: a JSON message will be passed to the 'state_formatter_update' and 'state_formatter_rib' methods and will be later retrieved from the database. Since the 'state_formatter' methods were invoked to process the data, it will do all the processing automatically. The expected result was compared with the data that was retrieved from the database. If the received data is the same as what is expected, the test will pass. This test ensures that the application follows the BGP RFC and that a consistent state could be created, or that a correct update file could be created.

9.5.8. Other tests

Other tests have also been performed to see if the application works as it should. Content that has been saved in the database, like the body message, were compared with the original body messages that resides in the ExaBGP messages. It was concluded that these messages were the same, but also that the MRT formatted messages, saved in the database, was the same as was expected. Some of these manually performed tests were later also added in the unit tests.

The files that have been created using the state machine, were also tested with several parsers. The parsers libbgpdump and mrtparse[25] were used to see if the resulting files could be parsed. With the final prototype, both parsers could parse the files that were created by the state machine.

[25] mrtparse: https://github.com/YoshiyukiYamauchi/mrtparse (mrtparse, Github, YoshiyukiYamauchi)

10. Deployment and testing

The deployment and test stage were meant for testing the prototype and deliver a product that could be released to third parties. In a meeting with the staff of the RIPE NCC, it was decided that the first week would be spend on pushing all code changes of ExaBGP upstream. The arguments for pushing these code changes upstream are mainly for maintainability and compatibility. Now that all the code changes are in the main stream branch of ExaBGP, no further maintenance has to be done on our code changes, since it is now being maintained by the original developer of ExaBGP.

Due to the new versions of ExaBGP, and since some JSON elements were renamed, some code changes needed to be done on the HBase consumer and state machine application. This was all done in that one week of time and the rest of the time could be used for testing the prototype.

Since the time was limited, it was decided that the testing should be more functional, instead of benchmarking. The focus of the tests are on the ‘integrity of data’ requirement, since this requirement is one of the most important ‘should’ requirements. The other ‘must’ requirements are either met or needs further testing, which could be a project on its own. It is also better to first test the integrity of the prototype, since the prototype its “core” is building a state. It would be more beneficial to first make the “core” solid and then perform some tweaks to improve the processing speed. In chapter ’12. Recommendations’ it is outlined which tests should be performed for testing the scalability and the ‘less delay’ requirement. These recommendations could be used for another project, which is more focussed on testing the prototype’s processing and throughput. If the reader wants to know if the ‘must’ requirements are met, it is advised to read sub-chapter ‘10.6 Conclusion’.

10.1. Integrity test

Two Quagga servers, later referred to as announcement servers, will have a peering relationship with two RRCs, later referred to as test RRCs. One of the test RRCs will run Quagga with the same configuration of the current RRCs that are in use, the other test RRC will run ExaBGP with all of it necessary components (i.e., queue systems, consumers etc.). The two announcement servers are artificial routers and will have a static configuration file, which results in the route announcements. Since the configuration file is static, and it is assured that the same routes will be announced to both the test RRCs, the generated MRT files can be compared with each other. These MRT formatted files will be used to detect any state differences.

The two test RRCs will be started approximately one minute past the hour and will generate a MRT formatted RIB dump on the hour and the update files every five minutes. The test will be stopped when both RRCs have generated their full RIB dump files, which approximately have the effect that the test will run sixty-five minutes.

This test will be run twice for comparing the resulting RIB files. If both files are the same, besides the timestamps and some of the other attributes (e.g. peer AS number), it can be concluded that the integrity of the data is as good as Quagga. The configuration file will also be compared with the resulting MRT formatted files for testing if the resulting state of the RIB file was what it was expected to be.

10.2. Integrity test with live RRCs

The two test RRCs, using Quagga and ExaBGP, will be connected to two live RRCs that will announce their routes to the test RRCs. The test RRCs will use the same time interval of generating the MRT formatted files as in the previous test. This will also result in approximately twelve update files and one RIB file, both formatted using the MRT specification. The test RRC using ExaBGP will be configured with a process-per-peer model were one ExaBGP process connects with one of the live RRCs.

It is harder to determine the integrity of the data since it can not be guaranteed that both test RRCs generate the same RIB files, which reflects that their state differs. This has to do with the update messages that they receive and that it can not be guaranteed that both the test RRCs receive the same update messages at the same time, which could be caused by the state differences of the connected live RRC. For example, Quagga receives a certain BGP message earlier than ExaBGP. Since ExaBGP receives the update messages later, it is perfectly possible that the live RRC received an update message during that time interval and that it resulted in a state change at the live RRC. The Quagga test RRC will receive a different update message than ExaBGP, which could result in state difference between ExaBGP and Quagga.

One other use case that should be kept in mind, is that Quagga is suspected of still processes the incoming update message while generating the MRT files. The state machine consumer, stops consuming form the queues when it received a timestamp from all the queues, which is higher than the maximum allowed timestamp. This will result in a situation where ExaBGP generates a RIB dump for exactly that timestamp and Quagga generating a MRT file and still changing the state of the RIB.

This test will be ran twice for comparing the resulting RIB files. If both files are the same, besides the timestamps and some of the other attributes (e.g. peer AS number), it can be concluded that the integrity of the data is as good as Quagga. If there are any differences it will be determined what the root cause of this problem is.

10.3. Monitoring

Zabbix will be used for monitoring and also for creating metrics. These metrics could be used during the tests and if there are any differences in the RIB files, to determine what the cause is of these differences. Some scripts are written or modified to gather metrics from the queueing systems, but also for gathering metrics of CPU and memory usage.

10.4. Test setup

During this stage a test setup has been created to simulate a real setup of the prototype. Several virtual machines were needed to run all the instances that were necessary to run the prototype. One server was used for virtualisation and was connected with two different switch ports. This test setup was used during the integrity tests and a schematic scheme of the network is showed below.

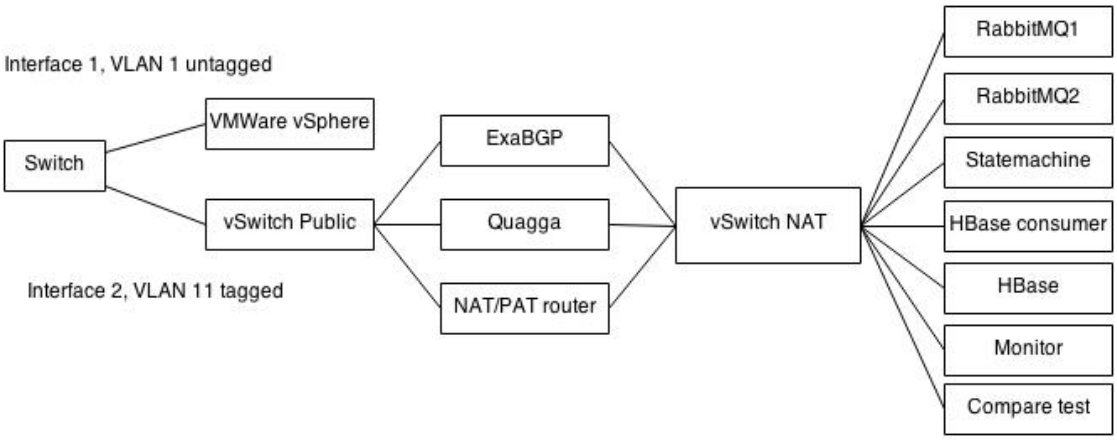

Figure 16 - High level overview of the test setup.

Three virtual machines have a public IP address and are connected to another virtual switch with its second interface. All the other virtual machines are connected to the NAT virtual switch and have a private IP address. All traffic from the virtual machines with only one interface, and which are only connected to the NAT virtual switch, will use the NAT/ PAT router for internet access.

10.5. Test results

This subchapter describes the results of the above described tests. For testing the integrity of the files, an application has been developed that compares the two resulting RIB dumps. The RIB dumps will be parsed by libbgpdump, and parsed once again by the developed application and inserted in the database. All the attributes are compared with each other.

10.5.1. Results of the integrity test

The test on 23th of May failed, but the test on 24th of May completed successfully. It was concluded that the integrity of the data is as good as Quagga offers. During the first test, an issue was discovered with the prototype were the AS number of the peer would be deleted when it received multiple state messages. This issue has been fixed an in the next test it was concluded that this patch was indeed solving the problem. Besides from the AS number that differed in the first test, and not in the second test, all BGP attributes of the 5040 prefixes were the same. All messages were accepted by the state machine and no messages were lost.

10.5.2. Results of the integrity test with live RRCs

During this test several differences were experienced in the attributes of the prefixes, but could be explained by the differences in time when the BGP message arrived. In the first test it was concluded that prefix 2a03:200:10::/44 was missing because the allowas-in option was not configured on the Quagga test RRC, this has been fixed in the second test. The prefix 84.205.70.0/24 was missing and this was because the prefix was later received by Quagga than ExaBGP. Other prefixes, which did not have the same attributes, were the result from messages that arrived at a different time, this was the case with two prefixes. The total amount of prefixes that have been captured by both the prototype and Quagga are 1029348 prefixes.

When the second test was executed, two prefixes were missing in the RIB dump of ExaBGP, which was again due to the differences in the arrival times of the BGP message. The BGP messages are tagged with the timestamp by ExaBGP on the moment they are processed, and are only processed by the state machine till a certain maximum allowed timestamp. There were three prefixes were the attributes differed from each other, again this was due to the differences in the arrival time of the BGP messages. In one case a prefix did not make it in to the RIB of Quagga, but it did make it in the update dumps. After researching why the prefix did not make it in the RIB dump, the only likely cause could be the AS path that the prefix had. The AS number of the test RRC appeared multiple times in the AS path of the received route. There were no other attributes that could cause the prefix not to be inserted in the RIB. During this test around the one million prefixes have been received by the test RRCs.

During these tests it was also concluded that ExaBGP saves more attributes than Quagga. Even the unknown attributes are saved in the MRT RIB dumps, whereas Quagga only dumps the attributes that it can parse. This way, ExaBGP provides users with more information, even if the attributes may not be known or can not yet be parsed, they are still inside the MRT files.

10.5.3. RabbitMQ performance

During the tests it was concluded that there was a serious performance bottleneck with the prototype. After trying to determine the cause, it was concluded that the bottleneck was caused by RabbitMQ and especially when adding messages to the queue. With acknowledgements turned on, the time it took to add all the messages to the queue was around the twenty to thirty minutes. When the acknowledgements were turned off, it only took about two to three minutes.

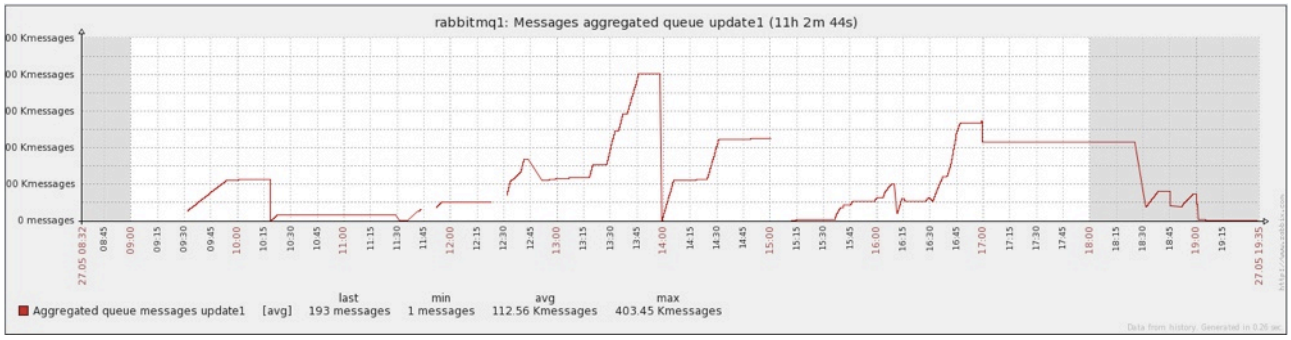

Figure 17 - Messages in RabbitMQ.

Figure 17 shows that the first test with acknowledgements, around 09:32, was executed. Around 13:00 a test was performed without the acknowledgements. As the figure reveals, with acknowledgements turned off, more messages can be added per five minutes.

Figure 18 shows that hundred thousand messages are added in 5 minutes without acknowledgements.

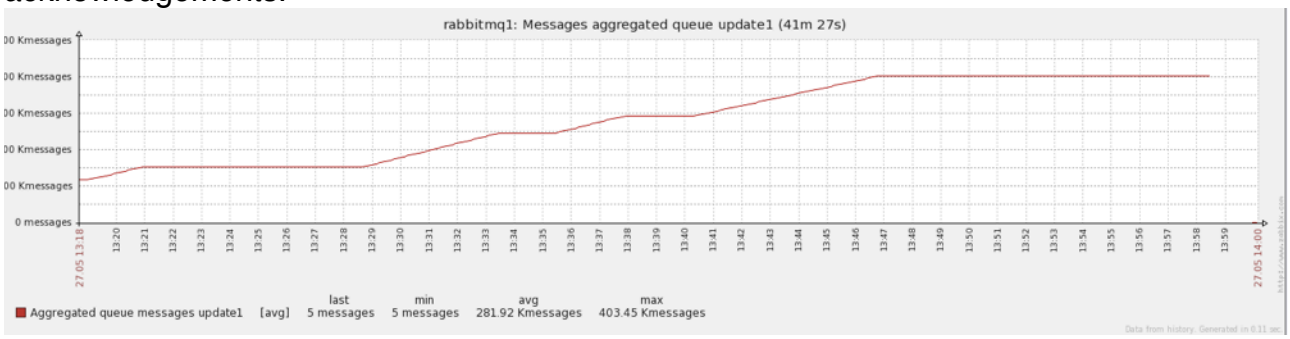

Figure 18 - Messages in RabbitMQ.

Figure 19 shows that twenty thousand messages can be added in 5 minutes with acknowledgements.

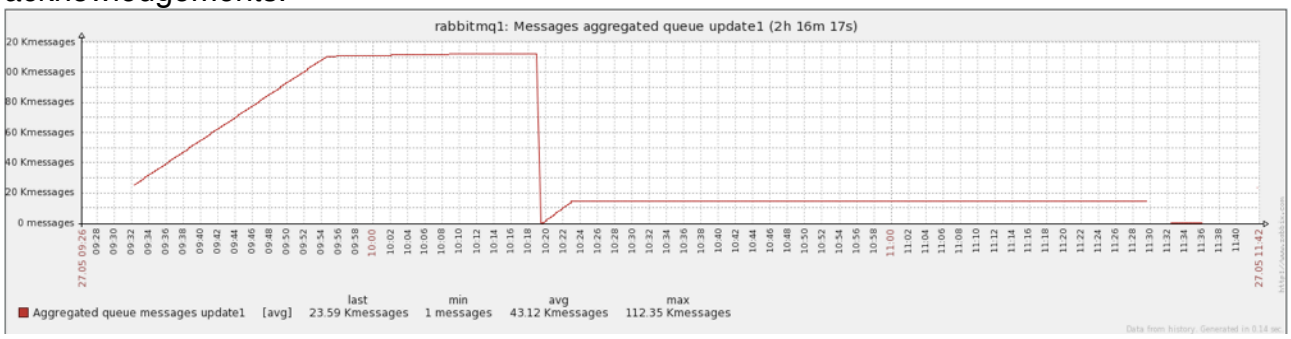

Figure 19 - Messages in RabbitMQ.

It could be the reason that the disk has to perform to many operations, the peak was around nine hundred operations per second, which is showed in the image below.

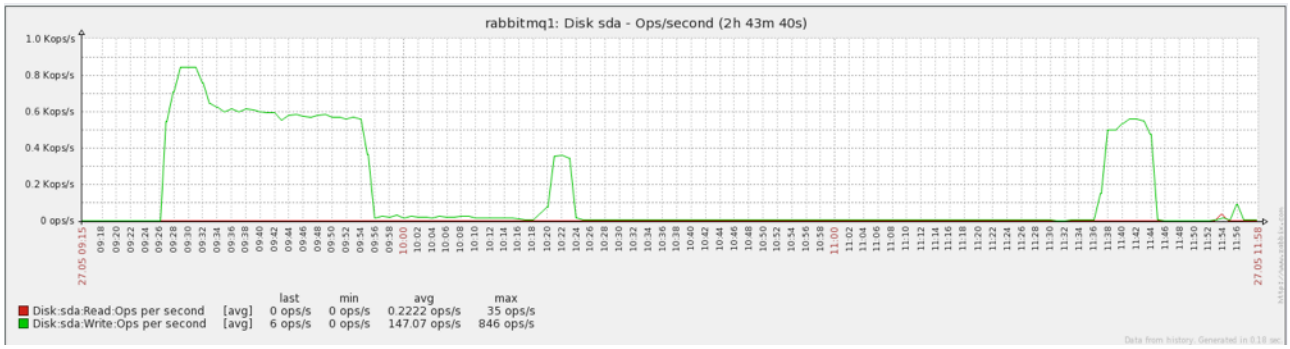

Figure 20 - Disk I/O of RabbitMQ server 1.

Note how relatively low the amount of operations are that need to be performed when the acknowledgements are turned off.

10.6. Conclusions

This subchapter describes if the prototype meets the requirements that are listed in chapter “5.3 MoSCoW scheme.”[26]

[26] If the readers wants to have more information about the requirements, it is advised to read the Research Report that has been added as an appendix to this document

10.6.1. Announcements of anchor and beacon IPs
This requirement is supported by ExaBGP. Although it has not been tested during the tests described in this chapter, it has been confirmed that routes can be announced to other peers.

10.6.2. MRT formatted files
This requirement is supported by this solution. The state machine can produce MRT formatted files, containing the BGP open, keepalive, update messages but also state changes. The state machine can also produce the RIB dumps formatted according to the MRT specification.

10.6.3. Raw data
This requirement is supported by this solution. There is a new option in ExaBGP, which has been developed during this project, to add the raw message inside the JSON string. The messages can be retrieved from the RabbitMQ queues and are also used in the state machine.

10.6.4. Metadata
This feature is supported by ExaBGP. There is a new option in ExaBGP to receive the keepalive messages from the peer and send it to the helper application. The helper application, in this case the producer, adds these messages in the RabbitMQ queues. These keepalive messages can later on be retrieved by consumers and could be used to determine if an RRC is still alive. Since keepalive messages are always send from a peer and when the RRC is offline, no keepalive messages will be added to the queues. If all peers are offline then there are no keepalive messages, but then there is also nothing to process, which basically means that the RRC is offline. The state changes, which are also a part of the metadata, as is described in chapter “5.19 Metadata” of the research paper, is also added in the queues. These state changes could be used to determine the state of the peer.

10.6.5. Ordering
The messages are stored in an ordered sequence in the RabbitMQ queues, as is described in chapter “9.3.7 Tests performed.”

10.6.6. Less delay
This solution allows that future work will bring the overall processing delay down. It provides messages in a stream oriented manner and consumers can process this data. The overall process delay, that is involved in RIS, depends on the overall RIS system and not only this solution.

10.6.7. Same attributes
The same attributes and maybe even more attributes are available in the JSON strings and MRT formatted files. This is because the raw data is added in the JSON string, so no attributes are lost from the moment ExaBGP receives a BGP message till the moment the message has been added in the queue.

10.6.8. Scaling
As stated earlier, this needs to be further tested. All the components of itself, in the overall prototype, are scalable but it needs to be tested how scalable it is. More nodes can be added to the RabbitMQ cluster and more processes can be launched for having several processes to handle multiple connections. But it must be tested where the bottlenecks are and how scalable the system itself is.

10.6.9. eBGP multihop
This feature is supported by ExaBGP and has been used in the test setup, since the test RRCs are connected with live RRCs.

10.6.10.Live data stream
Yes, this requirement is supported by the prototype. The messages that the producer receives from ExaBGP are added in the RabbitMQ queues. When consumers connect and pull messages from the queue, they will receive a stream of data instead of the batch oriented process used in the current RRCs. The delay that is involved from the moment that ExaBGP receives the message and when the consumers receive the message, depends on the processing speeds of ExaBGP, the producer, RabbitMQ and the consumers. It is near to impossible that there is a live data stream, but this is near close to a 'live' data stream.

10.6.11.Integrity of data
This solution provides even higher integrity than Quagga, since it also dumps the unknown attributes, from libbgpdump's point of view. To clarify this requirement, the integrity of the data means that no data is lost from the moment a BGP message is received till the moment the MRT formatted files are generated. During the tests it was concluded that the same attributes with the prefixes have been received by ExaBGP and that the same MRT files have been generated when you compare its content.

10.6.12.Extensible
This requirement is supported by the prototype. ExaBGP, the producer and the consumers are developed in such a way that extra features could be added to these systems in a modular way. When it is necessary to create a whole new application for processing, the application can connect to a queue, and pull the messages from these queues. In such a way independent applications can be developed and run concurrently without intervening each other. But as stated earlier, it is also possible to extend the feature set of ExaBGP, the producer and consumers.

10.6.13.High resolution timestamp
As described in chapter "9.3.1 ExaBGP modifications" this feature has been added to ExaBGP.

10.6.14.Authorisation and encryption
At this moment the traffic between the producer, consumers and RabbitMQ are not using authorisation or encryption. However, it is possible to add this, since authorisation and encryption is supported by RabbitMQ[27]. The current library that is in use, the pika Python library, currently only supports authentication and encryption[28]. To add support for this in the producer and consumers it requires some code changes. If the RIPE NCC decides that only internal applications can connect with the queues, it requires that the RabbitMQ queues can only be reached from within RIPE NCC's network.

---

[27] For more information about the authorisation and encryption capabilities of RabbitMQ, please visit the following websites: http://www.rabbitmq.com/blog/2011/02/07/who-are-you-authentication-and-authorisation-in-rabbitmq-231/, https://www.rabbitmq.com/ssl.html (RabbitMQ).

[28] For more information about the authentication and encryption support in pika, please visit the following websites: http://pika.readthedocs.org/en/latest/modules/credentials.html, http://pika.readthedocs.org/en/latest/examples/using_urlparameters.html (Pika, Read the docs).

11. Conclusion

This chapter describes the conclusion of the following main question:
“How can the current implementation of the RIS Route Collector be replaced with a better alternative, aimed to process updates faster, make information easier to integrate in the RIPE NCC Hadoop storage backends (e.g. through XML, JSON, or YAML), and meet the gathered requirements?”

As outlined in the research report and chapter 7 and 8 of this document, the best solution is not one application or program, but a modular approach with several programs that depend on each other. As outlined in chapter 8, the proposal was to use a custom solution based on ExaBGP and a queueing mechanism that holds the messages. Updates can be processed faster, and can be processed by several applications, as the update messages resist in a queue. Processing speed depends on the consumers, which consume messages from the queue, and can also insert the messages in the HBase tables. The messages of ExaBGP are structured in a JSON format and can be parsed by well known programming languages.

It is also outlined in chapter 10.6 that all requirements are met, except authorisation and encryption. This can easily be added to the current prototype and is not a requirement that must be in this version of the prototype. The integrity of the state, and its resulting RIB and update dumps, can be proven to be as good as Quagga. It is also the case that all attributes are saved in the RIB dumps, also attributes that Quagga drops.

The only thing that needs to be done is more scalability testing, which is also outlined in chapter “12. Recommendations.” All the must and could requirements are met in the current version of the prototype. The only ‘could’ requirement that is not met with the current version of the prototype is the ‘Authorisation and encryption’ requirement. With a few modifications, as outlined in chapter 10.6.14, it is possible to meet this requirement in a new version of the prototype.

Therefore, the end conclusion is that the best solution, to replace the current RIS Route Collectors, is a custom solution based on ExaBGP. It is proven, by testing the prototype, that it is indeed the solution that the RIPE NCC is looking for, based on the requirements. Only more testing needs to be done before this prototype can be used in a production environment. The integration of storage backends, like a Hadoop storage backend, is also possible, which is outlined in chapter 9.4, using the JSON messages in the queue.

12. Recommendations

This chapter describes some recommendations for this project.

12.1. Prototype recommendations

There are several recommendations for the overall prototype. These recommendations are for improving throughput or improving the overall working of the prototype.

12.1.1. Process manager

Only applies to the ExaBGP producer. Currently, when one of the processes exits without knowledge of the other processes, the application will hang or crash. Some kind of manager should be created that monitors the processes and when one process crashes that the manager will respawn that process.

12.1.2. Pipe connections

Only applies to the ExaBGP producer. It could be the case that a deadlock occurs when too many messages are added in the pipe connections. During the time that the prototype was used, this problem did not occur. However, tests should be performed or another implementation should be used, for communicating between the different processes in the ExaBGP producer.

12.1.3. Implementation of local saved messages

Only applies to: Producer. The save_local function in the producer should be redesigned and more tested to see if the save_local functions saves all messages locally on high load.

12.1.4. Maximum allowed timestamp

Only applies to: State machine. Currently, the timestamp is used for determining the maximum allowed time interval and the filename of the MRT file. The maximum allowed time interval is used to determine if its time to stop reading from a queue or to check if a dump file should be created. After the file has been created the maximum allowed timestamp is set to zero and on the next iteration it is determined what the next maximum allowed time interval should be. It could lead to situations were one timestamp is significantly higher than the other timestamps in the other queues. This could be improved by just increasing the maximum allowed time interval after the dump file has been created, based on the previous known maximum allowed timestamp.

12.1.5. Less delay

The goal of the new RIS system is to have less delay involved than with the current RIS implementation. The prototype is build to allow future work to gain less delay with the overall RIS system. It involves more work to make the overall system faster and to gain less delay, since the post processing mechanisms still rely on the MRT formatted files. The prototype is backwards compatible and still creates the MRT formatted files, but faster processing could be gained by making the current mechanisms be able to work with the JSON messages.

12.1.6. Database

The state machine currently uses SQLite for storing the update messages and for storing its state. If the database gets somehow corrupted, it can only re-create state when the RRC is restarted. Newer versions of the state machine should use the data, which is saved in HBase, to re-create state. It may also be useful to switch from SQLite to HBase and store state in there, so that other applications can query for the state in HBase.

12.1.7. RabbitMQ acknowledgements

As explained in chapter 10.5.3 processing efficiency could be gained when turning of the acknowledgements. The only problem with turning off acknowledgements is that the connection can get closed and it could be a problem in the library that is being used[29]. It is advisable to retain the acknowledgements, so that the producer knows if the message has been added to the RabbitMQ queues. To determine if the problem may be caused by the relatively high I/O operations, it is advised to test how RabbitMQ behaves when dedicated servers are used instead of virtual machines.

12.1.8. Queue deadlock

Only applies to: State machine. It could be the case when one queue stays empty that a deadlock occurs. The state machine waits before every queue has reached the maximum allowed timestamp. It should be the case when one queue stays empty, for some configurable amount of time, the state machine stops waiting for that queue.

12.1.9. Multi processes or threads for state machine

Only applies to: State machine. Efficiency might be gained by having a multi-process architecture were every process connects to one queue.

12.2. Testing recommendations

This subchapter describes tests that should be performed on the prototype. The recommendations outlined here could be used as input for a next project that is focussed on testing the prototype. These test recommendations could be used for testing the application for production readiness, whereas the original prototype was only meant for testing if it could meet all the requirements.

[29] Github issue: https://github.com/pika/pika/issues/397 (Pika, 30/05/2014)

12.2.1. Scalability

For properly testing the scalability of the overall solution, several tests need to be performed, and the results need to be compared with each other. The most important part that needs to be tested is the scalability of the producer and consumers. Since the whole concept, from the beginning, is generating state per RRC, it should also be tested if the current producers and consumers are doing this efficiently and provide mechanisms to scale when more peers are willing to connect with one of the RRCs based on the prototype.

For testing this, it is recommended, that the prototype is connected with a subset of the peers. Tests need to be performed what the influence is of having only one ExaBGP process connect with all of the RRCs, but also the other way around, testing how the application behaves when it is connected on a process-per-peer basis. Using such a model, an ExaBGP process gets launched for every peer, and process all the messages (e.g., putting it in the queue or saving it locally).

Another test that needs to be performed is the scalability of the state machine. Can it handle the messages of several peers and are there any bottlenecks in this one process model and that uses SQLite for saving the state? Tests need to be performed if a solution based on HBase or another SQL solution can better scale that the current implementation.

The queueing mechanism is also a component in the overall architecture that needs to be tested. Is the queueing mechanism scalable enough or is there a point when adding more nodes to the cluster is not useful anymore? Such scenarios should be tested.

12.2.2. Unknown attributes

One of the other requirements, listed in the requirements list, is the RAW data that exists in the ExaBGP JSON messages. First tests show that indeed all the RAW data is included in the JSON string, but it also needs to be tested how ExaBGP behaves when unknown attributes are send to ExaBGP.

Since the intention is to gather as much as data as is possible, it also involves gathering messages where the attributes are unknown to ExaBGP. It needs to be tested how ExaBGP behaves, which could be done by replaying BGP messages where the attributes are unknown, and send these messages to ExaBGP itself. The resulting state of the state machine consumer could then be determined by looking at the RIB dumps. If the same attributes are present, as in the original BGP message, it can be concluded that even unknown attributes, from ExaBGP's point of view, are saved in the MRT files.

13. Reflection

This chapter describes the process that has been undertaken during this project, but this chapter will also be used to reflect on the skills, which were necessary to carry out the job.

13.1. Skills

This subchapter will reflect per skill if it was indeed necessary for this graduation project. It will also outline why, from the author's point of view, the skill was achieved.

13.1.1. I.An3

The official description of this specific skill is: *"An existing, complex, large-scale or worldwide research on technology or methodology and analysing alternatives."*

This is in fact the whole first stage of the project, the *Research* stage, which was a whole stage dedicated on researching possible alternatives for the RRC systems and to create a proposal based on the research. The size of the research will be qualified under, *'complex'* and *'large scale'* research, since a lot of parties are involved, but also because it needs to be a solution for the problems that the RIPE NCC is facing with the current system. All the research that has been done can be found in the research report, which has been included as an appendix.

13.1.2. I.Ad3

The official description of this specific skill is: *"Ability to apply argumentation from a tech, business, costs/benefits, risks and legislation perspective."*

This specific skill was used during the first stage of this graduation project. It was necessary to have good arguments for defending the decisions that were made in the proposal, but also to defend why the proposal is the best decision that could be made. Arguments were from a technical point of view, extensibility and maintainability, and a costs/benefits points of view. The cost was calculated in time, the time that was needed to be invested in the solution (e.g., adding extra functions). All the arguments can be found in the research report, which has been included as an appendix.

13.1.3. I.On3

The official description of this specific skill is: *"Design a secured, multi-site, worldwide business network including possible security measures with specialistic and state-of-the-art technology."*

The network and infrastructure that is necessary for the prototype and eventually for the production site, is a secure multi-site business network. Although it may seem that not too much effort has been in invested in creating this whole multi-site, secured, state-of-the-art network, keep in mind that the whole solution is based from this point of view. The prototype does not send its traffic encrypted to the RabbitMQ servers, but only minor code changes are necessary to support such a feature, since RabbitMQ supports this. Generating the MRT formatted files which are lated processed by the legacy system, all these different components of the network needs to be supported by the application.

Therefore, from the author's point of view, these specific skills has indeed been necessary during this project. Without this skill, it was not possible to deliver an application which keeps all these different factors in mind, while developing and deploying such a prototype.

13.1.4. I.Re3

The official description of this specific skill is: *“Being able to prepare a customised application for deployment and testing.”*

This specific skill was necessary during the last stage of this project. The prototype needed to be tested, a deployment of the test setup was necessary, all of this was done by using this specific skill. Also using monitoring software and use the results as a basic of a conclusion falls under this skill. It is not only deploying and testing the prototype, but also correlating the results of the test, are the times when it is necessary to use this skill.

13.2. Reviews

On 28th of May a mid-term review was held to review the student. The review has been added as an appendix of this document and has the following conclusion :
“Wouter has performed well throughout his project, and designed and delivered a useful prototype system in a well-organised and coordinated manner.”

13.3. Personal goal

One personal goal, of the author, was to improve verbal and written expression. During the project the author was exposed to the English language and improved his writing and verbal skills. Documentation needed to be written, but also presentations needed to be given for the colleagues of the RIPE NCC.

The technical skills that the author wanted to improve on, such as BGP and knowledge about inner workings of systems, were a personal goal of the author. During this project a lot of knowledge was gain about BGP, the inner workings of systems, but also how different systems can be created to work with each other (e.g. the producer, which adds messages in the queue and the consumers that read these messages). During the development stage, more experience was gained with Python during the project, since the producer and consumers were written in Python.

13.4. Meetings

During the graduation project there were weekly meetings where the status of the project, and the next steps that should be taken, were discussed. These weekly team meetings were also used for taking decisions on the directions that the project should go to. Input for these meetings were both from the author and the supervisors of the RIPE NCC. They wanted to know what the status of the project was, and the author wanted to offer proposals with arguments, for taking decisions related to the direction of the project. These proposals were mostly presented during a presentation for the staff of the RIPE NCC or during the meetings.

14. Resource list

This chapter lists all the resources that have been used during this project.

Books:

- Schagen, J.D, van der Kwaak, W., Leenstra E., Smit. W. en Vonken F. *Bachelor of ICT - domeinbeschrijving*, Amsterdam, 2009

Publication of an institute:

- *A Guide to the Business Analysis Body of Knowledge*, International Institute of Business Analysis, Whitby, ON (Canada), 2009

Websites:

- http://www.openbgpd.org/
- http://www.ripe.net/
- http://www.rabbitmq.com/
- http://hornetq.jboss.org/
- http://zeromq.org/
- http://kafka.apache.org/

Specific page of websites:

- http://www.ripe.net/lir-services/ncc/staff/ripe-ncc-staff-structure (RIPE NCC staff structure)
- https://wiki.python.org/moin/GlobalInterpreterLock (Information about Python GIL)
- https://docs.python.org/dev/library/multiprocessing.html#multiprocessing.Connection (Information about the multiprocessing pipe connections)
- http://wiki.apache.org/hadoop/Hbase/%20DataModel#row (HBase row lexicographical ordering)
- https://code.google.com/p/dpkt/ (Google Code DPKT project page)
- https://github.com/YoshiyukiYamauchi/mrtparse (Github mrtparse project page)
- https://github.com/pika/pika/issues/397 (Issue listed on Pika Github page)
- http://www.rabbitmq.com/blog/2011/02/07/who-are-you-authentication-and-authorisation-in-rabbitmq-231/ (Authentication and authorisation mechanisms in RabbitMQ)
- https://www.rabbitmq.com/ssl.html (Encryption mechanism RabbitMQ)
- http://pika.readthedocs.org/en/latest/modules/credentials.html (Pika authentication mechanisms)
- http://pika.readthedocs.org/en/latest/examples/using_urlparameters.html (Pika SSL mechanism)
- http://bgpmon.netsec.colostate.edu/index.php/join-the-peering/peering-faq (BGPmon Peering FAQ)
- http://www.slideshare.net/charmalloc/apache-kafka (Apache Kafka presentation)

RFCs or drafts:

- https://tools.ietf.org/html/rfc6396 (MRT RFC)
- http://tools.ietf.org/html/draft-ietf-grow-bmp-07 (BMP draft)
- http://www.ietf.org/rfc/rfc4271.txt (BGP RFC)

15. Appendixes

All documents and source code, which has been developed during this project, are added as an appendix. It is decided, with the approval of the Hogeschool van Amsterdam[30], that all appendixes will be delivered on a CD. If a CD was not included with the reader's copy of this document, the reader can ask for a copy using the contact details in this document.

[30] During the mail conversation on Tuesday 13th of May 2014