LEDE Firmware optimization for wired deployments using BGP and BMX6 for routing by enhancing and extending Bird Daemon's configuration and UI integration

Theale, June 2017

Major: Computer Science
Minor: Open Source Software

Author:
Eloi Carbó Solé

External consultant:
Víctor Oncins Biosca
Routek SL

Director:
Joan Manuel Marquès Puig
Computer Science, Multimedia and Telecomunications Department (DPCS-ICSO)

Universitat Oberta de Catalunya
2017

Project Key Data

Títol del treball:	Optimització del Firmware LEDE per desplegaments basats en xarxes cablejades utilitzant Bird Daemon i BMX6; Extensió de la integració gràfica de la configuració i anàlisis respecte desplegaments en Quagga
Project's Title:	LEDE Firmware optimization for wired deployments using BGP and BMX6 routing protocols by enhancing and extending Bird Daemon's configuration and UI integration
Nom de l'autor:	Eloi Carbó Solé
Nom del consultor:	Víctor Oncins Biosca
Data de lliurament:	06/2017
Àrea del Treball Final:	Sistemes Distribuïts
Titulació:	Màster Universitari en Programari Lliure
Paraules clau:	OpenWrt, Bird Daemon, Guifi.net, Xarxes comunitàries

Resum del Projecte

En aquest treball s'ha millorat la integració de l'eina Bird Daemon en sistemes LEDE/OpenWrt. Aquesta eina permet a administradors de xarxes comunitàries controlar seccions on diferents topologies i protocols d'enrutament convergeixen i requereixen de mètodes, generalment manuals, per intercanviar prefixes i per discriminar el trànsit de la xarxa per al seu bon funcionament.

Les millores aplicades al paquet complementari bird-openwrt, es basen en continuar l'automatització de la configuració d'aquest programari, que consta de la seva pròpia sintaxis, i en augmentar el nombre de funcionalitats que es poden configurar des de l'entorn gràfic (web) del sistema. Amb això reduïm tant la necessitat d'aprendre la sintaxis específica del programari com d'haver d'accedir a línia de comandes per configurar manualment el sistema, reduint la possibilitat d'introducció errors.

Per altra banda, s'han analitzat possibles línies d'implementació per tal d'obtenir informació de les sessions establertes per Bird Daemon. Aquesta informació en viu permetria monitoritzar tant els protocols i els prefixos compartits, com possibles problemes al sistema i a la informació històrica.

Finalment, els resultats obtinguts per aquest projecte són l'actualització del paquet bird-openwrt amb la inclusió de millores mencionades, així com un nombre de noves funcionalitats que s'implementaran en el futur i una proposta per a un projecte per tal d'integrar informació dels protocols en viu mitjançant interfície web, seguint les conclusions de l'anàlisi fet.

Abstract

This project has enhanced current Bird Daemon's integration with the OpenWrt/LEDE firmware system. This utility allows network administrators to manage network sections where different routing protocols and technologies may coexist, thus requiring mechanisms to exchange their network prefixes and to allow routing through configured filters to ensure network´s correct behaviour.

This integration is managed through the bird-openwrt package, which automates Bird's configuration syntax and presents it through a web-based environment. This package's improvements have been focused on adding graphical integration to Bird's features and to reduce the number of instances where an administrator is required to go to command line in order to tweak system's settings to configure a specific behaviour.

Moreover, there is a theoretical analysis of the future implementation paths to include monitoring capabilities and to integrate them in the graphical environment.

Finally, this project's results have been an update to bird-openwrt package including the mentioned enhancements and a project's proposal and a series of recommendations to implement monitoring capabilities to bird-openwrt package.

Acknowledgements

I want to thank Joan Manuel Marquès Puig for helping me to get in touch with Víctor, thus being able to agree and produce this dissertation. To Víctor Oncins Biosca, for supporting bird-openwrt solution and pushing me getting back to its development and, later, to become my tutor in this dissertation, helping me to understand administrators perspective and their requirements in order to implement a much better solution which, definitively, will keep rolling. Moreover, I would like to say thank you for all the time spent not only during our review meetings, but working together with the testing environments, which were a big challenge and an invaluable amount of experience that I could not have had otherwise. To Andreu Bassols Alcón for his continuous support managing all the resources I did require during this project's development. Finally, special thanks to Sílvia, who has patiently supported me during the whole process.

Index

List of Figures

Listings

Chapter 1

Introduction

This project aims to simplify and enhance the management and monitoring capabilities of network's administrators using Bird Daemon software on top of an OpenWrt/LEDE-based Firmware. It is a second iteration in the development of an existing configuration integration package already being used by OpenWRT/LEDE's community. Section 1.1 introduces the key concepts used in in the introduction and as part of the dissertation itself.

1.1 Background Concepts

The following subsections are intended to be self-contained definitions for contents recurrently referenced in this dissertation, sorting them by scope (from top to bottom). The two highest-level topics covered are OpenWrt and LEDE environments and project's target network.

1.1.1 OpenWrt/LEDE Project

OpenWrt, and its Fork LEDE (Linux Embedded Development Environment), are Open Source Linux-based firmwares primarily focused on commodity routers, but aiming to work in any Linux-based system. This firmware supports a wide variety of manufacturer's hardware and also a wide range of software, services and routing protocols to enhance, secure and efficiently operate as a standalone router and service provider. This community-driven firmware uses different internal GIT source repositories as well as the GitHub service, which is the most widely used. As any other open source project using GIT, it allows users to contribute to it by doing changes in their own repository forks and then creating Pull Requests in GitHub's website in order to get their changes approved and integrated in the official stream. This procedure is followed in all the different repositories that the project owns and has a big number of reviewers to protect the project's code quality and the system's stability.

1.1.1.1 UCI

The Unified Configuration Interface aims to centralise OpenWrt's packages and the system configuration. This mechanism is widely used for almost, if not all, the packages working in OpenWrt. UCI allows you to easily, and in a human-readable manner, simplify administration overheads. For reference, the following UCI configuration item configures the loopback interface:

```
config interface 'loopback'
    option ifname 'lo'
    option proto 'static'
    option ipaddr '127.0.0.1'
    option netmask '255.0.0.0'
```

Listing 1.1: Loopback network interface's configuration using UCI.

1.1.1.2 LUCI

Lua UCI is OpenWrt's implementation provide a graphical to manage UCI settings via web pages (UI is generated on Server side) following a Model-View-Controller software pattern implementation and using Lua programming language.

LUCI's main components are:

- CBI (Model): CBI files include the UCI definition and *describe* HTML's presentation (e.g optional/mandatory properties, order of apparition, and all the required logic to validate entered data through the forms).
- Controller: Web pages definition (e.g data, pages order or rendering target) and communication functions required by the Model (CBI) page.
- View: HTML templates defining how a page, section or specific element is represented.

1.1.1.3 LUCI2

LUCI2 is the 2nd generation of OpenWrt's UCI UI modeling architecture. This second version uses client side page generation technologies (HTML, CSS and JavaScript) to enhance and simplify UI creation and also releasing router's resources from handling UI creation and management. Moreover, this new version uses a standard communication process between client's browser and router's backend http server: standard XMLHTTPRequest (XHR) API. This standard API uses HTTP Requests to transfer objects by embedding them in the URL but avoiding page reloads. This mechanism's most common object format is JavaScript Object Notation (JSON).

Because of the new requirements emerged as consequence of moving UI's creation from the server side to the client side, we need a mechanism together with an architecture supporting it (explained in section 3.3.1), to query router's data:

- Universal Bus (uBus) is a service that allows communication between different services. It is implemented to act as a pipe for packages or protocols to register their APIs in a shared *namespace*. uBus provides its own httpd server plugin to communicate with client's browser using it.

In order to use uBus capabilities, services must integrate it in their code. For those packages not being able (or not willing) to do it, uBus also provides an alternative registration method:

- RPCd (Remote Procedure Calls Daemon) is an HTTP backend server implementation for client-server communication calls in network-based systems. In OpenWrt's case, queries are mainly sent from client's browser to the backend server (RPCd).

  This backend server allows non-uBus-integrated services to register an API with it, using Shell commands to define required functions. Services registering to RPCd will be able to query uBus through client calls but these APIs will need to handle request's data and data object's format.

Finally, this new version is still experimental. There is not many documentation of its API definition, how to use or integrate it in a new or existing package and, therefore, it is complex to get the full picture of how this architecture works or how to integrate it.

1.1.2 Bird Daemon

Bird Daemon (from now onwards Bird)[1] is an open source Internet Routing service that allows network administrators to simplify route sharing configuration, management and monitoring of different routing protocols by using Routing tables as *transferable* knowledge and a powerful filtering c-like language to achieve it with really fine-grained results. Bird manages its own configuration also following the same c-like scheme, which was the main goal of my 2014 project: to automate and simplify it by using UCI instead and letting the Package do the translation to Bird configuration.

Bird current version is 1.6.3 and its functionality is split in two different Daemons: one for IPv4 (Bird4) and one for IPv6 (Bird6). This version supports the following routing protocols:

1.1.2.1 Routing Protocols

- Babel: IGP (Interior Gateway Protocol) distance-vector protocol stated as being in alpha stage of adoption. This protocol is not available to automate through our Package yet.
- **Open Shortest Path First - OSPF**: IGP link-state protocol fully supported for both IPv4 OSPFv2 and IPv6 OSPFv3. This protocol has some functionality available for IPv4 in the Package but is not fully functional and there is no UI supporting it. OSPF support is one of the top priorities for future Package improvements.
- **Routing Information Protocol - RIP**: IGP distance-vector protocol fully supported. This protocol is completely deprecated by other distance-vector protocols as OSPF, which are less constrained by network's scale. This protocol is not available to automate through our Package and, because it is obsolete, there are not plans to implement it in short term.
- **Border Gateway Protocol - BGP**: EGP (Exterior Gateway Protocol) path-vector protocol fully supported. This is the most common protocol for backbone networks and the key protocol for this project. This protocol is available to automate through the Package but only the most common or relevant options have been implemented. BGP's support finalisation is one of the top priorities for future Package improvements.

.

1.1.2.2 Bird Support *Protocols*

The following list of Bird utilities have been named protocols. However, these are not real routing protocols but a number of supportive implementations of different functionalities or standard services to enhance and simplify Bird's management. This naming convention unifies Bird's implementation and configuration management, but it must not be mistaken for a real protocol.

- **Static *Protocol***: Bird's mechanism to implement *smart* static routes. It allows origin or pattern discrimination and modification. For example, to configure any route in 10.0.0.0/16 to be unreachable or to extend it with an attribute: `ospf_metric = 100`. Static Protocol is available to automate through the Package.
- **Pipe *Protocol***: Routing Tables are the main, and OS standard, knowledge units for Bird (e.g BGP & OSPF primary tables). Pipe Protocol allows to connect different Routing Tables and to apply discrimination to the routes they share (e.g `BGP->OSPF accept all` and

`OSPF->BGP accept` filtered to: `if net ~ 10.0.0.0/8`). Pipe Protocol is available to automate through the Package.

- **Direct *Protocol***: Route generator for any targeted interface. Bird uses pattern matching to include/exclude any network interface that we want to be encapsulated as a single bunch of *device* routes. Direct Protocol is available to automate through the Package.
- **Device *Protocol***: This functionality is required in most of the Bird configurations and its main purpose is to gather key data from system's network interfaces in order to facilitate Bird's operation. Device Protocol is available to automate through the Package.
- **Kernel *Protocol***: Bird's implementation to allow sharing routes between the Operative System Kernel Routing Tables and the ones designated by Bird. Kernel Protocol is available to automate through the Package. There are plans to improve this protocol as there is one process left to be automated through UI and it is top priority.
- **RAdv *Protocol***: Implementation of the Router Advertisement Protocol (IPV6's Neighbour Discovery) allowing a fine-grained control of how often the neighbour discovery information is sent and which information is shared on them per target. RAdv Protocol is not available to automate through the Package. There are no plans to implement it in the short term.
- **Biderectional Forwarding Detection - BFD *Protocol***: This utility is a standalone tool for neighbours monitoring in order to foresee some protocol service disruptions by monitoring peers in a more efficient way than most protocols do. This protocol consists in a session created by real routing protocols (e.g OSPF and BGP) and its sole role is to notify them in case of an event. This protocol is almost fully supported (except of verbose mode and authentication). BFD Protocol is not available to automate through the Package. There are not plans to implement it in short the term.

1.1.2.3 Bird Filter and Functions

One of Bird's key functionalities is network routes filtering through scripting. Bird's main configuration allows an administrator to define rules that routes imported or exported will need to be compliant with to avoid pass discrimination using its own programming simple *language* (loops are not implemented). This language provides:

- Functions: self-contained functionality required in several places. Their function is to reduce code's repetition but not to accept or drop routes.

- Filters: instructions executed by Bird on each route received (or about to be sent) through the different protocols in order to modify route's properties and to, according to the defined requirements, let the route pass or not.

1.1.3 OpenWrt/LEDE's configuration integration package

Bird-OpenWrt Package (from now onwards the Package) is an open source OpenWrt/LEDE-specific solution compound by four separated packages for UCI and LUCI configuration management and presentation for both IPv4 (bird4-uci and luci-app-bird4) and IPv6 (bird6-uci and luci-app-bird6) Bird implementations.

These packages provide administrators using Bird an user-friendly configuration scheme (UCI), automated configuration processes and graphical interface capabilities (LUCI) instead of its c-like configuration scheme. Specific implementation details are covered in chapter 3.

This Package is currently part of OpenWrt's official stream (version 0.2) in GitHub and developers can contribute to it in the same way as with any other OpenWrt/LEDE component or service.

1.1.4 Guifi.net

Guifi.net is a community-driven network working for and by its own users providing an affordable alternative for anyone willing to connect to the Internet. This network's principles are freedom; open design, administration and management; and neutrality. It was born in Catalonia as a wireless network (Access Point to Endpoint and Access Point to Access Point) but it has spread all over the world with about 33.124 active nodes (as 26/05/17) using roof antennas and, more recently, optical fiber deployments. This has made auspicious for other network topologies as mesh networks, making of Guifi.net a rich and heterogeneous network.

This network has its own Government Foundation, La Fundació Guifi.net, which main goals are to promote and protect network's principles defined in an operational and behavioural common regulation named XOLN, support any company that adheres to XOLN's principles allowing them to be able to professionally operate and advertise themselves as Guifi.net service providers and, last but not least, to represent Guifi.net network and the internal service providers as a member (peer) of Catalan's Internet Exchange Point (CATNIX) bringing the possibility for any internal service provider to peer with other members to exchange traffic and routes[1].

[1] For reference, one of the latest incorporations as a member to CATNIX is AKAMAI service provider, thus making feasible for Guifi.net internal service providers to peer with this worldwide Content Delivery Network.

Finally, although Guifi.net main routing protocol is BGP for infrastructure and OSPF for internal routing, there are several isles operating as Mesh Networks using BMX6 dynamic routing protocol.

1.1.5 Infrastructure vs Mesh Network Routing Protocols

Routing protocols' job is to receive a route and, according to its attributes and the information stored in the system, to redirect this route to the next step towards its destination or to drop it. However, each protocol follows different principles in order to achieve the best performance using different algorithms or paradigms. Moreover, depending on who is consuming the network and which are its requirements, we could prioritise either scalability, stability or resilience and prioritise how much critical are the previous characteristics for our consumers:

- **Infrastructure Protocols**: commonly used in *backbone* networks, which are commonly deployed in robust and powerful devices following a specific architecture. These protocols' strength is to be stable, robust and highly scalable. However, their main handicap is that they suffer from big overheads on topology changes. This means that these protocols are low/non fault tolerant and this is translated into longer convergence times (seconds to few minutes) in large-scale networks.
- **Mesh Protocols**: oppositely to backbone networks, mesh networks' strength is to be able to converge almost instantly after any topology event. These networks work in a cooperative manner in order to achieve a fully connected network (point-to-multipoint) where all the nodes share network's knowledge in order to optimise routes and nodes floods the network in order to keep network's topology knowledge up to date. This is a strong requirement due to the heterogeneity of these networks and that, some of them, are configured in unpredictable environments with high feasibility of suffering from topology changes. Nevertheless, this network's topology is built on the premise of being able to interconnect as many nodes as possible to each other to be able to quickly redirect network's transit to reach any node in the system, even on a topology change event.

1.1.5.1 BGP

BGP is a dynamic infrastructure IP routing protocol designed for large-scale internet topologies Autonomouos Systems routing entities (Autonomous Systems are internally compound by from small to mid scale networks acting as a single entity). Its routing algorithm relies on the best path according to route's attributes.

1.1.5.2 BMX6

Batman-eXperimental6 [2] is a BATMAN (Better Approach To Mobile Adhoc Networking) mesh protocol's fork using different routing algorithms. This routing protocol is compatible with most of linux-like systems, it is optimised for IPv6 networks while it is also able to announce and receive IPv4 prefixes. This routing protocol uses a table-driven distance-vector approach (BMX6 tries to compose a routing table with all the source-destination entries and routes traffic according to the best path to reach the destination.

1.2 Motivation

Back in 2014, while I was working on my BSc. dissertation at Universitat Politècnica de Catalunya (UPC), the department and, specifically, the investigation team I was working with, gave me the opportunity to participate in a Google Summer of Code project under the umbrella of Freifunk, to design, develop and demonstrate a package that would help to simplify the configuration of Bird as a software able to share routes between BMX6 mesh and BGP infrastructure networks. This solution was meant to be installed in *border* nodes deployed in the Catalan community network Guifi.net and, therefore, production ready.

That project was successful and the result was an integration package using OpenWrt's well-known UCI/LUCI configuration mechanism to set up Bird through an user-friendly Web UI even without knowledge of Bird's syntax. However, GSoC's time frame was not enough to polish the package and to add other non-critical routing protocols. Moreover, the Package stopped being maintained later that year. Nevertheless, this OSS project has been on my *backlog* of things I want to keep improving and also I have been queried some times by Víctor Oncins as this package is really helpful for network administrators but it is not mature enough for complex production environments available in Guifi.net.

Therefore, I have been really fortunate to have the opportunity to retake the development of this package as my MSc. project and to work together with Víctor as this has meant that I have had direct feedback from administrators using the tool in production environments and to improve its most critical features. Moreover, Víctor has also published a report on GitHub [3] describing the main challenges found using the old version of the Package and a deep description of the environment.

1.2.1 Bird Daemon's administration issues

As part of the GSoC project, the solution provided was not mature enough to fulfil all the requirements:

- Tight time-frame forcing to prioritise the key capabilities to implement.

- Some key protocols were not enabled in the final solution because they were not relevant for GSoC's scope (e.g Pipe or Direct).
- Some routing protocols were not enabled in the final solution because they were not relevant for GSoC's scope (e.g OSPF or Babel).
- Some basic processes require manual (terminal) changes.
- No possible way to edit Filters or Functions files through Web UI.
- No Bird Daemon Status feedback (e.g no way to know if bird is running or failed to start through Web UI).
- No possible way to see Bird Daemon's Log information through Web UI.
- Bird's API changed (from Bird 1.4.3 to 1.6.3) making bird crash using base Package configuration.
- No possible way of monitoring Bird's current status (e.g full information for BGP connections).

1.3 Scope of the project

This project's scope is to adopt some of the mentioned enhancements that are in consonance with command line based processes automation, to improve current UI's user experience and to align the Package with the current Bird Daemon's API in a 3 months time frame.

As a result of a *backlog* prioritization, the following items were agreed (shown in priority order):

- Update the Package to the latest Bird API.
- Update old version's disruptive issues (e.g disabled critical Protocols).
- Add Status, Log, Functions and Filters Bird capabilities graphical integration.
- Do a theoretical viability investigation of OpenWrt's uBus integration in order to gather Bird's live data and to be able to use it in the Package.

1.3.1 Deviations from the original plan and future work

While agreeing the original scope of the project, few extra ideas and tasks were planned but, as a matter of priorities and time constraints, some were dismissed or set as future work.

- Add missing routing protocols: adopt OSPF routing protocol, other less critical routing protocols and functionalities, hence increasing the range of capabilities that are automated for administrators tu use with no command line expertise required.
- Integrate next generation of Web UI using LUCI2: JavaScript-powered UI using uBus architecture instead of current Lua-based one.
- Implementation of uBus integration according to the results of the investigation done in this project.
- Comparative set of tests between Quagga and Bird Daemon solutions.

Most of these extra tasks are already documented as part of the Package Documentation Repository [4] and open for discussion if new requirements arise.

1.3.1.1 Bird Daemon Vs. Quagga deployments

There is a special reasoning behind not doing a comparative analysis of these two solutions. Of course the timing constrains have strongly influenced the decision of dropping this comparison from this project's scope, but there is also the big amount of evidence already collected for my GSoC2014 project, plus some new evidence found either in some reputable sources as well as from Bird's own OSS Community proving that Bird Daemon has become stable, less resource eater and more flexible (thanks to its Filter&Function scripting language) than other well-known enterprise level solutions. This evidence and where it is coming from is available in Appendix B.

1.4 Methodology and communication

This project starts with the premise that there is no need for a wide initial investigation phase as the Package used was designed and developed by myself. Nevertheless, there are three foreseen introductory tasks:

- Refresh the Package to the latest Bird Daemon version API.
- Investigate, understand and document the production environment.
- Update Documentation and prepare the repositories required (documentation, package and dissertation).

After this initial phase, the implementation tasks will be executed in a Kanban-like approach:

- Features will be executed following Backlog's priority order and one at a time.

- Each *feature* or *requirement* must be self-contained and the Package should be releasable at any time.
- There is no Board or framework to introduce the data (e.g time spent or state of the tasks) as such as the overhead of doing it is not proportional to the number of tasks or value of the data that could be collected. However, during the first *Cycle* of the project (first two weeks), in order to illustrate how could this project look like using Kanban, I did use an online OSS tool called *Taiga.io*. Appendix C includes some screenshots showing initial tasks created using the tool.
- There will be weekly/bi-weekly meetings with the Stakeholder in order to discuss progress, any blocker or issue and rearrange priorities if required.
- There will be a *demo* for the Stakeholder in order to show him the progress in a weekly/bi-weekly basis.

The communication, as already mentioned, will be done through regular meetings with the External Consultant (Stakeholder) using the Jitsi VoIP conference service, which allows screen sharing and text communication while in conference, simplifying demoing and code reviews. Regular communication will also be done through Hangouts instant messaging service and by email to share progress, risks or blockers.

1.4.0.1 Gantt Diagram

Tasks' delivery forecast can be seen in Figures 1.1 and 1.2:

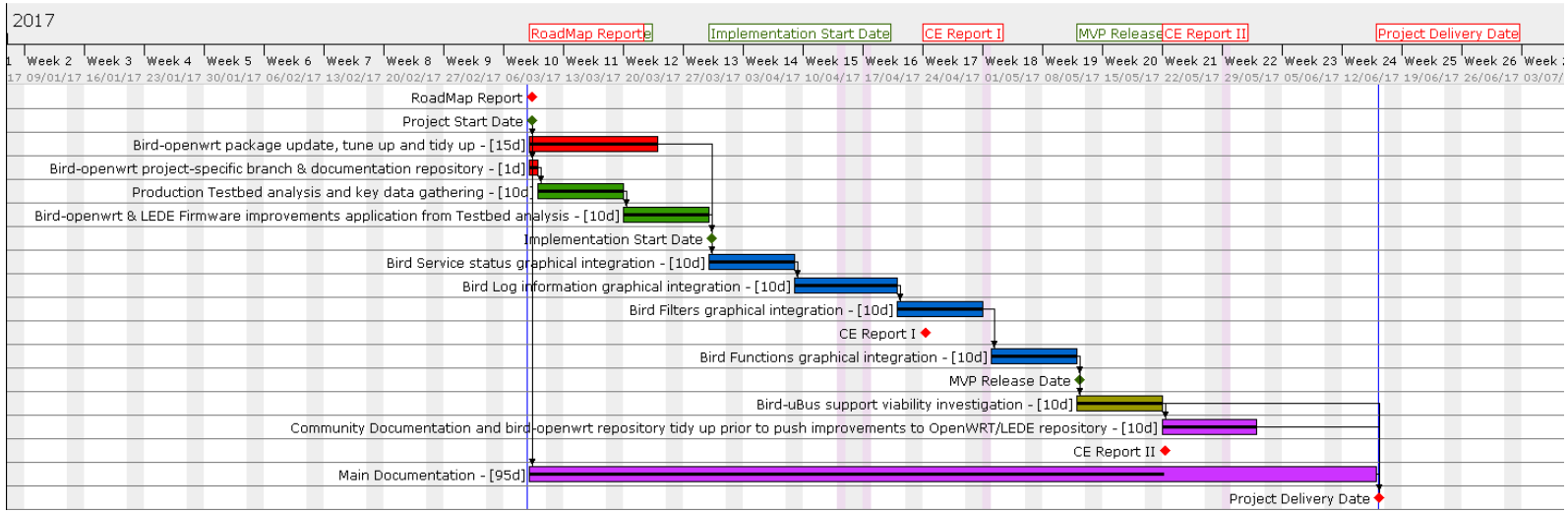

Figure 1.1: Tasks schedule

Key milestones:

- **Project's start date & RoadMap Report** (09/03/17): initial Package refresh and production environment investigation. Project's goals formal report and when they are expected to be delivered.
- **Project's implementation start date** (30/03/17): beginning of features' implementation.
- **Continuous Evaluation Report I** (22/04/17): formal report to present Project's progress, pending work, any issue or blocker and updated timeline.

- **MVP Release** (12/05/2017): forecast delivery date of the final version of the Package. No extra changes planned unless the investigation task requires them.
- **CE Report II** (22/05/17): optional progress report prior to Project's delivery.
- **Project Delivery Date** (12/06/17): final date to deliver the dissertation, slides, recording and any extra archive required.

GANTT project

Duration	Name	Begin date	End date
0	RoadMap Report	09/03/17	09/03/17
0	Project Start Date	09/03/17	09/03/17
15	Bird-openwrt package update, tune up and tidy up - [15d]	09/03/17	23/03/17
1	Bird-openwrt project-specific branch & documentation repository - [1d]	09/03/17	09/03/17
10	Production Testbed analysis and key data gathering - [10d]	10/03/17	19/03/17
10	Bird-openwrt & LEDE Firmware improvements application from Testbed analysis - [10d]	20/03/17	29/03/17
0	Implementation Start Date	30/03/17	30/03/17
10	Bird Service status graphical integration - [10d]	30/03/17	08/04/17
10	Bird Log information graphical integration - [10d]	09/04/17	20/04/17
10	Bird Filters graphical integration - [10d]	21/04/17	30/04/17
0	CE Report I	24/04/17	24/04/17
10	Bird Functions graphical integration - [10d]	02/05/17	11/05/17
0	MVP Release Date	12/05/17	12/05/17
10	Bird-uBus support viability investigation - [10d]	12/05/17	21/05/17
10	Community Documentation and bird-openwrt repository tidy up prior to push improvements to OpenWRT/LEDE repository - [10d]	22/05/17	01/06/17
0	CE Report II	22/05/17	22/05/17
95	Main Documentation - [95d]	09/03/17	15/06/17
0	Project Delivery Date	16/06/17	16/06/17

Figure 1.2: Schedule details

1.5 Obtained results' brief summary

This project has two main achievements:

The first one is the self-contained major update for bird-openwrt package, which has not only brought support for the latest Bird Daemon API, but also introduced a number of new functionalities and process automations critical for its adoption by our target administrators and the rest of OpenWrt's community. The second main achievement is the detailed uBus integration analysis including some implementation examples and foreseen future development paths. As part of the analysis, I have also detailed a possible project proposal following the most promising implementation path for future students willing to learn about OpenWrt/LEDE, Bird Daemon and uBus communication architecture.

All contents mentioned in this dissertation are available on GitHub:

- Bird-OpenWrt Package:
  https://github.com/eloicaso/bird-openwrt
- Project's extra documentation:
  https://github.com/eloicaso/bgp-bmx6-bird-docn
- This dissertation:
  https://github.com/eloicaso/msc_dissertation

1.6 Structure of the document

This dissertation is organized as follows: Chapter 1 introduces background concepts to understand this project's scope and basic management information. Chapter 2 presents target and tested network architectures and its requirements. Chapter 3 goes in depth detail about which have been Package's main improvements and also briefly describing previous capabilities. Moreover, this chapter also details the theoretical analysis done on Bird's uBus integration feasibility. Chapter 4 brings exhaustive information regarding performed tests in both initial development and virtual real scenario networks, detailed information of the main differences between the original Package version and this project's enhancements, and also, common Bird errors and how to solve them. Finally, Chapter 5 summarises main achievements, main challenges and whether they have been solved or not, and also future development lines.

Chapter 2

Network Architecture

This project's target network is a new experimental-production network located in Barcelona. It has been deployed in order to prove its stability, robustness and feasibility to replicate it in other Barcelona's locations with similar topologies. This network's main particularity is that it merges different network topologies requiring the use of Exterior Gateway Protocols (BGP) together with Internal Gateway Protocols (BMX6) in order to be able to share routes between different BGP ends (named E and F in the figure 2.1) going through a Mesh Network. The following information has been extracted from the report "Integració entre BMX6 i BGP en dispositius basats en LEDE" [3].

As shown in the figure 2.1, network's elements are:

- **Infrastructure Super Node 1** (ISN1): BGP Supernode connected to the BGP network (Guifi.net, section 1) via wireless and to the MXN1 Router through Ethernet.
- **Mesh eXchange Node 1** (MXN1): LEDE/OpenWrt router connected via Ethernet to the ISN1 and to the antenna (or an Ethernet port) providing access to the Mesh Network. This frontier node provides BGP to BMX6 route-sharing capabilities using Bird.
- **Mesh Network**: A number of Nodes connected using BMX6 forming an isle between BGP nodes.
- **Mesh eXchange Node 2** (MXN2): LEDE/OpenWrt router connected via Ethernet to the ISN2 and to the antenna (or an Ethernet port) providing access to the Mesh Network. This frontier node provides BGP to BMX6 route-sharing capabilities using Bird.
- **Infrastructure Super Node 2** (ISN1): BGP Super Node connected to the BGP network (Guifi.net, section 2) via wireless and to the MXN2 Router through Ethernet.

2.1 Routing requirements

The routing requirements to successfully ensure that all routes are shared between both BGP ends are:

- Routes must be shared/announced between ISN1 (**E**) and ISN2 (**F**).
- Mesh Network's Routes (BMX6 - **C**&**D**) must be shared/announced to ISN1 and ISN2 (**A**&**B**). Therefore, shared/announced to Guifi.net network.
- ISN1 and ISN2 Routes (BGP - **A**&**B**) must be shared/announced to the Mesh Network (**C**&**D**).
- MXN1/2 must configure Bird to use a custom Routing Table that will be shared with BMX6.
- MXN1/2 must configure BMX6 to use the *Table* plugin in order to redirect its routes from Kernel's Table to a custom one.
- MXN1/2 must configure Bird to set them both as BGP Peers to stablish an iBGP session between them (AS2).
- MXN1 must configure Bird to set ISN1 as BGP Peer AS1
- MXN2 must configure Bird to set ISN1 as BGP Peer AS3

2.1.1 Caveats

There is an important caveat with this network distribution:

The current version of BMX6 is not able to handle the number of announced prefixes that Barcelona's Guifi.net BGP sessions are currently sharing (3.500+). Therefore, in order to be able to handle those received network prefixes, BMX6 will have to start aggregating its routes, which eventually would shut-down the service and leave the node overloaded or even unresponsive.

In order to avoid this disruptive behaviour, Bird filter scripting capabilities available in MXN1 and MXN2 allow to reduce the geographical scope of the prefixes imported and exported to and from the Barcelonès zone (including cities: Badalona, Barcelona, Hospitalet del Llobregat, Sant Adrià del Besos and Santa Coloma de Gramanet).

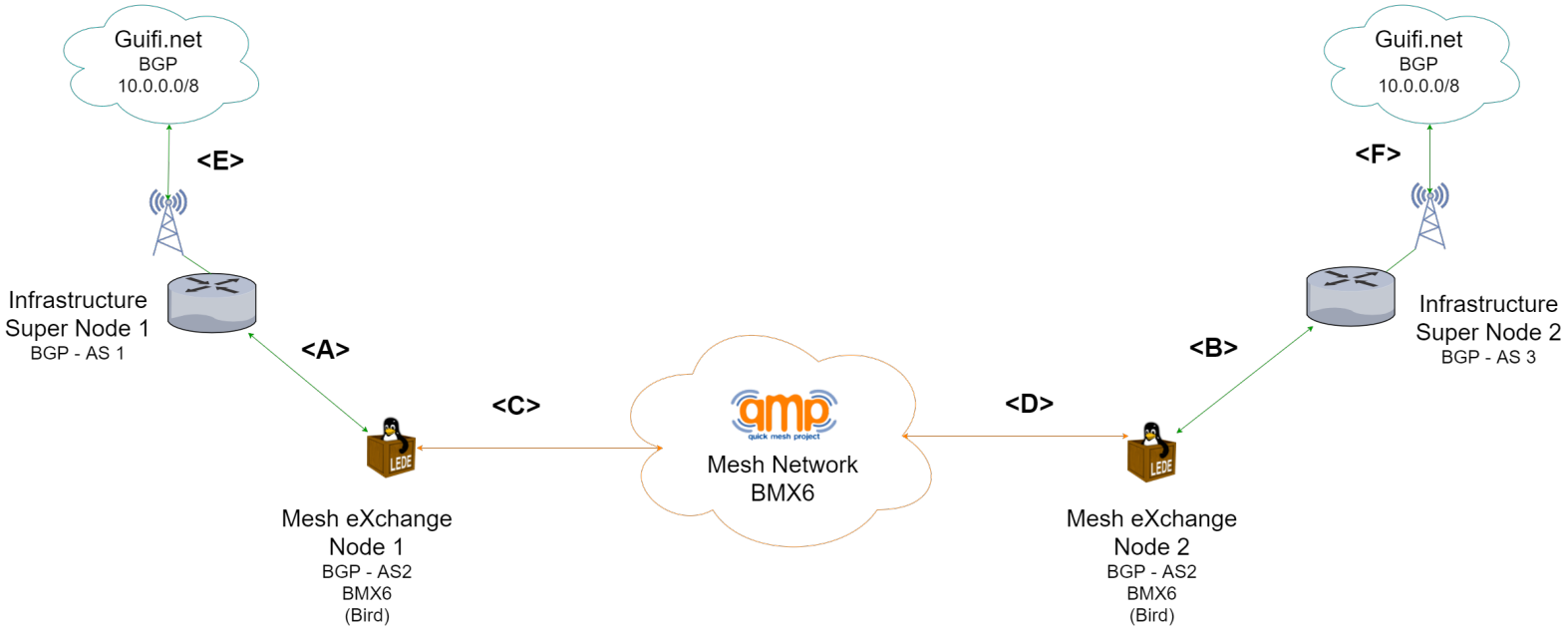

Figure 2.1: Production Network targeted in this project

2.2 Testing Scenarios

Although it has not been possible to test Package's improvements in the target production network as the changes would incur in service disruption for up to 1500+ nodes in the Barcelonès Zone, it is foreseen to update target *MXN* Nodes after agreeing it with all the involved parts.

Nevertheless, this Package's improvements have been tested in two different environments connected directly with Guifi.net thanks to Universitat Oberta de Catalunya and Víctor's support, who have provided and helped me configuring a number of Virtual Machines, virtual network resources and also agreed permission to get through another university's network, Universitat Pompeu Fabra, in order to connect to a different network section, but still in Barcelona, thus following the same approach of different BGP Peers being able to route and share prefixes.

2.2.0.1 Internal Development Testing

Package development tests have been done inside UOC's network using a really simple network topology where our target Bird Node VM is connected to Guifi.net using UOC's Infrastructure Node (UOC receives/announces 3000+ BGP Routes).

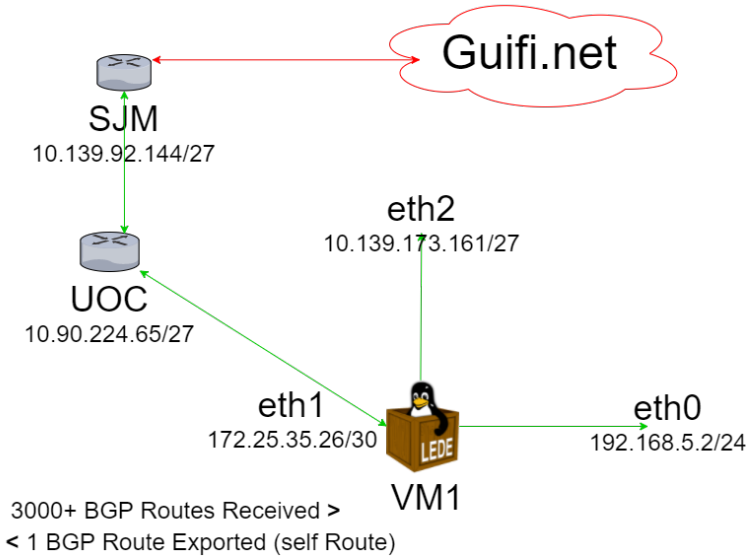

Figure 2.2: Minimal test environment.

2.2.0.2 Final Package Testing

As can be seen in the Figure 2.3, the final testing environment is a virtual mirror of the target network. This environment has been created in order

to do the final tests after *releasing* the final version of the Package to test it without the risk of damaging the production network or flooding unwanted routes to Guifi.net. This network section routes to two Infrastructure Super Nodes connected to two geographically-separated Barcelona well-known Universities, being almost, if not exactly, a mirror of what our target network is.

The components provided in order to achieve this network have been:

- VPN access to the Guifi Network using UOC's resources.
- 4 Virtual Machines using LEDE17.01 Firmware in University's network.
- Virtual Bridge to connect the VMs simulating a fully connected Mesh Network.
- Network connection through two different Barcelona's network sections.
  - Virtual Machine 1 connects through UOC's Super Node
  - Virtual Machine 4 connects through UPF's internal network, to a near Guifi network section.
  - Both Infrastructure SuperNodes are importing all their routes to our network
  - UPF's SuperNode throughput has been limited to avoid disruption in their internal network.
    * Connection through UPF's internal network using a L2TP Tunnel.
    * We have agreed to do this testing in a limited time-frame to avoid disruption in their services as we are sharing routes between Guifi.net-UOC-UPF-Guifi.net.

2.2.0.3 Testing environment elements

- **SJM**: Guifi.net Node BCNSantJoanDeMalta51. Infrastructure Node with 6 Point-to-Point connections to other Super Nodes. As we have no control on this node, we will consider that it is importing and exporting any received route to/from Guifi.net.
- **UOC**: Guifi.net Node BCNRamblaPobleNou156. Infrastructure Node located in the Universitat Oberta de Catalunya and connected to the Super Node BCNSantJoanDeMalta51. This node is shown as AS1.
- **VM1**: KVM Virtual Machine acting as Frontier Node (MXN1). This node is configured with Bird (BGP AS2) and BMX6 in order to connect

to its BGP neighbour AS1 (ISN1), to its iBGP Peer AS2 (MXN2) and to the BMX6 Mesh network.

- **VM2 & VM3**: KVM Virtual Machines acting as plain Mesh nodes.
- **VM4**: KVM Virtual Machine acting as Frontier Node (MXN2). This node is configured with Bird (BGP AS2) and BMX6. It connects to its BGP neighbour AS3 (ISN2) over a GRE Tunnel configured in UPF's internal network, also to its iBGP Peer AS2 (MXN2) and to the BMX6 Mesh network.
- **UPF**: Guifi.net Node BCNUPFPobleNou. Infrastructure Node located in the Universitat Pompeu Fabra and reached through UPF's internal network. Therefore, VM4 connects using an internal IP Address supplied by UPF's administrators. This node is shown as AS3.

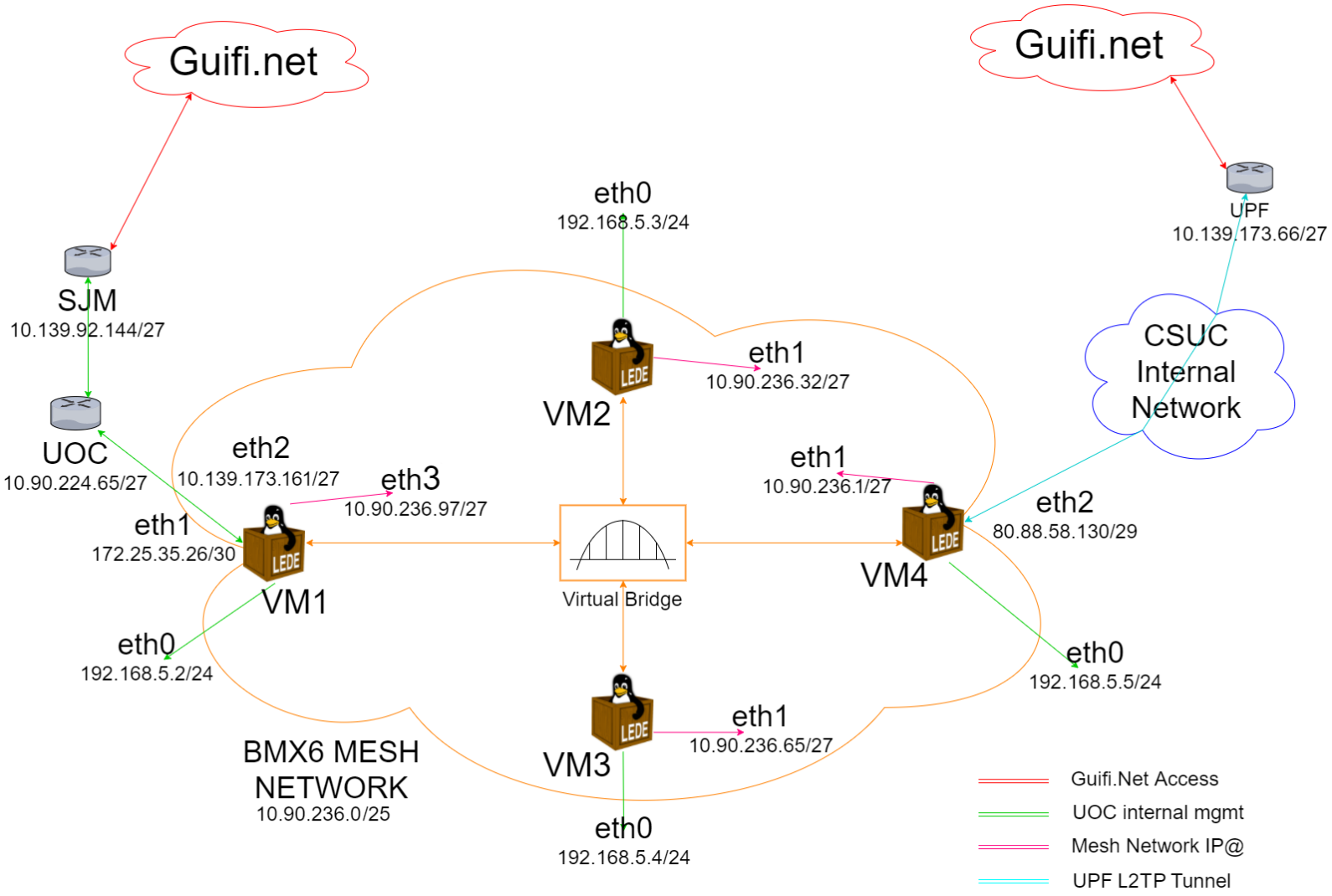

Figure 2.3: Development Network simulating production's environment

Chapter 3

Package improvements' implementation

3.1 Administration requirements

One of the administrator's main tasks while managing networks is, if required, to facilitate the coexistence of different Routing protocols in the same network section. Hence the requirement of rich routing tools as Quagga or Bird to act as facilitators of this route *crossed-announcement* -and even attributes translation- between protocols.

Administrators require:

- A full-featured tool with an easy and intuitive UI to manage and monitor protocol health and data efficiently and avoiding any handmade/-custom edit, reducing configuration's complexity.
- Use of LEDE/OpenWRT-based firmwares widely used in the target network.
- A Routing Protocols' management tool that, at least, supports BGP Static Routing Protocol and it is able to share routes with BMX6 Dynamic Routing Protocol in a manageable way.
- Use Bird instead of Quagga to make use of its proven efficiency, low resource consumption and powerful filter capabilities, which is critical to some of the widely used commodity hardware in Guifi.net.
- Use and improve Bird's configuration integration package (bird-uci and luci-app-bird) available in the official Routing Repository of LEDE/OpenWRT.
- Avoid project-specific customisations in the integration package that would not benefit all the community. If required, add those custom enhancements in a development branch.

- Update Package's documentation and create new topics to cover Web UI interface and any manual process not covered by package's improvements.
- Update Bird integration package in order to be compliant to the latest API (v1.6.3 when this document was written).
- Enhance Web UI to support user-friendly configuration and visualisation of the following:
  - Bird service status.
  - Bird events information (Logs).
  - Filters and Functions editing using an embedded HTML text editor.
  - Update old configuration Web pages, fix some outdated options and re-sort them to a more logically order.
- Do theoretical viability investigation to use uBus as a mechanism to communicate with Bird and get health information and current-status information for handled protocol using JSON messages.

3.2 Implemented changes and improvements

The following sections summarise what has been changed as part of this project's development, which changes have been successful, challenges found and lessons learnt that will help towards future versions of the Package.

3.2.1 Building and deployment process documentation update

One of the first challenges I found during the initial investigation was that the documented process for building the Package was wrong. Because this process was originally written and tested on 2014 using OpenWrt, it was misaligned and led into few days testing this process using the latest version of the development environment in order to tune it. Main changes are:

- Generalise Makefiles: remove hardcoded references to bird4/bird6 where possible on Makefile and post installation scripts.
- Re-test and update steps: Refresh steps required to compile (`make`) and deploy the Package in the target test environment.
- Add extra useful information: Package version's information, dependencies, links to documentation and known issues,

3.2.2 Apply code standards

As a daily Bash user, one of the main concerns that I had once I did retake the project was the state of the code because I did stop giving support to the Package on 2014 and I have improved drastically my consciousness towards clean, standard and following-best-practises code. Therefore, the top priority task I got was to normalise the code, apply best practises and, where possible, refactor it to follow a *library*-pattern to be able to formalise an API and, in future releases, even to create unitary tests that would automate Package's tests.

The first challenge I found was that LEDE/OpenWRT firmwares use the light compound of Linux tools BusyBox. This all-in-one tool comes really handy in embedded environments where performance and storage are critical but some of its tools are limited versions of the original ones.

Particularly, the tool that has been more challenging is *ash*, which is the built-in Shell Command Line included instead of *Bash* and, although it includes most of its features, there are few others like Arrays that are not available and require the developer to re-think the solution (Ash readme page suggest the use of `set` command).

Some examples of the improvements applied are:

- Encapsulate variables with curly brackets to avoid wrong substitutions or other common issues where mixing variable names and other strings:

```
root@LEDE:~# path="/etc/"
root@LEDE:~# ls $pathconfig/bird4
ls: /bird4: No such file or directory
root@LEDE:~# ls ${path}config/bird4
/etc/config/bird4
```

Listing 3.1: Variable encapsulation.

  This is a forced example but there are some instances where, in really complex scripts, unexpected substitutions could happen.

- Encapsulate Strings appropriately to avoid unexpected substitutions or code execution (e.g. script injection): this is an uncommon situation that could happen with commands like `sed`, where quotes and other special symbols are crucial to get the expected output.

- Use of 4 spaces instead of tabs for code readability.

- Use of simplified `if` statements only with clear and single-line occasions. Avoid using simplifications on instances where more than one line is required or the command is too large and would be more reasonable to split it using backslash (\).

  Acceptable:

```
root@LEDE:# var="true"
root@LEDE:# [ "$var" = "true" ] && install_package="y" ||
    install_package="n"
var is true
```

Listing 3.2: If statement simplification (I)

Unacceptable:

```
root@LEDE:# [ "$(uname)" = "Linux" ] && { . /etc/os-release;
 echo -e "\n $LEDE_RELEASE \n"; } || echo "Not Available"

LEDE Reboot SNAPSHOT r3969-8322dba

root@LEDE:#
```

Listing 3.3: If statement simplification (II)

3.2.3 init.d script and service management

Bird's init.d script (`/etc/init.d/bird4`) manages the service on boot or on demand. Bird's init.d file is substituted on the installation time by `/etc/bird{4|6}/init.d/bird{4|6}` script, and the original one backed up as `/etc/bird{4|6}/init.d/bird{4|6}.orig` to be restorable in the event of uninstalling the Package.

Improvements introduced are:

- Refactor init.d script and split it in two files. `bird{4|6}` for service management and `bird{4|6}-lib.sh` as an *API/function* holder for UCI-bird configuration translation.
- Store a backup of the current configuration each time the service is started. However, this backup is overwritten each time and it is administrator decision to take a copy of this file before reloading the service.
- Add *smart* service management to avoid multiple start/stop/restart calls to the service, causing service disruptions if not required (e.g. multiple start calls should be ignored). See figure 3.1.
- Add extra management functions for LUCI Web management. These new functions call the original ones but forcing plain text outputs. See figure 3.2.
- Re-sort UCI translation script's in order to fix an issue with Functions and Filters.
- Enhance service handling and error information logging, previously dismissed.

```
root@LEDE:~# /etc/init.d/bird4 status
bird4 start status: [ RUNNING ]
root@LEDE:~# /etc/init.d/bird4 stop
bird4 Daemon Stop Status: [ OK ]
root@LEDE:~# /etc/init.d/bird4 stop
bird4 Daemon Service already stopped. [ FAILED ]
root@LEDE:~# /etc/init.d/bird4 restart
bird4 Daemon Service already stopped. [ FAILED ]
Starting bird4 Service [ ... ]
bird4 Daemon Start Status: [ STARTED ]
root@LEDE:~# /etc/init.d/bird4 start
Starting bird4 Service [ ... ]
bird4 Daemon already started. Status [ RUNNING ]
```

Figure 3.1: Service management for Terminal.

```
root@LEDE:~# /etc/init.d/bird4 status_quiet
bird4: Running
root@LEDE:~# /etc/init.d/bird4 stop_quiet
bird4 - Stopped
root@LEDE:~# /etc/init.d/bird4 stop_quiet
bird4 already stopped
root@LEDE:~# /etc/init.d/bird4 restart_quiet
bird4 already stopped
 ...
bird4 - Started
root@LEDE:~# /etc/init.d/bird4 start_quiet
bird4 already started
```

Figure 3.2: Service management for Web UI.

3.2.4 UCI Configuration improvements

The Unified Configuration Interface (UCI) aims to centralise OpenWrt's settings and it is widely used for almost, if not all, the packages in OpenWrt. UCI allows you to easily, and in a human-readable manner, configure any system following the same scheme, simplifying administration overheads.

However, *bird-uci* Package uses the UCI configuration file in a non-classical manner. Instead of making use of the configuration as it is, the Package only acts as a translator between what the user wants (written in UCI-scheme) and what bird needs to work (c-like configuration file). As stated in section 1.2, the first version of the package successfully manages Bird, but there have been some API changes since Bird v1.4.3 and the integration is not completed yet:

- As part of Bird's v1.4.3 to v1.6.3 API reviewing, some options have required tweaking in order to be compliant to the latest API.
- Most of the UCI improvements are tied to LUCI improvements (see section 3.2.5) in order to enhance UX. For example, BGP Protocol

allows you to execute an action once a number of routes is reached (imported, exported or received). This is shown as a pair of settings in the UI. Previously, each setting was independent, which was a problem as both are optional and hidden for simplicity reasons. By adding the extra option, I have been able to tie both options graphically and make them work as expected. From /etc/config/bird4:

```
config bgp 'bgpAS1'
        option import_trigger '0'
        option export_trigger '0'
        option receive_trigger '0'
        option disabled '0'
        option template 'test123'
        option neighbor_address '192.168.1.100'
        option neighbor_as '1'
```

Listing 3.4: Tied options using UCI (I)

As shown in the snippet, we have three `_trigger '0'` options that state that there is no Limit set in this BGP session. However, if we set one through the UI:

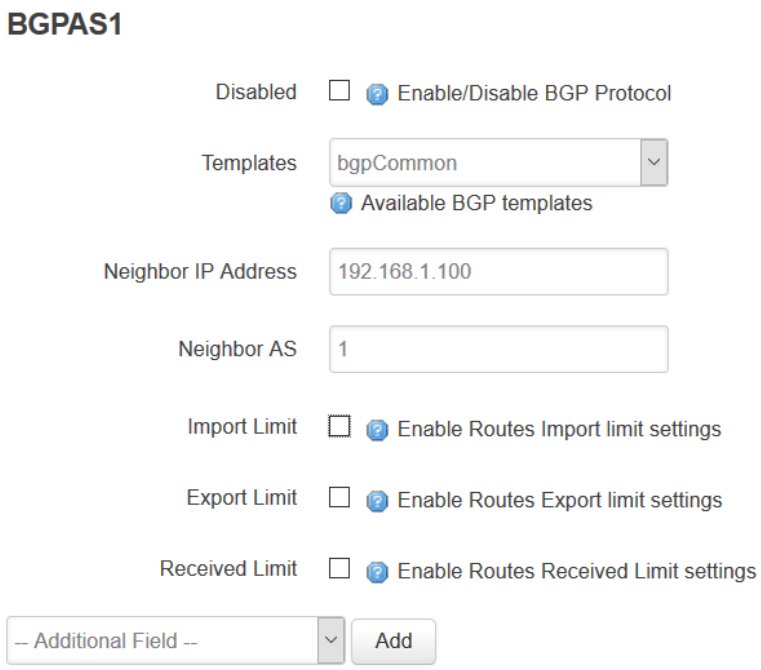

Figure 3.3: Import Limit Trigger not selected.

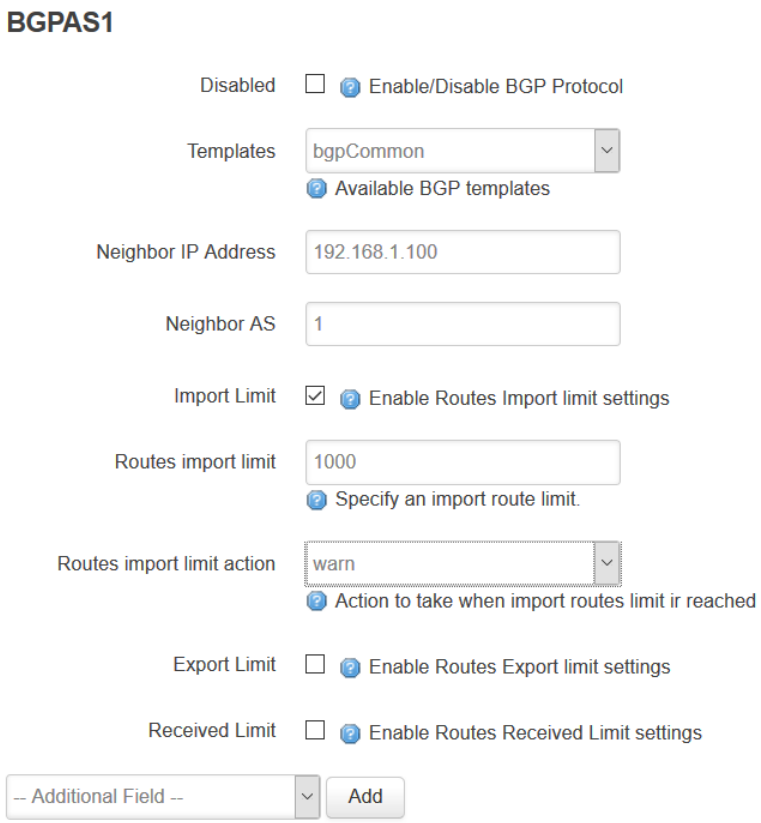

Figure 3.4: Import Limit Trigger selected.

As shown in both figures 3.3 and 3.4 LUCI brings both options together, making it clear to the administrator that these settings must be filled. The UCI result for 3.4 is the following:

```
config bgp 'bgpAS1'
        option import_limit_action 'warn'
        option export_trigger '0'
        option receive_trigger '0'
        option disabled '0'
        option template 'bgpCommon'
        option neighbor_address '192.168.1.100'
        option neighbor_as '1'
        option import_trigger '1'
        option import_limit '1000'
```

Listing 3.5: Tied options using UCI (II)

This tying improvement has been done in the web UI's and not in UCI translation time because, as can be seen in the following code snippet, it is easier to let LUCI configuration management process to

add/remove those attributes automatically on settings save time, than doing some hand-made if statements.

In the following Lua snippet, LUCI creates a UI Flag option (our trigger) which is mandatory (`optional = false`). The other two options (`limit` and `limit_action`) are both optional and dependant on the value of our flag (`depends(import_trigger = "1")`.

```
[...]
import_trigger = sect_templates:option(Flag, "
    import_trigger", "Import       Limit", "Enable Routes
    Import limit settings")
import_trigger.default = 0
import_trigger.rmempty = false
import_trigger.optional = false

import_limit = sect_templates:option(Value, "import_limit"
    , "Routes import   limit", "Specify an import route
    limit.")
import_limit:depends({import_trigger = "1"})
import_limit.rmempty = true

import_limit_action = sect_templates:option(ListValue,
                        "import_limit_action", "Routes
    import limit action", "Action to take when    import
    routes limit ir reached")
import_limit_action:depends({import_trigger = "1"})
import_limit_action:value("warn")
import_limit_action:value("block")
import_limit_action:value("disable")
import_limit_action:value("restart")
import_limit_action.default = "warn"
import_limit_action.rmempty = true
[...]
```

Listing 3.6: LUCI tied options implementation.

3.2.5 LUCI UI improvements

Following previous section's UCI/LUCI example 3.2.4, there have been other UI improvements coupled with changes in the UCI implementation. The following subsections will cover each UI Page to summarise its role and which changes have been done to it. Nevertheless, the last subsection will explain the number of challenges faced during project's development. More detailed images of each page are located in the Appendix D. The ones shown here are for reference.

3.2.5.1 Status Page

New Page allowing an administrator to manage Bird Service. This page shows 3 buttons tied with the init.d functions explained in section 3.2.3 figure 3.2.

The contents of this page are:

- Three buttons to trigger the service management: Start, Stop and Restart in Quiet mode.
- Dynamic Text Box showing Bird service's status. This text will be updated if any service update through the buttons is triggered.

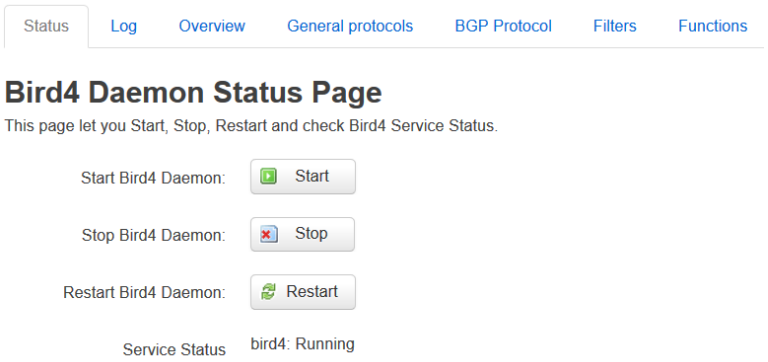

Figure 3.5: Status Page

Although contents of this page are simple and straightforward, the result shown in figure 3.6 is not the desired one. The initial idea was to use a single button for starting and stopping the service, and none for restart. Moreover, the button would be automatically switched between states showing, when started, service's PID. In Appendix D you can see the expected behaviour in the implementation of Privoxy's OpenWrt Package.

The reason behind not doing it is that this simple change would require to change from using LUCI's CBI (Lua) to LUCI2 (JavaScript). Lua's implementation allows you to trigger a number of actions according to a specific

element state, if an action (e.g. button clicked) is triggered or during page's rendering. However, Bird's startup can take few seconds and LUCI has no granular control or a polling mechanism on rendering time to allow this behaviour. To do so, it would be required to force the page to wait (manual OS `sleep` command) which would block page's loading and leave it in an incorrect state depending on Bird's time to start.

Nevertheless, I did do a test implementation (with and without a `sleep` OS call) and, because of the way CBIs work, the button action and rendering action were somehow triggering service calls multiple times, starting and stopping the service in an random way and not refreshing the contents correctly, hence showing wrong status and PIDs.

3.2.5.2 Log Page

New Page showing contents in Bird's configured Log file. This page is automatically refreshed *each second* with the following information:

- Name of the Log file. For reference and easier administration.
- Size of the Log file. Critical information in Bird configurations where Debug information is also enabled. Log file can grow really fast if there are more peers sharing information or if the debug mode is set to log most, or all, the possible events in the system. If the Log file fills the partition where it is located, Bird will automatically shutdown and prevent any start up attempt until this is resolved.

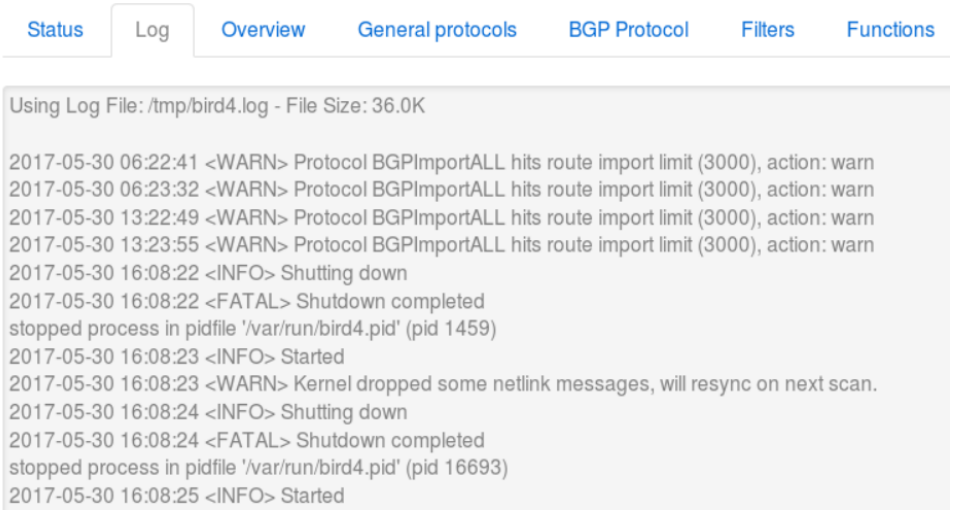

Figure 3.6: Log Page: service restart example

This page has been implemented using mixed capabilities from LUCI and LUCI2. Because of the requirement to show a rolling Log page with a

reasonably regular auto-refresh process, it was necessary to use JavaScript's XHR capabilities embedded in the HTML page itself together with Lua code to read the file and send the text as a PlainText HTML response (object).

Because of the lack of documentation in this area, I spent weeks investigating it by my own using the available package sources in OpenWrt's repositories. However, because what the other packages usually do is to use XHR polls as a communication broker method (e.g. get a specific data from a specific UCI file, apply some filtering/transformation if the data is not in the desired state and, finally, populate specific UI fields available in the HTML page) it was not clear what exactly I had to do as I only needed to: read last 30 lines of a file (Bird's Log File) and populate the Text View (Plain text data).

Finally, after spending some days trying to contact experienced LEDE/ OpenWrt developers using community's contact channels, I could have few conversations with one of the main contributors of LEDE, OpenWrt and also main author of LUCI/LUCI2 `jow-`[1], who gave me some hints, examples of LUCI2 pages and also helped me debugging some issues I had during the development that allowed me to finish both Log and Filters/Functions pages.

Log File Source key highlights:

```
if luci.http.formvalue("refresh") then
[...]
luci.http.prepare_content("text/plain")
```

Listing 3.7: Lua - Log Code (I)

- Only execute the polling function (update log information) if *refresh* is passed as parameter.
- Send the outputs back as plain text.

```
lf = sys.exec("tail -n30 " .. log_file):gsub("\r\n?", "\n")
[...]
luci.http.write("Using Log File: " .. log_file .. " - File Size
    : " .. log_size .. "\n" .. lf)
```

Listing 3.8: Lua - Log Code (II)

- Get the last 30 lines of the file `log_file`.
- Return this text as HTTP Response.

[1] Jo-Philip Wich: LEDE-Project and OpenWrt core developer. Github Profile.

```
<script type="text/javascript">//<![CDATA[
    // Refresh page each second. Use "refresh=1" as trigger.
    XHR.poll(1, '<%=url('admin/network/bird4/log')%>', {
        refresh: 1 },       function(xhrInstance) {
         var area = document.getElementById('log')
         area.value = xhrInstance.responseText;
    });
//]]></script>
```

Listing 3.9: JavaScript - Log Code (III)

- Embed JavaScript code in the HTML Page.
- Execute an XHR Poll periodically (1 second) against the URL `admin/network/bird4/log` (itself) with parameter `refresh=0`. In this instance, `refresh` is a *JSON object* parsed to the backend script in Lua that will trigger our File reading and return the last 30 lines of log information (a really simple case). However, this mechanism is designed to receive complex JSON object structures that would include all the data required for the backend scripts to execute some transformations into UCI or into system calls and, finally, to return the outputs that would refresh the data in the UI.
- The internal function `function(xhrInstance)...` gets the HTML code below with ID *log* (our HTML Log Text Box) and injects the Text Object received by our XHR instance.

```
<textarea readonly="readonly" style="width: 100%" wrap="on"
    rows="32" id="log"><%=lf:pcdata()%></textarea>
```

Listing 3.10: HTML - Log Code (IV)

This is all our page's real code. We have a simple readonly Text Box (32 lines) with ID *log* and instantiates the variable `lf` from Lua's code.

3.2.5.3 Overview Page

The Overview Page shows base Bird Service settings:

- File used to store the UCI translated Bird.conf file.
- Definition of any Routing Table that will be used in the Protocols.
- Router's ID (some protocols can overwrite it for their own purposes)
- Debug and Log settings and where to store them. This option is currently configured to use a single file for logging. However, Bird allows

to set any number of different instances with any set of options. Although this would be the desired state, given the resources of most of LEDE/OpenWrt's nodes, the log partition gets filled fast enough with a single file. It is important to know that if the Log Partition is filled and Bird is unable to access it, Bird service will be stopped and blocked until the issue is solved.

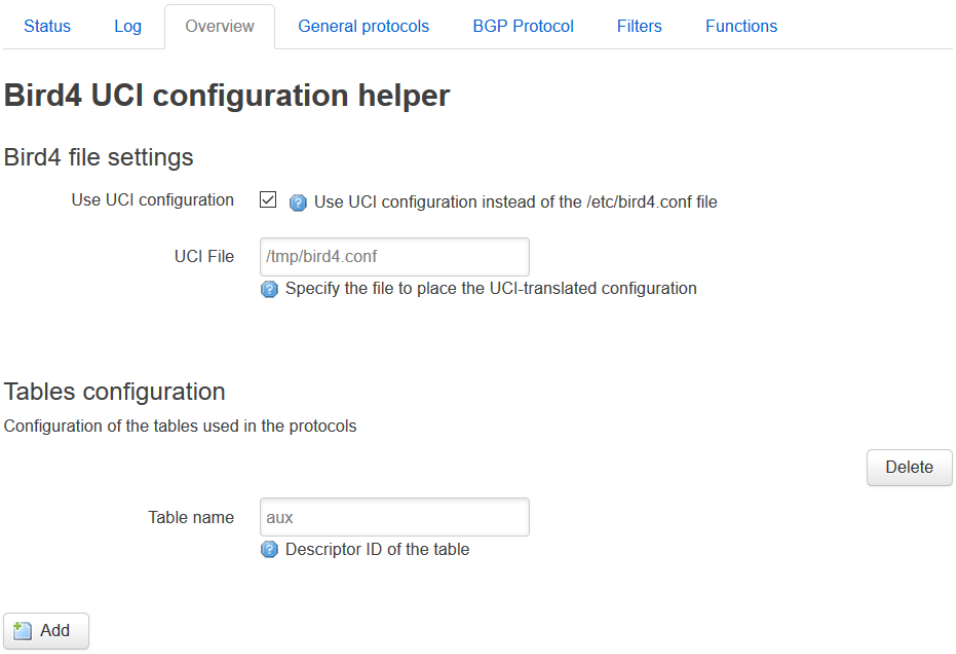

Figure 3.7: Overview Page

Changes and Improvements: This section has not been severely modified. The only big change has been the Log&Debug settings, because there were some issues using *All* together with other options. By definition, *All* implicitly includes the other options and should not be passed to the configuration.

3.2.5.4 General Protocols Page

The General Protocols Page shows the configuration of some of the Bird support protocols described in section 1.1.2.2.

- Kernel Protocol (1 mandatory as base Routing Table talking to the OS. Any extra kernel protocol is optional).
- Device Protocol: optional.

- Pipe Protocol: optional.
- Direct Protocol: optional.
- Static Protocol: optional.
- Routes: Routes are just the representation of entries in a Static Protocol. These are optional and tied to a single Static Protocol.

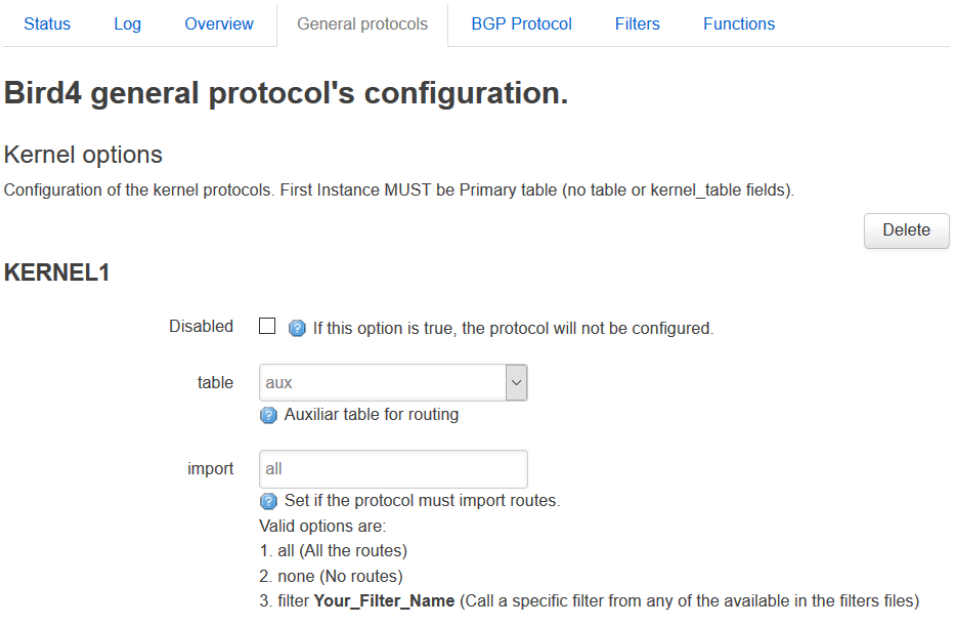

Figure 3.8: General Protocols Page

Changes and Improvements: The main change in this page has been the recovery of both PIPE and Direct Protocols, disabled during the original Project because they were not relevant (there was no coexistence of protocols in Bird). These two protocols are now required as they allow to communicate routing tables (PIPE) and to mark the routes as device ones on specific *local* interfaces (DEVICE) to feed the kernel table.

3.2.5.5 BGP Protocol Page

The BGP Protocol Page is the most important for this project as it allows us to configure the main protocol used in Guifi.net. Together with OSPF, BGP is the most complex protocol to configure (and translate) on Bird. With a big number of different *options* as well as another big number of *attributes* to configure, the automation of this protocol has been the main goal of this Package.

- BGP Templates: allow to set a number of common options repeated among a number of different BGP Sessions. Their use simplifies configuration's maintainability as well as improves readability of the file. Templates are optional and you can configure as many as required.
- BGP Instances: Instances are the definition of each BGP session to be created. They can feed from Templates or set all the options manually. Instance options will always prevail over template ones. Instances are optional and you can configure as many as required.

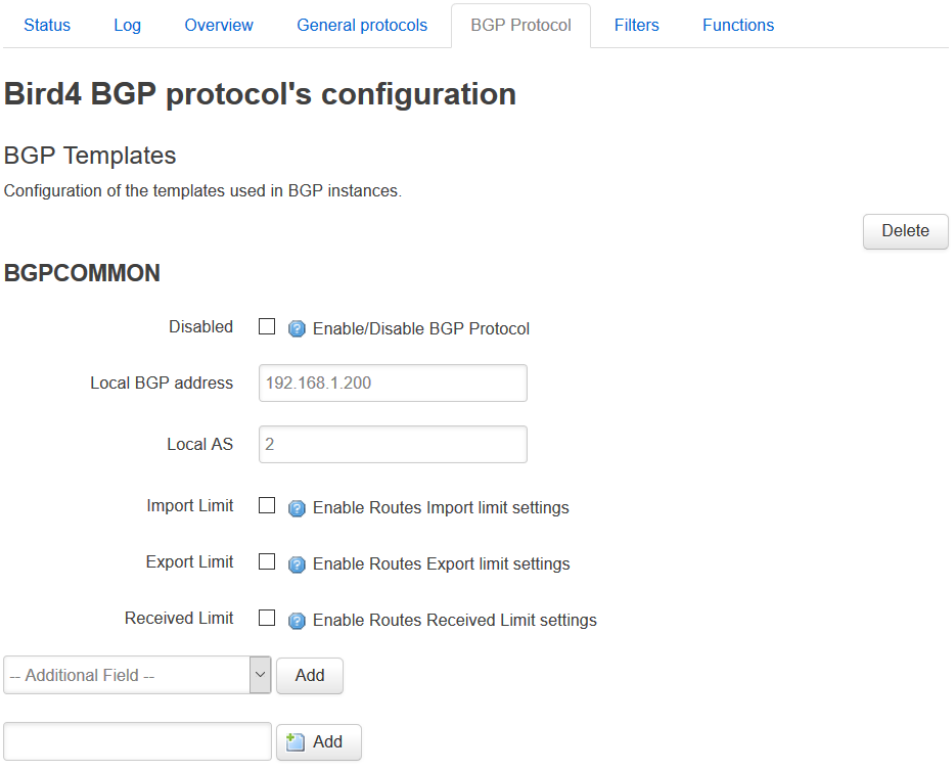

Figure 3.9: BGP Protocol Page

Changes and Improvements: BGP Templates and Instances options have been re-sorted in order to follow a logical order when configuring them. Moreover, all the shown options have been reviewed and re-targeted as optional or mandatory and also, as explained in previous section 3.2.4, some of the options required to configure route sharing limits were not displayed together, which was prone to misconfiguration.

3.2.5.6 Filters & Functions Page

Two new Pages empowering Filters&Functions file edition without requiring terminal-based processes. This feature is one of this project's main goals because it completely removes the need for administrators to go to command line, which is a barrier for some non-expert administrators. The page's composition is the following (for both functions and filters):

- Disclaimer: informative message about how to rename new files and how to visualise them after creation.
- Recognised files dropdown menu: List of the files detected in the correct folder and available for administrators to use. Supported folders are:
  - Filters: `/etc/bird{4|6}/filters/*`
  - Functions: `/etc/bird{4|6}/functions/*`
- Load Button: This button will select the file shown in the Text Area and populate the *read-only* Text Box stating which file is actually being edited. Internally, this button will set a Lock variable in the filesystem containing target file name, in order to prevent editing files.
- Editing File read-only Text Box: This read-only field is populated with the contents selected in the dropdown menu when the *Load* Button is clicked. Unless this field is populated with a full path to a file, any contents in the TextArea will dropped (`Submit` button will just refresh the contents of the page).
- Text Area Box: This editable 30 lines field will be populated with the contents of the file Loaded (or empty if it is a new file). Any contents in this field will be stored in the selected file, or dropped otherwise.
- Submit Button: This button will trigger the save mechanism. If a file has been correctly loaded, it will store the contents in the Text Area to the target file. Protocol's configuration can be stored before applying it by clicking the *Save* button instead of the *Save & Apply* on. However, this process uses direct system calls and is not reversible, hence it is named *Submit*.

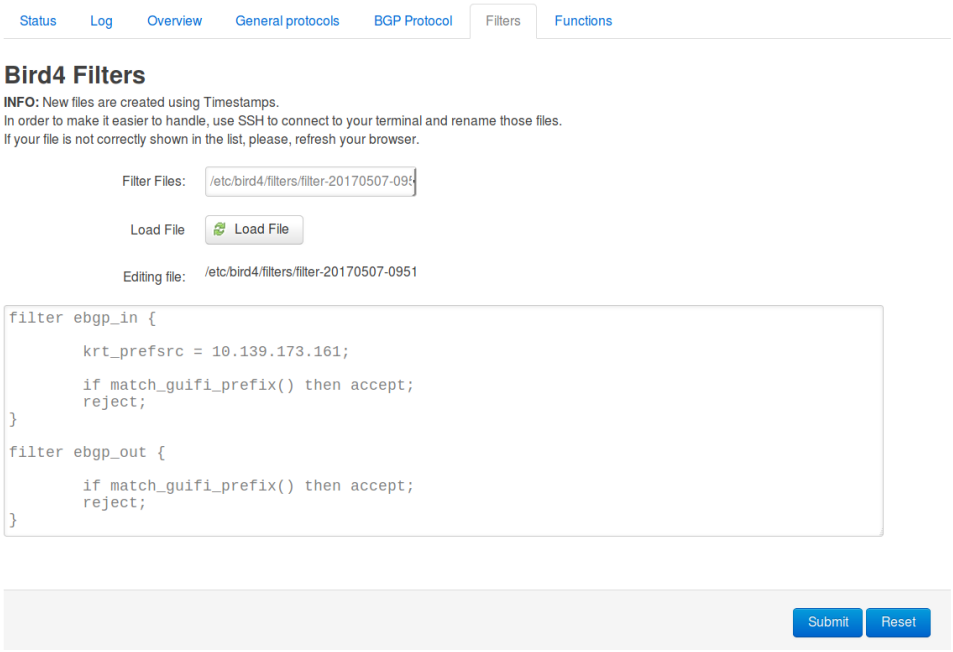

Figure 3.10: Filters Page

Figure 3.11: Functions Page

These two pages have been implemented using LUCI capabilities only (Lua Model) plus a simple HTML template, with some Lua embedded to inject text and properties, in order to enhance the Text Area: set to 30 lines, enlarge text's size and make use of `Courier New` font in order to make it more *console-alike* highlighting that it is code and not plain text:

```
<%+cbi/valueheader%>
    <textarea class="cbi-input-textarea" <% if not self.size
        then %>          style="width: 100%; font: normal 11pt
        'Courier New'"<% else %> cols="<%=self.size%>"<% end
        %> data-update="change"<%= attr("name", cbid) ..
        attr("id",    cbid) .. ifattr(self.rows, "rows") ..
        ifattr(self.wrap, "wrap")  ..
        ifattr(self.readonly, "readonly") %>>
    <%-=pcdata(self:cfgvalue(section))-%>
    </textarea>
<%+cbi/valuefooter%>
```

Listing 3.11: Enhanced Edit Text Box Template.

- `<%+cbi/valueheader%>` and `<%+cbi/valuefooter%>`: Lua snippets acting as delimiters for top and bottom cbi's contents.
- `<%[...]%>`: Lua code injection.
- `<%=[...]%>` Lua variable injection.

Caveats

The biggest challenge has been to be able to select different files to store our filters and functions. This Page has required a couple of weeks of development because of its functionality, the already mentioned file selection flexibility and what is required from the Text Area is not something easy to do with current LUCI documentation.

In the same way as the Log Page 3.2.5.2, current documentation refers to the most common uses but pages where no UCI configuration is involved are just briefly mentioned. Some of the challenges found are:

- `SimpleForm` Page format is required due the lack of UCI settings to modify. However, the documentation explaining which is SimpleForm's API, differences between this solution and `Map`[2] or possibilities using SimpleForm are not detailed enough.
- Local variables are not reliable. Because pages can be re-rendered depending on the event triggered, your stored variables could be overwritten with new values by these rendering functions. An example of this issue was my attempt use the Read-only field to store the path. Because `cfgvalue` (render) function is triggered several times in an uncontrolled manner (we have no control of it) your variable may have been overwritten several times. Therefore, if you switch the selected file, you were storing your data in the wrong file.

[2] Map is LUCI's alternative format page to define UI by *mapping* fields into UCI properties

- UI values can only be gathered or saved during specific events (e.g. on render or save functions).
- Unable to refresh the file list on a render function. It has bee required to do it as part of Page's creation, therefore, needing to refresh the page to update the file list. This is *annoying* if you have created a new file, because it may not be listed until you refresh the page. This has been documented as a known issue and will be resolved by using LUCI2 in a future release.
- Trying to use nested events usually cause data misconfiguration. An example of that was my first attempt to use the Load File Button to lock the target file. Current process to edit files is:
  - Select a file
  - Press Load Button
  - The file is Loaded (write Lua function): filename in the readonly field and contents in the TextArea.
  - Edit File contents
  - Press Submit (write Lua function)

  Because the Submit button also triggers a write Lua function, it was, somehow, also activating Load Button's one. This was a big issue if, before saving the file, the administrator selects a different file from the dropdown menu. This issue was finally solved by adding an extra level of complexity on both write Lua functions as well as storing our target path in a file in the system (`/etc/bird{4|6}/filter_lock` and `/etc/bird{4|6}/ffunction_lock`).

3.2.6 Align documentation and upgrade to Markdown

Documentation has been one of the biggest disappointments while working with LUCI/LUCI2. To use a project or a technology and find that there is not enough detail, if any, about what you can do and how, is really disappointing and causes frustration while trying to figure it out. Therefore, I have put a lot of efforts on updating all the documentation available in the Package, as well as added an extra repository with other data that supports it, without bloating it (this Package will be Pulled to the official repository, so it is desirable to have the minimum contents in this one, and everything else linked). Moreover, because the old documentation was plain text, it was unpleasant to read. In order to facilitate Package's configuration and daily usage, I have upgraded old documents to MarkDown.

The main documentation is now separated between:

- UCI: UCI definition and examples and any terminal-based function or command required to use the Package without UI.
- LUCI: Web UI Pages' composition, field description, default values and any known issue.
- README.md: Development notes about how to build and enhance the Package and known issues on current Package version.
- Changelog.md: Exhaustive list of enhancements added as part of this project (v0.3)
- COPYING.md: Markdown version of GPLv3.0
- AUTHORS.md: List of the contributors.

Documentation repository with extra information:

- Repository-Contents.md: This file includes the tree structure of Packages repository as well as a description of what each file is.
- Manual-Procedures.md: List of manual procedures that have not been automated in this Package's version and may require an administrator to do them on command line. Currently, the use of secondary Routing Tables with custom table IDs has to be done following the procedure shown.
- TODO.md: Unsorted and non-prioritised list of tasks pending for future releases.

3.3 Bird Daemon uBus integration investigation

uBus theoretical investigation is driven by the need of continuously monitor the status of our protocols and the critical historical data that it could bring any administrator managing any size and complexity networks. Current Bird's implementation provides a low consumption and powerful routing solution but its monitoring capabilities lay in a CLI mimicking other well-known plain text-based tools as Quagga, Cisco or Mikrotik's Consoles.

OpenWrt/LEDE's Bird plain-text CLI tools are `birdc{4|6}` and `birdcl{4|6}`. Birdcl is the lightweight version of birdc. It does not support some commands (e.g. service management) or command history. Both tools provide the administrators an API [5] in order to manage the live system and gather information from it using UNIX Standard domain sockets (`bird.ctl`). As already mentioned, this CLI provides human readable plain-text information on specific queries, thus most of its outputs are not suitable or automatizable for scripting. Some example of Birdc's outputs are:

```
root@LEDE-MXN1:/etc/config# birdc4
BIRD 1.6.3 ready.
bird> show status
BIRD 1.6.3
Router ID is 10.139.173.161
Current server time is 2017-06-03 19:06:59
Last reboot on 2017-06-03 00:54:23
Last reconfiguration on 2017-06-03 00:54:23
Daemon is up and running
bird> show protocols
name     proto    table    state  since       info
kernel1  Kernel   aux      up     00:54:22
static1  Static   aux      up     00:54:22
device1  Device   master   up     00:54:22
BCNRamblaPobleNou BGP      aux      up     00:54:27
   Established
UOCBGPMesh BGP       aux       down   00:54:22
bird> exit
root@LEDE-MXN1:/etc/config#
```

Listing 3.12: Birdc Console mode.

This example shows two output examples of Bird CLI working in Console mode. The administrator enters into a Bird-protected-mode in order to visualize the required information. The tool also includes a helper function (command: ?) according to the specific command being visualised:

```
bird> ?
add roa ...                                    Add ROA record
configure ...                                  Reload
    configuration
debug ...                                      Control
    protocol debugging via BIRD logs
delete roa ...                                 Delete ROA
    record
disable <protocol> | "<pattern>" | all         Disable protocol
down                                           Shut the daemon
    down
dump ...                                       Dump debugging
    information
echo ...                                       Control echoing
    of log messages
enable <protocol> | "<pattern>" | all          Enable protocol
eval <expr>                                    Evaluate an
    expression
exit                                           Exit the client
flush roa [table <name>]                       Removes all
    dynamic ROA records
help                                           Description of
    the help system
mrtdump ...                                    Control
    protocol debugging via MRTdump files
```

```
quit                                                    Quit the client
reload <protocol> | "<pattern>" | all                   Reload protocol
restart <protocol> | "<pattern>" | all                  Restart protocol
restrict                                                Restrict
    current CLI session to safe commands
show ...                                                Show status
    information
bird>
```

Listing 3.13: Help on Birdc Console mode.

```
bird> show ?
show bfd ...                                            Show
    information about BFD protocol
show interfaces                                         Show network
    interfaces
show memory                                             Show memory
    usage
show ospf ...                                           Show
    information about OSPF protocol
show protocols [<protocol> | "<pattern>"]               Show routing
    protocols
show rip ...                                            Show
    information about RIP protocol
show roa ...                                            Show ROA table
show route ...                                          Show routing
    table
show static [<name>]                                    Show details of
    static protocol
show status                                             Show router
    status
show symbols ...                                        Show all known
    symbolic names
bird>
```

Listing 3.14: Help on `show` level of Birdc Console mode.

Finally, the following code snippet shows our target information: BGP Session live data.

```
root@LEDE-MXN1:/etc/config# birdc4 show protocols all
BIRD 1.6.3 ready.
name     proto    table    state  since       info
BCNRamblaPobleNou BGP      aux      up     00:54:28
    Established
  Preference:     100
  Input filter:   ebgp_in
  Output filter:  ebgp_out
  Routes:         3023 imported, 0 exported, 3023 preferred
  Route change stats:     received   rejected   filtered
      ignored   accepted
    Import updates:          108200          0          0
                13     108187
    Import withdraws:         25629          0        ---
                 6      25623
```

```
    Export updates:          108187     108187          0
                ---          0
    Export withdraws:          25623        ---        ---
                ---          0
  BGP state:          Established
    Neighbor address: 172.25.35.25
    Neighbor AS:      59361
    Neighbor ID:      10.90.224.65
    Neighbor caps:    refresh AS4
    Session:          external AS4
    Source address:   172.25.35.26
    Hold timer:       164/180
    Keepalive timer:  28/60
```

Listing 3.15: Birdc Simple `Show Protocols all`. Truncated to show BGP.

As shown in the snippet above, we can query Bird to get base BGP information: name, session status, path preference, neighbour's ID, etc. Or live data as Keepalive time left or number of routes imported, exported, blocked by our filters, etc. This information would be valuable for administrators if it was shown in the UI as a chart or rolling text field, working both as live status checker and also as a health screener.

Understanding that our main goal is to, somehow, integrate monitoring capabilities into the Package, the next step is to see which tools would support us doing it.

3.3.1 LUCI2 new architecture: WebUI-uBus-RPCd-Service

LUCI2 introductory information is available in section 1.1.1.3, see it for reference of the acronyms.

LUCI2 main components are:

- Client-Side Browser: HTML/CSS/JS provided by service (e.g. luci-app-bird{4|6}) is executed on client's side.
- uBus Service: Client-Server-based query broker to communicate different system services.
- uHTTPd Server: *Server-side* HTTP server to satisfy queries.
- RPC Daemon: HTTP backend server acting as a query broker for packages not integrated with uBus service but providing an API to use it.

3.3.1.1 Example overview

Using Bird4, luci-app-bird4 and a fictional *RPC-uBus Bird4 Service* as an example, this process' architecture is the following:

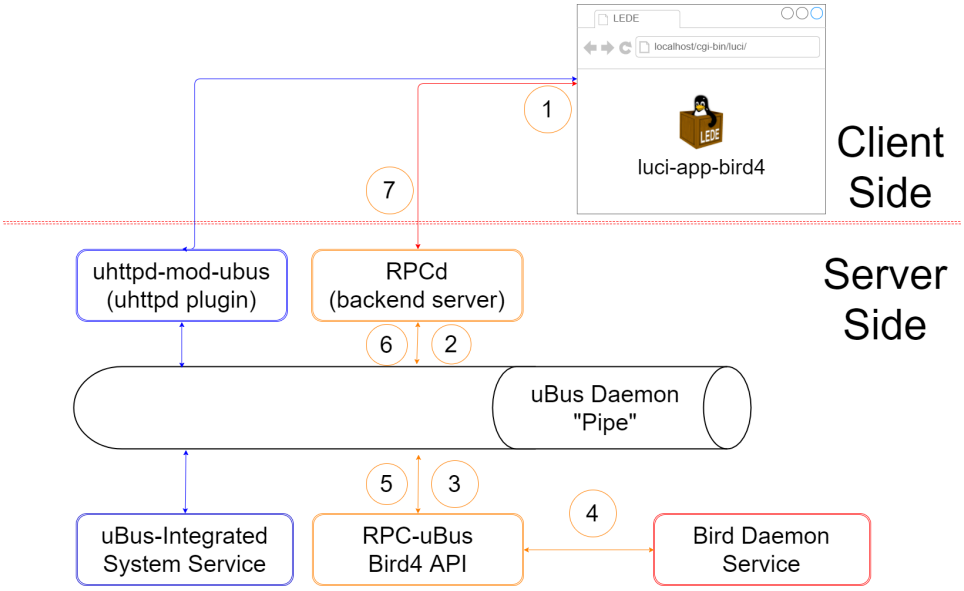

Figure 3.12: LUCI2 Communication architecture for Bird4 Service

Taking in consideration that Bird has no integration with uBus, we need to connect to it through an RPCd service. The steps followed for a Client to interact with Bird Service are shown in figure 3.11:

1. Client's browser wants to query Bird service to populate a number of fields the UI (luci-app-bird4). These queries are done using JavaScript using RPCd's API.

2. RPCd backend server passes Bird's UI query to uBus, who will check if Bird has an API registered in its namespace.

3. If uBus finds the given API, it checks the specific call sent and it is executed.

4. Bird4's API executes shell script commands to directly query Bird4 and returns any outputs generated.

5. The API will need to handle Bird's outputs and to encapsulate them into JSON objects. After doing that, any generated output is sent back to uBus.

6. uBus sends any object received to the RPCd server.

7. RPCd Server sends this object back to the client and it refreshes any targeted component with this new data.

3.3.1.2 Server-side's implementation details

Following the bullets above, the first thing required is to create our *Shell* API under the folder `/usr/libexec/rpcd` for uBus to be able to add it. This API needs to provide its own description (`list`) and functions (`call`) required to communicate your service through JSON objects with the Client. This example shows a Bird service management implementation using four RPC calls (see figure 3.11: step number 3).

```
#!/bin/sh

case "$1" in
    list)
        echo '{ "status": { }, "start":{ }, "stop":{ } }'
    ;;
    call)
        case "$2" in
            status)
                status=$(/etc/init.d/bird4 status_quiet)
                echo "{ \"status\" : \"$status\" }"
            ;;
            start)
                start=$(/etc/init.d/bird4 start_quiet)
                echo "{ \"start\" : \"$start\" }"
            ;;
            stop)
                stop=$(/etc/init.d/bird4 stop_quiet)
                echo "{ \"stop\" : \"$stop\" }"
            ;;
        esac
    ;;
esac
```

Listing 3.16: RPCd Bird4 Service Management.

On a new system, where no other services have been registered on RPCd, if we check what is registered on uBus, we will see only system services:

```
root@LEDE:/usr/libexec/rpcd# ls -la
drwxr-xr-x    2 root     root          4096 Jun  4 10:29 .
drwxr-xr-x    3 root     root          4096 Jun  4 10:29 ..
root@LEDE:/usr/libexec/rpcd# ubus list
dhcp
log
network
network.device
network.interface
network.interface.lan
network.interface.loopback
network.wireless
service
session
system
uci
```

Figure 3.13: uBus registered services (I)

We can see in the figure that no other services are providing an API using RPCd folder (it is empty) and we can only find services like network, UCI or system itself providing uBus integration.

After adding our code shown in the snippet 3.16 in RPCd's folder and restarting RPCd service, we can now start querying it:

```
root@LEDE:/usr/libexec/rpcd# /etc/init.d/rpcd restart
root@LEDE:/usr/libexec/rpcd# ls
bird4
root@LEDE:/usr/libexec/rpcd# ubus list
bird4
dhcp
log
network
```

Figure 3.14: uBus registered services (II)

The truncated output shows that uBus is now able to call bird functions. If we query uBus for specific and verbose (`-v`) information about the provided API this would be the output:

```
root@LEDE:/usr/libexec/rpcd# ubus -v list bird4
'bird4' @6ab40983
        "status":{}
        "start":{}
        "stop":{}
```

Figure 3.15: uBus registered services (III)

As show in the figure 3.15, this Bird RPCd API provides 3 functions requiring no parameters (an example of a parametrized API will be shown as an example for Client-side's example).

Now we can test this API, which is interacting (in an over-simplified manner) with Bird:

```
root@LEDE:/usr/libexec/rpcd# ubus -S call bird4 status
{"status":"bird4: Running"}
root@LEDE:/usr/libexec/rpcd# ubus -S call bird4 start
{"start":"bird4 already started"}
root@LEDE:/usr/libexec/rpcd# ubus -S call bird4 stop
{"stop":"bird4 - Stopped"}
root@LEDE:/usr/libexec/rpcd# ubus -S call bird4 start
{"start":"bird4 - Started"}
```

Figure 3.16: uBus registered services (IV)

As shown in the code snippet 3.16, we are executing `*_quiet` functions returning plain text responses from Bird service's management script. This output is wrapped in a JSON object and uBus will send this output back to the Client.

With these steps, we have all the necessary elements to make RPCd communicate from server's side.

3.3.1.3 Client-side implementation details

This client-side example is different from the server-side one and has not been implemented in the development environment. It adds extra complexity in order to be able to tie it with the next subsection 3.3.1.4, which will cover the required changes in Bird itself in order to support LUCI2.

Code snippet 3.17 could be part of a future version of luci-app-bird{4|6} or a new package adding this functionality separately as a plugin (e.g. `bird-mod-ubus`). This example explores the possibility of a future BGP monitoring page (or section in BGP Protocols Page) giving administrators live information about their sessions.

```
Luci2().ui.view.extend({
[...]
    getPeers  : Luci2().rpc.declare({
        object: 'bird4',
        method: 'getBGPNeighbour',
        params: '[ "BGPSession" : "' + targetBGPSession + '"
            ]',
        expect: { BGPNeighbourID: {} }
    }),
[...]
    getNeighbourStatus : function() {
        var targetBGPSession = "BCNPeerToTGN";
        self.getPeers(targetBGPSession).then(
            function(JSONOutput)
        {
            // Parse any JSON Data received into a pre-defined
                and sanitised format
            var bgpData = self.parseJSON(JSONOutput);

            // Update the UI according to the data received
            self.updateUIData(bgpData);
        });
    },
[...]
    execute: function () {
        [...]
        // Force RPC call to be called each second
        self.repeat(self.getNeighbourStatus, 1000);
        [...]
    }
}
```

Listing 3.17: RPC Call Script.

This possible JavaScript code defines the API used in order to communicate with the backend server (`declare`) calling bird4's RPCd and querying the `getBGPNeighbour` function. This function would, somehow, query `birdc4` (e.g. `birdc4 show protocols "BCNPeerToTGN" -json '{ 'bgpPeerID' : { } }'`) or any other tool designed for Bird to answer live data queries with JSON-formatted responses instead of human-readable ones. This snippet follows an oversimplified approach where we only ask Bird for a specific data and we use only a little portion of it (BGP Neighbour AS ID). However, I mention both `parseJSON` and `updateUIData` on purpose foreseeing that this RPC function would really return all BGP's data and populate the UI with it in a reasonable time lapse (snippet shows 1 second repetition).

3.3.1.4 Bird Service's required changes

Previous sections have explained how we could implement an integration of Bird in uBus presupposing that it is already providing JSON-formatted

outputs. This section focuses on Bird's implementation and which changes would be required to support this new output format.

For this investigation, I have reviewed three possible approaches:

1. **Generate a new OpenWrt package to parse Bird's outputs from human-readable to JSON**:
   The first approach is to use Bird Remote Control's command outputs in the same human-readable format (see example shown in the snippet 3.15). This would require specific parsing functions for each possible output and variation (e.g. while established BGP sessions show detailed route data, non-established ones show no information). This solution would have no impact in Bird's official code as JSON parsing would occur in a separate software.

   This option is totally dismissed as it adds two levels of complexity by forcing the new package (or functionality) to be parsing messages and being completely coupled with a non-standard and unreliable communication *API*. Therefore, any change in Bird's human-readable outputs or data's structure shown would cause our package to stop working and to require a parser update to conform to the new API. All this process would take some time and could be challenging according to which non-standard part of the output has changed, affecting any users on OpenWrt/LEDE systems.

2. **Modify Bird Remote Control's implementation to parse from human-readable to JSON**:
   This second approach moves the JSON parsing closer to where it is originated. Birdc and Birdcl tools just `echo`es the data according to the CLI command executed. However, unlike the first approach, because the outputs are parsed on Bird's side, their community and developers would manage any required integration change in order to support the latest version of the Daemon. Hence, any external administrator or developer would be able to rely in a standardised API with JSON-formatted outputs.

   This option has also been dismissed because, although the parsing method occurs closer to the data, there is still a parsing method from Bird's human-readable output to JSON. Moreover, any external developer requiring live data would be forced to:

   - Forced to use or, at least, to install a tool not fit for purpose (CLI).
   - Use a socket to Bird which is not exclusive to his processes.
   - Use a socket to Bird which its *private* API could be accessible from the CLI tool.

- In order to get some data, it could be required to use CLI-based commands.

3. **Modify Bird's code to switch from human-readable to JSON outputs**:
   This third approach moves the problem even further. The solution proposed is to modify Bird's code in order to add extra functions that would, together with the human-readable ones, present protocols' live status in JSON format. This means that we have to review Bird's code in order to find where each output is generated and create an extra function (or add a parameter) to send the data in JSON's format.

   This proposal requires a high number of changes in the official solution as well as tests to prove that all generated outputs are RFC compliant [6] and that this changes are not causing performance issues in the Daemon. Following this solution, it could open three new options to solve our current issues:

   - Continue using Bird Remote Control but receiving JSON outputs instead of human-readable ones (may require improvements in birdc/birdcl tools). This option has some of the 2nd proposal issues.
   - Create a new bird *semi-official* (while not accepted) utility following the same approach of birdc (UNIX socket to connect to Bird) only being used for monitoring purposes. This solution would require its own UNIX socket or to use the same as Bird Remote Control, which could create a conflict.
   - Create an integration package or specific Bird implementation with uBus Socket. This would be the desired solution as the communication and integration would be complete and Bird could be announced as uBus *compliant* service. Moreover, on the one hand there would be no conflicts with Bird Remote Control but, on the other hand, there could be some performance issues derivatives from this (currently unknown) socket communication implementation.

   Finally, as an example of Bird's current output printing implementation, the following snippet shows part of BGP's information presentation [7]:

```
static void
bgp_show_proto_info(struct proto *P)
{
    struct bgp_proto *p = (struct bgp_proto *) P;
    struct bgp_conn *c = p->conn;

    proto_show_basic_info(P);
```

```
    cli_msg(-1006, "  BGP state:          %s",
        bgp_state_dsc(p));
    cli_msg(-1006, "    Neighbor address: %I%J",
        p->cf->remote_ip, p->cf->iface);
    cli_msg(-1006, "    Neighbor AS:      %u", p->remote_as);
[...]
}
```

Listing 3.18: bgp_show_proto_info function in BGP.c.

This code snippet 3.18 shows the first lines of BGP's information, available even if the session fails. The main idea from this snippet is that all the different protocols available in Bird are using C data structures in a non-obscured manner, with some of the almost identical to JSON: E.g. `p->rr_client ? " route-reflector" : ""`,
Next code snippet shows BGP Connection structure (one of the different BGP structures):

```
struct bgp_conn {
  struct bgp_proto *bgp;
  struct birdsock *sk;
  uint state;               /* State of connection state
      machine */
  struct timer *connect_retry_timer;
  struct timer *hold_timer;
  struct timer *keepalive_timer;
  struct event *tx_ev;
  int packets_to_send;          /* Bitmap of packet types to
      be sent */
  int notify_code, notify_subcode, notify_size;
  byte *notify_data;
  u32 advertised_as;            /* Temporary value for AS
      number received */
  int start_state;          /* protocol start_state snapshot
      when connection established */
  u8 peer_refresh_support;      /* Peer supports route refresh
      [RFC2918] */
  u8 peer_as4_support;          /* Peer supports 4B AS numbers
      [RFC4893] */
  u8 peer_add_path;         /* Peer supports ADD-PATH
      [RFC7911] */
  u8 peer_enhanced_refresh_support; /* Peer supports enhanced
      refresh [RFC7313] */
  u8 peer_gr_aware;
  u8 peer_gr_able;
  u16 peer_gr_time;
  u8 peer_gr_flags;
  u8 peer_gr_aflags;
  u8 peer_ext_messages_support;     /* Peer supports extended
      message length [draft] */
  unsigned hold_time, keepalive_time;   /* Times calculated
      from my and neighbor's requirements */
```

```
};
```

Listing 3.19: Bird's BGP Protocol Session C Structure.

Therefore, we could make use of some available implementations to automate their parsing (e.g. JSON-C) and make the integration process much easier and facilitate its maintainability.

3.3.1.5 Analysis conclusions

The conclusion I reached from this theoretical analysis about the feasibility of integrating uBus on Bird is that, although it would require a lot of efforts until we can viably get JSON-formatted live data from Bird, it is possible to achieve it if the third approach I have presented is followed. This approach will require Bird community and developers' support in order to implement it in the official stream and this solution must follow JSON standards. After achieving that, we can start defining how we want current LUCI or LUCI2 implementation's to adopt this new API and how much it can be delivered in an initial phase (from simple auto-refreshed text fields to historical health charts, simulated connectivity maps and other relevant data).

Joining everything together, these requirements would easily become a feasible future Open Source Software MSc. project. This project's requirements would be: to have network's knowledge or willingness to learn about it; to have some level of C, Shell and JavaScript expertise; to be prepared to agree project's requirements, ideas and approaches with Bird's community and developers; and also, patience to deal with the lack of LUCI2, uBus and RPCd documentation in OpenWrt/LEDE, being a bonus if, together with the community, the student helps to fill any knowledge gaps and to feed them back to them.

Chapter 4

Tests Results

4.1 Package Testing

Testing an integration/translation Package, and this one specifically, is a rather complex task to evaluate as Bird configuration files are modular and desired settings can be achieved in different ways. Even more, although a *it works/it does not work* policy could be accepted, it does not mean that there are no other possible implementations that could work in a better way. For example, filters and functions can be either written in the *.conf* file or included using `%include` mechanism, being the second one a better approach as it enhances code readability as well as it avoids bloating the configuration file unnecessarily.

With this introduction in mind, the following sections will explain how this package has been tested following Bird's configuration base requirements and service behaviour and some *future work* ideas to achieve automatic and unit tests.

4.1.1 Configuration of the *translation* tests (future work)

To perform configuration integrity tests in current package, it is required to repeat the execution of `/etc/bird{4|6}/init.d/bird4 restart` in order to trigger the UCI-bird.conf translation from a target UCI file. The code to do this translation has been refactored in an functional manner to allow future unit tests or, at least, make it easier to integrate in an automated test framework or process. For example, an automated CI/CD build process could build an update of the package, push it into a test node, execute the translation process and compare it against the previous (or a stable) version, as well as check its correctness by querying Bird's status.

4.1.1.1 Reviewing v0.2 against v0.3

Testing the outputs from the old and new packages, and taking into account that there are some manual changes in the old one, the following example is configured as follows:

- Router IDs follow node's IP Address.
- Kernel, Device and Static Protocols have been set by default.
- A Static Route has been added (identical).
- BGP Template and Instance have been configured following v0.2 scheme with matching settings to avoid Bird failures.
- BGP Instance AS and Neighbours are dummy values.
- A BGP Filter called "all_ok" (accept all routes) has been added using each version's process.

In the new package, we have instantaneous configuration correctness feedback as we can check Bird's status in the Status Page. In the old package, after executing `/etc/bird{4|6}/init.d/bird4 start`, Bird will fail and it is required to move the Filter "all_ok" to the top of the document. Bird will start correctly after this modification.

After checking that both daemons are running, we can then perform a *diff* between the configuration files and look for any noticeable difference.

```
3,9d2
<     #Filter filter1:
<     filter all_ok
<     {
<         accept "all ok";
<     }
<
13c6
<     router id 192.168.1.200;
---
>     router id 192.168.1.100;
17a11,17
>     #Functions Section:
>     #End of Functions --
>
>     #Filters Section:
>     include "/etc/bird4/filters/filter1";
>     #End of Filters --
>
19c19
<     protocol kernel {
---
>     protocol kernel kernel1 {
```

```
46c45
<     source address 192.168.1.200;
---
>     source address 192.168.1.100;
57c57
<     neighbor 192.168.1.201 as 1002;
---
>     neighbor 192.168.1.101 as 1002;
```

Listing 4.1: Differences in Bird configuration using v0.2 and v0.3 of the Package.

As shown in this *diff* snippet, almost all the translated configuration is identical apart from:

- Different Router IDs and BGP neighbours (expected)
- Kernel Protocol definition (minor change in the API)
- BGP Filter definition (major change in the API)

4.1.2 Bird Daemon Errors

Bird Daemon provides an error exit code together with different text outputs in order to highlight errors in the configuration. Although most of the times it can be easily spotted using Bird's feedback, there are also instances where the Daemon's documentation may be required to fix them.

4.1.2.1 Bird Daemon Error examples

Most common errors that an administrator may need to resolve are:

- A configured field has incorrect syntax. Bird will give you hints about what is wrong most of the times: wrong IP address format `bird: /tmp/bird4.conf, line 7: Invalid IPv4 address 1921.68.1.1`. But some *rare* times the message is less helpful and you may need to check the contents of the file and understand the error.

  As an example of this: `bird4: Failed - bird: /tmp/bird4.conf, line 65: syntax error`. We need to check the bird4.conf file and see that in line 65:

```
64:     protocol bgp BGPExample {
65:         import Filter NonExistingFilter;
66:     }
```

Listing 4.2: Bird4.conf contents.

  We will need to find out that the shown filter used in the import field of BGP Protocol, does not exist.

- Non-compatible configuration. The other set of common errors is non-compatible fields in a Protocol.

  As an example of this: `bird: /tmp/bird4.conf, line 76: Only internal neighbor can be RR client`. We need to remove the Route Reflector Client setting from the BGP Instance to fix this behaviour.

- Missing filter or function. If you include a filter name in any of the Protocols or if any of your filters use a non-existing function, Bird will fail to start showing an error as follows: `bird: /tmp/bird4.conf, line 71: No such filter`.

- Syntax errors in a filter or function. This error follows the same approach as the first bullet: `bird: /etc/bird4/filters/filter-20170507-0951, line 4: syntax error`. You are required to go to command line and fix the problem checking the configuration and filter or function files.

- Filter calling to non-existing functions. If your filter executes a command that is not defined by Bird's syntax, it will handle it as a function. If that function does not exist in any of the handled files, it will show this error: `bird: /tmp/bird4.conf, You can't call something which is not a function. Really.`

- Filters not accepting/rejecting routes. Bird Daemon filters must return an *accept* or *reject* policy per route received. If any of your filters does not return any policy per route, it will be silently ignored and substituted with an "accept".

  As an example of this issue:

```
filter doNothing
{
    print "HelloWorld";
}
```

Listing 4.3: Filter definition.

Bird Daemon will succeed starting up but, if we check the log information in the Log Page, this error message will be shown:

```
<ERR> Filter doNothing did not return accept nor reject. Make
    up your mind
<INFO> HelloWorld
```

Listing 4.4: Filter printing message.

4.1.3 Real Scenario: VM with simple BGP configuration connected to Guifi.net

As part of the acceptance tests, a VM was set up by a sysadmin in the Universitat Oberta de Catalunya to act as a pre-production machine. This

VM is connected to a *Mikrotik* Router acting as Gateway to *Guifi.net* but this scenario does not connect or communicate through any Mesh Network using BMX6, so it is an end point.

The configuration of this system is almost identical, component-wise, to the ones available in Guifi.net. However, this system will only route itself (1 route) and import any.

Bird UCI configuration set through the web UI and its translation into Bird4 configuration can be reviewed in appendix A.

This VM is communicating to Guifi.net through a Mikrotik which is already doing some filtering but, in any case, it is still able to import 3000+ Routes and export itself:

```
root@LEDE-eloi:~# birdcl4 show protocols all
[...]
BGPImportALL BGP      master   up     2017-05-10  Established
  Preference:     100
  Input filter:   ebgp_in
  Output filter:  ebgp_out
  Import limit:   3000 [HIT]
    Action:       warn
  Routes:         2999 imported, 1 exported, 2999 preferred
  Route change stats:     received   rejected   filtered
      ignored   accepted
    Import updates:        1208383          0          0
               88    1208295
    Import withdraws:       337268          0        ---
              300     336968
    Export updates:        1208298    1208295          2
               ---           1
    Export withdraws:       336968        ---        ---
               ---           0
  BGP state:          Established
    Neighbor address: 172.25.35.25
    Neighbor AS:      59361
    Neighbor ID:      10.90.224.65
    Neighbor caps:    refresh AS4
    Session:          external AS4
    Source address:   172.25.35.26
    Route limit:      2999/3000
    Hold timer:       160/180
    Keepalive timer:  29/60
```

Listing 4.5: Bird BGP query.

Using Bird Lightweight Remote Control (**birdcl4**) we can verify Bird's BGP instance. As key information:

- BGP Instance: BGPImportALL
- Filters applied: *ebgp_in* and *bgp_out*

- We are connected to our neighbour 10.90.224.65 with Autonomous System ID 59361
- The number of routes received fluctuates but the data shown presents 2999 routes imported.
- We do not know when, but the import Limit reached (HIT) and that generated warnings. From our Package's Log Page: `2017-05-21 22:09:13 <WARN> Protocol BGPImportALL hits route import limit (3000), action: warn`
- We are exporting 1 Route.

As a health check, we can query Bird of its last reconfiguration, reboot time or status using `bircl4 status`:

```
root@LEDE-eloi:~# birdcl4 show status
BIRD 1.6.3 ready.
BIRD 1.6.3
Router ID is 10.139.173.161
Current server time is 2017-05-22 00:20:23
Last reboot on 2017-05-10 19:31:09
Last reconfiguration on 2017-05-10 19:31:09
Daemon is up and running
```

Listing 4.6: Bird status query.

Chapter 5

Conclusions

In this project, the existing Bird Daemon's integration package has been reviewed and refactored making it more robust and compliant with the latest OpenWrt/LEDE-supported Bird API. New graphical configuration pages have been added in order to add missing critical functionality highly used in most of Bird configurations. These new pages integrate some functionalities that, in the previous version, were forcing administrators to do manual changes in command line, thus being a big improvement to make this tool more user-friendly. Moreover, because one of the biggest challenges during this project has been the lack of documentation in some of the areas that were being improved, this dissertation has aimed to include reference information to facilitate its understanding and the Package's documentation has been updated and extended. Finally, the theoretical analysis performed has shown a promising integration opportunity to enhance the Package's capabilities adding monitoring data.

This project's objectives have been successfully achieved on schedule even with the challenges occasioned due to the lack of official documentation. Even more, as an extra task for this project to prove its correctness, a network section has been deployed in the real environment (using virtual nodes) following the same topology and challenges as the targeted one. The network's configuration has been a real challenge as the environment and the management system were in Catalonia and the connection was done through a VPN in the UK. Moreover, because of the number of different technologies being used to get connection between both endpoints, it has required weeks of work to configure it as expected and to be able to get reliable data from it. Therefore, because this task was started after achieving all the requirements, it has not been possible to monitor it and to analyse the data but to confirm its correct behaviour.

Regarding this project's schedule and methodology followed, I have worked in a kanban-like manner, which has helped me focusing in one requirement at a time and also has been positive for Víctor who, as a Stakeholder, received

regular updates, new features demos and progress reports to help him manage the project and find possible risks or blockers. One of the recurring risks we did flag was the above mentioned network deployment. Because of its late deployment and big number of unknowns, it did cause a big unplanned overhead on the project and its delivery.

5.1 Future work

This project has opened a number of future goals that have been recorded either in this dissertation or in the Package's documentation repository:

- Finish Package repository's wrapping up in order to deliver this improvements to OpenWrt/LEDE's official stream.
- Continue improving Bird's integration by enhancing the Package (e.g. add OSPF protocol's graphical integration).
- Finish requirements' definition on uBus integration and UI capabilities foreseen to include from Bird's live data gathering and which implications it has on the system (i.e. performance or storage issues if we keep too much historical data).
- Analyse latest Bird Daemon v2.0.0 (currently in alpha state) and plan for any API disruptive change, new features and capabilities to take advantage of it while improving the current version.

List of Terms

BusyBox : light and optimised UNIX tools including most of the widely used terminal commands (e.g. `ash`, `cat` or `rm`). Main documentation. 24

Exterior Gateway Protocol : routing protocols managing connectivity and paths on networks compound by Autonomous System entities. 4

GIT : version control system widely used to track file's changes in a decentralised manner. 1

Google Summer of Code : annual open source software development program (6 months) driven and sponsored by Google to support students looking for OSS projects and getting rewarded for developing them. 8

Interior Gateway Protocol : routing protocols being used in internal Autonomous Systems (single-administrated network behaving as an entity) to manage their connectivity and paths. 4

JSON-C : a JSON implementation for parsing C language data structures. This implementation states that it is RFC7159 compliant. 53

Kanban Lightweight Agile development process. Unlike other Agile project management methodologies, Kanban is just a number of principles about how to empower team's performance and not a bounded framework specifying what needs to be done and in which manner.. 10

L2TP : Tunneling method to support VPNs using the Point-to-Point and Layer2 Forwarding Protocols together.). 19

MarkDown : programmatic documentation syntax to convert plain text documentation into HTML. There are several different implementations with slight differences. Original specification - 2004. 40

Quagga : Network software solution providing an implementation for most of the infrastructure (classic) routing protocols (e.g. BGP, OSPF and RIP). 10

Taiga.io : online project management tool working either with Kanban or SCRUM Agile methodologies. This tool is widely used in OSS projects due to its power, simplicity and plugins (open API) and has also enterprise options (https://taiga.io/). 11

XOLN : el Comuns de la Xarxa Oberta, Lliure i Neutral document. Guifi.net's principles to guarantee a Free, Open and Neutral Network. This principles guide both users and service providers how to interact in a fair manner. La Fundació Guifi.net is the network's institution to arbitrate any issue arisen due to this principles non-compliance. 6

List of abbreviations

API Application Programming Interface. 2

CLI Command Line Interface. 41

GPLv3.0 General Public License version 3.0. 41

KVM Kernel-based Virtual Machine. 19

OS Operative System. 4

OSS Open Source Software. 8

PID Process IDentificator number. 30

UI User Interface. 2

UOC Universitat Oberta de Catalunya. 18

UPC Universitat Politècnica de Catalunya. 8

UPF Universitat Pompeu Fabra. 18

UX User eXperience. 26

VPN Virtual Private Network. 19

XHR XMLHTTPRequest. 2

Bibliography

[1] "Bird daemon official repository. nic.cz research labs.." https://gitlab.labs.nic.cz/labs/bird.

[2] A. Neumann, E. López, and L. Navarro, "An evaluation of bmx6 for community wireless networks.," in *WiMob*, pp. 651–658, IEEE Computer Society, 2012. http://dblp.uni-trier.de/db/conf/wimob/wimob2012.html#NeumannLN12.

[3] "Integració entre bmx6 i bgp en dispositius basats en lede." https://github.com/guifi-exo/doc/blob/master/knowledge/bmx6-bgp-lede.md.

[4] "Project's documentation repository: Bird-openwrt package todo list." https://github.com/eloicaso/bgp-bmx6-bird-docn/blob/master/EN/TODO.md.

[5] "Bird remote control api and documentation." http://bird.network.cz/?get_doc&f=bird-4.html.

[6] "Json's rfc definition.." https://tools.ietf.org/html/rfc7159.

[7] "Bird's bgp protocol implementation file." https://gitlab.labs.nic.cz/labs/bird/blob/master/proto/bgp/bgp.c.

Appendix A

Bird Daemon's Configuration using v0.3 Package - UOC's VM in Guifi.net

A.1 UCI Configuration

```
config bird 'bird'
        option use_UCI_config '1'
        option UCI_config_file '/tmp/bird4.conf'
        option UCI_config_File '/tmp/bird4.conf'

config global 'global'
        option log_file '/tmp/bird4.log'
        option router_id '10.139.173.161'
        option log 'all'

config table
        option name 'aux'

config kernel 'kernel1'
        option import 'all'
        option export 'all'
        option scan_time '10'
        option learn '1'
        option disabled '0'

config device 'device1'
        option scan_time '10'
        option disabled '0'

config bgp_template 'BGP_COMMON'
        option receive_limit_action 'warn'
        option local_as '92099'
        option igp_table 'bgpTable'
        option export_limit_action 'warn'
```

```
        option import_limit_action 'warn'
        option next_hop_self '0'
        option next_hop_keep '0'
        option rr_client '0'

config table
        option name 'bgpTable'

config bgp 'BGPImportALL'
        option receive_limit_action 'warn'
        option template 'BGP_COMMON'
        option neighbor_as '59361'
        option neighbor_address '172.25.35.25'
        option export_limit_action 'warn'
        option import_limit_action 'warn'
        option import_limit '3000'
        option import 'filter ebgp_in'
        option export 'filter ebgp_out'
        option next_hop_self '0'

config kernel 'Kernel_BGP'
        option disabled '0'
        option table 'bgpTable'
        option kernel_table '251'
        option scan_time '10'
        option learn '1'
        option import 'all'
        option export 'all'

config pipe 'pipe1'
        option disabled '0'
        option peer_table 'bgpTable'
        option table 'aux'
        option import 'all'
        option export 'all'
        option mode 'transparent'

config direct 'direct1'
        option disabled '0'
        option interface '"br-lan","br-wan", "br-mgmt"'

config static 'static1'
        option disabled '0'
        option table 'aux'
```

Listing A.1: UCI Configuration.

A.2 Bird Configuration

```
#Bird4 configuration using UCI:

log "/tmp/bird4.log" all;
```

```
#Router ID
router id 10.139.173.161;

#Secondary tables
table aux;
table bgpTable;

#Functions Section:
include "/etc/bird4/functions/function-20170507-1038";
#End of Functions --

#Filters Section:
include "/etc/bird4/filters/filter-20170507-0951";
#End of Filters --

#kernel1 configuration:
protocol kernel kernel1 {
#    disabled;
     learn;
     persist;
     scan time 10;
     import all;
     export all;
}

#Kernel_BGP configuration:
protocol kernel Kernel_BGP {
#    disabled;
     table bgpTable;
     kernel table 251;
     learn;
     persist;
     scan time 10;
     import all;
     export all;
}

#static1 configration:
protocol static {
     table aux;
}

#device1 configuration:
protocol device {
#    disabled;
     scan time 10;
}

#direct1 configuration:
protocol direct {
#    disabled;
     interface "br-lan","br-wan", "br-mgmt";
}
```

```
#pipe1 configuration:
protocol pipe pipe1 {
#    disabled;
     table aux;
     peer table bgpTable;
     mode transparent;
     import all;
     export all;
}

#BGP_COMMON template:
template bgp BGP_COMMON {
     local as 92099;
#     next hop self;
#     next hop keep;
     igp table bgpTable;
#     rr client;
}

#BGPImportALL configuration:
protocol bgp BGPImportALL from BGP_COMMON {
     import filter ebgp_in;
     export filter ebgp_out;
#     rr client;
     import limit 3000 action warn;
     neighbor 172.25.35.25 as 59361;
}

========================================
BGP Filters and Functions:
root@LEDE-eloi:~# cat /etc/bird4/filters/filter-20170507-0951
filter ebgp_in {

        krt_prefsrc = 10.139.173.161;

        if match_guifi_prefix() then accept;
        reject;
}

filter ebgp_out {

        if match_guifi_prefix() then accept;
        reject;


root@LEDE-eloi:~# cat
    /etc/bird4/functions/function-20170507-1038
function match_guifi_prefix()
{
        return net ~ [ 10.0.0.0/8{9,32} ];
}
```

Listing A.2: Bird4.conf Configuration.

Appendix B

Bird Daemon vs. other solutions analysis from IXNs perspective

On large-scale routing deployments, for example an Internet eXchange Point (IXP), the most well-known OSS solution is Quagga. However, there are other alternatives as powerful as Quagga and even less resource eater. Although Bird Daemon is a less known technology, it has presence in most of the key exchange points and its capabilities matches Quagga's and even overcomes it thanks to some key features along with an almost negligible resource consumption in comparison to other solutions.

After looking for good references and projects about IXPs implementing and analysing Bird Daemon and asking its community, I have triaged some relevant presentations showing Bird's potential and its increasing interest over the years, along with the complexity of the solutions implemented:

- Amsterdam Internet eXchange (AMSIX): 2010 presentation analysing the use of different routing solutions and resource consumption data. `http://ripe60.ripe.net/presentations/Jasinska-_Ab_Using_Route_Servers.pdf`
- AMSIX (II): 2012 presentation analysing different routing implementations in a specific testbed. https://ams-ix.net/downloads/ams-ix-route-server-implementations-performance.pdf
- CZ.NIC (Bird core developers): 2013 presentation in NANOG57 conference about the latest version of Bird Daemon, key IXP using it and features added since the previous NANOG conference the developers attended to. `https://www.nanog.org/meetings/nanog57/presentations/Wednesday/wed.general.Filip.BIRD.16.pdf`

- DE-CIX: 2016 presentation in RIPE73 conference about scaling Bird servers to load-balance queries. `https://ripe73.ripe.net/presentations/115-e-bru-20161026-RIPE73-scaling-bird-routeservers-final.pdf`
- DE-CIX: 2017 presentation in Euro-IX conference showing a framework developed in order to generate stress tests in large-scale BGP sessions to automate these tests with minimum impact in their infrastructure. `https://euro-ix.net/m/filer_public/81/dd/81dda868-dd3b-46be-8ee2-9881d112af5a/de-cix_route_server_testframework_euro-ix_30.pdf`

Appendix C

Kanban Project Management using Taiga.io Service

As part of the initial investigation, I did some research on Open Source Project Management tools that could help me monitoring my progress as well as adding some value to the final project. Because of this project's scope and time-frame, the size of the team (me) and the number of Stakeholders (Víctor), the only Agile *approach* that I could use was Kanban[1]. The following sections present the tests and initial usage of Taiga Kanban service.

[1] Kanban approach summarised: project with a continuous prioritised backlog, one task per team resource, the project must be releasable after closing any task, reduce to the minimum the number of required ceremonies and tasks go from *ToDo* (left) to *Done* (right).

C.1 EPICS View

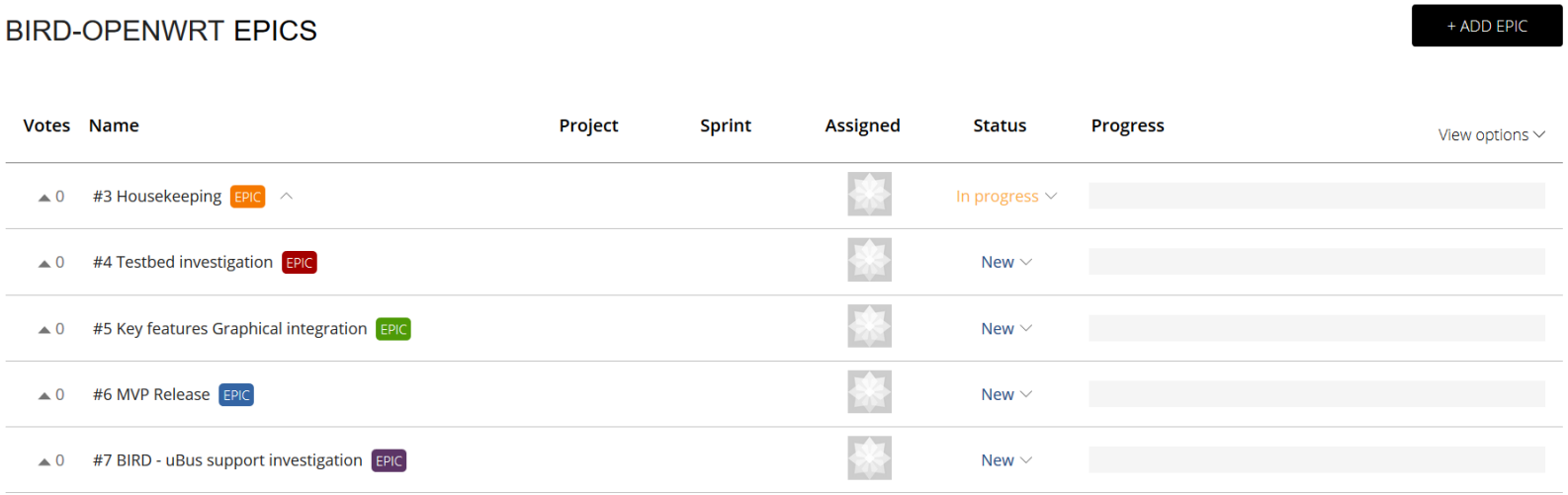

Figure C.1: Project EPICs overview

This view presents the information about the big tasks represented in project's schedule, who is working on each one, other useful information and how far is the task from being delivered.

C.1.1 EPIC Detail View

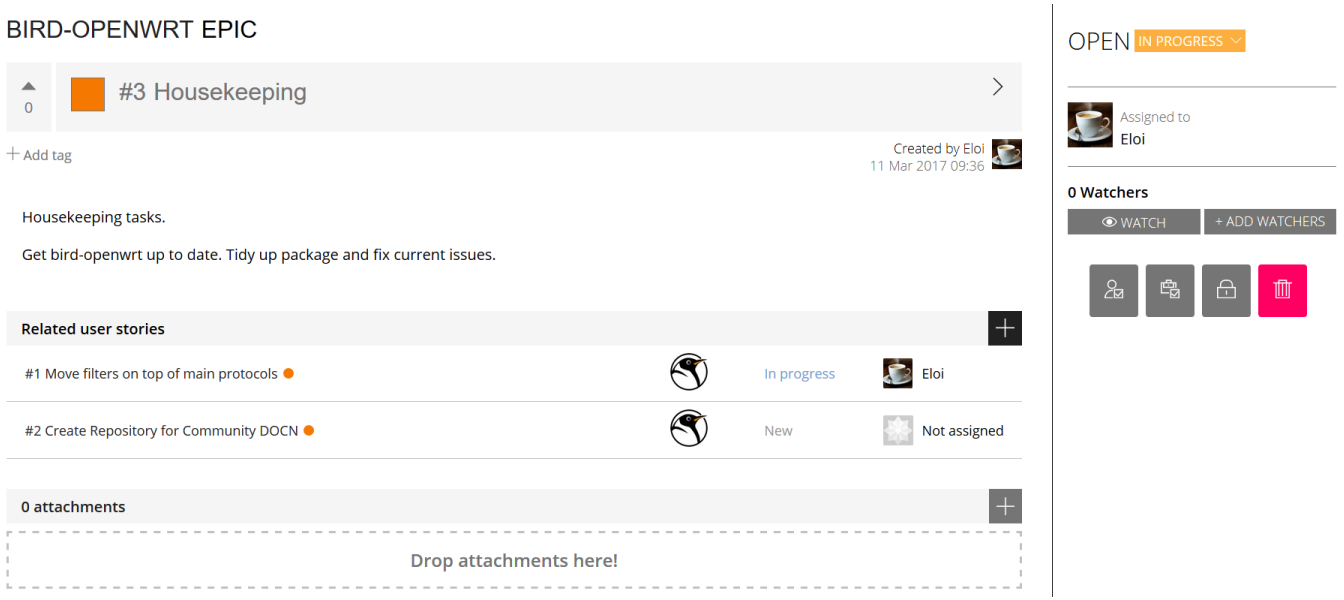

Figure C.2: Housekeeping EPIC detailed view

This view presents the detailed information of a specific Epic. It presents the first EPIC *Housekeeping* which is In Progress, assigned to *Eloi* and two tasks, one already in progress and another waiting for resources.

C.2 Timeline View

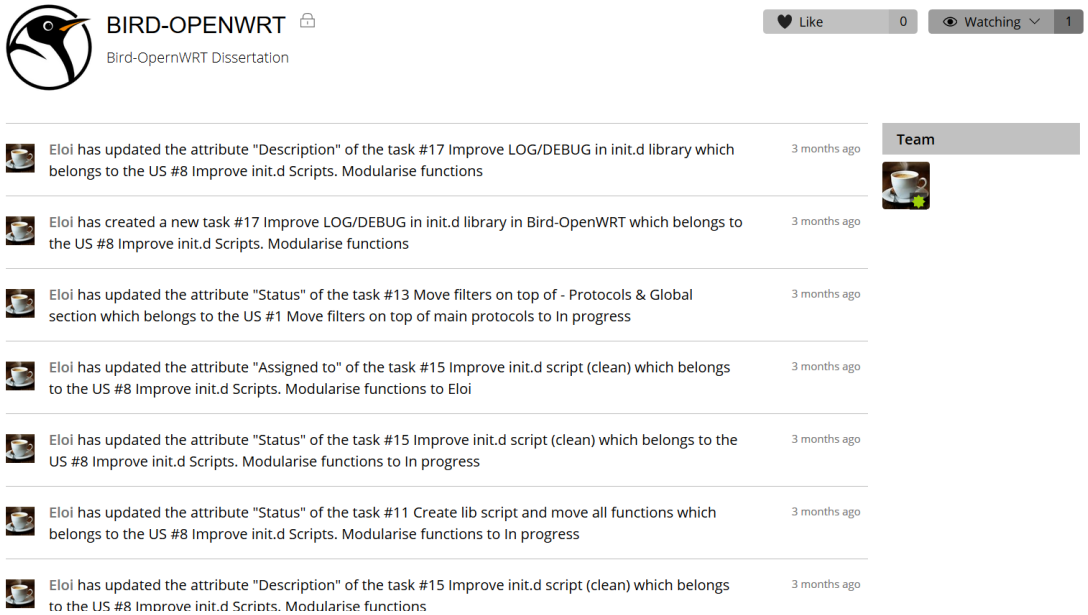

Figure C.3: Project timeline status information

The Timeline View presents Project's log information. Any action applied to any of the tasks, stories or epics will be logged and shown chronologically in this page.

C.3 Kanban Board View

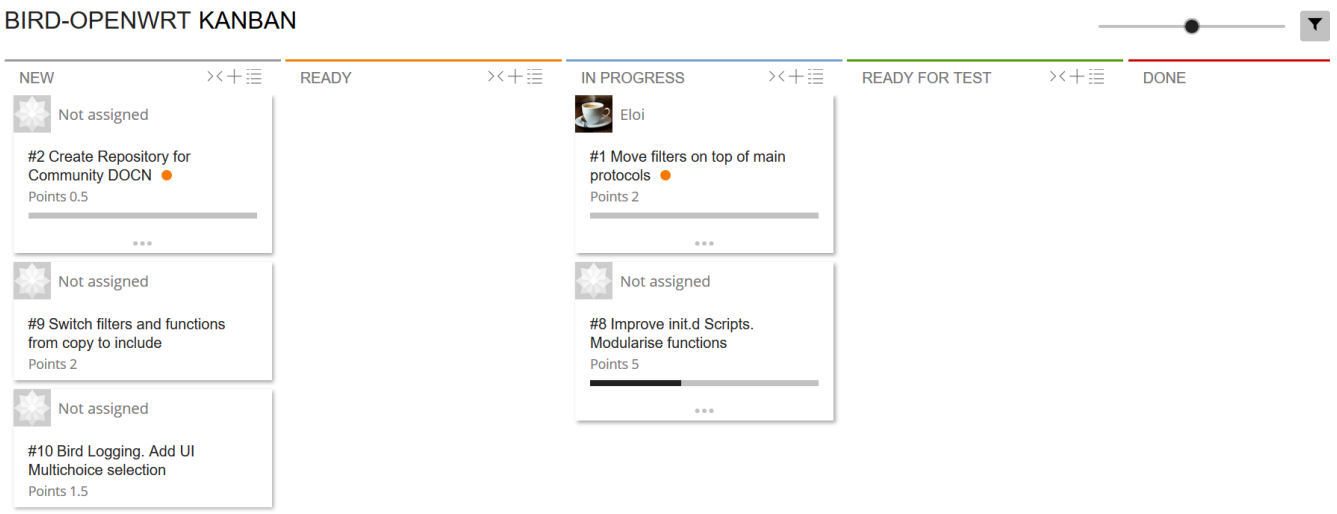

Figure C.4: Kanban User Stories board view

The Kanban Board shows the state of the User Stories being addressed, in which state and how far they are from being completed.

C.3.1 Cycle/Sprint View

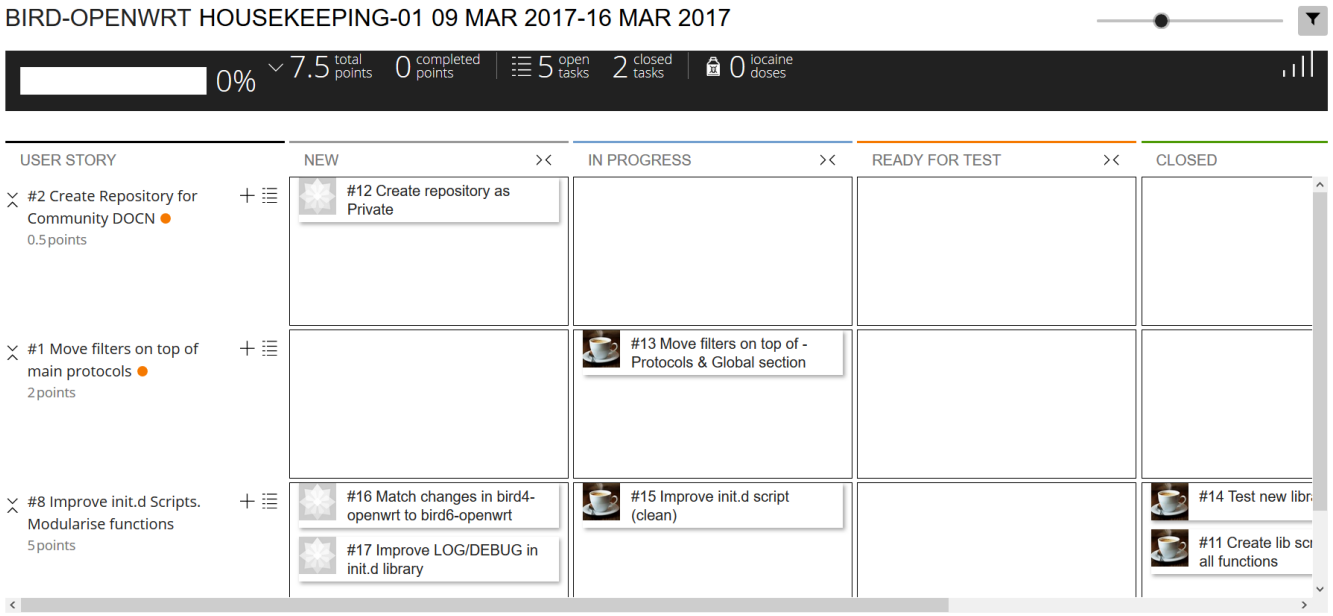

Figure C.5: Kanban current cycle/sprint tasks board view

The Cycle/Sprint View presents the user stories being addressed in priority order (in the left) and all the involved Tasks required to complete each of these user stories, to who they are assigned and their state. This is a detailed view of the Kanban Board View.

Appendix D

Extra LUCI Example Pages

D.1 Privoxy LUCI2 Status Page

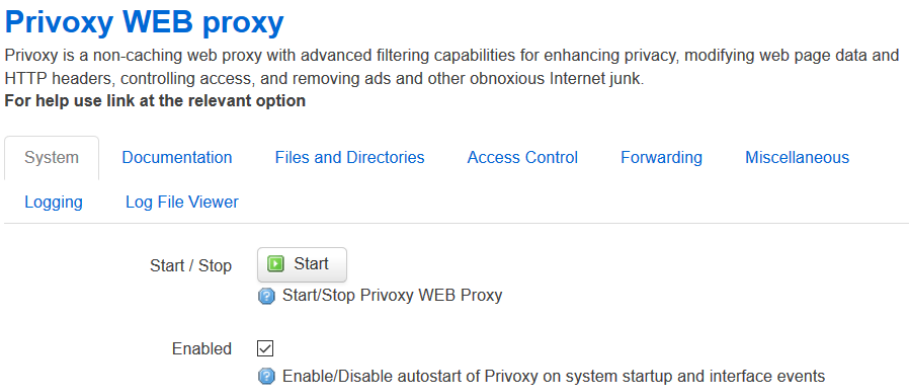

Figure D.1: Privoxy disabled service.

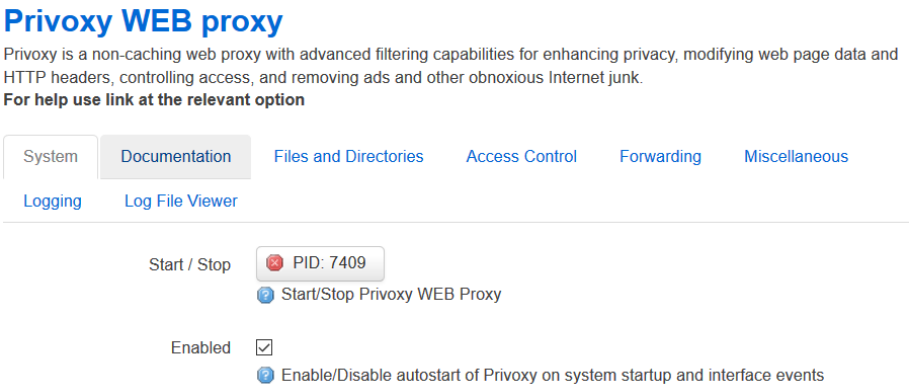

Figure D.2: Privoxy enabled service.