LEDE Firmware optimization for wired deployments using BGP (Bird Daemon) and BMX 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

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

External consultant:
Víctor Oncins Biosca
Routek SL

Universitat Oberta de Catalunya
2017

Index

List of Figures

Listings

Glossary

- OSS: Open Source Software.
- Bird Daemon: OSS internet routing protocol service.

Chapter 1

Introduction

1.1 Structure of the document

1.2 Motivation and description of the problem

This project aims to simplify and enhance management and monitoring capabilities of network administrators' using Bird Daemon software on top of an OpenWRT/LEDE-based Firmware. This project is a second iteration in the development of an existing configuration integration package already being used by OpenWRT/LEDE's community.

1.2.1 Motivation

Back in 2014, while working on my BSc. dissertation in the *Universitat Politècnica de Catalunya*, the department and, specifically, the investigation team I was working with, gave me the opportunity to participate in a GSoC[1] project under the umbrella of Freifunk, to design, develop and present a package that would help simplifying the configuration of Bird Daemon as a software able to share routes between BMX6 mesh and BGP infrastructure networks deployed in *frontier* nodes deployed in the Catalan community network Guifi.net.

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 a user-friendly Web UI even without deep knowledge of Bird's syntax. GSoC's time frame though was not enough to polish the package and add some secondary protocols and the package stopped getting maintenance from myself later that year. However, it has been an OSS project that has been on my *backlog* of things I want to keep improving and also been queried some times by Víctor Oncins as it is really helpful for network administrators

[1] Google Summer of Code (2014)

but it is not mature enough for complex production environments available in Guifi.net.

Therefore, I have been really lucky to have the opportunity to retake this package as my MSc. project while doing my MSc. and work together 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 [1] describing the main challenges found using the old version of the Package and a deep description of the environment.

1.2.2 Bird Daemon

Bird Daemon[2], from now onwards Bird, is an open source Internet Routing service (daemon) 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 a c-like scheme and this 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:

1.2.2.1 Routing Protocols

- **Babel**: IGP[3][4] 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

[2] Bird Daemon: Link

[3] Interior Gateway Protocol: routing protocols being used in internal Autonomous Systems to manage their connectivity and paths.

[4] Autonomous System: Single-administrated network behaving as an entity.

Package and, because it is obsolete, there are not plans to implement it in short term.

- **Border Gateway Protocol - BGP**: EGP[5] 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 / relevant options have been implemented. BGP's support finalisation is one of the top priorities for future Package improvements.

.

1.2.2.2 Support Protocols

The following list of *Protocols* are not exactly routing protocols but supportive implementations of different capabilities or services to enhance and simplify system's management.

- **Static**: Bird's mechanism to implement *smart* static routes. It allows origin or pattern discrimination and modification. For example, 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**: Routing Tables are the main knowledge units for Bird (i.e. BGP & OSPF primary tables). Pipe Protocol allows to connect different Routing Tables and to apply discrimination to the routes the share (i.e. BGP->OSPF accept all and OSPF->BGP accept if part of 10.0.0.0/8). Pipe Protocol is available to automate through the Package.
- **Direct**: 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**: This *protocol* is required in most of the Bird configurations and its main purpose is to gather key data from system's interfaces in order to facilitate Bird's operation. Device Protocol is available to automate through the Package.
- **Kernel**: 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.

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

- **RAdv**: 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 not plans to implement it in short term.

- **Biderectional Forwarding Detection - BFD**: This *protocol* 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 session created by real routing protocols (i.e. 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 term.

1.2.3 OpenWRT/LEDE's configuration integration package

Bird-OpenWRT Package, from now onwards *the Package*, is an open source OpenWRT/LEDE-specific solution (*.ipk*) integrated by four separated packages (two for Bird IPv4 (*bird4-uci*) and IPv6 (*bird6-uci*) UCI integration and the other two for Web UI management (*luci-app-bird4* and *luci-app-bird6*) providing Bird Daemon a user-friendly configuration scheme (UCI) and a graphical interface in OpenWRT/LEDE-based routers. All the implementation details are covered in chapter 3.

1.2.4 Bird Daemon 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 (i.e. Pipe or Direct).

- Some secondary protocols were not enabled in the final solution because they were not relevant for GSoC's scope (i.e. OSPF or)

- Some basic processes require manual (terminal) changes

- No possible way to edit Filters or Functions files through Web UI.

- No Bird Daemon Status feedback (i.e. 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 (i.e. full information for BGP connections)

1.3 Scope of the project

This project's scope is to adopt as many of the mentioned enhancements that are clearly aligned with eradicating required manual changes in command line, improve the UX[6] and to align the packet with current Bird Daemon API in the given time frame of 3 months. As a result of a *backlog* prioritization, the following items were agreed (in priority order):

- Update the package to the latest Bird API.
- Update old version's disruptive issues (i.e. disabled Protocols).
- Status, Log, Filters and Functions Graphical integration.
- Theoretical viability investigation of uBus integration.

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 secondary protocols: adopt more key features from Bird and increasing the range of administrators being able to take advantage of this Package.
- Integrate next generation of Web UI using LUCI2: HTML/JavaScript-based UI instead of LUA-based.
- 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.

[6] UX: User eXperience

Most of these extra tasks are already documented as part of the Package Documentation Reporitory[7] and open for discussion and Pull Requests[8] to add extra requirements.

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 it from this project's scope, but there is also the big amount of evidence already collected for my GSoC project as well as some new evidence found either in some reputable sources as well as from Bird's own OSS Community proving that Bird Daemon has been far more stable, less resource eater and flexible (thanks to its Filter&Function scripting language) than other well-known enterprise level solutions. This evidence is available in the Appendix B.

1.3.2 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/requirement* must be self-contained and the Package should be releasable at any time.
- There is no Board or framework to introduce the data (i.e. 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

[7] GitHub TODO List: Link.

[8] Pull Request: Changes pushed to a repository by an external party (i.e. a fork repository pushing changes to its parent.

Kanban, I did use an online OSS tool called *Taiga.io*[9]. See Appendix C in order to see some captures of the 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* to the Stakeholder to show 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[10] conference service, which allows screen sharing and text communication while in conference, simplifying demoing and code reviews. Regular communication will be also done through Hangouts instant messaging service and by email to share progress, risks or blockers.

1.3.2.1 Gantt Diagram

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

[9] 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.

[10] Jitsi: Open Source multiplatform VoIP conference service.

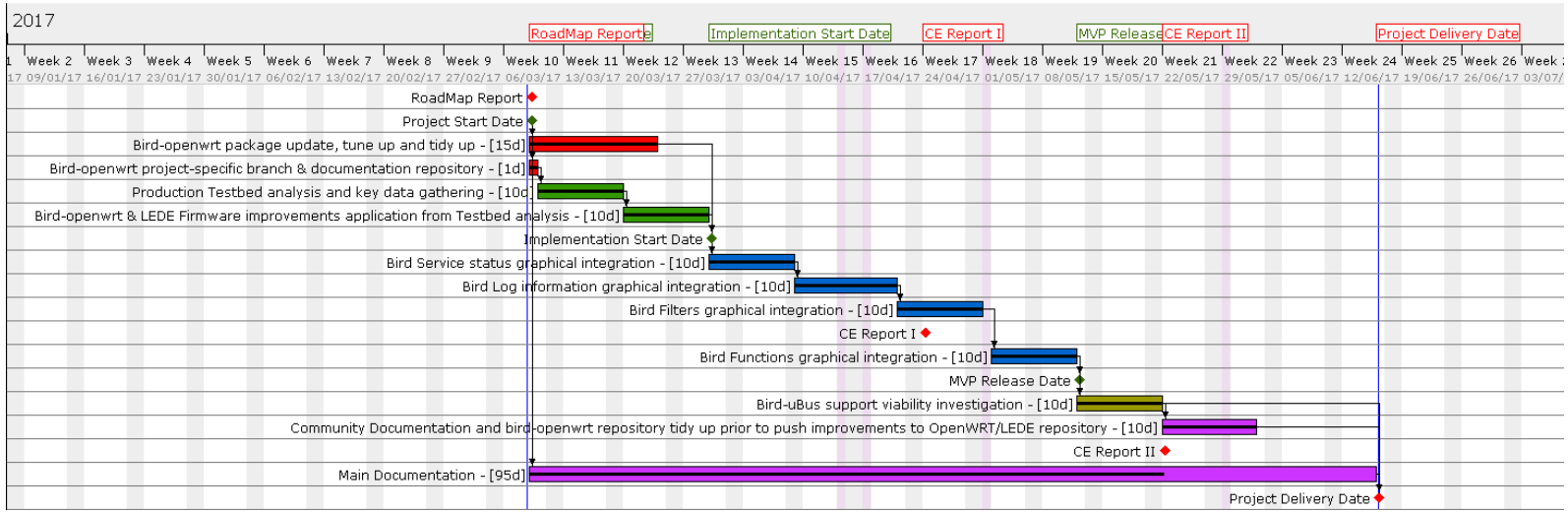

Figure 1.1: Tasks schedule

Key milestones:

- **Project's start date & RoadMap Report** (09/03/17): initial Package refresh and production environment investigation. Formal report of which are project's goals and when are they 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

1.4 Background information

1.4.1 Guifi.net

Guifi.net is a community network working for and by its own users (self-organised) giving an affordable alternative for anyone willing to connect to the Internet. This network's principles are freedom, open design, administration and management and neutrality. This network was born in Catalonia as a wireless network but it has spread all over the world and with about 33.124[11] active nodes (as 26/05/17) using roof antennas and optical fiber deployments.

This network is connected to the Catalan Internet Exchange Point (CATNIX[12]), has its own Government Foundation[13] to promote and protect network's principles defined in an operational and behavioural common regulation (Comuns - XOLN[14]) and it is open for any company that adheres to the XOLN/FONNC principles, to professionally operate and advertise itself as a Guifi.net Internet or services provider.

Finally, although Guifi.net main routing protocol is BGP for infrastructure and OSPF for internal routing, there are several isles[15][16] operating as Mesh Networks using BMX6 dynamic routing protocol.

1.4.2 OpenWRT/LEDE Project

OpenWRT, and its Fork LEDE-Project[17], 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.

1.4.3 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,

- **Infrastructure**: commonly used in structural networks. Stable, robust and scalable. Their main handicap is to suffer of big overheads

[11] Guifi.net live statistics: Link
[12] CATNIX: Link
[13] Fundació Guifi.net: Link
[14] XOLN/FONN: Compact for a Free, Open & Neutral Network
[15] Guifi.net Mesh Networks: Link
[16] qMp: Most commonly used firmware in Guifi.net for mesh networks. This OpenWRT fork aims to simplify and automate mesh deployments.
[17] LEDE-Project: Linux Embedded Development Environment.

on topology changes (i.e. low/non fault tolerance) and have big convergence times in large-scale networks.

- **Mesh Networks**: oppositely to classic dynamic 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 the network topology knowledge up to date.

1.4.3.1 BGP

BGP is a dynamic infrastructure IP routing protocol designed for large-scale internet topologies (EGP[18]). Its routing algorithm relies on the best path according to route's attributes.

1.4.3.2 BMX6

Batman-eXperimental6 [2] is a fork of the Mesh protocol BATMAN[19]. This is a mesh networking routing protocol is compatible with most of linux-like systems but only operates with IPv6 networks. This routing protocol uses a table-driven[20] distance-vector approach[21].

1.4.3.3 UCI

The Unified Configuration Interface (UCI) aims to centralise OpenWrt's packages and 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.

1.4.3.4 LUCI

LUCI is OpenWrt's solution to graphically represent and allow configuring UCI settings via web pages (UI is generated on **Server-side**). It uses Lua[22] language following the MVC[23] software pattern.

LUCI's components are:

[18] EGP: Exterior Gateway Protocol. This includes all the protocols that routes between Autonomous systems.

[19] B.A.T.M.A.N: Better Approach To Mobile Adhoc Networking.

[20] Table-driven: Compose a routing table with all the source-destination entries.

[21] Distance-vector: Best path (cost of going) from source to destination.

[22] Lua: open source powerful language optimised for embedded environments.

[23] MVC: Model View Controller Software architecture pattern. This is **just one** of the different patterns of logically separate a program's *intelligence* between data, logic and UI avoiding tangled *spaghetti code*.

- CBI (Model): CBI files include UCI definitions and *describe* HTML's final form (i.e. optional/mandatory properties, order of apparition, and all the required logic to validate entered data through the forms.
- Controller: Web pages definition (i.e. data 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.

1.4.3.5 LUCI2

LUCI2[24] is the next version of LUCI using **client-side** web technologies to enhance and simplify UI creation, releasing these router's resources. Moreover, this new version uses standard communication process between browser and http server (JavaScript's XHR[25]) using the RPC Daemon[26] and uBus Daemon[27] as query brokers between UI's Front End, Server's Back End and router's internal Daemons providing services under RPCd.

This new version is not completely defined and it has been evolving together with OpenWrt/LEDE and LUCI. There is no much documentation of its definition, API, how to use it or integration examples and, therefore, it is complex to get the full picture of how it works and how to start using it and it is common to see packages using both CBIs(LUCI) and HTML/CSS/JS(LUCI2) approaches together according to the complexity or level of customisation required to represent the data.

The already mentioned lack of documentation in LUCI2 will be repeated during section 3.2.5 as it was a critical handicap while implementing some UI pages with *simple* data structures, but requiring some customisation and process automation, which LUCI2 solves efficiently if you know how to do it.

1.5 State of the art

[24] LUCI2: 2nd generation of OpenWrt UCI UI modeling. This second version uses HTML/CSS/JavaScript and communication through JSON messages instead of LUA.

[25] XMLHttpRequest(XHR): communication method to transfer queries as objects. Most common communication object format is JSON.

[26] Remote Procedure Calls: Client-Server communication mechanism for network systems (router).

[27] Universal Bus (uBus): is a daemon providing a *space* for packages or protocols to register their APIs allowing external access to it acting as a pipe between them. uBus use RPC calls through the RPC Daemon available in OpenWrt.

Chapter 2

Network Architecture

As shown in the figure 2.1, our targeted network is a mixed section requiring the use of Exterior Gateway Protocols (IGP) and Internal Gateway Protocols (EGP) in order be able to share routes routes between the two BGP ends (named *E* and *F*) going through a BMX6 Mesh Network (from report [1] - in Catalan).

As shown in the figure:

- **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 Daemon.
- **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 Daemon.
- **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

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:

Current version of BMX6 is not able to handle the number of routes that this Guifi.net BGP section is sharing (2.500+). Therefore, BMX6 starts aggregating routes, which eventually shut-downs the service and leaves the node overloaded as it is not able to achieve it.

In order to avoid this disruptive issue, Bird Daemon filter scripting capabilities available in MXN1 and MXN2 allow to reduce the geographical scope of the routes imported and exported to/from the Barcelonès[1] Zone.

[1] Barcelonès: Network Zone including Badalona, Barcelona, Hospitalet del Llobregat, Sant Adrià del Besos and Santa Coloma de Gramanet

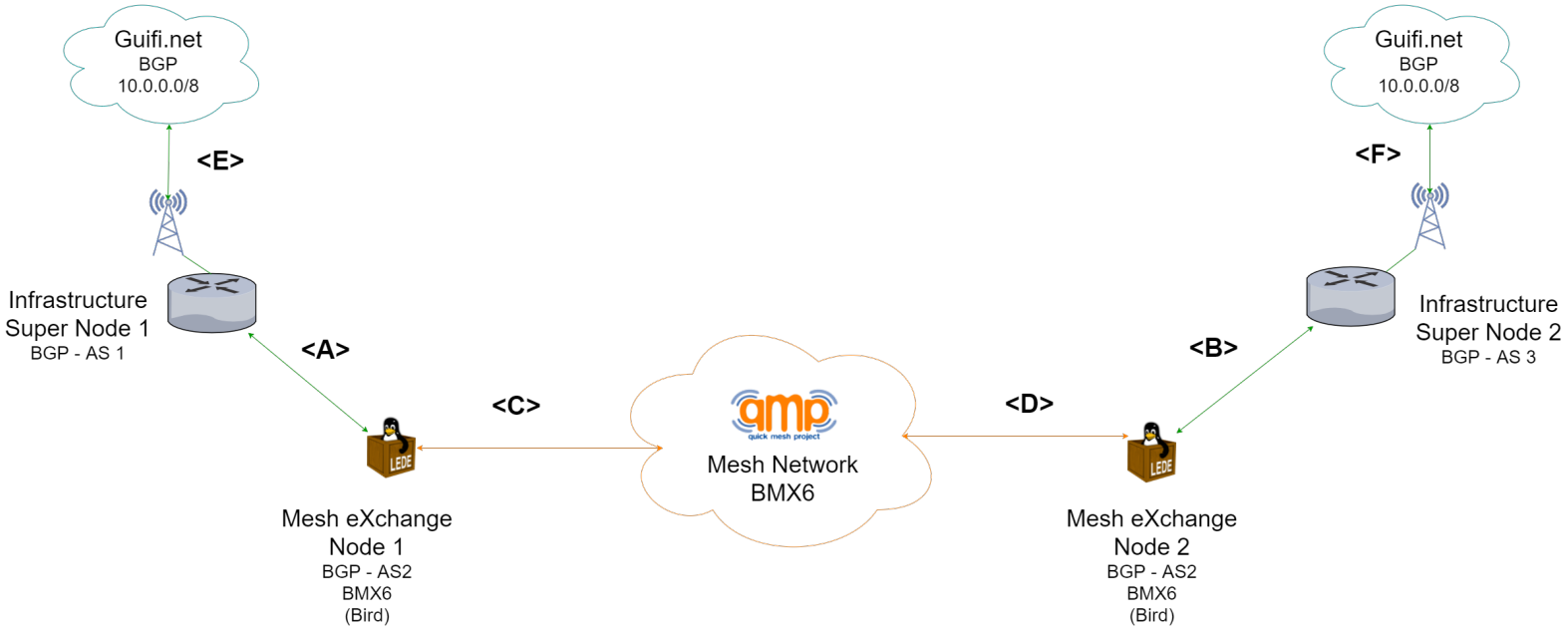

Figure 2.1: Production Network targeted in this project

2.2 Testing Scenarios

Although it has not been possible to test Package's improvements in the target production network as the changes would be incur in service disruption for up to 1500+ nodes[2] 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 environments connected directly with Guifi.net thanks to *Universitat Oberta de Catalunya* and Víctor's support, who have provided me and helped me configuring a number of Virtual Machines, a number of virtual network resources and also agreed permission to connect through another university's network (UPF[3]) in order to simulate the network section required in each case:

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).

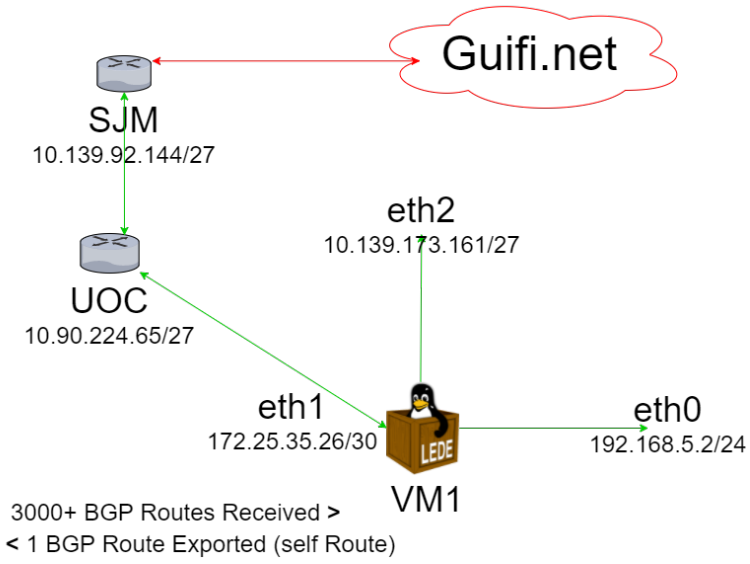

Figure 2.2: Minimal test environment.

[2] Font: https://guifi.net/en/node/2435/view/nodes
[3] UPF: Universitat Pompeu Fabra (Barcelona)

2.2.0.2 Final Package Testing

As can be seen in the Figure 2.3, the final testing environment is 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 in order to test it but without the risk of damaging the production network or flooding unwanted routes to Guifi.net. This network section routes to two Infrastructure SuperNodes connected to two geographically-separated Barcelona well-known Universities, being almost, if not exactly, a mirror of what our target network is.

- 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 Mesh Network.
- Network way through two different network sections.
  - Connection using UOC's Super Node
  - Virtual Machine 4 connects through UPF[4]'s internal network to find path out to a near Guifi network.
  - Both Infrastructure SuperNodes have Import ALL policy applied 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 GRE Tunnel[5]
    * 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.

[4] UPF: Universitat Pompeu Fabra (Barcelona)

[5] Generic Routing Encapsulation (GRE): Cisco's IP Layer virtual tunnelling Point-to-Point protocol.

- **VM1**: KVM[6] Virtual Machine acting as Frontier Node (MXN1). This node is configured with Bird Daemon (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 Daemon (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.

[6] KVM: Kernel-based Virtual Machine.

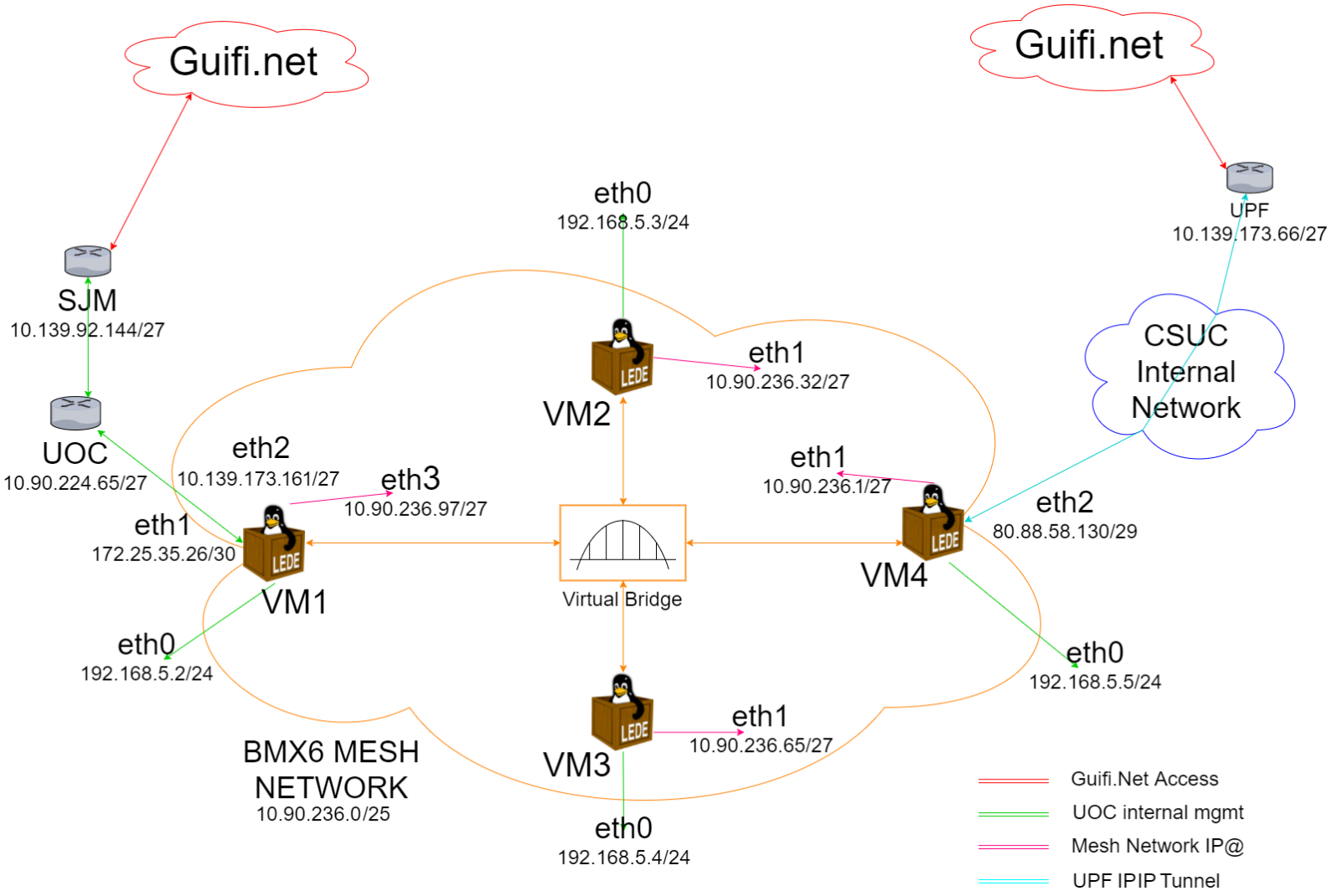

Figure 2.3: Development Network simulating production's environment

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 Daemon instead of Quagga to make us 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 Daemon'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 Daemon service status
  - Bird Daemon 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 Daemon as a mechanism to communicate with Bird Daemon and get health information and current-status information for handled protocol using JSON messages.

3.2 Changes and improvements implemented

The following sections summarises 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 Build 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. This misalignment led into few days testing the build process using the latest version of the development environment in order to tune it (the original building process was documented in 2014).

- Generalise Makefiles (bird4/bird6) and post installation scripts.
- Add detailed steps to get Package's source, add bird required packages into the Image Build settings, compile it (`make`) and deploy it in the target router.
- Add alternative mechanisms, Package information and (after finishing changes in the Package), Package 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[1]. 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 requires 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 (i.e. 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 (\).

[1] BusyBox: light and optimised UNIX tools including most of the widely used terminal commands (i.e. `ash`, `cat` or `rm`).

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 Daemon's init.d file is substituted on 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 (i.e. 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 1.2.1, 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 User eXperience. 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 states that there is **no** Limit set in this BGP session. However, if we set one through the UI:

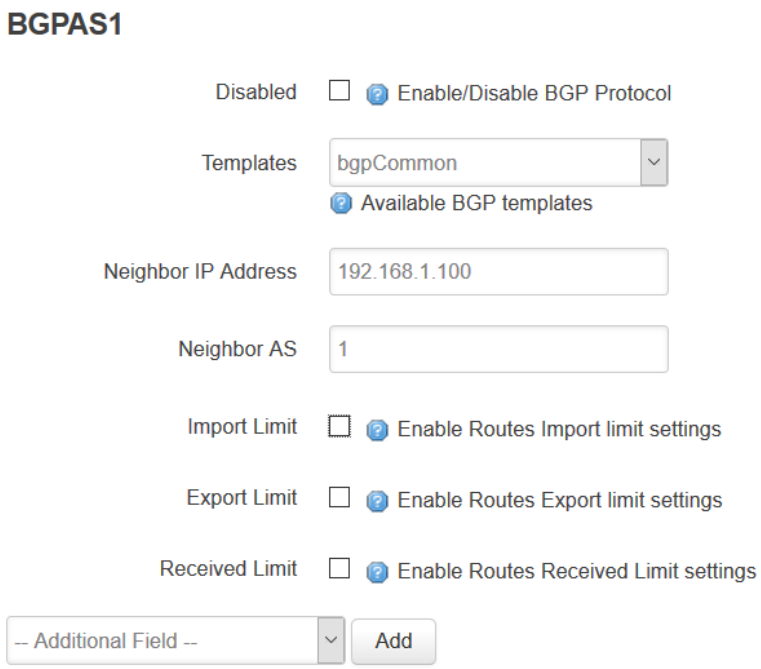

Figure 3.3: Import Limit Trigger **not** selected.

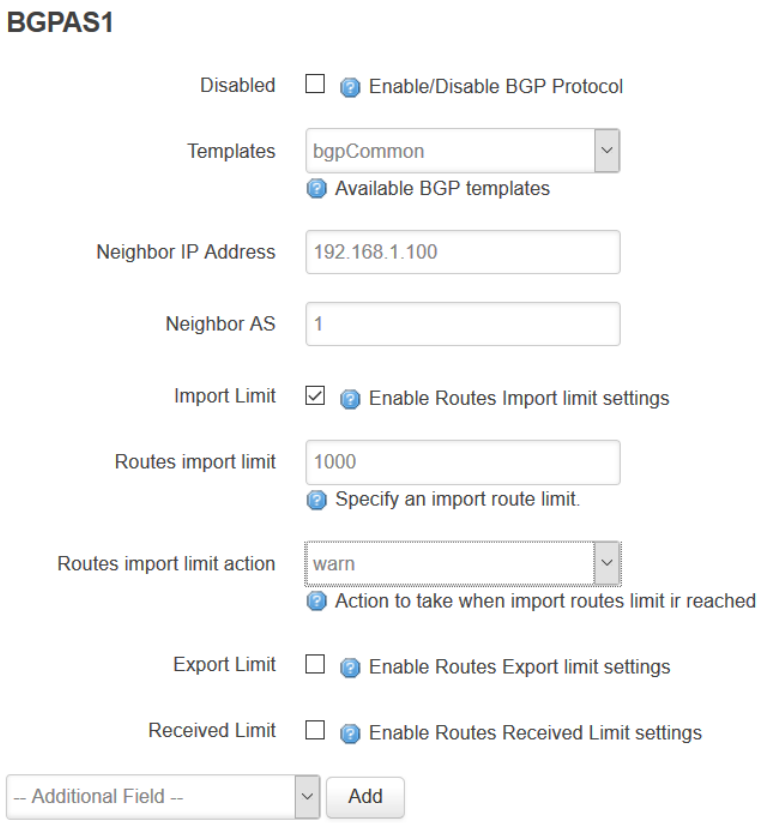

Figure 3.4: Import Limit Trigger **selected**.

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 example3.2.4, 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 you trigger any service update through the buttons.

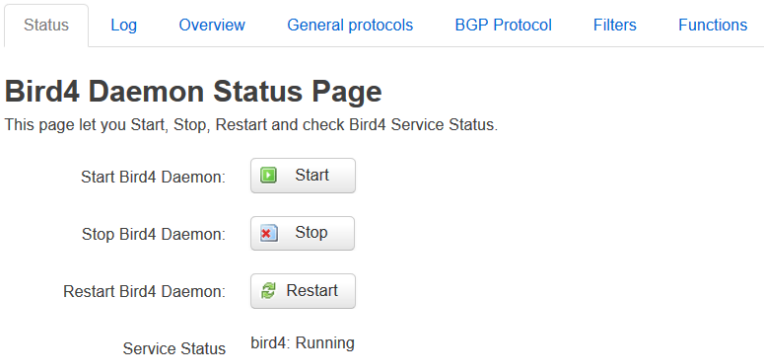

Figure 3.5: Status Page

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 switch between states showing, when started, service's PID. In Appendix D you can see the expected behaviour shown in the LUCI 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 (i.e. button clicked) is triggered or during page's rendering. However, there is no granular control on it or a polling mechanism that could automate the transition between service disabled and enabled. To do so, it would be required to force the page to wait (manual OS `sleep` command) which would also **block** page's loading.

Moreover, I did do a test implementation (without a `sleep`) 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 unexpected way and not refreshing the contents (wrong status and PID displayed).

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/all the possible events in the system. If the Log file grows enough to fill the /var partition, Bird will **automatically** shutdown and **prevent** any start up attempt until this is resolved.

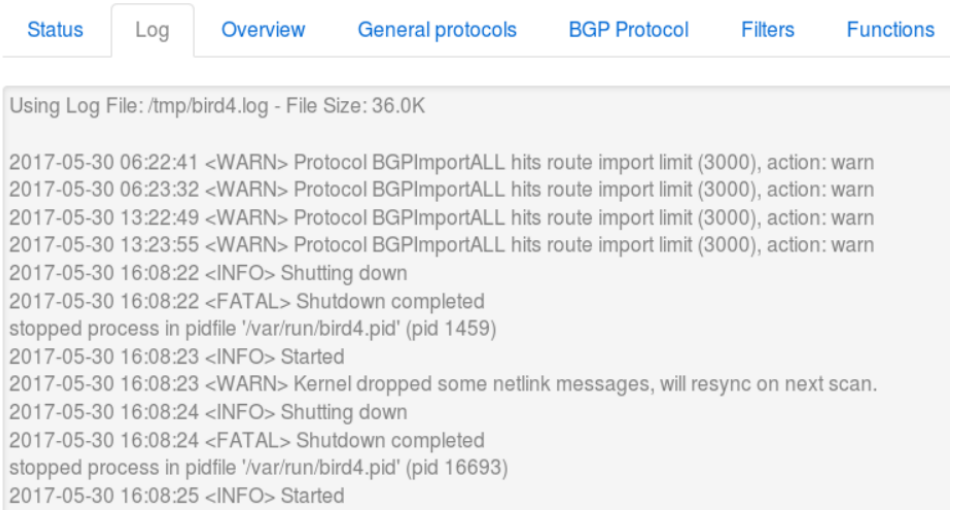

Figure 3.6: Log Page: service restart example

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 did spend 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 (i.e. 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[2], I could have few conversations with one of the main contributors of LEDE, OpenWrt `jow-`[3] and also main author of LUCI/LUCI2, who gave me some hints and examples of LUCI2 pages that helped me fixing my issues with the 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.

[2] LEDE community contact methods: Link
[3] Jo-Philip Wich: 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 (**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[4].

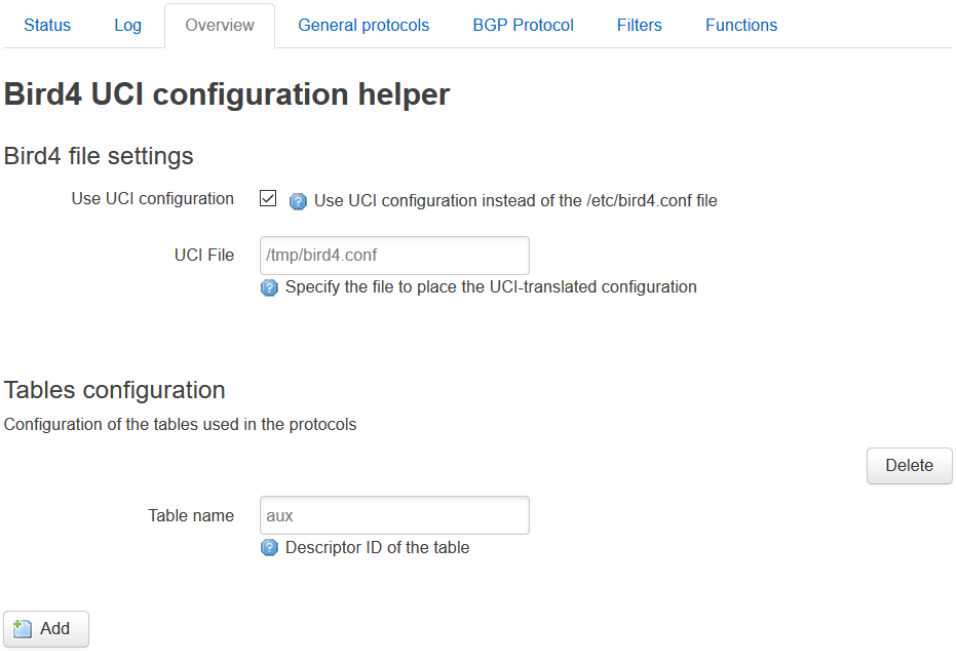

Figure 3.7: Overview Page

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 be not passed to the configuration.

3.2.5.4 General Protocols Page

The General Protocols Page shows the configuration of some of the supportive protocols described in section 1.2.2.2.

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

[4] If the Log Partition is filled and Bird is unable to access it, Bird service is dropped and blocked until more space is added.

- 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.

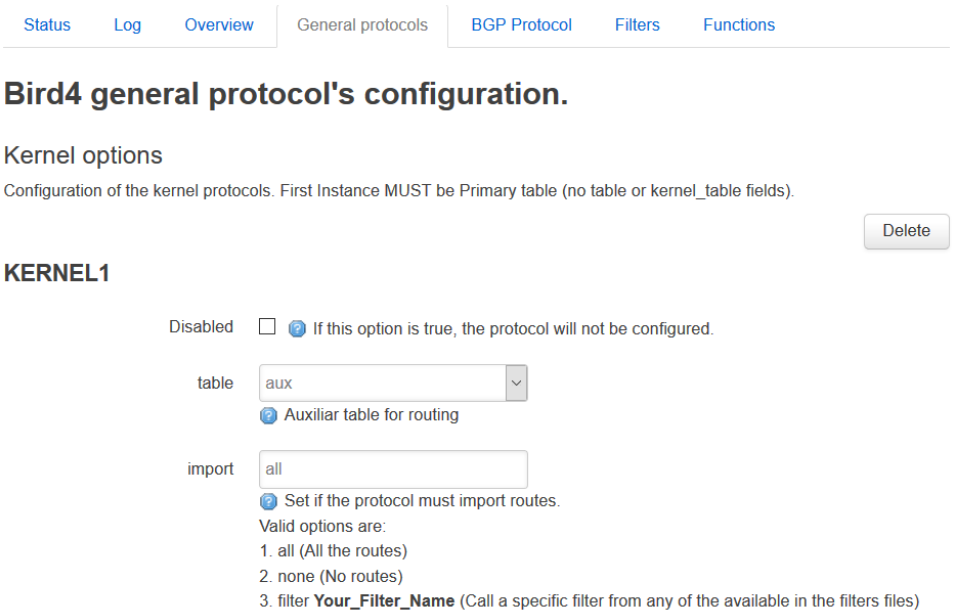

Figure 3.8: General Protocols Page

Changes and Improvements: 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.

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

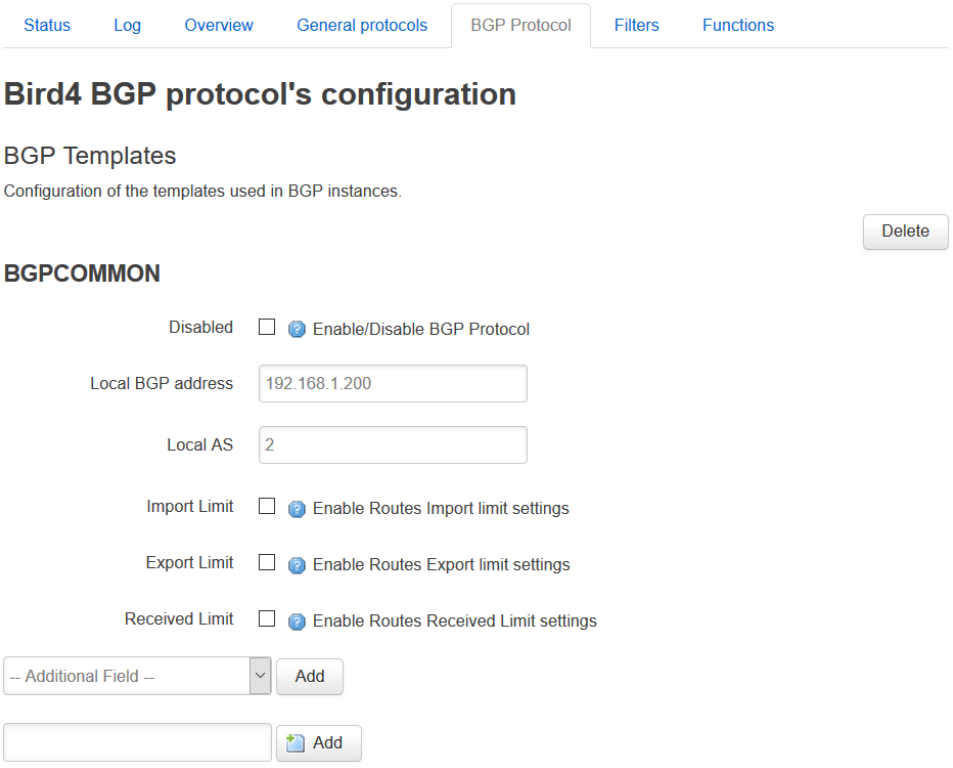

Figure 3.9: BGP Protocol Page

Changes and Improvements:

3.2.5.6 Filters & Functions Page

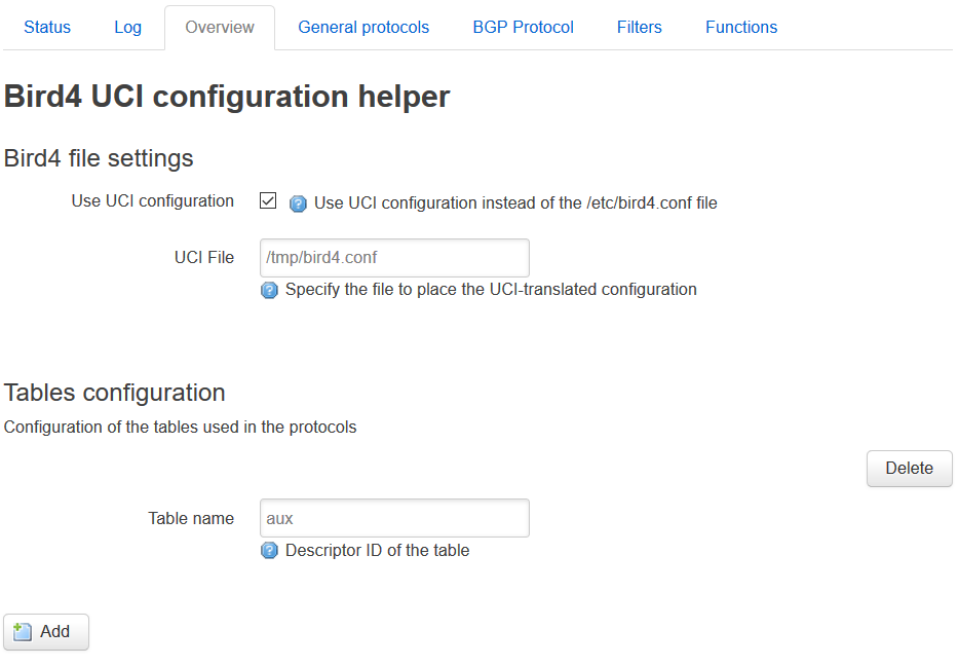

Figure 3.10: Overview Page

3.2.6 Align documentation and upgrade to Markdown

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 not 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 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: Battlemesh experiment code

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 printing message

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: UCI Configuration

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: UCI Configuration

Chapter 5

Conclusions

5.1 Future work

Appendices

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 presence in Worldwide IXPs and other institutions

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

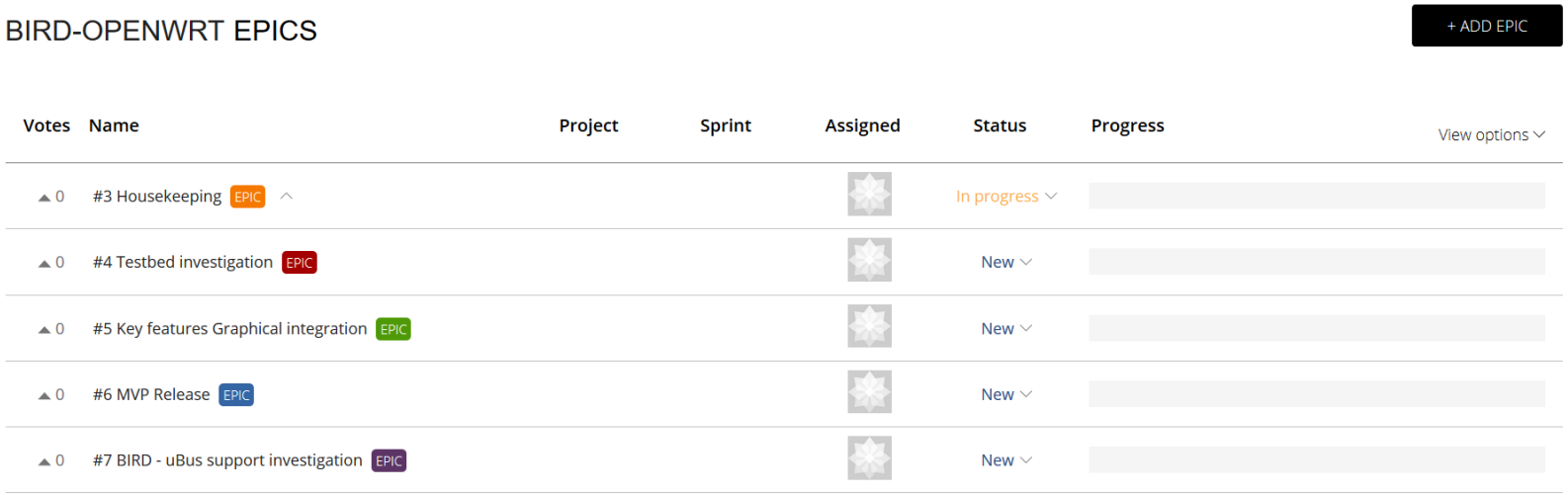

Figure C.1: Project EPICs overview

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 it is the task of being delivered.

C.1.1 EPIC Detail View

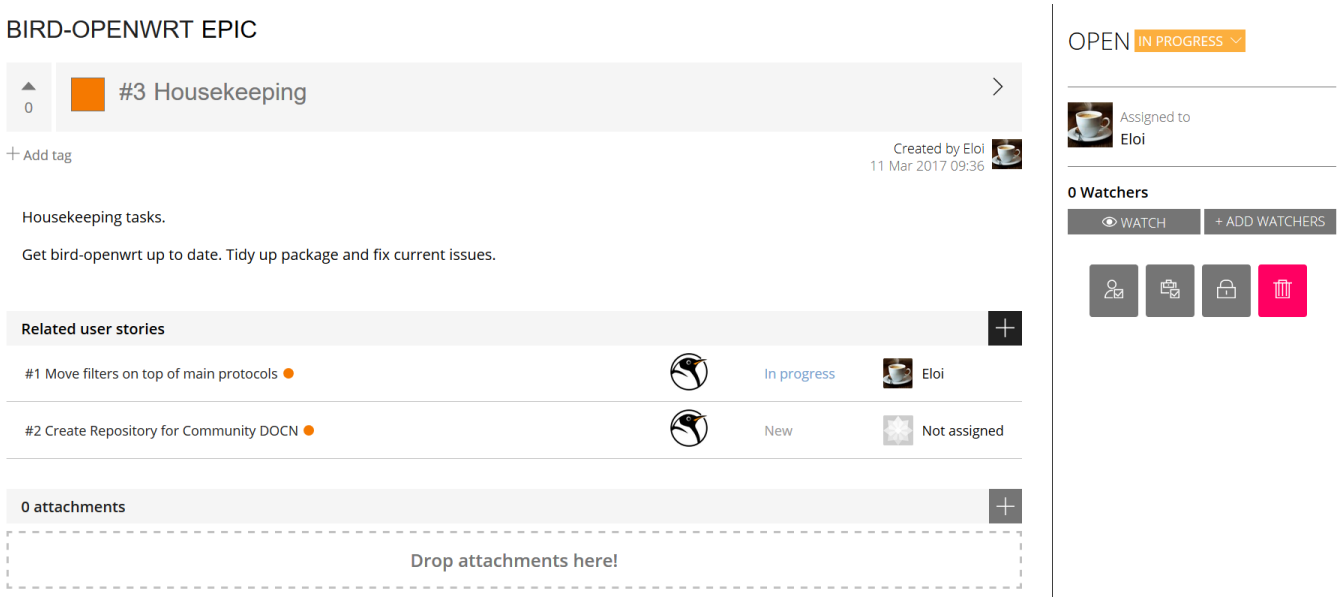

Figure C.2: Housekeeping EPIC detailed view

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

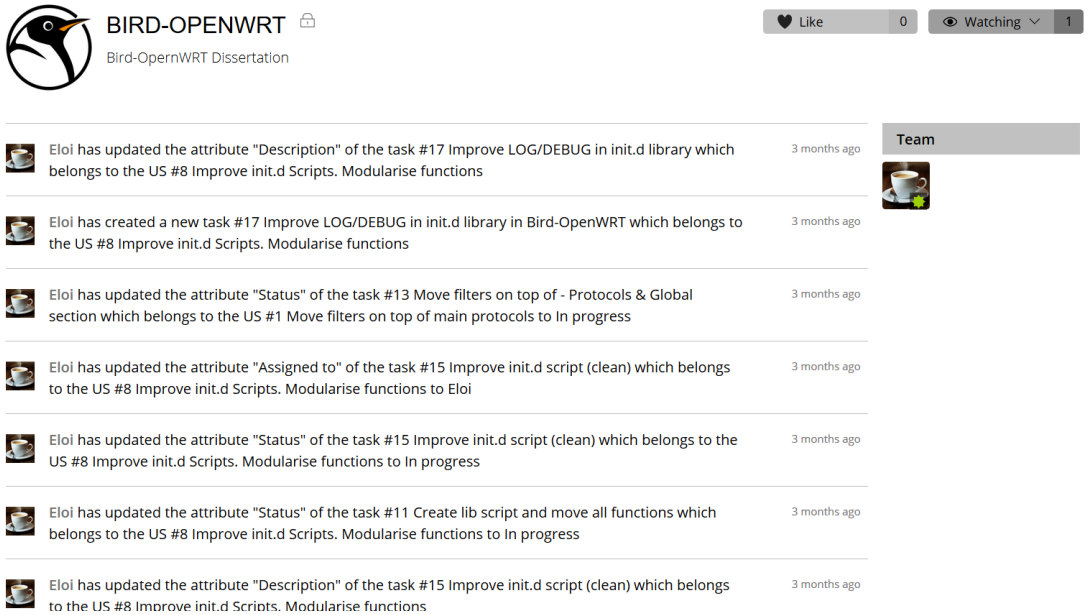

Figure C.3: Project timeline status information

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

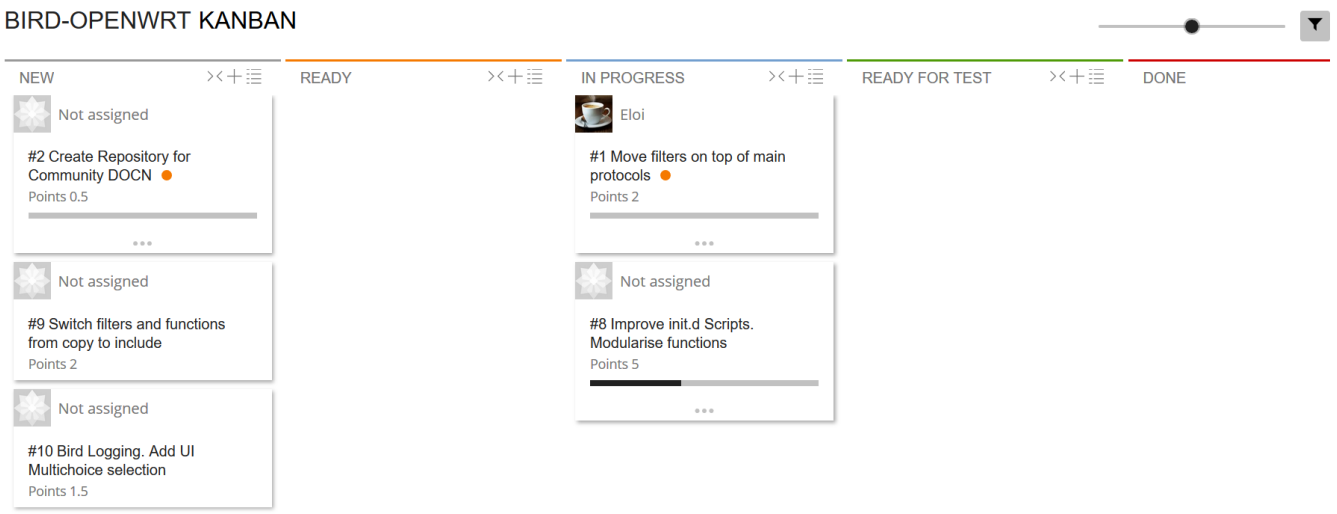

Figure C.4: Kanban User Stories board view

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

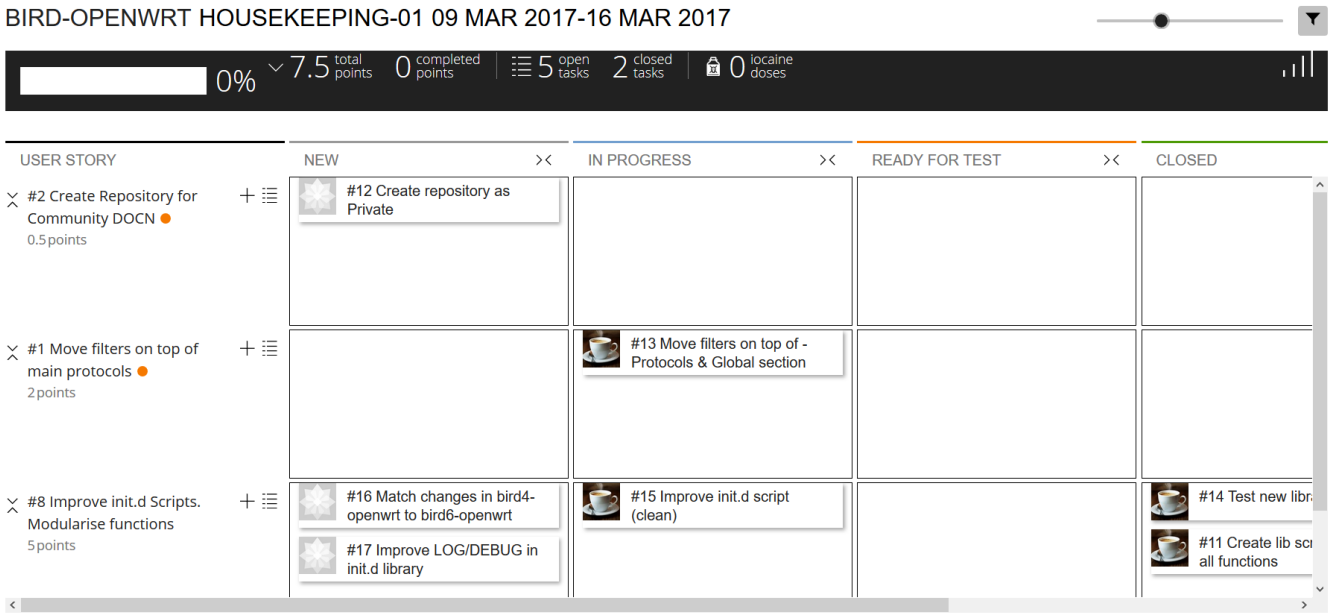

Figure C.5: Kanban current cycle/sprint tasks board view

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 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

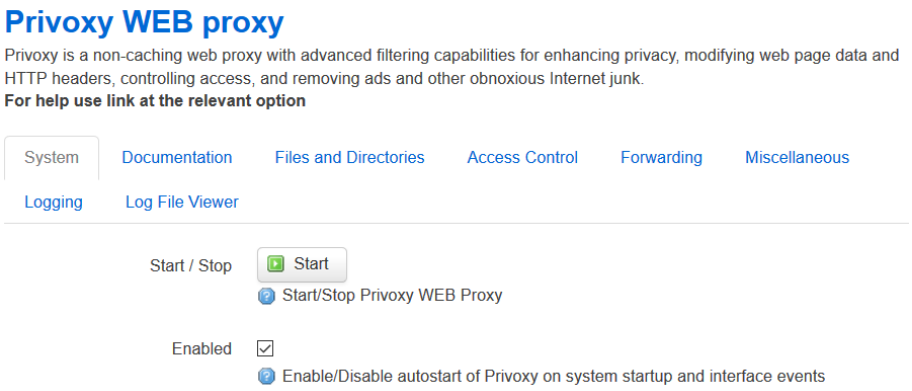

Figure D.1: Privoxy **disabled** service.

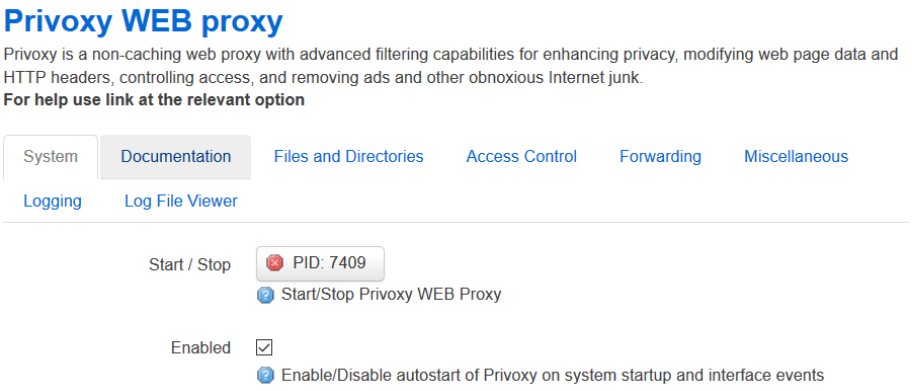

Figure D.2: Privoxy **enabled** service.

Bibliography

[1] "Integració entre BMX6 i BGP en dispositius basats en LEDE." `https://github.com/guifi-exo/doc/blob/master/knowledge/bmx6-bgp-lede.md`.

[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.