Scaling AMS-IX Route Servers

University of Amsterdam, Security and Networking Engineering

David Garay

August 18, 2019

Supervisors (AMS-IX):
Stavros Konstantaras

Supervisors (UvA):
Prof. dr. ir. Cees de Laat

Abstract

In 2018, more than 2,730 Autonomous Systems on the Internet were affected by 12,600 incidents[1] (including outages, attacks, route leaks and hijacks); and new incidents appear on a daily basis[2]. BGP policies are a key countermeasure operators have against these incidents[3]. The two route servers at AMS-IX connect more than 714 clients each and manage more than 340,000 prefixes. Because of their central position in the IXP's topology, they play an important role in enforcing the BGP policies of their clients. To ensure the consistency of these policies, the IT systems of AMS-IX query the Internet Routing Registry database (IRRDB)[4] on an hourly basis. As the number of clients and prefixes increase yearly, concerns about their scalability and maintainability arise. What are the limits of the current architecture? Are there other ways to receive updates in real time? And if so, what are the consequences of these alternatives? To address these concerns, we identify, reproduce and measure the impact of the current BGP policy update process on the network infrastructure. Our tests revealed that the route server's blocking time (CPU utilization at 100% during reconfiguration) is the main bottleneck in the system, and that it increases with the size of the route server's configuration file and the amount of route server clients undergoing policy changes. Furthermore, by studying the frequency of the policy changes based on historical data, we are able to evaluate and discuss the blocking probability of the system, and discuss ways to improve its scalability. Finally, we propose a Publish-Subscribe messaging system that enables real time processing of BGP policy updates and explain the changes required on the current architecture. We discuss how to decrease the processing time by improving the way policy updates are processed at IXPs, and show the feasibility of the design with a proof of concept. We conclude recommending IXPs to monitor usage statistics of BGP policy updates in production and compare them against actual measurements on their network infrastructure.

Keywords *– BGP Policies, IXPs, RPSL, Route Servers, Bird, PubSub, Messaging*

1 Introduction

The Amsterdam Internet Exchange (AMS-IX) is the second largest internet exchange in the world, in terms of peak throughput[1]. Internet exchange points (IXPs) facilitate the interconnection of organizations (typically Internet Service Providers, ISPs), by offering a high capacity local network.
Route servers (RSs) occupy a central position in the IXP's infrastructure in order to help IXP clients establish *multilateral interconnections* and exchange of network topology information, and in this way avoid a full-mesh topology. Because of their position, route servers also contribute in applying policies that the network operators specify.
The IT systems of the AMS-IX update on an hourly basis their Border Gateway Protocol (BGP) policy information of its route server clients. While this is a relatively short interval when compared to other IXPs (see Section 2.2.1), it still leaves the system working with stale data for up to one hour, in the worst case. Because the number of clients and prefixes handled by the route servers of AMS-IX keep increasing[2], concerns arise about the scalability and maintainability of the current systems. AMS-IX would like to improve the current policy update process, make it more timely and the systems more scalable. AMS-IX would also like to know the requirements and consequences of these improvements.
Since data sources and tools used to process BGP policies are developed, maintained and widely adopted by the community[5][6], it is likely that the challenges faced by AMS-IX are also present in other large IXPs. To the best of our knowledge, there is no scientific research on the improvement of the policy update process of route servers at IXPs.
The aim of our research is to evaluate the performance of the current BGP policy update process, identify and measure bottlenecks. More concretely, we design experiments to measure the CPU, memory and network traffic impact on the route server and its clients, and study the influence of the size of configuration files and protocol updates triggered by policy changes.
In Section 2.2 we provide an overview of data sources that are involved in the BGP policy update process. We chose to evaluate in detail the Internet Routing Registry Database (IRRDB) because it has been operational the longest and it is likely to offer more opportunities for improvement.
The main contributions of our research are:

- We identify the main bottleneck in the system, and describe how the size of the configuration file and the number of route server clients undergoing protocol updates (see Section 4.1.2) increase it. Furthermore, we show the trend and frequency of BGP policy changes at the AMS-IX using historical data from RIPE[4].
- We introduce the usage of the Erlang B formula[7] to predict the blocking probability of the system, and hypothesize over the system's behavior in normal and busy scenarios. Furthermore, we discuss how the Erlang B formula can help to identify capacity expansion alternatives and optimization opportunities.
- We propose a Publish-Subscribe messaging system that enables processing in real time of policy changes, we suggest improvements to the IXP IT

[1] https://www.pch.net/ixp
[2] https://stats.ams-ix.net/rs-stats.html

and route server systems to decrease the blocking time in the current architecture and show the feasibility of the design with a proof of concept.

Our paper is structured as follows: Section 2 provides background information on the domain of BGP policies, route server architecture and related data sources. In Section 3 we discuss the setup required for the experiments and their design. We present the results in sections 4. In Section 5 we list the requirements and propose our Publish-Subscribe messaging system and countermeasures to decrease the blocking time.
Finally, in Sections 6 through 7, we reflect on the results and proposed design, present our conclusions and suggestion for future research.

1.1 Research question

To help us achieve the goals aimed in this paper, we answer the following research questions:

- *Regarding the route server's BGP policy update process, what are the main performance and scalability indicators? What are the bottlenecks of the process? What is the impact of these bottlenecks?*
- *If required, how can we improve these indicators in a new, achievable design?*

1.2 Literature review

Vouteva and Turgut describe[8] the RS configuration architecture of AMS-IX, its tooling and current BGP configuration practices, and propose a tool to automate BGP configuration as a whole. Automated BGP configuration is a key component in the process of BGP policy configuration, and therefore, to our research.
Regarding the scalability of BGP route server implementations, Jenda Brands and Patrick de Niet looked at BGP Parallelization[9] as a way to overcome the CPU bottlenecks in route servers, which cause long convergence times. While in their research focuses on convergence time of the systems, it helps our research in confirming the relation between number of peers undergoing `BGP UPDATES` and the time the CPU runs at 100% utilization.
We reevaluate the integration options between the data sources and the route server, and propose a design based on a Publish-Subscribe messaging integration pattern, described by Hohpe et al. in Enterprise Integration Patterns[10].
We make extensive reference to relevant RFCs, namely on: BGP, Routing Policy Specification Language (RPSL), Resource Public Key Infrastructure (RPKI), and the operational and security aspects of route servers.
Finally, practical aspects of the experiments and proof of concept are possible thanks to the following projects/services: the Bird routing daemon[5], aroute-server configuration parser [6] and Google's PubSub cloud service.

2 Background

In this section we cover the necessary background information to understand the rest of the paper. We point to additional information where appropriate.

2.1 BGP Policies

Policies help network operators achieve certain goals. For instance, network operators might want to filter out non-optimal routes, to avoid paying transit

charges.
Caesar et al.[11] categorize policies in four groups, namely business relationship policies, resulting from the relationship between the network operator and its neighbor; traffic engineering policies, which are meant to control the quality of service and traffic load on the peering links; scalability, meant to reduce control traffic and load on routers, and security policies, to protect the network operator from malicious or accidental attacks.
RSs might contribute in applying policies of RS clients due to its central location in the IXP infrastructure. For example, they can filter of prefixes depending on the policies specified by administrators of RS clients on IRRDB.
RFC-7454: BGP Operations and Security [12] summarizes common existing guidelines defined by the Internet community and help operators implement BGP policies.

2.2 Baseline Architecture

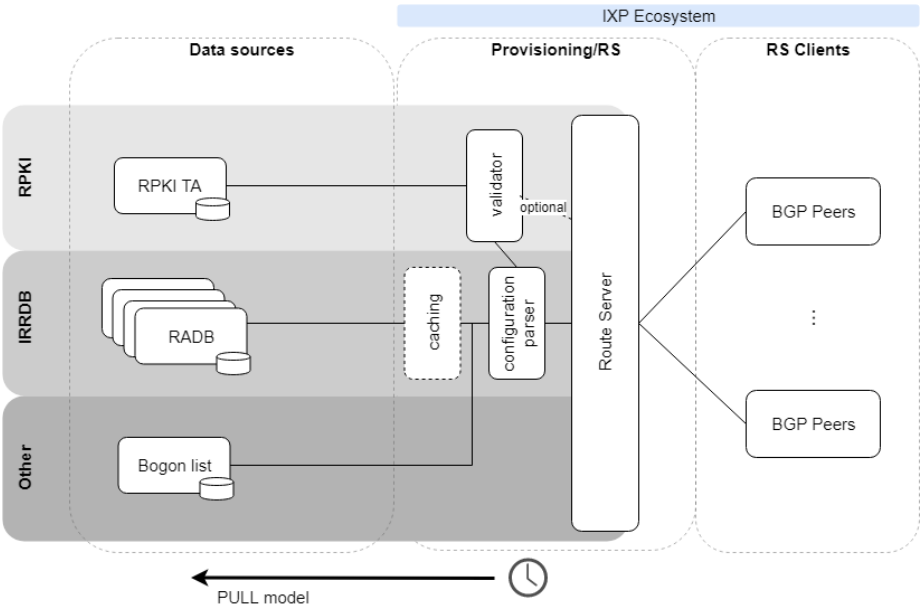

Figure 1: Baseline architecture: IRRDB, Resource Public Key Infrastructure and Bogon prefix validation processes, components and interfaces

To illustrate the interactions between the components in the architecture depicted in Figure 1, we describe below the main components in the architecture.

2.2.1 The route server

The route server's main function is to broker network reachability information among its RS clients, forming in this way a multilateral interconnection. In a multilateral interconnection, three or more external BGP speakers exchange routing information via a route server[13]. In this way a full mesh topology (and the resulting higher complexity and operational costs) is avoided. RSs are different from a route reflector because they use eBGP instead of iBGP, and they

need to support path distribution on a per-client basis. This requires special handling of BGP UPDATE and path calculation functionalities, not present in normal BGP implementations[13]. Because of the broker role of route servers, they also play an important role in enforcing BGP policies.

IXP	Clients	Connected to RS	Update frequency
AMS-IX[14]	845	714	1 hour
DE-CIX[15]	870	846	6 hours
LINX[16]	819	640	at least 3 hours[17]

Table 1: Route server clients and update frequency

Table 1 shows the amount of networks peering with the route servers and the frequency with which route servers update BGP policies in major IXP's.
RFC-7948: Internet Exchange BGP Route Server Operations[18] and Brands et al.[9] deal with the main aspects relevant to scaling RSs on the network side.
Bird is the main route server implementation used by IXPs[5]. Bird runs on a single thread and uses only one CPU core, and this explains the blocking behavior during reconfigurations (CPU utilization becomes 100% while the new configuration file is processed)[9].
The configuration of Bird is done with a text file which contain directives, data structures, functions and filters. For example, an RPSL route-set object from the IRRDB is typically translated by the Configuration Parser into a set data structure containing prefixes in the configuration file. Other data structures include the Bogon prefix list, and the Routing Origin Authorization (ROAs) used by the Resource Public Key Infrastructure (RPKI). In general, the main contributors to the file size are:

- ROA records: this table contains information about prefixes, network mask lengths and their allowed origin ASNs. This part of the configuration is static, meaning it does not increase with the number of peers.
- RS client configuration: typically containing protocol details, such as peer ASN number and IP addresses. Additionally, to support the policies applied at the route server, functions (for example, check_prefix_lengths) and filters (prefix_list for a given ASN) are typically used in the configuration file. This part of the configuration varies with the number of peers, and the size of the objects they refer to (the size of the prefix_list, in our example).

Example of Bird configuration files are available in our project repository[3].
To take new configurations into use, Bird reads them without restarting the BGP protocol, by default. However, changes in filters (for instance, those introduced by an updated BGP policy) will usually lead to a restart of BGP[19]. In Section 4.2 we show how frequently changes in BGP policies occur.

2.2.2 The Configuration Parser

The Configuration Parser periodically aggregates the information retrieved from data sources in Figure 1, plus any other changes that might have happened in the environment (e.g. new peers, new interface addresses), and generates a new configuration file for the route server. The Configuration Parser typically

[3]https://bitbucket.org/david-garay/rp2

includes in the configuration file a set of common rules derived from best practices among IXPs and network operators, such as basic filters and prefix / origin ASN validations. Examples of the former are minimum and maximum prefix length and bogon filters. Examples of the latter are IRRDB- and RPKI-based filtering).[6]

2.2.3 RPKI - Resource Public Key Infrastructure

RPKI provides mechanisms for validating the contents of BGP announcements and map an origin ASN to a list of prefixes. In this way, attacks such as route hijacks are prevented. To achieve this, it relies on three main elements[20]:

- The RPKI: defined following the current organization for IP block and ASNs allocation, with IANA at the root of the hierarchy, and the 5 RIRs (Regional Internet Registries) below, managing a defined geopolitical region. Currently, the RPKI Trust Anchor (TA) function is carried out by the RIRs[4].
- Routing Origin Authorization (ROAs): provides an explicit authorization that a given autonomous system is permitted to originate routes to a set of addresses. The ROA contains the ASN to be authorized, the prefix that might be originated by the ASN and the maximum length of the prefix.
- A distributed repository system: to validate all ROAs, network operators need to acquire all the certificates and certificate revocation lists.

2.2.4 IRRDB - Internet Routing Registry Database

The IRRDB is a set of databases maintained by different organizations on the internet (RADB[5], NTT[6] and RIPE NCC[7], to name a few). The information on the IRRDB is represented using the Routing Policy Specification Language. In the baseline architecture depicted in Figure 1, multiple databases compose the IRRDB, serving different regions of the world. The services these registries provide are exposed through the following interfaces[4]: whois, RDAP[8] and REST APIs[9]. Moreover, network operators can host their own copy of the database while receiving near real-time updates thanks to the Near Real Time Mirroring protocol (NRTM)[21] (denoted as caching in Figure 1). RIRs typically offer network operators web, email or API interfaces, so they can update their RPSL objects[4].
RPSL - Routing Policy Specification Language: RFC-2622[22] states that RPSL is designed to contain a view of the global Internet policy on a single-cooperatively maintained database. RPSL is object oriented, and uses classes to represent policy and administrative objects.

```
1 aut-num:          AS29263
2 org:              ORG-GDH1-RIPE
3 as-name:          HAAGNET-AS
4 ...
5 import:           from AS6777 action pref=200; accept AS-AMS-IX-
      ROUTESERVER
6 export:           to AS6777 announce AS-HAAGNET
7 ....
```

[4] http://localcert.ripe.net:8088/trust-anchors
[5] http://www.irr.net/
[6] https://www.us.ntt.net/support/policy/rr.cfm
[7] https://www.ripe.net/
[8] https://www.icann.org/rdap
[9] https://apps.db.ripe.net/db-web-ui/#/query

```
default:          to AS-KPN
admin-c:          HH3038-RIPE
tech-c:           HRA19-RIPE
status:           ASSIGNED
mnt-by:           RIPE-NCC-END-MNT
mnt-by:           HAAGNET-MNT
created:          2003-07-16T08:45:53Z
last-modified:    2017-11-15T09:21:21Z
source:           RIPE
```

Listing 1: Example content of aut-num object

Listing 1 is an example of an aut-num object. The attributes indicate the autonomous-system number (aut-num: AS29263), name (as-name: HAAGNET-AS), and administrative information. Multiple import/export fields are allowed, and in the example above we highlight the policies of HAAGNET-AS relevant to the route servers of AMS-IX (AS6777). The import policies refer to the prefixes the routers at HAAGNET-AS expect to receive. More concretely, they specify a BGP attribute the routers of HAAGNET-AS apply (pref=200) to the list of prefixes (`AS-AMS-IX-ROUTESERVER`) the routers of HAAGNET-AS are expecting. The policies on the export field refer to the prefixes being advertised by the routers of HAAGNET-AS: they only advertise the prefixes in the `AS-HAAGNET` prefix list[10].

2.2.5 Team Cymru bogon prefix list

Bogons are prefixes that should not appear on the routing tables (for instance, private ranges defined by RFC-1819 and prefixes that have not been allocated to a RIR). Filtering these prefixes helps in anti-spoofing and preventing their usage in DDOS attacks. The list of these prefixes is updated dynamically by Team Cymru[11].

2.3 Application integration patterns

Applications don't live in isolation. Integration deals with how applications interact with each other. Hohpe et al.[10] outline the main application integration approaches: file transfer, database, Remote Procedure Call (RPC) and messaging.

In a file system integration, one application writes and another reads. Applications need to agree on timing, filenames and locations, and who is in charge of deleting the files. In a shared database, multiple applications read from a single physical database, and one schema. The disadvantages of these approaches is that they produce tightly-coupled systems: each application must have specific knowledge about the files or database composition, structure and data format. Coming up with a unified schema suitable for different applications can be challenging. Another disadvantage of these systems has to do with the timeliness: the time it takes from when one application decides to share data until the other application can consume it can be large, which causes applications to work with stale data.

Ideally, one application would invoke behavior from another application when an event (such as an update) occurs. In an RPC system, applications communicate by means of synchronous calls over the network, the same way local calls are

[10] https://www.ams-ix.net/ams/documentation/ams-ix-route-servers
[11] https://www.team-cymru.com/bogon-reference.html

made. The communication occurs in real time and synchronously. The problem with this approach is that network performance problems and failures are often not evident, while these are pervasive problems in distributed systems.[10]
Integration using messaging promotes loose-coupling between systems, scalability and hides network unreliability.

2.3.1 Publish-Subscribe messaging

Messaging systems enable messages produced at one producing application to be transported to a consuming application. To accomplish this, they rely on a Messaging System, also called Message-oriented Middleware[23][10]. Compared to RPC, messaging systems allow, among other things, synchronous and reliable communications, throttling and mediation. A key feature that enables these capabilities is the store-and-forward operation. When a message is sent to the messaging system, the message is stored and then delivered. An offline consumer can retrieve the message when it comes back online.
Messaging channels are logical addresses in a messaging system[10]. They are meant to transfer a particular type of information. The sending application does not know what particular application will consume its message, but it can be assured it is interested in the information. A Publish-Subscribe a type of channel well suited to broadcasting information. In a Publish-Subscribe channel, endpoints need to subscribe to so-called topics before they can receive notifications. Whenever an endpoint publishes a message to this topic, the message is either consumed by a subscriber immediately, or remains in the messaging queue until it expires.

2.4 The Erlang B formula

Whenever a system is able to receive request for event processing simultaneously, in the absence of a queue, events might be lost because the system is busy handling other requests. A typical example of this is a telephone switch with a limited amount of lines serving a bigger number of subscribers. The probability that a call is lost is called the blocking probability.[7]
The Erlang B formula provides the blocking probability, as long as the arrival process is a Poisson process (criteria satisfied with a large amount of independent events)[7]. The blocking probability is a function of the number of resources (telephone lines, in our previous example) and the traffic intensity ($T_i = n * t_p$, where T_i is the traffic intensity, n the number of events and t_p is the average time the resource is kept).[7] Other approaches to modeling the blocking probability are discussed in the section "Analyzing service capacity" of Distributed Systems[23]. However, we chose for the Erlang B formula since it is well-known in telephony applications and recommended by ITU-T. The formula and its parameters are described in detail by Parkinson.[12]
Our current policy update process works in Pull mode, which means updates are retrieved at configured time intervals. In this mode we are in control of the time to pull information, and throttling is achieved naturally because pulling only happens when the system is free. The downside of this mode of operation is that when the retrieval interval is large, the data becomes stale. On the other hand, pulling more frequently comes at the cost of higher resource usage.
To have a more timely processing of events, we want to consider a Push mode of operation. Here, BGP policy updates are sent to the IXP as soon as they

[12]http://www.kt.agh.edu.pl/ brus/kolejki/Tutorial.pdf

are produced. It follows then that with an increased blocking time, collisions are more likely to happen. The Erlang B formula can provide the blocking probability of the route server, help us predict scalability limits resulting from the blocking probability and elaborate strategies to expand the system capacity.

3 Methodology

Our goal is to describe the impact BGP policy updates have on the CPU, memory of the route server and the traffic between the route server and clients. Our approach consists of three steps:

- Evaluating the impact of updates: in our exploratory tests, we identify the file size and the number of route server clients undergoing `BGP UPDATEs`[24] to have an effect on the blocking time of the route server. The route server sends the `BGP UPDATEs` whenever policy changes are processed.
- Evaluating the frequency of BGP policy changes: here, we assess the amount of changes coming from the IRRDB by counting the version history of the `aut-num`, `route` and `route6` RPSL objects.
- Profiling of the Bird routing daemon: we use a C profiler with Bird to learn which functions are called and how much CPU time is spent in them during reconfigurations.

3.1 Evaluating the impact of updates

To evaluate the impact of updates, we look at the effect of the file size and the amount of route server clients on the network infrastructure.

3.1.1 Test environment

In our setup the route server is configured with rules equivalent to the ones used in production networks[6] (for instance RPKI/IRRDB validations, bogon filters, etc).

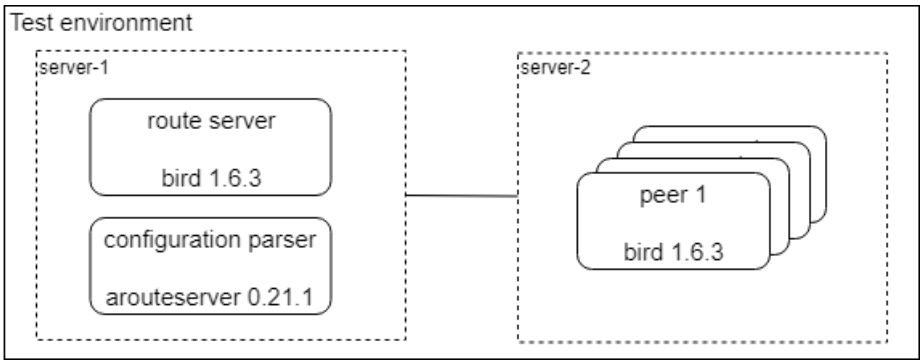

Figure 2: Test environment, component locations and dedicated link

Figure 2 depicts the main components of our setup:

- Configuration Parser - arouteserver[6] v0.21.1: Python tool that automatically generates configuration for IXP route servers.
- Route Server - Bird 1.6.3[5]: our route server instance runs on a physical server with `Intel(R) Xeon(R) CPU E3-1220L V2 @ 2.30GHz` CPUs.
- RS clients - Bird 1.6.3[5]: running on a different server, the deployment in containers[25] of Bird RS clients allow to have multiple copies running in parallel and perform scalability tests - each Bird client consumes less than 5 MB of memory during execution.

A dedicated network link of 1000 Mbps between the servers provides the connectivity and ensures no external disturbances.

3.1.2 Experiment 1: RS reconfiguration with different file sizes

Our exploratory experiments show that the route server CPU utilization goes to 100% during reconfiguration, and the time spent in this state varies with the size of the configuration file in use. We use the standard GNU time (version 1.7), top (version 3.3.12) and tcpdump (version) Linux tools to measure the time, CPU, memory and capture the network traffic under three different scenarios:

- Scenario 1: Bird configuration file of 2.5 MB, which we generate with the Configuration Parser[6] and one RS client as input.
- Scenario 2: Bird configuration file of 61 MB, which we obtain with the Configuration Parser and the list of the 714 RS clients at AMS-IX[13].
- Scenario 3: Bird configuration file of 116 MB, obtained with the Configuration Parser and the 2472 route server clients as input, chosen from the top five IXPs (IX.br[14], DE-CX[15], AMS-IX; LINX[16] and MSK-IX[15]).

The number of RS clients and their policy specifications are responsible for the difference in the sizes of the configuration files. The contents of the configuration file are described in Section 2.2.1.
Unfortunately bigger files could not be tested because of an error in arouteserver, which prevented us from configuring more than 2,500 ASNs[16].

3.1.3 Experiment 2: Reconfiguration with different peers sending BGP updates

Reconfigurations of the route server after a policy change trigger a `BGP UPDATE` procedure, which increases the load on the network and the time spent in the reconfiguration process. Such an policy change happens, for example, when the administrator of an RS client adds a new prefix to its policy specification.
To test this behavior, we configure the route server clients with stub networks. We then reload the route server multiple times using two different configuration files, each containing different prefix sets (differing in one prefix).
We measure the time taken to reconfigure Bird while varying the number of route server clients. We chose the work with the arbitrary values 2, 4, 8 (testing with more than 700 RS clients would provide a more realistic view of AMS-IX's current load, and is recommended for future work).
To ensure the effect of different file sizes are accounted for, we perform experiments without `BGP UPDATEs`, with only one node, and with multiple nodes sending `BGP UPDATEs`.

3.1.4 Experiment 3: Route server peering with large number of clients (>1400 route server clients)

To test the limits of our setup, we designed an experiment where 1400 RS clients (approximately twice the number of clients on AMS-IX's RS) are peering with the route server. For this experiment, we reuse the list of clients obtained in Experiment 1 - Scenario 3. Details are available in the project repository.[17]

[13] https://www.ams-ix.net/ams/connected-networks
[14] https://ix.br/
[15] https://www.msk-ix.ru/en/
[16] https://github.com/pierky/arouteserver/issues/48
[17] https://bitbucket.org/david-garay/rp2

3.1.5 Experiment 4: ROA and Bogon list updates

The objective of this test is to identify how other BGP policy changes (in particular, changes to the ROA and Bogon list) might trigger a `BGP UPDATE` procedure. To accomplish this, we evaluate the traffic between the route server and its clients while adding prefixes to the ROA and Bogon lists. For these experiments, we reuse the configuration file in Experiment 1 - Scenario 1.

3.2 Evaluating the frequency of BGP policy changes

Configuration changes might result from different processes. For instance, a new ROA record, a bogon prefix introduced or due to a peering with a new route server client. In our evaluation, we focus on BGP policy updates done at the IRRDB.

In Listing 1 we show that policies can be specified on the `import` and `export` attributes. These attributes contain in turn policies which refer to other RPSL objects (`aut-num`, `route/route6` and `as-set/route-set`). Changes on any of these attributes or objects result in a BGP policy change. Still, not every change is relevant to a given IXP, so we need to exclude changes affecting other IXPs. With these considerations, we collect historical information (timestamp of object changes) using the Historical Whois API from RIPE STAT[18]. The list of `aut-num` objects to be queried comes from the list of connected route-server clients at AMS-IX. The list of `route` objects come from the IPv4 and IPv6 prefixes present at the route-server routing table and kindly provided by AMS-IX for the purposes of this research.

A limitation with our method is that RIPE STAT APIs does not provide information about objects from another region.

3.3 Profiling of the Bird routing daemon

To understand which functions are responsible for the CPU blocking observed in our experiments, we run Bird using a profiler. Valgrind[19] offers C-program analysis capabilities by providing a just-in-time recompilation of the code to an intermediate *Ucode*, and in doing so it enables different tools to debug and profile. Valgrind requires the program under analysis to be compiled with debugging symbols on. We use Kcachegrind[20] to interpret the results. The source code of Bird is publicly available[21] and can fulfill this requirement. The experiments to identify the top functions consuming CPU cycles during reconfiguration are:

- Experiment 1: 2.5 MB file reconfiguration, no BGP Updates
- Experiment 2: 116 MB file reconfiguration, no BGP Updates
- Experiment 3: 2.6 MB file reconfiguration, 4 clients triggering BGP Updates

The content and specification of the test configuration files are described in Section 3.1.2.

[18] https://stat.ripe.net/docs/data_api
[19] http://www.valgrind.org/
[20] https://github.com/KDE/kcachegrind
[21] https://github.com/BIRD/bird

4 Results

The results of the experiments and measurements in Section 3 are shown below. Despite having monitored CPU, memory and traffic, we present our results in terms of blocking time because no significant impact on the route server or its clients was observed in terms of the other parameters. All results are available in the project repository[22].

4.1 Evaluating the impact of updates

This section contains the results of experiments to measure the impact on the network infrastructure.

4.1.1 Experiment 1: Reconfiguration with different file sizes

Following the definitions for Experiment 1 in Section 3.1, we obtain:

File size [MB]	Blocking time [s]
2.5	0.09 ± 0.003
61.0	2.01 ± 0.05
119.0	4.29 ± 0.10

Table 2: Reconfiguration time [s] as function of configuration file size

Table 2 shows the average blocking time for different file sizes, measured over 20 reconfigurations[26] with a confidence level of 95%. The main effect of the number of reconfigurations is that when we calculate the confidence interval, we use the t-distribution instead of the Normal distribution. We believe that in our case the difference between the two alternatives is negligible.
In Table 2 we see a nearly linear relationship between blocking time and file size.

4.1.2 Experiment 2: Reconfiguration with different peers sending BGP updates

Following the definitions for Experiment 2 in Section 3.1, we obtain the results:

#peers	No active BGP sessions	One node	All nodes
2	0.09	4.05 ± 0.02	4.51 ± 0.02
4	0.09	3.95 ± 0.03	5.41 ± 0.04
8	0.09	3.95 ± 0.02	7.48 ± 0.03

Table 3: Reconfiguration time [s] as function of the number of peers undergoing protocol updates

Table 3 shows the average blocking time for different number of peers in three scenarios: "No active BGP sessions", "One node" and "All nodes". The experiments are measured over 20 reconfigurations with a confidence level of 95%. Three factors contribute to the increase in time: firstly, the effect of the file size can be seen in Column "No active BGP sessions". Secondly, there is a process (which is seen when a node undergoes `BGP UPDATE`) causing approximately

[22] https://bitbucket.org/david-garay/rp2

3.95s, which can be seen in Column "One node". Finally, we observe an additional increase of approximately 0.5s per additional RS client in the topology undergoing `BGP UPDATE`.
When interpreting these results, we need to consider that we use test prefixes while testing the `BGP UPDATES`. This means that the route server filters them out, instead of aggregating and exporting them further to other route server clients. We believe that the actual time contributions due to `BGP UPDATES`, without the filtering, would be slightly higher because the prefixes need to be propagated by the RS and processed by its clients. We opt to keep the filters enabled to keep the experiments simple.

4.1.3 Experiment 3: Route server peering with large number of clients (>1400 peers)

Following the definitions for Experiment 3 in Section 3.1, we get a continuous restart of the route server instance when reaching 1013 sessions open. The error message is:

```
2019-06-27 12:37:49 <ERR> BGP: Error on listening socket:
Too many open files
```

The error is related to the Linux security configurations, which determine how many file handlers a process spawned from shell might open. In this case, the limit is set to 1024, causing the resource exhaustion. These settings are changed with the `ulimit` command.[23]

4.1.4 Experiment 4: ROA and Bogon list updates

The results of Experiment 4 defined in Section 3.1 are:

- Updating the Bogon list results in `BGP UPDATES`.
- Notably, we do not observe `BGP UPDATES` when we modify the ROA list, despite our tests via the command line interface and modifications on the static configuration file.

Apart from IRRDB policy changes, we note that modifications to the Bogon list also trigger `BGP UPDATE` procedures.

[23] https://www.manpagez.com/man/1/ulimit/

4.2 Frequency of BGP policy changes

Following the definitions Section 3.2 for the evaluation of the frequency of policy changes, we obtain:

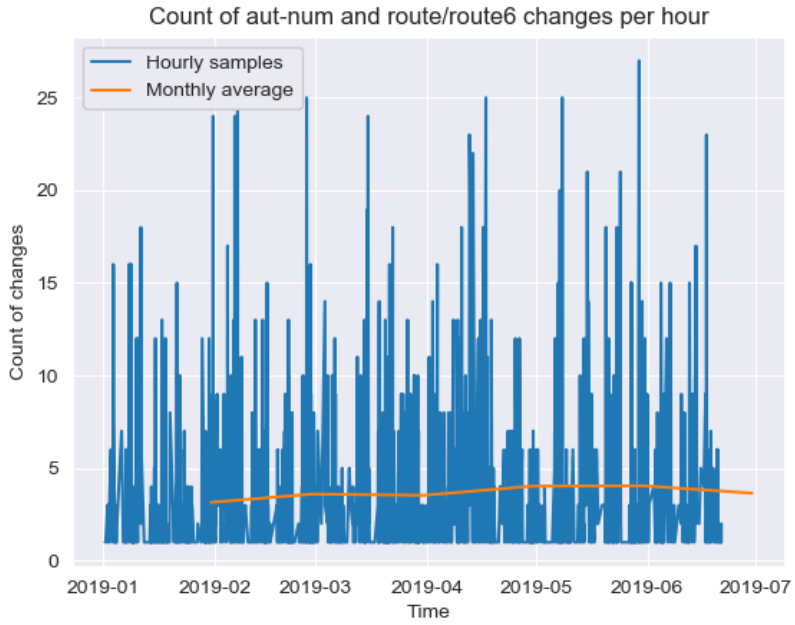

Figure 3: Aut-num and route object changes, aggregated per hour, 2019-01 till 2019-06

In Figure 3 each point represent the number of changes in an hour (we aggregate the timestamps on an hourly basis). The orange line represents the monthly averages calculated at the end of the month. When multiple RPSL objects related to the same ASN have changes, only the object with most changes is considered. Moreover, outliers (points with Z-score higher than 3) are not included. Based on the values show in Figure 3, we define the "normal scenario" to be 4 (the monthly average) and the "busy scenario" (the frequent spikes in policy updates) to be 28.
To show the trend in policy changes, it is useful to inspect the data in a longer time frame. Figure 4 below shows the rate of policy changes since 2004. Compared to Figure 3, we can see higher values, which is because in this time frame there are outliers with higher values, and the criteria to filter out outliers is higher.

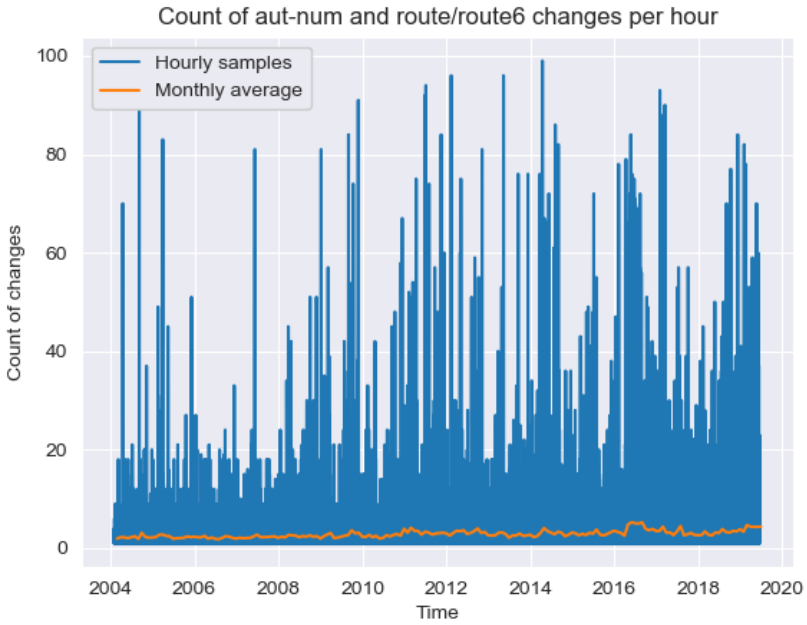

Figure 4: Aut-num and route object changes, aggregated per hour, 2004-01 till 2019-06

Examining the monthly average, we can see a slight increase along the years. If the number of ASNs and the usage of the IRRDB increase, we expect the monthly average of policy changes to increase.
In Sections 4.1 we show that there is a nearly linear relationship between the RSs configuration file size and the time taken to process it. Also, there is a near 4s increase in time when `BGP UPDATEs` are performed by at least one RS client. Finally, we observe an 0.5s increase per node undergoing `BGP UPDATEs` added to the topology. Our analysis of the amount of hourly changes of RPSL objects in Section 4.2 improves our comprehension of the system. In Figure 3, the monthly average of changes per hour is between 3 and 4, and there are frequent spikes above 20. We believe that in a Push model, if we restart the route server only when a BGP policy change arrives instead of on an hourly basis, the overall utilization of the system will go down. We also highlight the importance of measuring accurately the frequency of policy changes: for example, if we consider the "normal scenario" of 4 arrivals per hour, using the Erlang B formula (see Section 2.4) we obtain a blocking probability of 0.88% (with a service time of 8s, following the results discussed previously). On the other hand, in a "busy scenario" (28 arrivals per hour and a service time of 20s), we obtain 13.46%. Measuring the rate of changes in a production environment is therefore key for IXPs to dimension the system accurately.

4.3 Profiling of the Bird routing daemons

Following the definitions for the profiling of Bird given in Section 3.3, we obtain:

#	**Function**	**Module**	**% CPU cycles (including)**	**% CPU cycles (excluding)**
1	cf_parse'2	cf-parse.tab.c	97,13%	16,09%
1	cf_lex	cf-lex.c	69.83%	43.88%
2	cf_lex	cf-lex.c	36.76%	23.02%
2	trie_node_same	trie.c	26.57%	26.57%
2	i_same	filter.c	11.54%	11.52%
3	cf_lex	cf-lex.c	69.63%	47.30%
3	bgp_rx	packets.c	11.93%	0.02%
3	rte_update2	rt-table.c	11.84%	0.03%

Table 4: Top functions consuming CPU cycles during reconfiguration

Table 4 list the top functions of Bird that consume CPU cycles. The column `% CPU cycles (including)` counts CPU cycles made by the function and subsequent calls to other functions - so in Experiment 1, cf_parse might wrap a call to cf_lex. To have a better impression of the work done exclusively by a function, refer to the column `% CPU cycles (excluding)`.
At smaller file sizes, the majority of the processing effort goes into the parsing of the configuration file (Experiment 1). As the file size increases, the more CPU cycles are spent on the data structures containing IP prefixes (Experiment 2). When `BGP UPDATEs` are present, processing of the protocol procedures increasingly claims more resources (Experiment 3). There are no functions with a '2 suffix in the source code[24]. Instead, according to the output of Valgrind the suffix indicates a recursive call to the function output [25]. The rest of the CPU cycles is consumed on external libraries (for instance, libc string utilities such as strtol or strcmp).

[24]https://github.com/BIRD/bird
[25]https://bitbucket.org/david-garay/rp2

5 Solution Design

We present our design by first listing our main design goals. We discuss then the changes introduced by our new design, and their effect on the existing system. Finally, we evaluate the feasibility of our design with a proof of concept.

5.1 Design goals

The main goals for our design are:

- Manage system blocking: if we consider for example "busy scenario" and an arbitrary arrival rate of 28 changes per hour (Figure 3), we get a processing time of 20s (following the contributions discussed in Section 4). The Erlang B formula gives then a blocking probability of 13.46%[27]. Almost 1 in 8 notifications would be lost, and leave the RS with stale configuration, unless queuing and retry mechanisms are in place. Furthermore, we hypothesize that operational tasks and protocol keep alives[9] might also suffer from lack of resources under these situations.
- Decrease network load: more reconfigurations result in more `BGP UPDATES`. We require mechanisms to protect the network infrastructure from spikes of policy change events, and keep the network load low.
- Real time processing of notification: apart from decreasing the risk of working with stale data, reconfiguring the RS only when required and not at fixed intervals is likely to bring the overall system utilization down (specially when we configure the frequency of policy changes in Section 4.2).

5.2 Publish-Subscribe messaging systems

We apply the Publish-Subscribe messaging pattern[10] to enable asynchronous communications and loose-coupling between the IRRDB and the operators' network infrastructure. An ISP might have multiple interested parties to its policy changes (e.g. service providers and IXPs), and this make scalability an important factor for our design. The Publish-Subscriber offers better scalability than Point-to-point channels because it decouples producers from consumer(s), and new clients just need to subscribe to a topic to receive notifications.
In the current operation, IT systems at each IXPs query the data sources at fixed intervals of time, potentially leaving the systems with stale data and unnecessary restarts. In our design we introduce a push model: data sources publish BGP updates to a Publish-Subscribe topic in a Messaging System, which then relays these messages to one or more consumers at IXPs. To enable these messaging capabilities, existing applications require new functions:

- Messaging Gateway: Hohpe et al.[10] describe that the main function of a Messaging Gateways is to encapsulate access to the messaging system from the rest of the application. In our case, the Messaging Gateway is simply a thin layer around the Configuration Parser which enables it to receive messages. This component is typically implemented using client libraries provided by the Messaging System. A Message Gateway is also required at the IRRDB.
- Messaging System: this component provides store-and-forward delivery of messages, and implements higher level services such as message queues and channels, so that e.g. addressing of endpoints and message retries are hidden from endpoints.

5.3 Subscription and Notification flow

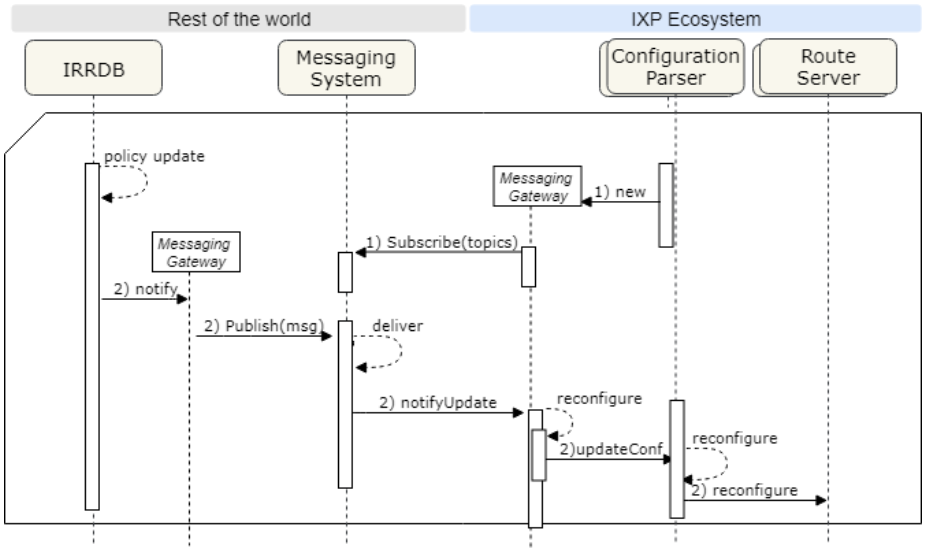

Figure 5: Publish/Subscribe sequence for the BGP policy update process

Figure 5 shows the procedures and interaction between components of the architecture, during a BGP policy update procedure. The IRRDB (and its Messaging Gateway) have the publisher role, and the Configuration Parser (and its Messaging Gateway) the consumer role. The Subscribe and Publish procedures are numbered 1 and 2, and the detailed description is provided below:

1. During the Subscribe procedure, the Configuration Parser initializes its Messaging Gateway. The Messaging Gateway sends a message to the Messaging System to subscribe to relevant topics (the channel where policy updates of a given ASN is published) and starts listening for notifications.
2. During the Publish procedure, the IRRDB uses its Messaging Gateway to publish messages to a topic. The Messaging System ensures that copies of the message are pushed to the interested endpoints, and remove the messages when they have been consumed (acknowledged). When a BGP update notification arrives to its Messaging Gateway, the Configuration Parser generates a new configuration file and then reconfigures the route server with it.

We describe further the following aspects:

- Flow Control/Queuing: in a Push model a policy update might arrive while the RS is still busy handling another request, prompting the need for a queue or retry mechanisms. Furthermore, a sustained high rate of BGP updates might bring the system close to a grinding halt[23], this, in turn, asks for Flow Control mechanisms. Message systems typically provide a message queue, and can adapt the frequency of messages delivered dynamically according to the rate of consumption. On the other hand, having a local queue at the Messaging Gateway offers implementation independence and provides more granular control (as messages are locally

available) to enable parallelism, for example. We opt for the implementation of Flow Control and Queuing both in the Messaging System and in the Messaging Gateway.

- Support for multiple route servers: if more route servers are introduced, our Message Gateway will benefit from the *dispatcher pattern*: an architecture for the Message Gateway where one master thread listens for incoming messages and one or more workers consume them from a local queue. In this way, the message gateway is able to receive notifications while still processing events.
- Message format and distribution: one option is to send the entire RPSL object in a *document message*[10]. However, in the scenario where the message remains in the queue long enough, this might present the consumer with stale information. A *command message* expects a reply from the consumer, increasing the complexity also at the producer side. For the purposes of updating policies, an *event message* that triggers synchronization at the consumer is more suitable. In this way, the consumer remains in control and decide, for instance, if it processes the update immediately on an individual basis or waits and performs batch processing.
- Idempotency: in messaging, a consumer is said to be idempotent when a message sent multiple times have the same result. Since an update notification invariably result in a synchronization with the IRRDB, repeated messages have no consequence other than the cost of executing again the procedure.

Perhaps the best alternative to our messaging based approach is an RPC system where BGP update notifications are sent in real time to the IXP. The advantage of an RPC system is that the implementation complexity is likely to be lower compared to a messaging system, because new systems are not required (although modification to existing ones are). The disadvantage of RPC is, however, that every endpoint in the architecture has to deal with details such as addressing, protocol parameters and queue/retransmission details, to name a few. These disadvantages make RPC not scalable. With loosely-coupled systems enabled by messaging, new applications and different protocols can be mediated by the messaging system.
We do not mention details about transport and addressing schemes, for instance, because these depend on the choice of Messaging System implementation.

5.4 Proof of Concept

To validate the feasibility of our design, we built a proof of concept that implements the Configuration Parser's Messaging Gateway, and rely on a commercial Publish-Subscribe[28] service for the delivery of messages.[26]

[26] Our criteria for selecting this solution was its simplicity and the availability of libraries in Python.

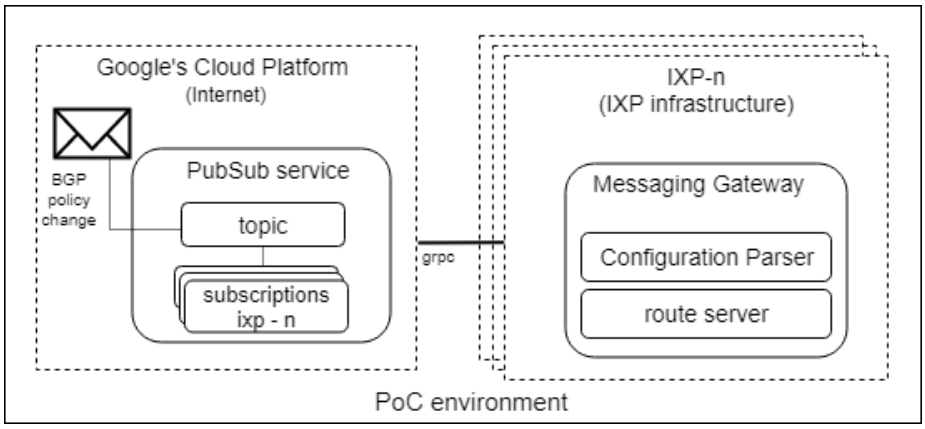

Figure 6: PoC environment, Messaging System (Google PubSub) and Message Gateway wrapping the Configuration Parser and route server to enable messaging

Figure 6 shows the components in our proof of concept. For the Messaging System we rely on Google's PubSub cloud service. This service also provides the libraries required to implement the Messaging Gateway.
In our proof of concept[27], we create a topic on our Messaging System and attach subscriptions to it: contrary to the classical description of Publish-Susbscriber systems in the literature[10], our Messaging System service adds a second layer of abstraction (the "subscription") to control the behaviour of the message queue. Without it, the consumers behave as Competing consumers[10]: only one endpoint consumes the message, which is then unavailable to other consumers. We use one subscription per consumer to achieve the desired broadcast behavior.
During the Subscribe procedure, the Messaging Gateway subscribes to a topic and spawns a listener thread that awaits for new messages. During the Publish procedure, when a new message is published to a topic the message is made available to the different subscriptions attached to the topic. In this way, multiple IXPs are able to receive the message.
The libraries from the Messaging System provide high level primitives such as `subscribe`, `create_topic` and `publish`[29]. These functions simplify the development of the Messaging Gateway. Except for an initial setup required for authentication, the client library hides the implementation details such as endpoint addresses, protocols and timers. During execution, we observe that the Messaging Gateway maintains a SSL/TCP session with the cloud server. When the application restarts and a new session is made, messages are still sent to the client. To recreate the complete behavior at the network infrastructure side, we have the Configuration Parser's Messaging Gateway parse a new configuration file and restart Bird with the new configuration.
Our main observations resulting from the proof of concept are:

- Asynchronous pull: our Messaging System offers an alternative to the Push model. Asynchronous pull relies on a permanent connection initiated by the Messaging Gateway. The advantage of this model is that messages are still pushed asynchronously and without noticeable delays

[27] https://bitbucket.org/david-garay/rp2

over the permanent connection, while decreasing the requirements on the Messaging Gateway (the "proper" Push mode implementation requires an HTTP server, DNS-resolvable URLs and not-self-signed certificates[30]). The permanent connection uses an HTTP2 / GRPC stack[28].

- Lack of ordering of messages:[29] the Messaging System does not guarantee the order of messages. This fact, however, does not impact our use case.
- Message queue size: for completeness, we note that the buffer size in our chosen implementation[30] of the Publish-Subscribe messaging system is 10 MB, which can be allocated to one or multiple messages.
- Partitioning of file and processing time: remarkably, adding the configuration parser to the chain highlights the inefficiencies in the current process. Currently, the Configuration Parser retrieves and parses the information for every single ASNs in the IXP's route server. Then, the reconfiguration is read by Bird, which also reconfigures all ASNs and process the `BGP UPDATES` of the affected ASNs. This inefficiency prompts the need for Data Partitioning[23] (which we cover in more detail in Section 6).

The proof of concept shows the feasibility of a Publish-Subscribe Messaging system to process notifications in real time. It highlights parts of the system that can be further improved (the Configuration Parser and the processing of BGP policy updates by Bird). Implementing the solution on different IXPs would require a small adaptation on the Messaging Gateway (in particular, in how the particular implementation of the Configuration Parser is done), and apart from user accounts, authentication and subscriptions, the Messaging System itself require no further modifications.

Although not tested in our proof of concept, flow control mechanisms and capabilities to support concurrent handling of events[30] are available in the Messaging System and its client libraries.

[28] https://cloud.google.com/pubsub/docs/pullstreamingpull

[29] https://cloud.google.com/pubsub/docs/ordering

[30] https://googleapis.github.io/google-cloud-python/latest/pubsub/subscriber/api/scheduler.html

6 Discussion

In the previous sections we investigated the current route server BGP policy update process in an effort to answer the question: what are the main performance and scalability indicators? What are the bottlenecks of the process? What is the impact of these bottlenecks?

We investigated the CPU and memory utilization, the time taken to process reconfigurations and the network traffic generated. In our results we show that the main indicator affecting performance and scalability is the CPU blocking time. During this time, the system is unable to process new updates —which might lead to loss of information in a push model, if countermeasures such as queuing are not taken. We also believe that operational activities and protocol keep alives might be affected.

Our research complements the previous work of Jenda Brands and Patrick de Niet[9], who studied the effects of high CPU usage on the BGP convergence time in the network infrastructure of the IXP. Our results show how the blocking time depends on the file size of the configuration file, and the number of RS clients undergoing `BGP UPDATEs` procedures. Because the configuration file size depends on the number of RS clients configured and their respective policies, we can predict the blocking time of the system based on the number of RS clients and the amount of policy changes arriving to the system. By profiling Bird we identified the functions where the route server is spending most of the CPU time during reconfiguration. In line with our experiments, we see that the main contributions are related to parsing of the configuration file and protocol processing. These observations allow us to hypothesize that the blocking time can be improved with more efficient parsing in Bird (for instance, using binary configuration files) and through better processing of RS client reconfiguration, both in the Configuration Parser and Bird.

Although Bird claims it has no hard limits in software that can limit its scalability, we observed a curious behaviour of Bird in combination with the default settings of Ubuntu: when enough BGP sessions are open, Ubuntu's default limit of 1024 file handlers per process is reached and Bird restarts after raising a segmentation fault. This is easily resolved by changing the `ulimit` settings for the Bird process. Still, we recommended Bird support to update the documentation.

When considering the scalability of the system, another important aspect is the frequency of BGP policy changes. We show in Figure 3 that during normal hours the rate of policy changes is on average 4, while frequently it goes above 20 changes per hour.

Considering our limitations in methodology during measuring the frequency of changes (and the fact that only IRRDB policy changes were considered), we believe that it is likely that the number of policy changes per hour is higher than what we show in Figure 3. Furthermore, as network operators increase the rate of policy changes and new clients join, we believe that the trend of policy change rate is to keep increasing.

We recommend IXPs to monitor the amount of changes arriving at their production Configuration Parser and evaluate their impact on the blocking time of the route server, to provide a more accurate view of the impact of policy changes, adjusted to their RS client profiles and network topology.

An important consequence of our evaluation of the impact and frequency of policy changes is that it allows us to predict the blocking probability of the system

using the Erlang B formula. It can be useful, for example, to express the scalability limit of the system in terms of its blocking probability: we hypothesize that at high values (e.g. above 80%) operational tasks, protocol keep alives and processing of new reconfigurations will suffer from lack of resources, and going to even higher values might bring the system to a grinding halt.
Our second research question deals with the improvement opportunities in a new design. We propose a design that enables the system to react in real time to policy changes by adopting a Publish-Subscribe messaging system that largely relies on the design patterns described by Hohpe et al.[10] We propose the introduction of a Messaging System, and Messaging Gateways at the IXP and data source sides, in order to enable messaging capabilities in these systems.
The first element required to support messaging is the Messaging System. In general, Message Systems are a mature technology and there are options to acquire commercial and open source solutions, both in the cloud and on premise. Perhaps the biggest investment required are related to operational costs —i.e. configuration, capacity and security management. We believe that the more frequently the messaging system is adopted for other use cases (e.g. Bogon prefix list update), the more economically justifiable it becomes. Another incentive to adopt a messaging solution is the possibility to unify existing processes into a unified process, while providing faster time to market to new applications (e.g. the introduction of a feedback notification, so that network operators can be aware of which IXPs implemented their new policies in real time). Regarding the position and operational responsibilities of the Messaging System, we consider two alternatives: at one hand, we can expand the scope of services provided by any of the existing data sources, or, rely on new organization that can provide this capability to the community. The evaluation of these alternatives fall outside the scope of our research.
The second functionality required to support messaging in our design is the Messaging Gateway. It introduces the functionality required on the Configuration Parser (at the IXP) and on the IRRDB to enable the Publish-Subscribe messaging capabilities. The responsibilities for their implementation fall at the organizations operating the endpoints. By using a dispatcher architecture with a local queue, we allow the Messaging Gateway at the IXP side to implement flow control and parallelism.
In our proof of concept, we implement a limited Messaging Gateway using Google's PubSub service and libraries, and describe how an event is processed in real time. Thanks to the proof of concept, the inefficiencies in the handling of BGP policy updates become evident: at every new update the Configuration Parser (arouteserver in our environment) refreshes the information about all clients and generates a new configuration file. Bird, the route server, reads and processes the complete configuration file, even if the change affect only one RS client. Data partitioning (processing subsets of the configuration file, in this case) offer an opportunity to improve the reconfiguration time. Another observation derived from our proof of concept has to do with the selection of the Messaging System implementation, and how it affects the capabilities available at implementation time (for instance, the asynchronous pull discussed in Section 5.4 sped up our implementation by lowering infrastructure requirements. Still, this feature might not be available in other Messaging Systems).
We believe our proposal can satisfy the design goals set (namely, manage system blocking, decrease network load and process notifications in real time) and is

technically feasible. Still, in our design we considered only the IRRDB BGP policy update process and relied on a Publish-Subscribe channel. New use cases (for instance, a feedback flow where the IXP informs which policies were refreshed) might require point-to-point channels, for instance. These additional capabilities need to be taken into consideration when selecting the Messaging System implementation. We did also not explore in detail other well-known scaling techniques: partitioning, replication and caching, and should be revisited as the design is implemented and improved.
We believe the design is future proof because it focuses on introducing the messaging capability to the architecture, which enables loosely-coupled applications to integrate with ease. Additionally, it offers potential to improving scalability by adding more route servers, more efficient processing of policy updates through data partitioning and more efficient parsing of Bird configuration files. These topics are recommended for future work.

7 Conclusion

This paper has investigated the current process handling BGP policy updates at an IXP's route server.
The results in this study indicate that the route server's blocking time (CPU utilization 100%) is the main bottleneck in the system. The blocking time during reconfiguration depends on the file size and the amount of peers that undergo protocol reconfiguration procedures. By studying the frequency of policy updates, we are able to describe the blocking probability in different situations using the Erlang B formula. We discuss the main considerations IXP need to make to dimensioning the route servers. Taking these results in consideration, we propose and evaluate a Publish-Subscribe messaging system that enables processing in real time of BGP policy updates, and demonstrate its feasibility with a proof-of-concept. Additionally, we highlight improvements to the Configuration Parser and Bird protocol processing that would further decrease the processing time. By profiling popular route server implementation Bird, we identify and discuss the modules which affects processing time the most, and suggest new research directions on these topics. As a short term countermeasure, we recommend IXPs to monitor usage statistics of this BGP process and get an accurate view of the update process impact using production data.

8 Future work

Future research should consider the potential effects of CPU blocking on other tasks more carefully, for example operational queries and protocol keep alive processing. In this way, IXPs can better determine the levels where blocking time become service affecting. Moreover, future research should be devoted to the development of data partitioning on
`arouteserver` and Bird, and optimizations in the functions behind the high CPU utilization times. Another topic that deserves further research pertains the introduction of new route servers, in order to partition the processing of policy updates, and the consequences on the network topology and BGP.
Additionally, the evaluation of the arrival of updates can be improved to include other RPSL objects.

Acknowledgments

We hereby would like to thank Stavros Konstantaras from AMS-IX for his support and guidance throughout our research. Stavros knowledge and motivation were key to the quality and progress of the research. We also would like to thank Paola Grosso from the UvA for the time she generously provided to discuss our results and their implications.

References

[1] Andrei Robachevsky, "Routing Security – Getting Better, But No Reason to Rest!" 2019, last accessed 7 july 2019. [Online]. Available: https://www.internetsociety.org/blog/2019/02/routing-security-getting-better-but-no-reason-to-rest/

[2] BGP Stream, "BGP Stream," 2019, last accessed 7 july 2019. [Online]. Available: https://bgpstream.com/

[3] MANRS, "Mutually Agreed Norms for Routing Security," 2019, last accessed 7 july 2019. [Online]. Available: https://www.manrs.org/

[4] RIPE NCC, "RIPE Whois Database," 2019, last accessed 7 july 2019, commit 1727a675dd136c67a0bdd7b351f01c0024a1280a. [Online]. Available: https://github.com/RIPE-NCC/whois

[5] CZ.NIC Labs, "BIRD Internet routing daemon," 2019, last accessed 7 july 2019. [Online]. Available: https://bird.network.cz/

[6] Pier Carlo Chiodi, "ARouteServer," 2019, last accessed 7 july 2019. [Online]. Available: https://github.com/pierky/arouteserver

[7] Frits C. Schoute, "The Power of the Erlang formula," 2000, last accessed 7 july 2019. [Online]. Available: http://www.ecoboot.nl/tudelft/Erlang.htm

[8] Stella Vouteva and Tarcan Turgut, "Automated configuration of bgp on edge routers," 2015.

[9] Jenda Brands and Patrick de Niet, "Parallelization of bgp for route server functionality," 2017.

[10] G. Hohpe and B. Woolf", *Enterprise Integration Patterns.* Addison-Wesley, 2003.

[11] Matthew Caesar and Jennifer Rexford, "Bgp routing policies in isp networks," 2005.

[12] IETF, "RFC7454: BGP Operations and Security," 2015, last accessed 7 july 2019. [Online]. Available: https://tools.ietf.org/html/rfc7454

[13] IETF, "RFC7947: Internet Exchange BGP Route Server," 2016, last accessed 7 july 2019. [Online]. Available: https://tools.ietf.org/html/rfc7947

[14] AMS-IX, "AMS-IX Route Servers," 2019, last accessed 7 july 2019. [Online]. Available: https://www.ams-ix.net/ams/documentation/ams-ix-route-servers

[15] DECIX, "DE-CIX Frankfurt Route Server Guide," 2019, last accessed 7 july 2019. [Online]. Available: https://www.de-cix.net/en/locations/germany/frankfurt/routeserver-guide

[16] LINX, "LINX - Members list," 2019, last accessed 7 july 2019. [Online]. Available: https://portal.linx.net/members/list

[17] LINX, “LINX Route Server Looking Glass,” 2019, last accessed 7 july 2019. [Online]. Available: https://portal.linx.net/cgi-bin/rslg

[18] IETF, “RFC7948: Internet Exchange BGP Route Server Operations,” 2016, last accessed 7 july 2019. [Online]. Available: https://tools.ietf.org/html/rfc7948

[19] CZ.NIC Labs, “BIRD User’s Guide (version 1.6.6),” 2019, last accessed 7 july 2019. [Online]. Available: https://bird.network.cz/?get_doc&f=bird.html&v=16

[20] IETF, “RFC6480: An Infrastructure to Support Secure Internet Routing,” 2012, last accessed 7 july 2019. [Online]. Available: https://tools.ietf.org/html/rfc6480

[21] RIPE NCC, “NRTM Mirroring,” 2019, last accessed 7 july 2019. [Online]. Available: https://www.ripe.net/manage-ips-and-asns/db/nrtm-mirroring

[22] IETF, “RFC2622: Routing Policy Specification Language (RPSL),” 1999, last accessed 7 july 2019. [Online]. Available: https://tools.ietf.org/html/rfc2622

[23] M. van Steen and A. S. Tanenbaum", *Distributed Systems*. Maarten van Steen, 2017.

[24] IETF, “RFC4271: A Border Gateway Protocol 4 (BGP-4),” 2006, last accessed 7 july 2019. [Online]. Available: https://tools.ietf.org/html/rfc4271

[25] Globo.com, “Bird Routing,” 2019, last accessed 7 july 2019. [Online]. Available: https://github.com/globocom/bird-routing/tree/0.2.0

[26] Russell A. Poldrack, “Statistical Thinking for the 21st Century,” 2019, last accessed 7 july 2019. [Online]. Available: http://statsthinking21.org/doing-reproducible-research.html#reproducible-practices

[27] Wyean Chan, “Erlang B Calculator,” 2019, last accessed 7 july 2019. [Online]. Available: https://www-ens.iro.umontreal.ca/~chanwyea/erlang/erlangB.html

[28] Google, “Google PubSub Documentation,” 2019, last accessed 7 july 2019. [Online]. Available: https://cloud.google.com/pubsub/docs

[29] Google, “Cloud Pub/Sub Client Libraries,” 2019, last accessed 7 july 2019. [Online]. Available: https://cloud.google.com/pubsub/docs/reference/libraries#client-libraries-install-python

[30] Google, “Google PubSub - Push documentation,” 2019, last accessed 7 july 2019. [Online]. Available: https://cloud.google.com/pubsub/docs/push