Interoperability between classic infrastructure and Libre-Mesh networks in Guifi.net

Contents

1 Introduction

The biggest community network in the world is Guifi.net [1] it spreads across Catalonia and beyond. IP routing on a such big community network is not a trivial task, in addition to the big size, the decentralized nature of the network encourage choice, so usually different parts of Guifi.net network uses different routing protocols.
In order to make those networks parts interoperate is now mostly a manual job, various groups have developed several OpenWrt based solutions to ease networking equipment configuration, but they didn't have achieved seamless protocol interoperability, moreover Guifi.net is mostly built with low cost embedded devices, so little performance footprint is always a requirement.
The scope of this work is to investigate mainly used routing protocols in particular BGP, BMX6 and BATMAN-adv, to study past interoperability experiments and to extend Libre-Mesh to implement seamless interoperability between classic infrastructure and Libre-Mesh networks in Guifi.net.
Bringing the new features at an usable and stable quality, testing them in real environment, protocols parameters adjusting for the optimal functioning and their deployment in production setups such as Barcelona Botanical Garden, AureaSocial offices and others clients are parts of this work too.

2 State of the art

2.1 BGP

BGP (acronym of Border Gateway Protocol) is a standard routing protocol designed to exchange routing and reachability information between autonomous systems (AS) on the Internet. BGP makes routing decisions based on paths, network policies, or rule-sets configured by a network administrator that is involved in making core routing decisions, because it needs information of each route reachability path to make decisions BGP is usually classified as a path vector protocol. Version 4 of BGP has been in use on the Internet since 1994, it's mayor improvement in respect to older version that are obsolete and unused on the Internet is the support for classless inter-domain routing (CIDR) and routes aggregation.
BGP may be used for routing within an AS. In this application it is referred to as Internal BGP, or iBGP. In contrast, the Internet application of the protocol may be referred to as Exterior Border Gateway Protocol, External BGP, or eBGP.
As BGP doesn't perform any type of neighbor discovery, peering are setup by manual configuration. For each peer router a connection through TCP port 179 must be established, after the connection is established BGP daemons synchronize with peers by sending the whole routing information they have to peers, after this initial sync only updates are transmitted to peers.

2.2 Guifi.net

Guifi.net networks uses different routing protocols such as BGP, OSPF, OLSR, BMX6, BATMAN-adv, Babel and more. Over time lot of Guifi.net nodes have been build sticking to the older but well known solution named *Modelo Híbrido*. This solution imply lot of manual work and a central provisioning server that know all details about the Guifi.net network, de facto limiting the resilience, the scalability and the growth of Guifi.net. To overcome *Modelo Híbrido* limitations lot of efforts developing new solutions have existed most of them based on mesh technology but none of them, mainly because of resource lack and consequent missing long term support, have been widely adopted causing even more fragmentation and making more difficult interoperability. Even if Libre-Mesh has been bred independently, it is already used in some Guifi.net zones, but like other mesh based solution it is at moment used only for last mile distribution.

2.2.1 Modelo Híbrido

As of *Modelo Híbrido* community network nodes can belong to two categories: *supernodos* and *nodos*, in this model *nodos* are not really nodes of the network as they do not do routing for other nodes of Guifi.net and just attach to a *supernodos*, while *supernodos* do route [2]. In *supernodos* usually a *Mikrotik routerboard* is used as router while externals WiFi AP are configured as plain WDS bridge [3], if two *supernodos* are connected by a WiFi link the router of each node see the router of the other node as if it was directly connected with an Ethernet cable. On top of the routers of each node BGP routing protocol is running, on each node connected BGP peering have to be established, each *supernodo* has his own unique private ASN assigned by the Guifi.net Drupal based network map, in this setup each *supernodo* of Guifi.net act as an AS with typical eBGP information exchange mode.
While BPG seems the more appropriate routing protocol for a neutral Internet exchange point, wireless community networks have

completely different needs that BGP cannot satisfy such as choosing WiFi path with more bandwidth available or automatic link establishment. *Modelo Híbrido* has helped Guifi.net to grow but over time new technology better suited for wireless community have emerged exposing many limitations of this model.

2.3 BIRD

BIRD is an open source implementation of multiple Internet routing protocols in form of a daemon for Unix like systems. It is distributed under the GNU General Public License. BIRD supports IPv4 or IPv6 as separate daemons [4], multiple routing tables [5], and BGP, RIP and OSPF routing protocols, as well as statically defined routes. Its design differs significantly from the better known routing daemons, GNU Zebra and Quagga [6]. Currently BIRD is included in many Linux distributions [6] in particular into OpenWrt [7].
BIRD provide very powerful route filtering framework including a simple C like programming language. There are two objects in this language: filters and functions. Filters are interpreted by BIRD core when a route is being passed between protocols and routing tables. The filter language contains control structures such as `if` and `case`, but it allows no loops. Filter gets the route, can read and write its attributes as needed and at the end, decides whether to pass the changed route through or whether to reject it. BIRD functions are meant to avoid code repetition. Functions can have parameters and they can define local variables. Recursion is not allowed. [8]
BIRD is notorious for his good performances and for his lightweight design because of this it is used in several Internet exchanges as a route server [6], where it replaced Quagga because of its scalability issues [9].

2.4 BMX6

BMX6 is an open-source, mesh routing protocol, developed by Axel Neumann. It originated as an independent branch of the BATMAN routing protocol and is already used successfully in mesh networks [10]. Being a mesh routing protocol, BMX6 distributes node descriptions which can contain node-specific configurations. It optimizes the communication between neighboring nodes for minimal protocol overhead. From a routing protocol point of view BMX6 is classified as **table-driven proactive, distance vector** routing protocol so even if there is no traffic in the network, the routes to all the network nodes are known by all nodes [11], while nodes does not have a global view of the network topology, so their knowledge is limited to next-hop to all destinations and the distance associated to it [12].

2.5 B.A.T.M.A.N. advanced

B.A.T.M.A.N. advanced (often referenced as batman-adv) is an implementation of the B.A.T.M.A.N. routing protocol in form of a Linux kernel module operating on layer 2. While usually wireless routing protocol implementations operates on L3 which means they exchange routing information by sending IP packets and bring their routing decision into effect by manipulating the kernel routing table. Batman-adv operates entirely on L2, not only the routing information is transported using raw Ethernet frames but also the data traffic is handled by batman-adv. It encapsulates and forwards all traffic until it reaches the destination, hence emulating a virtual network switch of all nodes participating. Therefore to the upper level all nodes appear to be link local and are unaware of the network's topology as well as unaffected by lower level network changes. Operating as a L2 routing protocol as there is no *L2 routing table* to manipulate, data frames must be processed directly by batman-adv and doing it in user-space would be very inefficient, as for each packet processed it would be `read()` and `write()` to the kernel and back causing multiple context switching that are computationally costly [13], which limits the sustainable bandwidth especially on low-end devices. To have good support for these devices as well and an overall good efficiency, batman-adv is implemented as a Linux kernel module. This way batman-adv avoid the context switching costs and only introduces a negligible packet processing overhead even under a high load. [14]

2.6 qMp

qMp do facilitate the creation of roaming enabled mesh networks, it is based on OpenWrt and BMX6 [15], qMp already supports configuration provisioning by guifi.net map *unsolclick* for mesh parameters adjusting [16] but it leave the interoperability with *Modelo Híbrido* mostly to manual static configuration done by the user [17] or in rare cases to manually configured Quagga-BMX6 module that is unreliable, resource intensive and requires manual configuration. Moreover qMp is designed around BMX6, in a way that to add support for another routing protocol would imply a big refactoring of qMp code.

2.7 Libre-Mesh

Libre-Mesh (often referenced as LiMe) is an initiative undertaken by community networks members of several continents that collaborate towards a common goal: to develop a set of tools that facilitate the deployment of Free Networks for any community in the world.
Main tool is LiMe meta-firmware: based on OpenWrt, eases the creation of community networks, and enables existing communities to add roaming enabled network clouds to their networks.
To accomplish his goal LiMe meta-firmware have an highly modular design and implementation, the core of LiMe meta-firmware is the lime-system package.

2.7.1 Network Architecture

The LiMe network architecture is a communication network of radio and cabled nodes organized in a mesh-like topology. LiMe network architecture has lot in common to wireless mesh network [18] but it provide further optimization and scalability taking advantage of the layered structure of the Internet protocol suite [19]. The creation of B.A.T.M.A.N. advanced has fostered the debate in community networks if it is better to build the mesh network with L2 routing protocols (like batman-adv) or with L3 routing protocols (like Babel, BMX6, OLSR). LiMe developers recognize the bad and the good parts of both worlds (L2/L3) and have decided to get the best of them. While LiMe let the community choice what protocol they prefer to use the default distribution include both batman-adv and BMX6 operating at same time but with different objectives.

L2 cloud

LiMe developers refers to L2 cloud (or just cloud) as a community network part in which the same broadcast domain is shared. A client connected to a cloud behave like if it is connected on the same Ethernet switch [20] together with all other hosts in the same cloud, all applications that take advantage of being connected to a LAN (as example: for neighbor discovery) can be used seamlessly on a cloud. As the broadcast domain is shared a client can roam without upper layer connectivity interruption across nodes of the same cloud because the assigned IP is still valid and the gateway is still “directly” reachable. On a cloud because all hosts share the same broadcast domain broadcast packets like ARP requests are flooded to the whole cloud [21], this pose limits to the scalability of a cloud, moreover sharing the same broadcast domain mean that typical L2 attacks such as ARP spoofing are possible [22]. Those two drawbacks of operating at L2 limits de facto the dimension of a cloud. Despite LiMe developers suggests to avoid to build too big L2 clouds around the world exists community networks using LiMe with L2 clouds spreading on entire small cities without noticing critical problems.

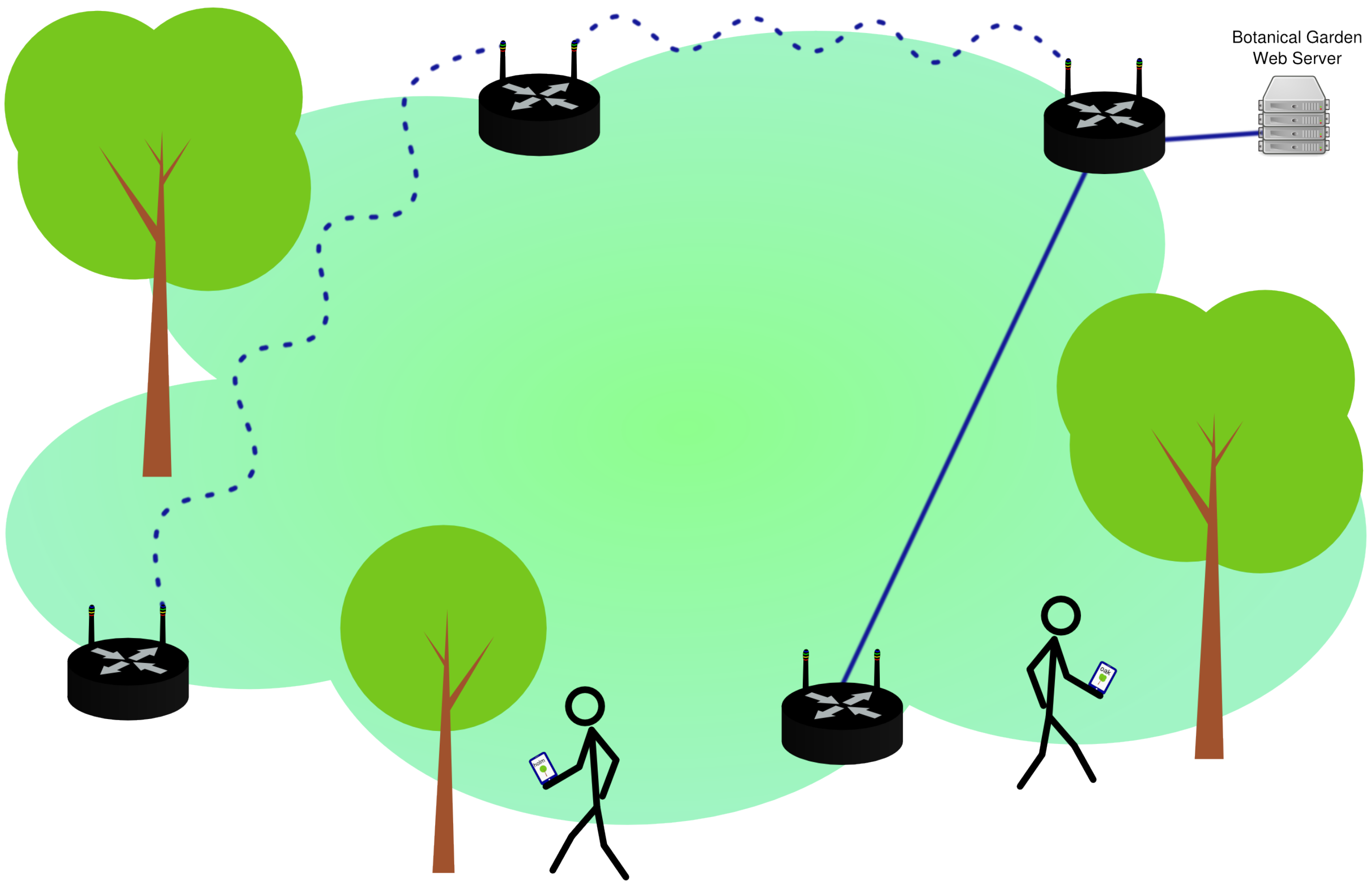


Figure 1: Simplified LiMe L2 cloud schema of Barcelona Botanical garden network.

Legenda

- Black routers: LiMe router that offer WiFi connectivity.
- Blue lines: Mesh links between routers.
- Persons with smartphone: Visitors accessing “Guia Multimèdia del Jardí Botànic” geolocalization capable web application via the WiFi connectivity provided by Libre-Mesh.

L3 network

LiMe developers refers to L3 network (or just network) as a community network part that may contain multiple broadcast domain and in which packet routing between them is done at L3. From the start LiMe has been bred with IPv6 as L3 reference protocol in mind but as of today important applications and a big part of Internet still rely on IPv4 support so LiMe does support both of them implementing a complete dual stack setup. Each broadcast domain must use one or more unique subnets that doesn't overlap with subnets used in other broadcast domains of the same network. Another key concept of LiMe is to build horizontal networks in which there is no discrimination between clients and servers. On a LiMe network every host connected can at same time offer services or be client of services offered by other hosts, because of that bandwidth offered to hosts is symmetric and NAT usage is banned.

Some of NAT drawbacks

- In presence of NAT direct connection between hosts is difficult, to open a connection special techniques are required [23].
- NAT break the end-to-end principle [24][25].
- NAT based network are unfair because the *natted* host lose the opportunity to be a server (listening for incoming connections), and the non *natted* automatically acquire a the privilege of listening for connection that isn't fairly distributed to all hosts anymore.

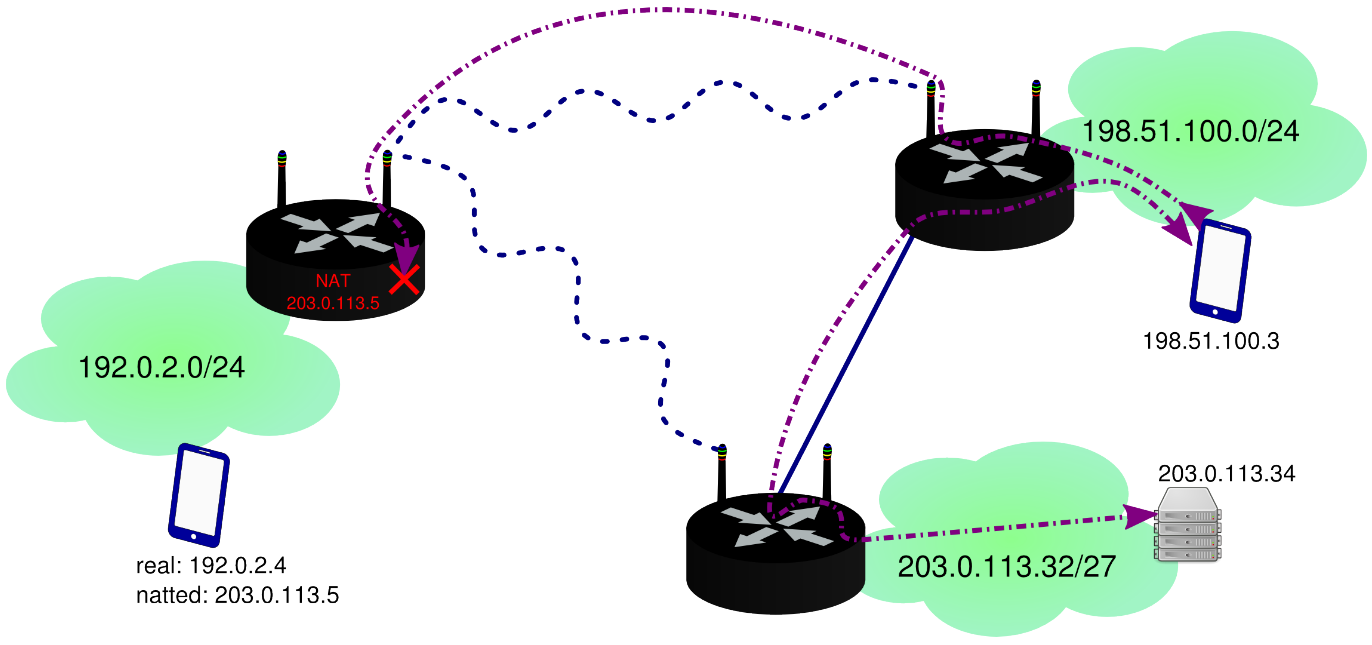


Figure 2: Connection tentative frustrated by NAT in an IPv4 network.

Legenda

- Blue lines: Mesh links between routers.
- Purple directed lines: Upper layer connections.
- Smart-phone 192.0.2.4: It can open connections towards non *natted* hosts like 198.51.100.3, but cannot receive connections because they get frustrated by NAT on his router.
- Smart-phone 198.51.100.3: It can receive connections from all hosts. It can open connections to all hosts but the the one towards the other smart-phone get frustrated by NAT.
- Server 203.0.113.34: Same as smart-phone 198.51.100.3.
- Documentation addresses conform to RFC 5737 [26].

Although the choice of banning NAT from LiMe networks may be imputed to courageous developers it is to be considered as a pragmatic choice too as networks **without** NAT has better performances [27], upper layer applications doesn't have to deal with lower layer asymmetry introduced by NAT, and network debugging is way simpler.

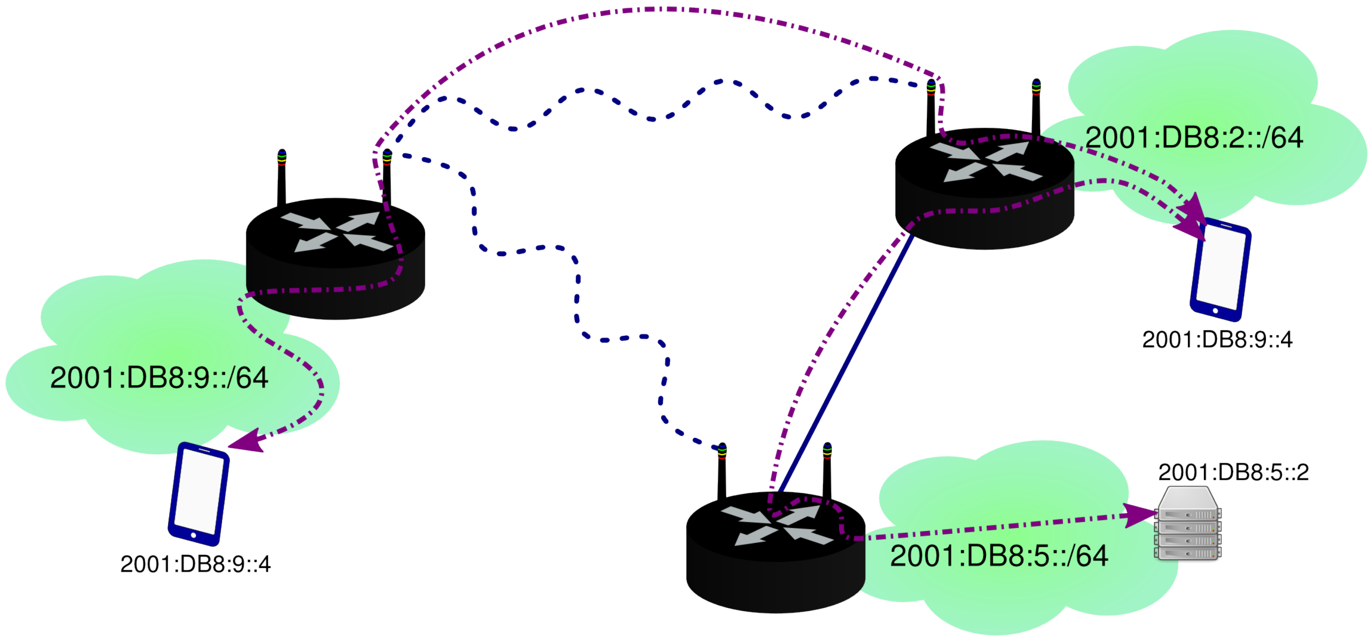


Figure 3: Connection tentative between clients works seamlessly in an IPv6 network without NAT.

Legenda

- Blue lines: Mesh links between routers.
- Purple directed lines: Upper layer connections. All hosts con open and receive connections each others.
- Documentation addresses conform to RFC 3849 [28].

Completing the puzzle

While in LiMe L1 isn't used for packet forwarding, L2 clouds show their limits as they grow oversize and L3 networks feel an unnecessary complication when they shrink too much. LiMe try to take the best from all of them. LiMe developers suggests to extends L2 clouds till the zone keep having homogeneous requirements, so it can be handled with common policies, moreover the probability to suffer a L2 cloud split must remain low enough, another factor to keep in mind is that if a L2 cloud grow too much broadcast packets will start to be significants bandwidth consumers and this is usually unwanted. Typical suggested L2 clouds can spread across areas like a school, a public garden or a small village. L2 clouds doesn't have mechanisms to connect each other by them self, at this point L3 network start to play his main role, to connect L2 cloud neighborhood each other. While hosts from different cloud can connect each others, as the broadcast domain is split broadcast traffic doesn't propagate. L3 networks moreover can afford splits and rejoins seamlessly. All of this that can seems some kind of "network magics" is actually done automatically by LiMe and in a quite understandable way thanks to its modular network and software architecture.
Let's dive into it from the ground up:

L1

At L1 all LiMe nodes of a network, including all clouds pertaining to it, can connect each other if in range, in the case of WiFi Ad Hoc this is obtained by using the same BSSID, in case of cabled Ethernet without further work other then connecting the LiMe nodes to the same switch and in case of WiFi ieee80211s it is obtained using the same `mesh ID`.

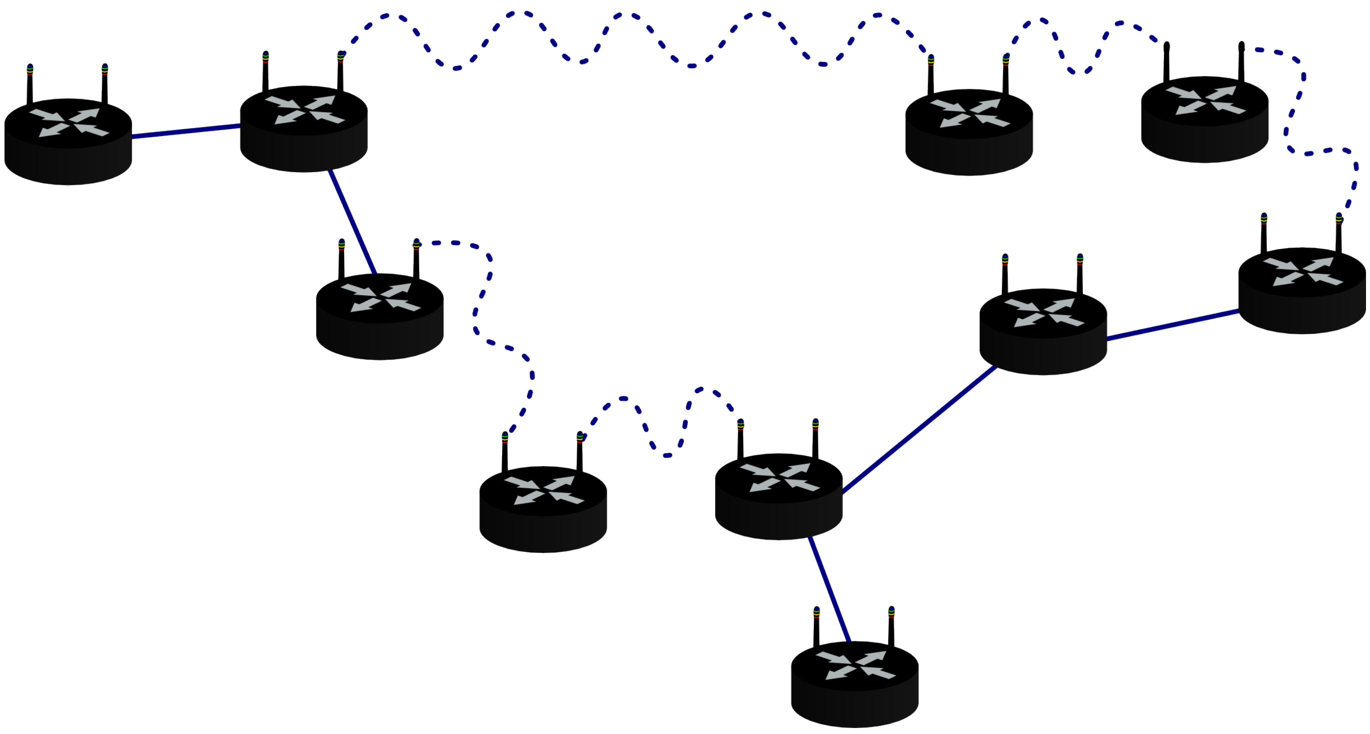


Figure 4: LiMe nodes from different clouds connecting each other at L1.

Legenda

- Black routers: LiMe routers on multiple clouds.
- Blue lines: L1 links between routers.

L1 + L2

Each cloud run L2 routing protocol (by default batman-adv) on top of a different 802.1ad VLAN, so although at L1 different clouds can overlap and connect each other there is no visibility from batman-adv point of view, this creates different broadcast domains also for hosts connected to different clouds.

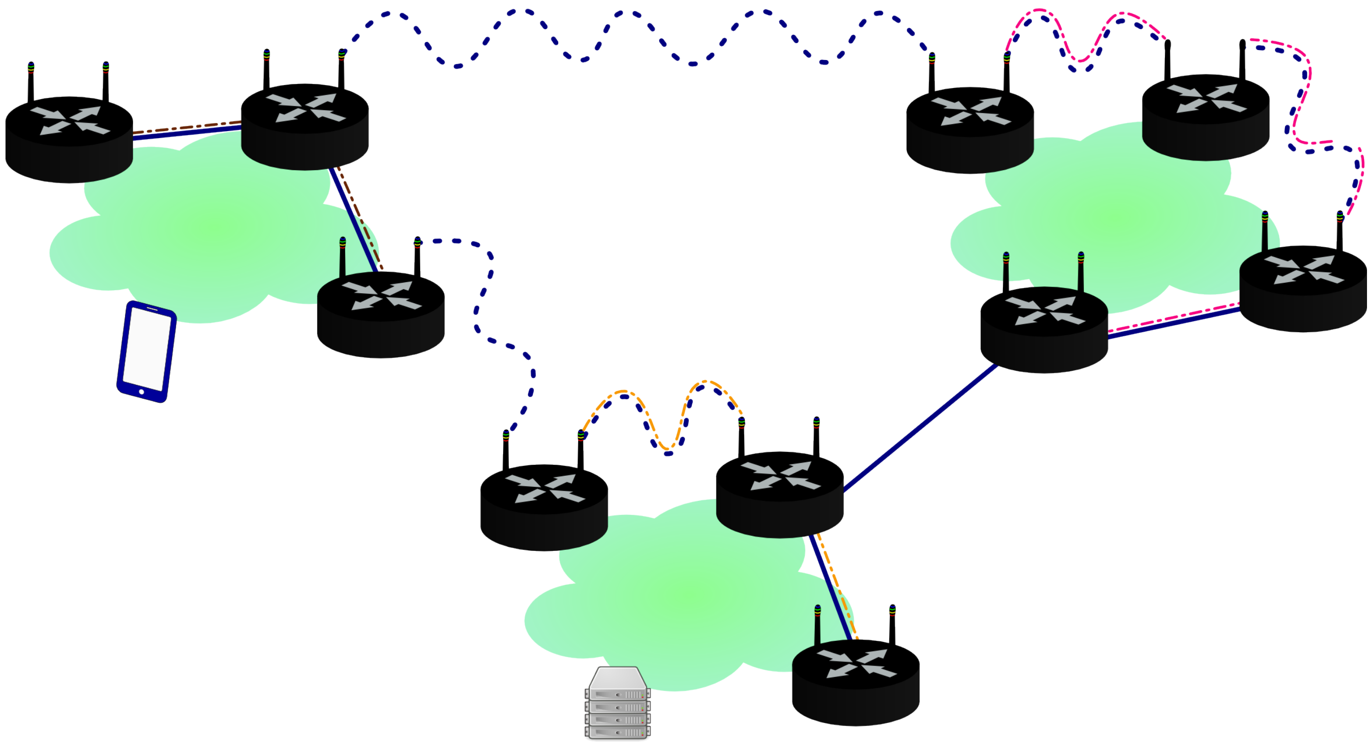


Figure 5: LiMe nodes from different clouds connecting each other at L1 but not at L2.

Legenda

- Black routers: LiMe routers on multiple clouds.
- Blue lines: L1 links between routers.
- Brown lines: L2 links of cloud 1.
- Orange lines: L2 links of cloud 4.
- Fuchsia lines: L2 links of cloud 9.

L1 + L3

All nodes run the L3 routing protocol (by default BMX6) on the same 802.1ad VLAN, so where there is a L1 link a L3 link exists too. This permits different clouds to automatically connect each other.

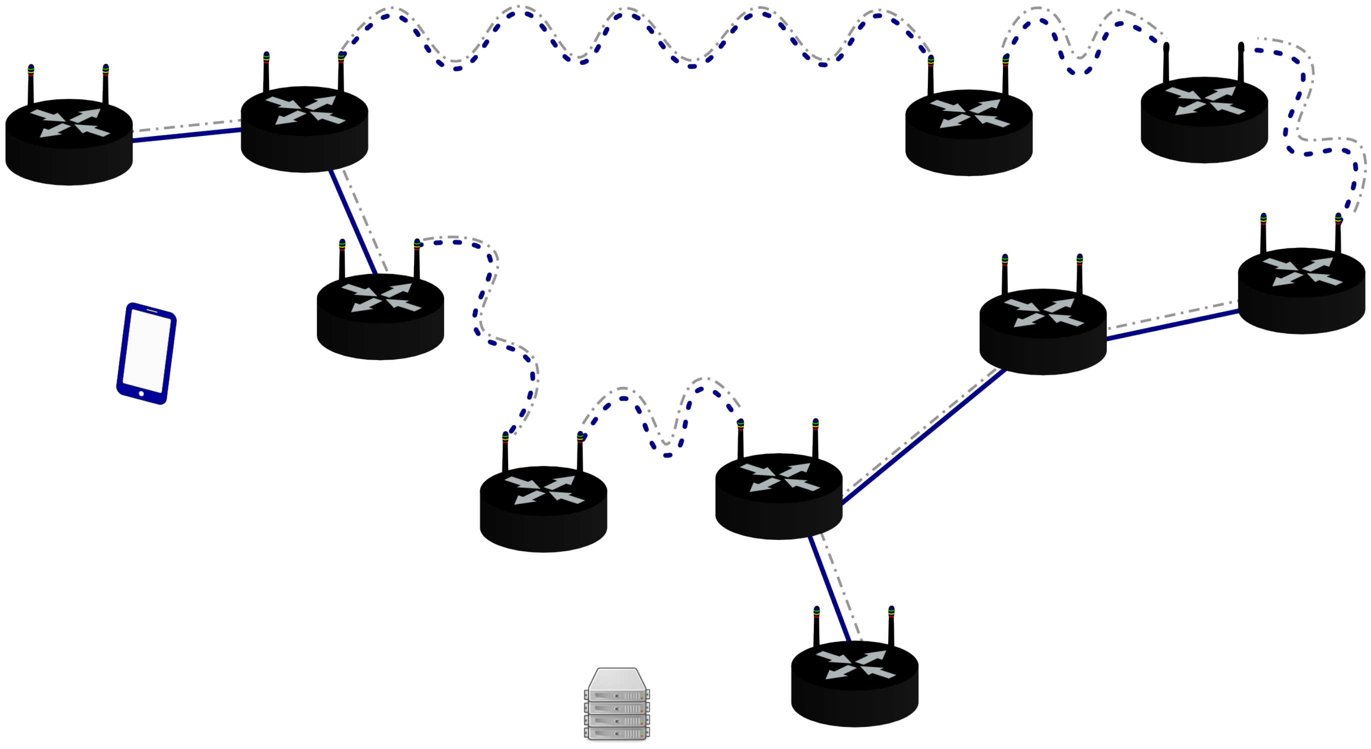


Figure 6: LiMe nodes from different clouds connecting each other at L1 and L3.

Legenda

- Black routers: LiMe routers on multiple clouds.
- Blue lines: L1 links between routers.
- Grey lines: L3 links between routers.

L1 + L2 + L3

Joining all pieces together we obtain what usually a community network using LiMe looks like. A bunch of roaming enabled clouds that route internal traffic at L2 and use L3 for communication outside the cloud.

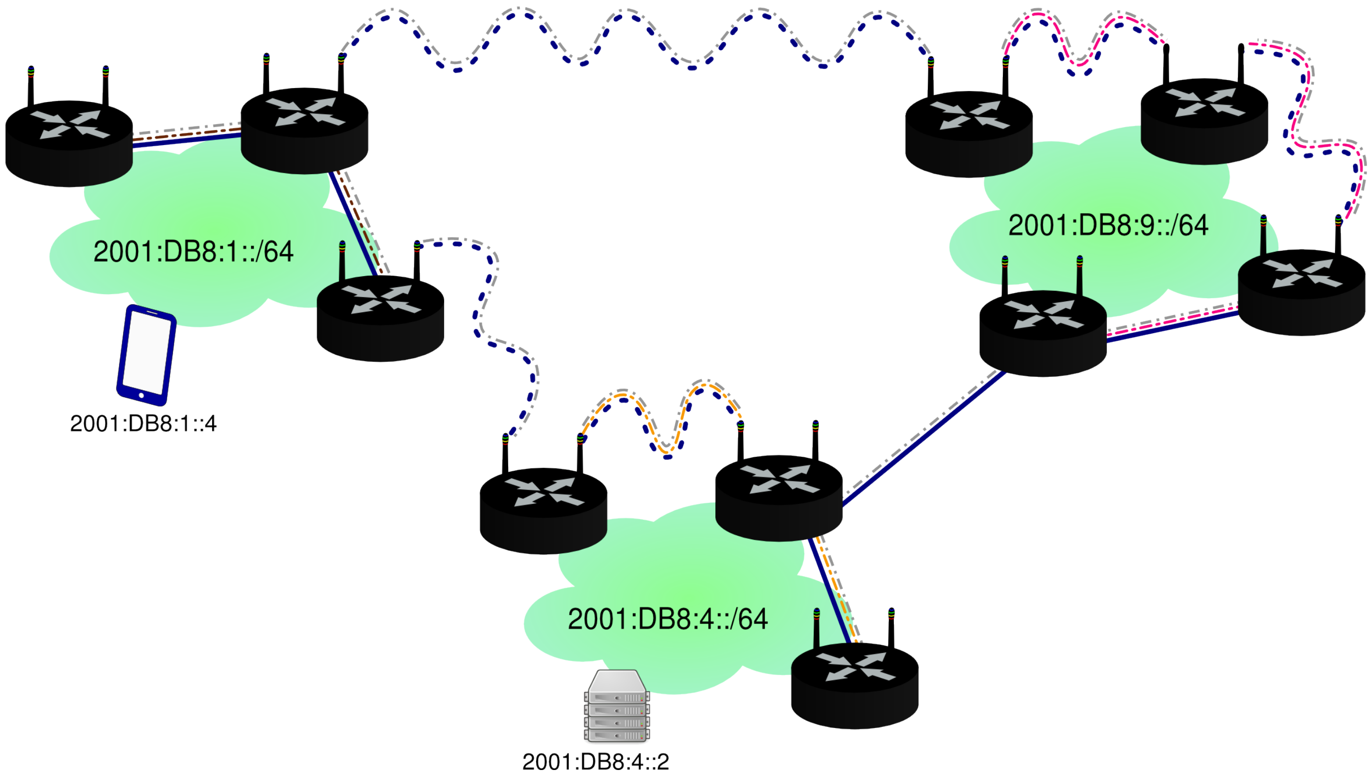


Figure 7: LiMe community network detailing links at L1, L2 and L3.

Legenda

- Black routers: LiMe routers on multiple clouds.
- Blue lines: L1 links between routers.
- Brown lines: L2 links of cloud 1.
- Orange lines: L2 links of cloud 4.
- Fuchsia lines: L2 links of cloud 9.
- Grey lines: L3 links between routers.

L1 + L2 + L3 over L2

A further yet unexplored setup is to automatically enable L3 routing only on nodes that connects multiple cloud and take advantage of the L2 cloud to exchange L3 information instead of use all L1 links, intuitively this could improve scalability but at least a prototype implementation and some empirical data is needed before feasibility could be evaluated.

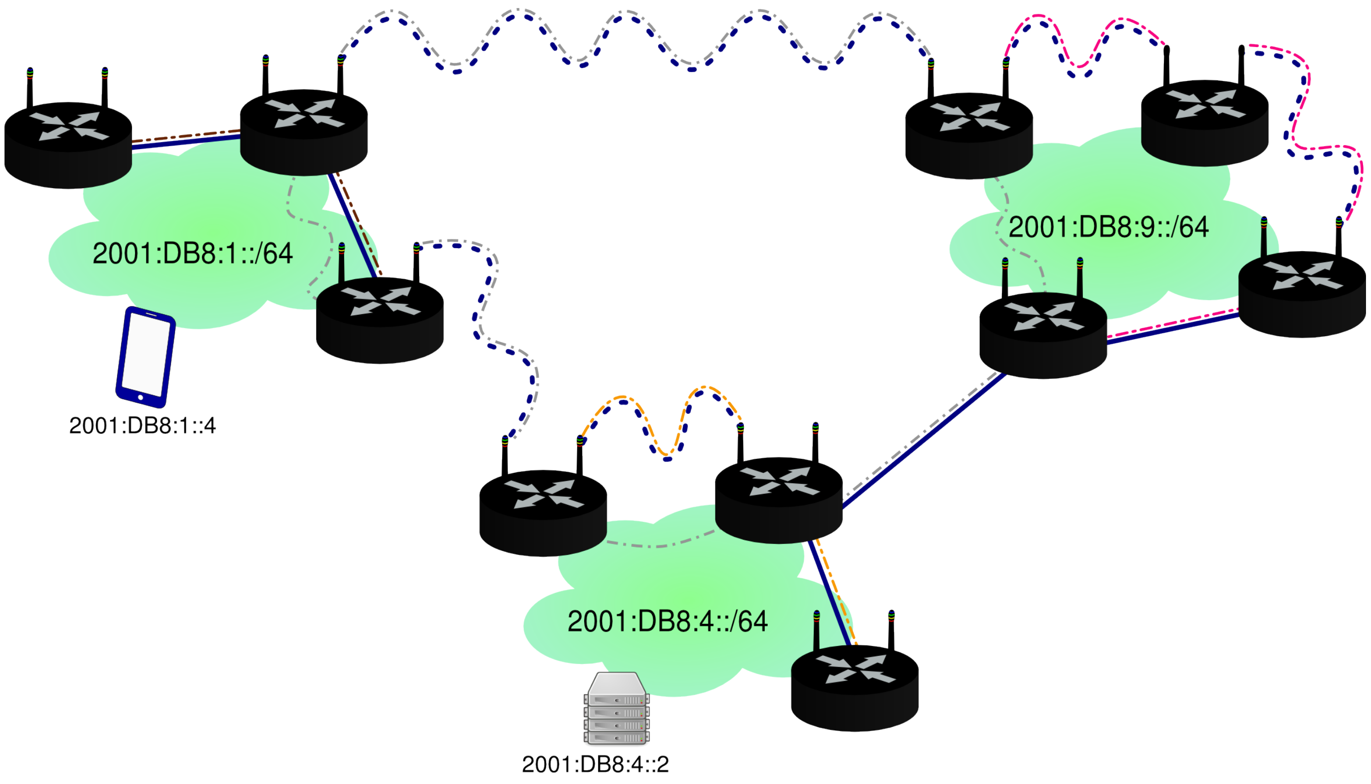


Figure 8: LiMe community network detailing links at L1, L2 and L3 over L2.

Legenda

- Black routers: LiMe routers on multiple clouds.
- Blue lines: L1 links between routers.
- Brown lines: L2 links of cloud 1.
- Orange lines: L2 links of cloud 4.
- Fuchsia lines: L2 links of cloud 9.
- Grey lines: L3 links between "frontier" routers.

2.7.2 Software Architecture

LiMe meta-firmware is structured as a set of OpenWrt packages, it is licensed under the GNU Affero General Public License v3 and written in Lua programming language in a custom object-oriented style [1], the choice of Lua is not a matter of tastes but of convenience, many of OpenWrt libraries are written in Lua or provides binding to it, moreover Lua interpreter fits in few kilobytes [29] that is very important working with embedded devices, while it is particularly suitable for scripting it offer elegant multi-paradigm without making it bloated or unreadable. LiMe joint effort is not the first time multiple communities try to join forces to create a common firmware but the modular structure and the usage of Lua have determined its success adapting

[1] In this custom object-oriented style objects are Lua tables. Object's members (attributes and operations) are table entry with the name of the member as key in the table. Objects may implement some abstract class. Abstract classes aren't written in code but specified in documentation. There is no check of conformance with abstract class specification.

to different wireless communities needs without loose code readability, in contrast to previous efforts that where not modular or tended to have mostly unreadable code written for Almquist shell and other ones that tried to use C but ended up writing more code to deal with memory management then for the original objective, LiMe meta-firmware as of today offer more than thirty OpenWrt packages structured and written in a comprehensible manner. LiMe meta-firmware provides a command line tool `lime-config` that when invoked simply delegate the work of updating the configurations to installed LiMe modules by calling their `clean()` and `configure()` methods, LiMe minimal installation already ships necessary modules but the architecture is meant to be extensible so more modules can be added if needed. LiMe modules are often modular themselves, an UML scheme may help the comprehension of LiMe architecture as a whole.

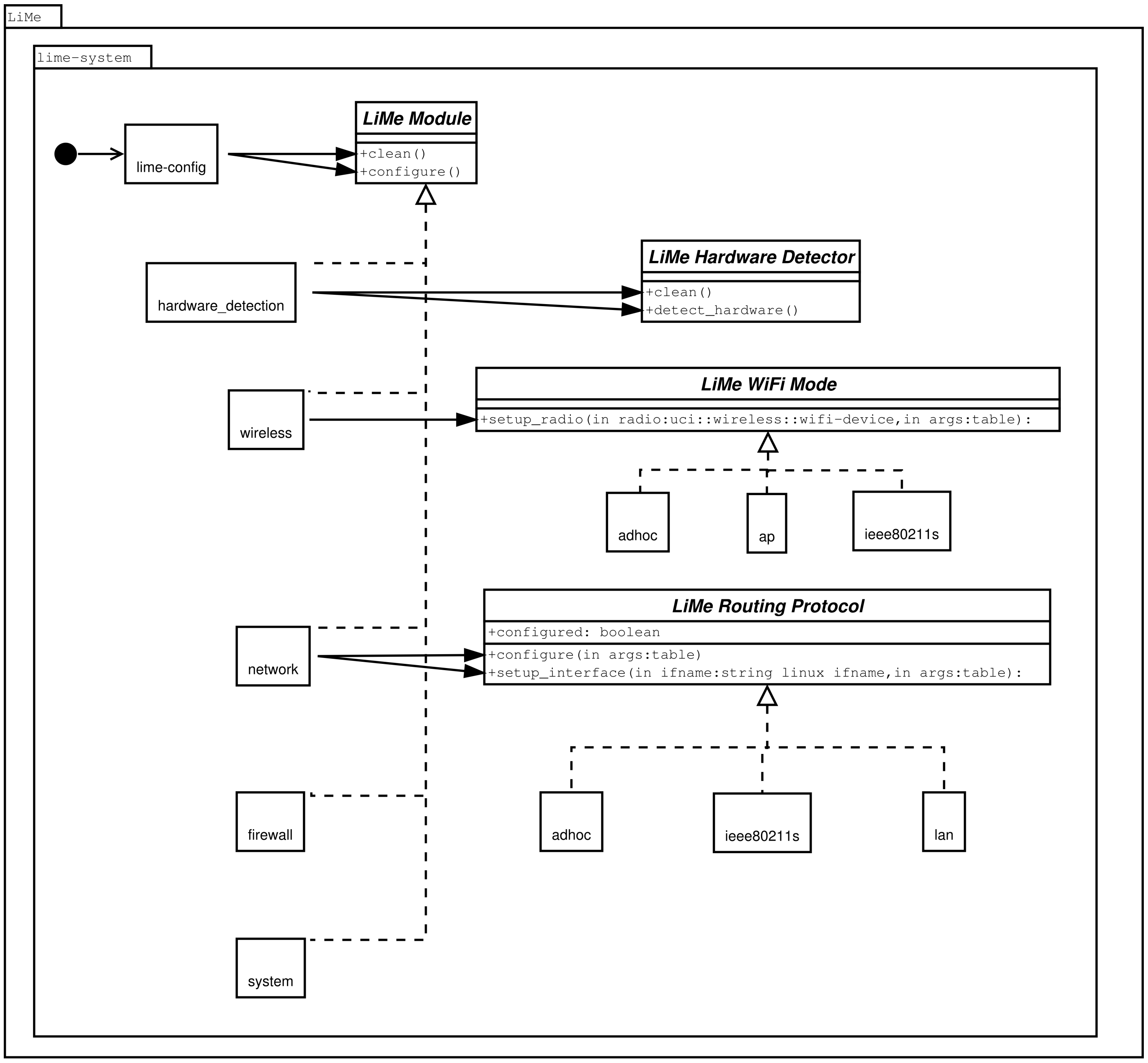


Figure 9: UML diagram of lime-system package.

Module: hardware detection

Hardware detection module is in charge of detecting hardware components of the router and if some of them need a special configuration in order to work according to LiMe needs generate that configuration, to accomplish its task as you cas see in

Figure 9 it is modular itself and it's submodules are called `detectors`. For each `detector` installed (such as `lime-hwd-usbradio`) it execute the respective `clean()` method that is supposed to cleanup outdated configuration and the `detect_hardware()` method that is supposed to configure the hardware and can write specific `config` sections in `/etc/config/lime` to be read by others lime components if necessary.

Module: wireless

Wireless module detect WiFi interfaces and prepare them for usage with LiMe, to accomplish its task it reads and write to `/etc/config/lime` and `/etc/config/wireless`, as depicted in Figure 9 it is modular itself and delegate part of the works to his submodules called `mode`. Each `mode` must implement a method called `setup_radio(radio, args)` that take as argument an `uci.wireless.radio` table and optional adjuntive arguments and is in charge of configure the WiFi radio to operate as needed by LiMe.

Module: network

Network module is in charge of configuring network interfaces and routing protocols, it is a big task to accomplish in a nifty way to do it is modular too. It offers a tight API to his modules that are called `proto` (such as `lime-proto-bgp`) while they must expose to network module a statefull interface that consists of:

- `configured` flag, it is false by default and setted to true when the `proto` is configured.
- `configure(args)` this method is in charge of configuring the underlying routing protocol, args is an array containing all parameters found in configuration file, parameters have positional meaning.
- `setup_interface(ifname, args)` this function is called once per each networking interface found on the router `ifname` is the Linux name of the interface and `args` is an array containing all related parameters found in configuration file.
- `apply()` usually just restart the underlying routing daemon to make the new configurations effective.

Module: firewall

Configure OpenWrt firewall if installed or iptables and ebtables according to LiMe needs.

Module: system

Configure system general settings like host-name and other miscellaneous stuff.

Hardware detector: ground-routing

Ground routing is a node setup similar to *Modelo Híbrido* in the sense of splitting routing and geographic linking task into different devices. In this setup a device running LiMe is in charge of routing while other devices justs deal with physic and link layer stuff. Those devices can keep their original fabric firmware, or being flashed with a plain OpenWrt image or even run a stripped version of LiMe. The name “ground routing” came from the habit of routing enthusiasts, to keep the L1/L2 devices on which they do not usually change the configurations on the roof (necessary to have good WiFi links) while the “ground router”, which is the subject of their experiments, is kept inside the house, “on the ground”, so if the configuration changes accidentally disrupt connectivity to the router, physical access is simple and there is no need to access the roof for recovery. As this functionality is not needed in default setup `lime-hwd-ground-routing` is distributed as a separate package and is not included in the default LiMe distribution.

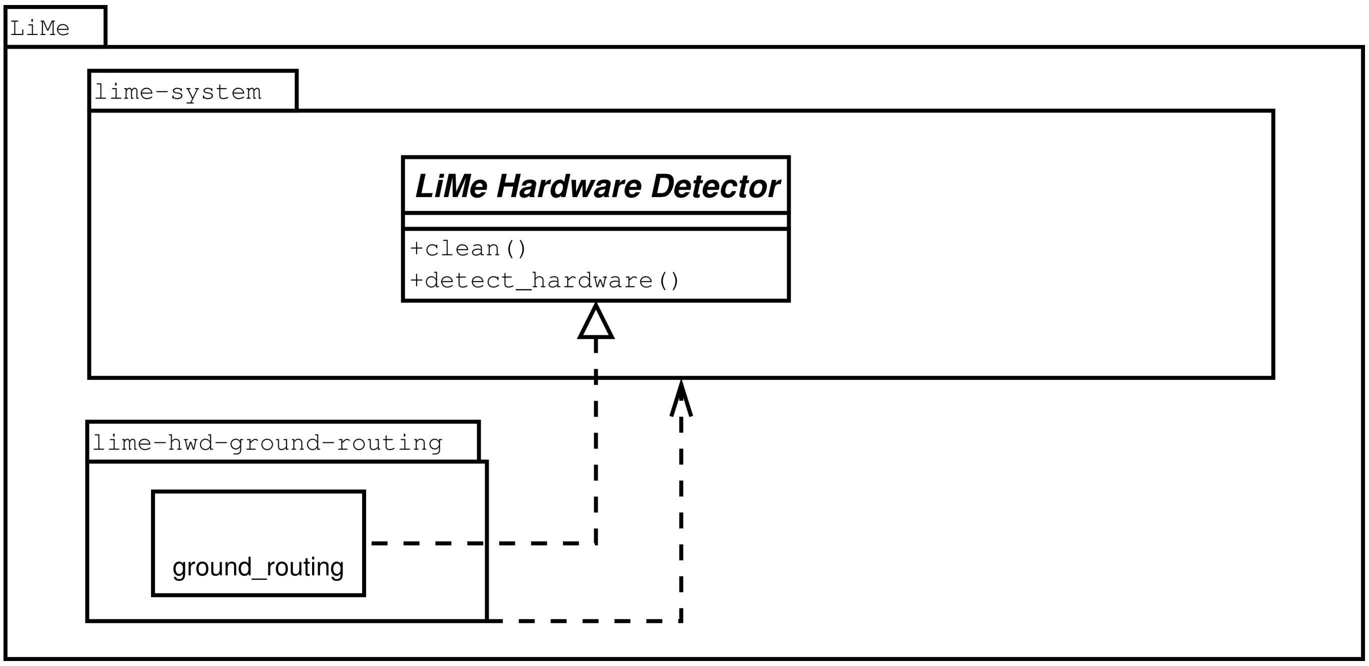


Figure 10: UML diagram of lime-hwd-ground-routing package.

When invoked by `lime.hardware_detection`, `lime.hwd.ground_routing.clean()` clean outdated ground routing sections, then `lime.hwd.ground_routing.detect_hardware()` creates ground routing configuration sections that eventually can include 802.11q interfaces or hardware switch configurations [30][31], according to node owner needs and hardware capabilities. In contrast to *Modelo Híbrido* and Ninix.org *Routing a terra* that are meant for a specific routing protocol (BGP the first [3], and OLSR the latter [32]) LiMe Ground Routing is routing protocol agnostic, thanks to the LiMe modular structure it is focused just on preparing the L1/L2 setup on the LiMe router and then the upper layer routing modules treat L2 interfaces created by Ground Routing the same as any other L2 interface [33]. This permit to LiMe networks to use any routing protocol on top of a Ground Routing setup and not being limited to a specific routing protocol, thanks to this abstraction LiMe can also “emulate” Ninix.org *Routing a terra* just installing both `lime-hwd-ground-routing` and `lime-proto-olsr`, and Guifi.net *Modelo Híbrido* just installing both `lime-hwd-ground-routing` and `lime-proto-bgp` that is part of the work done during this internship.

Hardware detector: openwrt-wan

This hardware detector is shipped as an optional package `lime-hwd-openwrt-wan`, but is installed and enabled by the default in LiMe distribution.

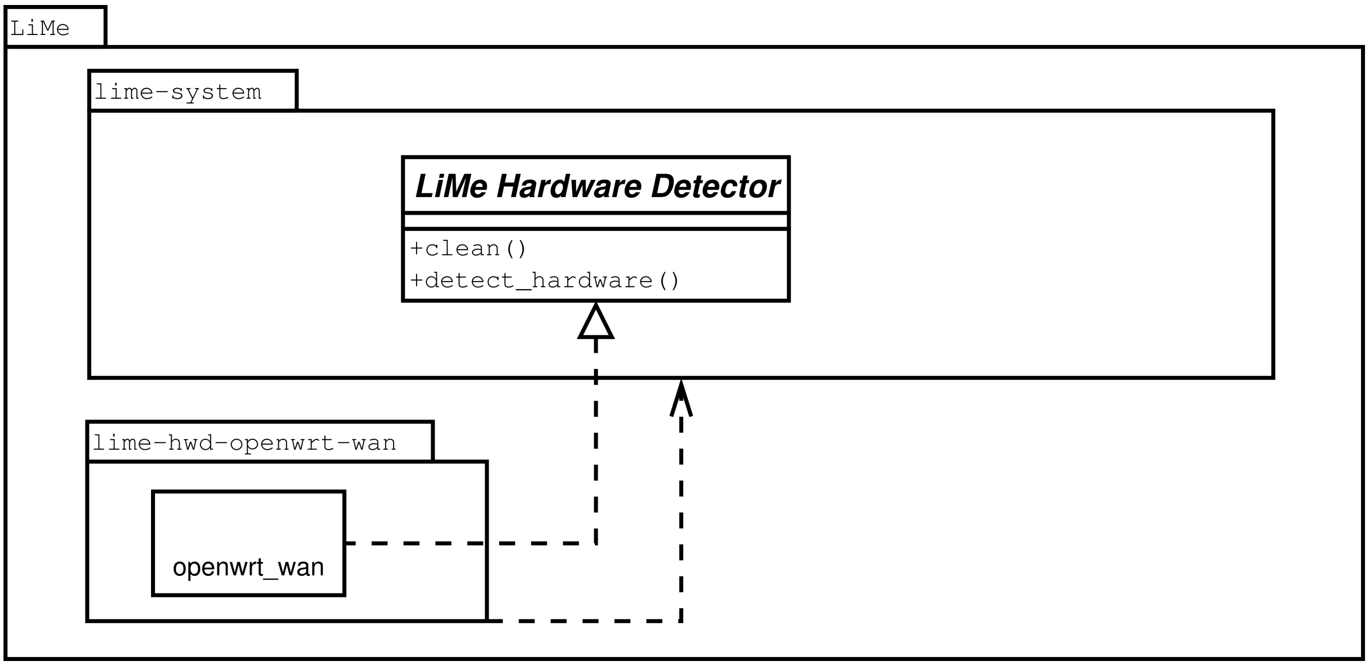


Figure 11: UML diagram of lime-hwd-openwrt-wan package.

Low cost consumer hardware, that community networks often use, have multiple Ethernet ports, although from a technical point of view those Ethernet ports are pretty much equivalent each other, vendors usually present them as different to the end user, in particular there is usually a separated port labeled WAN end the other ports usually grouped together labeled LAN [34]. The end user is told by the router vendor that the WAN port is meant to be connected to the Internet connection and the LAN ports to be connected to local computers. To avoid confusing the user this convention is usually respected both in OpenWrt and in LiMe, although because technically those ports are equivalents there is no automatic way to determine which port is WAN by software. OpenWrt developers when they add support and write the default configuration for a specific device they let the port labeled as WAN to auto-configure via DHCP while they configure as static the ones labeled as LAN by the vendor [35]. LiMe does support almost every device that is well supported by OpenWrt so, instead of duplicating the work of determine if there is a WAN labeled port and which port is it, delegate this task to `lime-hwd-openwrt-wan` that check which is the WAN port in the OpenWrt default configuration that is backed up into `/etc/lime/config.original/` by LiMe at first boot [36] and configure it as needed by LiMe and to behave as user expect to.

lime.hwd.openwrt_wan.detect_hardware() snippet that retrieves WAN ifname.

```
local uci_o = libuci:cursor()
uci_o:set_confdir("/etc/lime/config.original")
uci_o:set_savedir("/dev/null")
local ifname = uci_o:get("network", "wan", "ifname")
```

Hardware detector: usb-radio

This hardware detector is distributed as an optional package `lime-hwd-usbradio`, it is in charge of detecting WiFi USB radio.

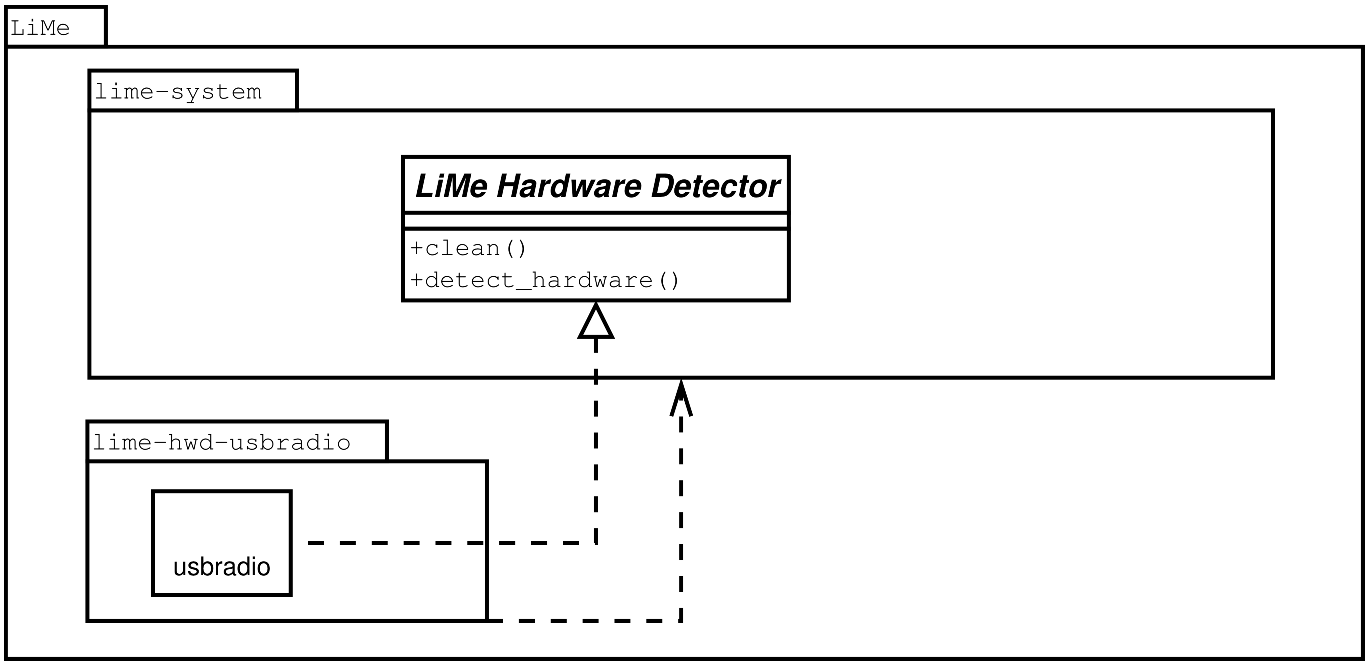


Figure 12: UML diagram of lime-hwd-usbradio package.

Most consumer routers features one or more USB ports, to setup print servers or share files from an attached disk. With OpenWrt those ports are usually usable as full USB and not only for specific tasks. Taking advantage of this community networks researcher started creating low cost multi-radio routers by plugging USB WiFi radios into those routers. Although OpenWrt configuration system doesn't recognize those USB radio automatically so the users had to configure them manually. To fill the gap between OpenWrt wireless configuration system and having USB radio automatically working `lime-hwd-usbradio` has been developed. As OpenWrt is compatible with Hotplug system [37] when an USB radio is connected to the router an Hotplug event is fired [38], `lime.hwd.usbradio` catch that event trough a crafted Hotplug hook [39] and configure the USB radio taking in account his capabilities. In the long run the lack of good quality support for `adhoc` or `ieee80211s` WiFi modes in Linux drivers for USB radios especially for the cheaper ones has discouraged users to use them for mesh links. Currently USB radios detected by `lime-hwd-usbradio` are mainly used in AP or STA WiFi modes.

Protocol: adhoc

This protocol is shipped as part of `lime-system` package as you can see in Figure 9 and it's enabled by default, it is charge of just generating the network configuration that bring up L2 interfaces corresponding to L1 radio interfaces configured as of *LiMe WiFi Mode* `adhoc`.

Protocol: anygw

This protocol is shipped as an optional package `lime-proto-anygw`, but is installed and enabled by the default in LiMe distribution.

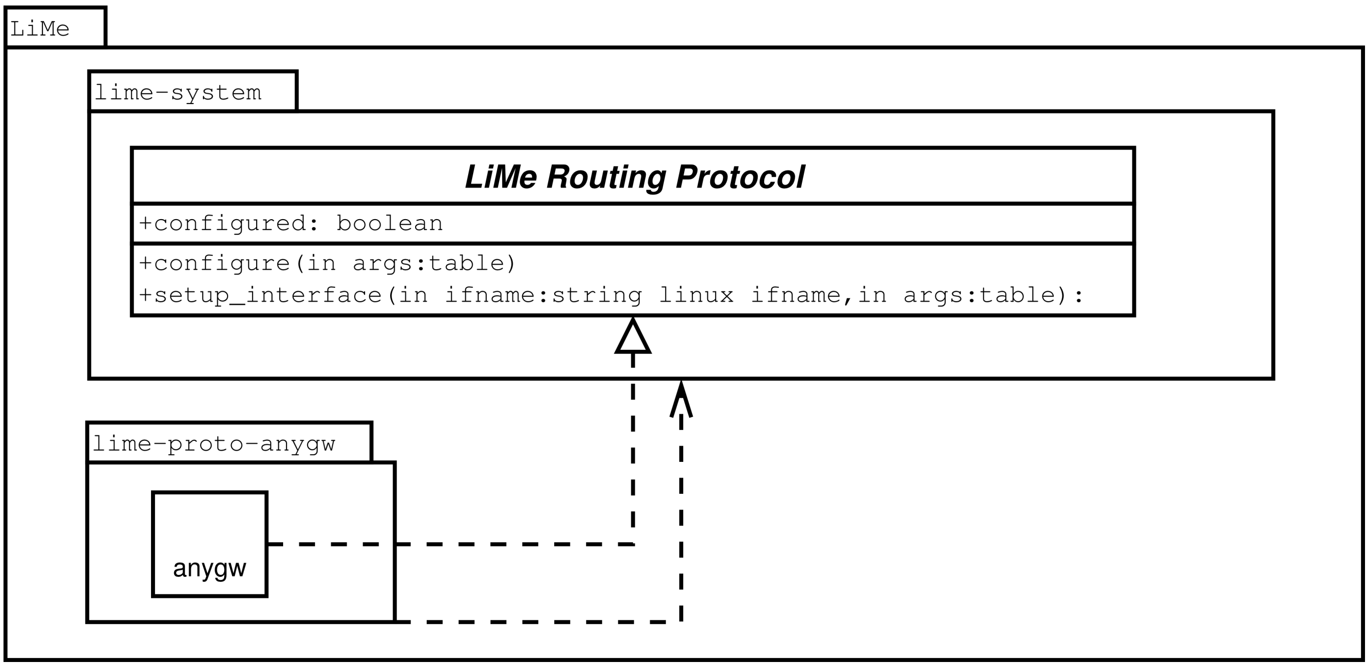

LiMe
lime-system
LiMe Routing Protocol
+configured: boolean
+configure(in args:table)
+setup_interface(in ifname:string linux ifname,in args:table):
lime-proto-anygw
anygw

Figure 13: UML diagram of lime-proto-anygw package.

In WiFi networks usually when a client move and get far from the AP it is associated, if it sense better signal from another AP with the same SSID it is expected to automatically disassociate from the far node and associate to the near one. When during this process upper network layers, specifically the user, doesn't experience a connection interruption it is commonly referred as roaming [40]. LiMe mesh networks does supports roaming at L2. In L2 roaming scenarios, because the clients are usually configured via DHCP and it operates at an higher layer then roaming, the client keep using the same gateway of when it associated first time without caring of it's position, this behavior may cause client's packets be routed in a suboptimal way, moreover if the client keep roaming far away from the initial gateway it may experience low network performances.

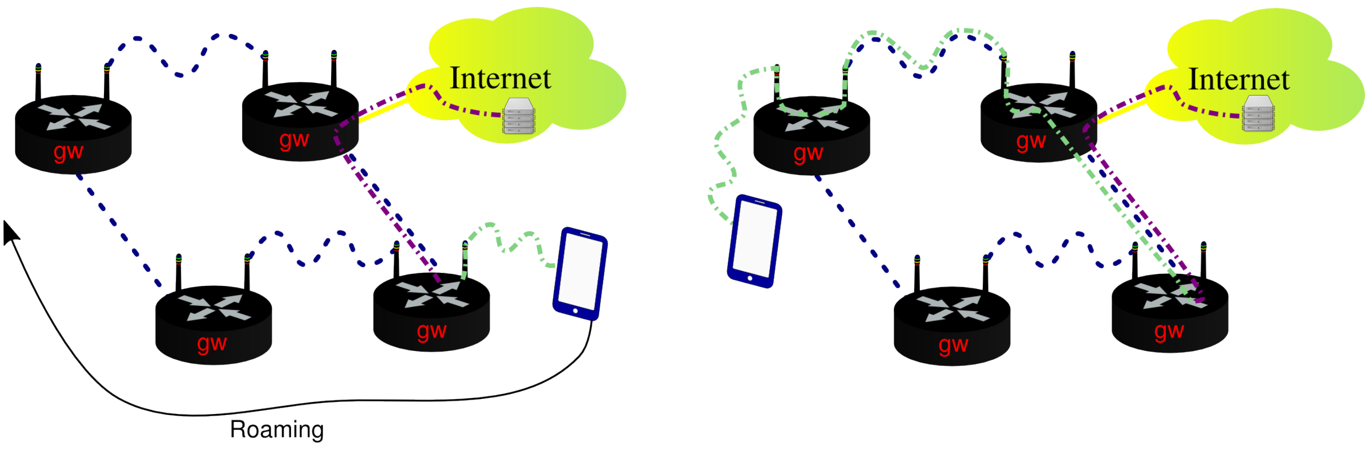


Figure 14: Roaming without anygw.

Legenda

- Blue dashed lines: Mesh links
- Purple dot dashed line: Client packets routed at L3
- Green dot dashed line: Client packets routed at L2

To avoid this unwanted behavior `lime-proto-anygw` offer a light implementation of anycast gateway. Anygw builds the virtual Ethernet interface `anygw` on top of LAN protocol client's access interface `br-lan` using the Linux kernel module `macvlan` [41][42]. When anycast gateway is enabled the first address of the mesh network subnet is reserved for anycast gateway operation, together with the IP also a crafted locally administered but valid **unicast** MAC address [43] is used, all anycast gateways on a LiMe L2 cloud share the same IP and MAC address.

anygw addresses generation.

```
local ipv4, ipv6 = network.primary_address()

local anygw_mac = "aa:aa:aa:aa:aa:aa"
local anygw_ipv6 = ipv6:minhost()
local anygw_ipv4 = ipv4:minhost()
```

In this way when a client sends a packets directed outside it's cloud it gets "intercepted" and optimally routed at L3 by the nearest router.

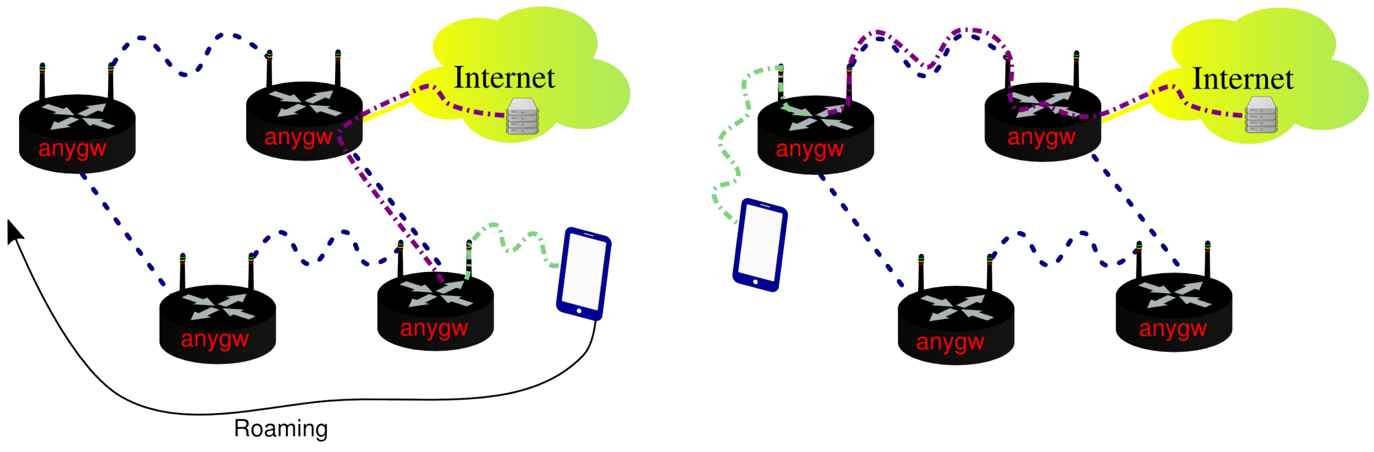


Figure 15: Roaming with anygw.

Legenda

- Blue dashed lines: Mesh links
- Purple dot dashed line: Client packets routed at L3
- Green dot dashed line: Client packets routed at L2

Protocol: batadv

This protocol is shipped as an optional package `lime-proto-batadv`, but is installed and enabled by the default in LiMe distribution. It configures batman-adv to create LiMe L2 clouds, and setup 802.1ad VLAN as needed by LiMe setup.

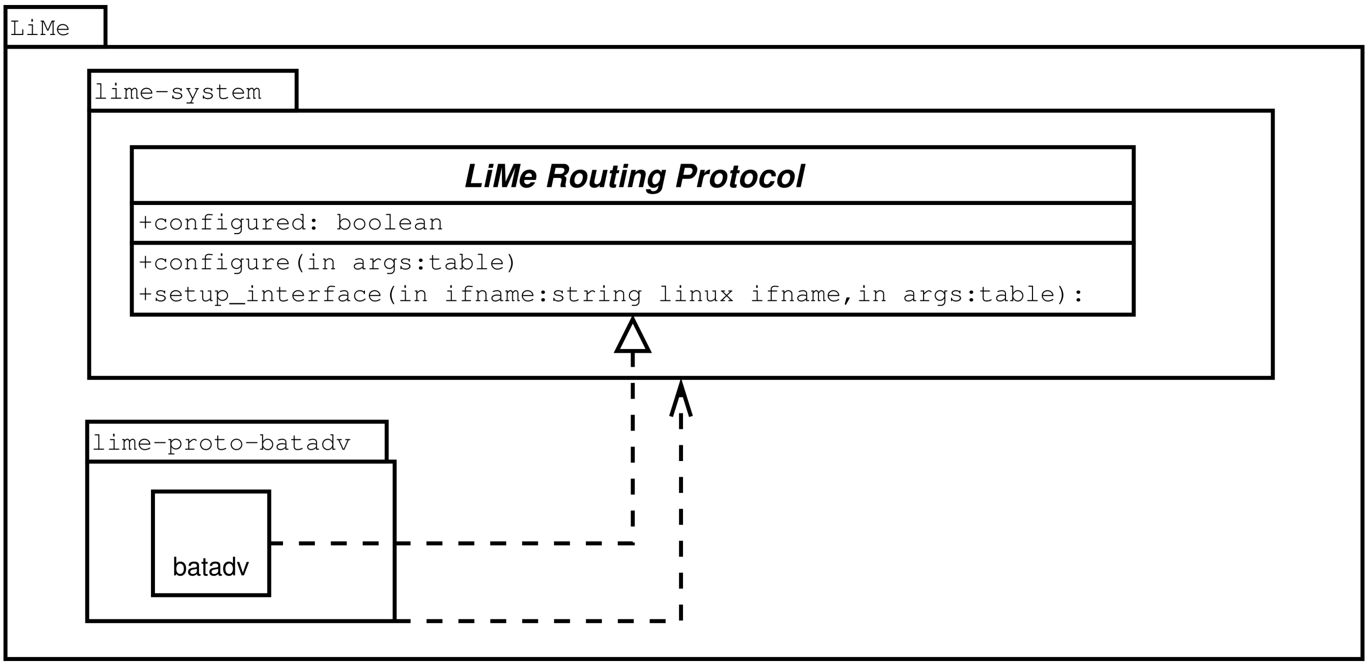

LiMe
lime-system
LiMe Routing Protocol
+configured: boolean
+configure(in args:table)
+setup_interface(in ifname:string linux ifname,in args:table):
lime-proto-batadv
batadv

Figure 16: UML diagram of package lime-proto-batadv.

Protocol: bmx6

This protocol is shipped as an optional package `lime-proto-bmx6`, but is installed and enabled by the default in LiMe distribution. It configures BMX6 for L3 routing between LiMe L3 networks and eventually with upstream connection, it also setup 802.1ad VLAN for BMX6 as needed by LiMe setup.

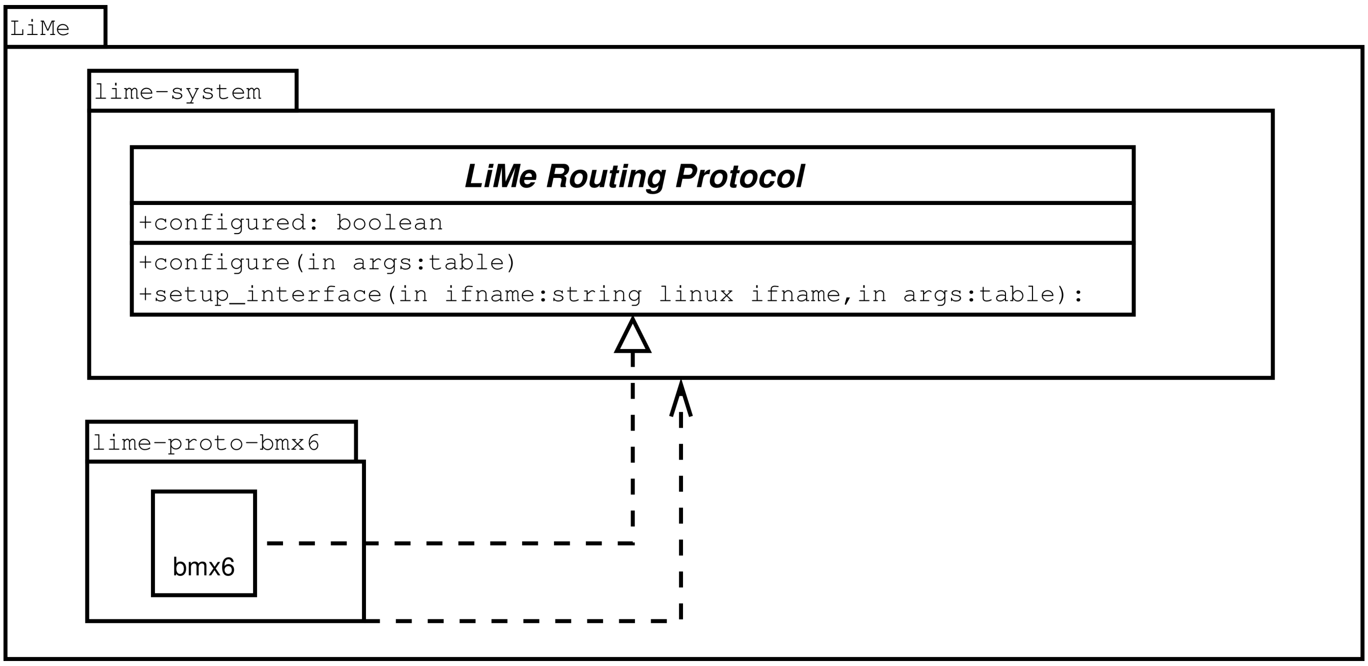

LiMe
lime-system
LiMe Routing Protocol
+configured: boolean
+configure(in args:table)
+setup_interface(in ifname:string linux ifname,in args:table):
lime-proto-bmx6
bmx6

Figure 17: UML diagram of package lime-proto-bmx6.

Protocol: ieee80211s

This protocol is shipped as part of `lime-system` package as you can see in Figure 9 but it is not enabled by default, `lime.proto.ieee80211s` is a relatively new born into LiMe family and the similarly used `lime.proto.adhoc` is preferred as default configuration because it has been more tested and is well known to community networks participants. Similarly to his older brother `lime.proto.adhoc`, `lime.proto.ieee80211s` is charge of just generating the network configuration that bring up L2 interfaces corresponding to L1 radio interfaces configured as of *LiMe WiFi Mode* `ieee80211s`.

Protocol: lan

As you can see in Figure 9 `lime.proto.lan` is shipped as part of `lime-system` package. LiMe LAN protocol is enabled by default and is meant to configure the network local to the node, in the default configuration it is also in charge of giving access to the mesh network at layer 2 to clients next to the node. It configure the node main IPv4 and IPv6 and bridge all network interfaces meant to give access to clients like wireless AP interfaces and Ethernet interfaces into `br-lan`.

WiFi mode: adhoc

As you can see in Figure 9 `lime.mode.adhoc` is shipped as part of `lime-system` package. LiMe WiFi mode adhoc is enabled by default and configures WiFi interfaces in adhoc mode that are used by upper layer protocols to establish links between LiMe nodes.

WiFi mode: ap

As you can see in Figure 9 `lime.mode.ap` is shipped as part of `lime-system` package. LiMe WiFi mode ap is enabled by default and configures WiFi interfaces as an AP. Those interfaces in default configuration are used by upper layer protocols to provide access to clients, but it is possible to use them for creating links too.

WiFi mode: ieee80211s

As you can see in Figure 9 `lime.mode.ieee80211s` is shipped as part of `lime-system` package. LiMe WiFi mode ieee80211s is not enabled by default as `lime.mode.adhoc` is generally preferred because it has been more tested over time. LiMe WiFi mode ieee80211s configures WiFi interfaces in ieee80211s mode but with `mesh forwarding` disabled, so ieee80211s is used only as link layer and not for packet forwarding that is delegated to upper layer protocols that take advantage of the WiFi interfaces created by `lime.mode.ieee80211s` to establish links.

3 Developed tools

3.1 lime-system

Even if LiMe is highly modular, to have a sane and modular itself BGP route exchange implementation required some modification to LiMe core, in particular the abstract class *Lime Routing Protocol* has been extended by adding the `bgp_conf(...)` method. A *Lime Routing Protocol* which want to exchange routes with BGP should implement that method in a way that it does the necessary configuration of the `proto` itself and returns to the caller a string containing a BIRD snippet configuration for route exchange with that `proto`.

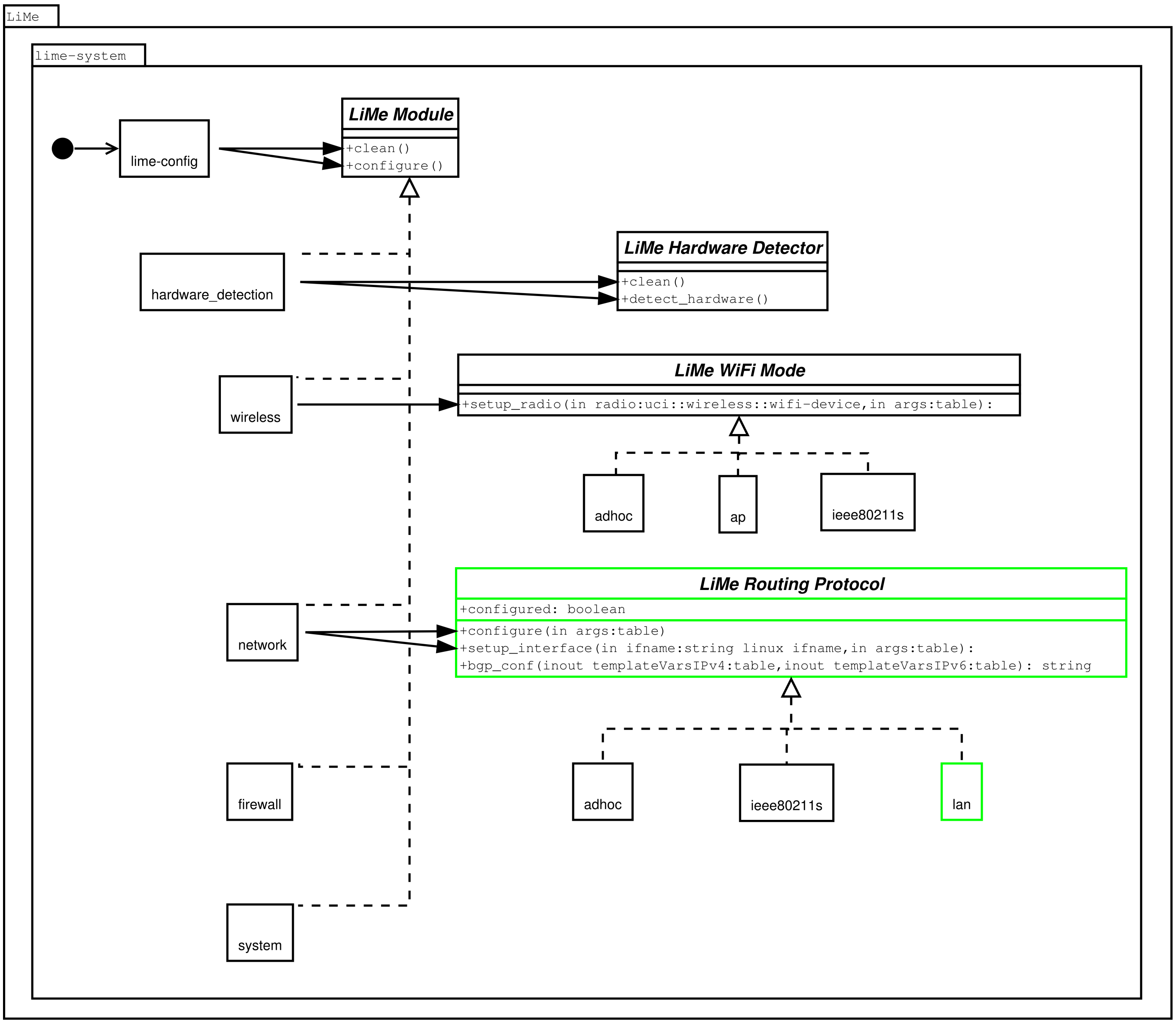


Figure 18: UML diagram of lime-system package, evidenced in green parts developed during the internship.

The new API call `lime.proto.bgp_conf(templateVarsIPv4, templateVarsIPv6)` is called by `lime.proto.bgp` for each `lime.proto` that has been requested to exchange routes with BGP, this function take as parameters two tables that are both readable to read already defined template variable and writable to eventually define additional template variables to pass back to `lime.proto.bgp` and returns a template snippet that is appended to the BIRD configuration file by `lime.proto.bgp`. Figure 20 display as an UML diagram the interaction between `lime.proto.bgp` and `lime.proto.*`.

3.1.1 Protocol: lan

As `lime.proto.lan` does very minimal work at L3 it doesn't need much interaction with BGP to announce his subnet route. Implementing `lime.proto.lan.bgp_conf(...)` has been quite simple as you can see in the following code snippet.

```
function lan.bgp_conf(templateVarsIPv4, templateVarsIPv6)
        local base_conf = [[
protocol direct {
        interface "br-lan";
}
]]
        return base_conf
end
```

3.2 Protocol: anygw

As the interaction of `lime.proto.anygw` with `lime.proto.bgp` is very similar to the one of `lime.proto.lan`. The modifications necessaries have been very minimal and limited to implementing `lime.proto.anygw.bgp_conf(...)` as you can see in the following code snippet, and in the UML diagram.

```
function anygw.bgp_conf(templateVarsIPv4, templateVarsIPv6)
        local base_conf = [[
protocol direct {
        interface "anygw";
}
]]
        return base_conf
end
```

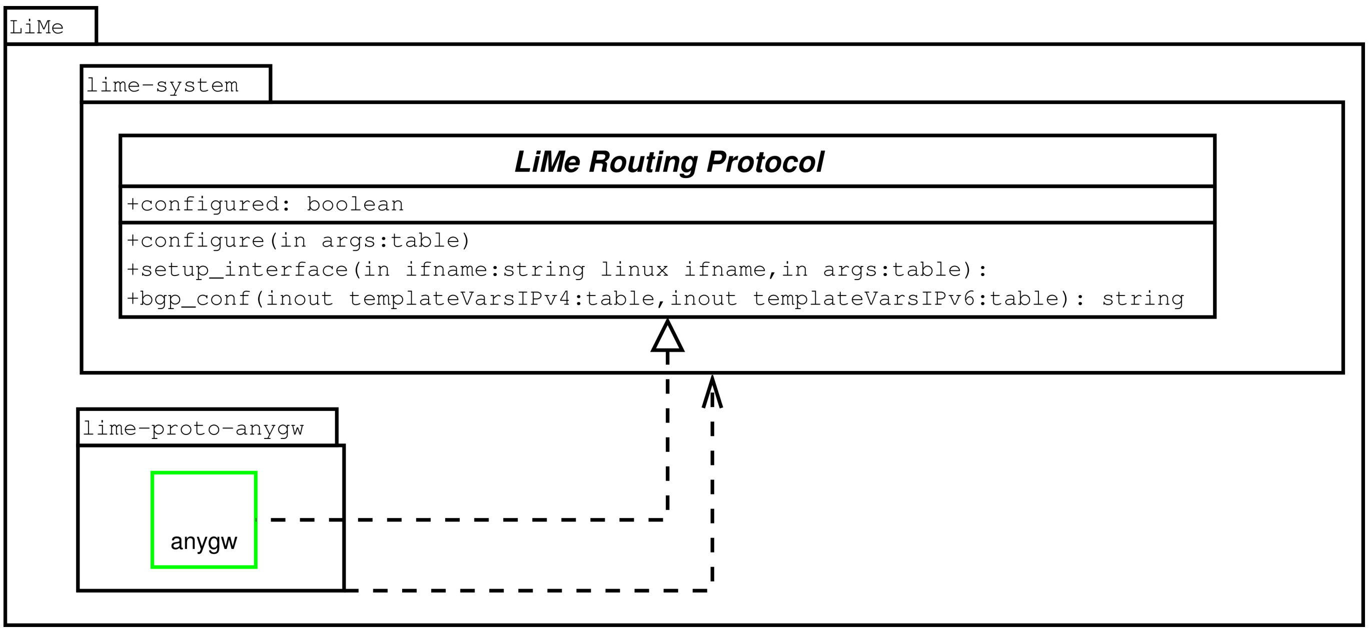

LiMe
lime-system
LiMe Routing Protocol
+configured: boolean
+configure(in args:table)
+setup_interface(in ifname:string linux ifname,in args:table):
+bgp_conf(inout templateVarsIPv4:table,inout templateVarsIPv6:table): string
lime-proto-anygw
anygw

Figure 19: UML diagram of lime-proto-anygw package, evidenced in green parts developed during the internship.

3.3 Protocol: bgp

To accomplish the task of interoperability between Libre-Mesh networks and Guifi.net BGP infrastructure, I have created `lime.proto.bgp` that implements the abstract class *LiMe Routing Protocol* and is shipped as the optional package `lime-proto-bgp`.

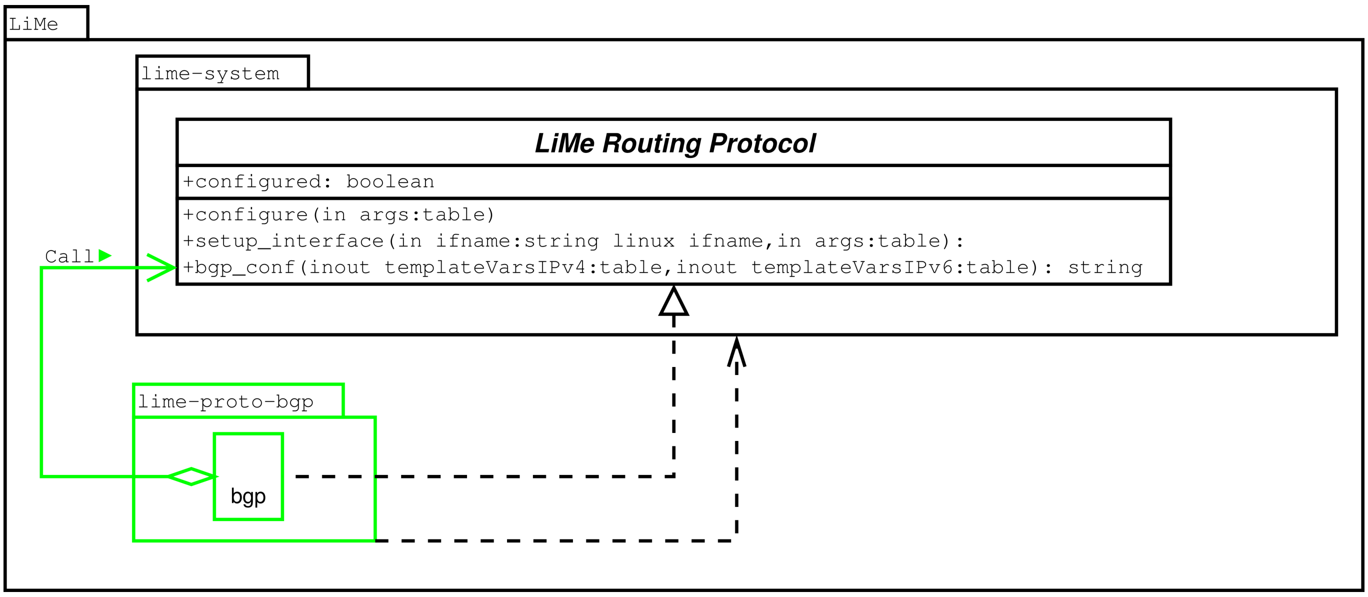

LiMe
lime-system
LiMe Routing Protocol
+configured: boolean
+configure(in args:table)
+setup_interface(in ifname:string linux ifname,in args:table):
+bgp_conf(inout templateVarsIPv4:table,inout templateVarsIPv6:table): string
Call
lime-proto-bgp
bgp

Figure 20: UML diagram of lime-proto-bgp package, evidenced in green parts developed during the internship.

This new *LiMe Routing Protocol* is in charge of configuring the LiMe system so it is capable of BGP routing using BIRD as back-end. To interconnect BGP zones with LiMe zones it establish peering with known BGP peer and export route learned via peering to the main kernel table. To instruct the BGP zones on how to reach the LiMe zones learning from main kernel table is enabled too, the route learned from the main kernel table are then exported to the BGP peering.

Snippet of lime-proto-bgp code showing how routes are learned and exported from and to main kernel table.

```
local base_template = [[
protocol kernel {
        learn;
        scan time 20;
        export all;
}
]]
```

Mesh routing protocols do usually have metrics to describe the quality of a path to a route, but BGP has not a similar concept, although it is possible to associate multiples attributes to BGP routes those aren't treated in an uniform way by different vendors implementations and in eBGP are often discarded or ignored [44]. Because there is no reliable way to convert *mesh quality* of path to a comparable BGP attribute, I decided to take a conservative approach and try to give to BGP zones the false but always worse then reality information that all mesh paths are to avoid if possible. The basic idea is that if a pure BGP path exists it should be preferred to a path including mesh networks because BGP can at least count hops on a pure BGP network but cannot understand if a mesh path is good or maybe the worse path possible. In eBGP setups (like Guifi.net) the most reliable way to share a route with low preference is to artificially enlarge it's AS path, this technique is called AS-path-prepending [45] and it's of common usages in situation where someone want to share a route but artificially lowering the preference.

Snippet of lime-proto-bgp code showing how mesh paths are avoided if possible.

```
local meshPenalty = args[4] or 8

local mp = "bgp_path.prepend("..localAS..");\n"
for i=1,meshPenalty do
```

```
        mp = mp .. "\t\t\tbgp_path.prepend("..localAS..");\n"
end
```

As in BGP direct interaction with network devices is almost absent it doesn't need any configuration for each interface so `lime.proto.bgp.setup_interface(ifname, args)` is an empty method, this may seems a cheep simplification but come with a not so cheap hidden cost, BGP doesn't offer any mechanism of node discovering so for each BGP peer peering configuration must be generated by LiMe.

Snippet of lime-proto-bgp code showing how peering configuration is generated for each BGP peer.

```
local peer_template = [[
protocol bgp {
        import filter fromBgp;
        export filter toBgp;

        local as $localAS;
        neighbor $remoteIP as $remoteAS;
}
]]

local function apply_peer_template(s)
        s.localAS = localAS
        if string.find(s.remoteIP, ":", 1, true) then
                bird6_config = bird6_config .. utils.expandVars(peer_template, s)
        elseif string.find(s.remoteIP, ".", 1, true) then
                bird4_config = bird4_config .. utils.expandVars(peer_template, s)
        end
end
config.foreach("bgp_peer", apply_peer_template)
```

While `lime.proto.bgp` is directly responsible of configuring BGP routing specific parts like BGP peering and BIRD filters, it delegates through the LiMe modularization pattern to the others *LiMe Routing Protocol* the specific configuration needed to make possible to automatize the inter-operation between each specific `lime.proto` and BGP, this has been accomplished calling for each of them the new introduced callback `lime.proto.bgp_conf(...)` and collecting the returned snippets into BIRD configuration file.

3.4 Protocol: bmx6

This package was already in charge of configuring BMX6 for L3 routing, now it also generate configuration for routes exchange between BGP and BMX6 networks, although it is not much code it has required time for investigation and for experimental tuning. The problems discovered during the experimental phase wasn't intrinsic of this module code, but were mainly BMX6 performance bottlenecks exposed by the configuration generated by this module.

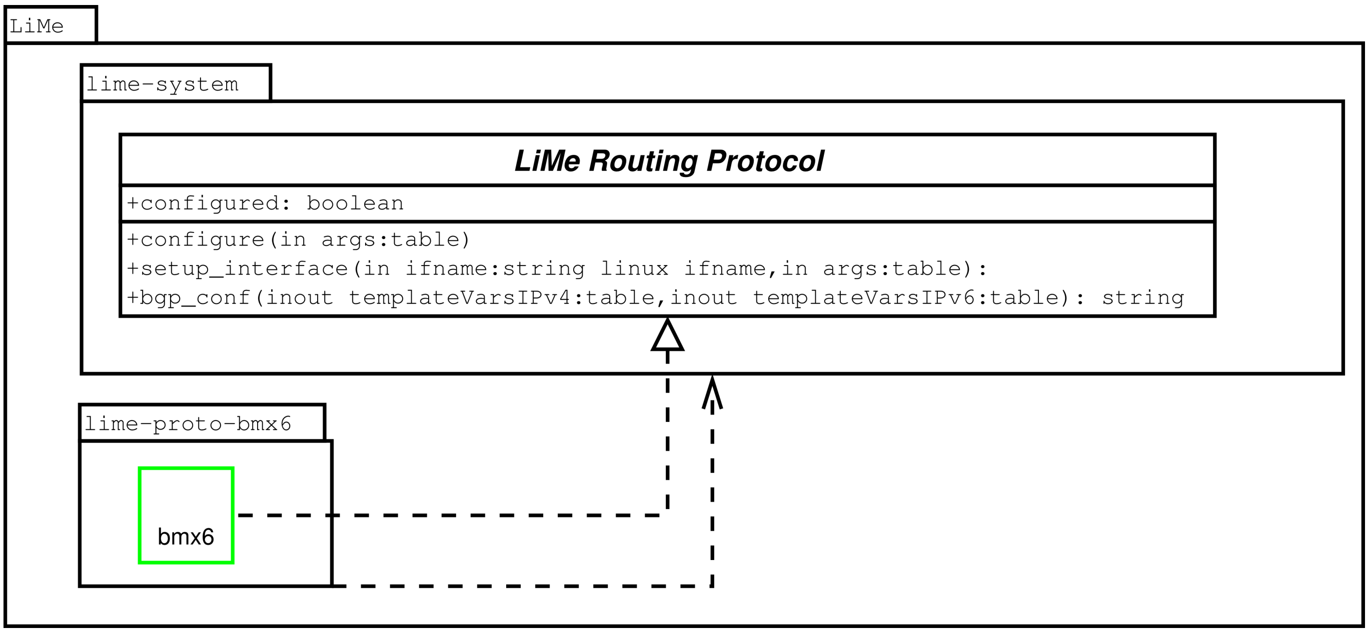

LiMe
lime-system
LiMe Routing Protocol
+configured: boolean
+configure(in args:table)
+setup_interface(in ifname:string linux ifname,in args:table):
+bgp_conf(inout templateVarsIPv4:table,inout templateVarsIPv6:table): string
lime-proto-bmx6
bmx6

Figure 21: UML diagram of lime-proto-bmx6 package, evidenced in green parts developed during the internship.

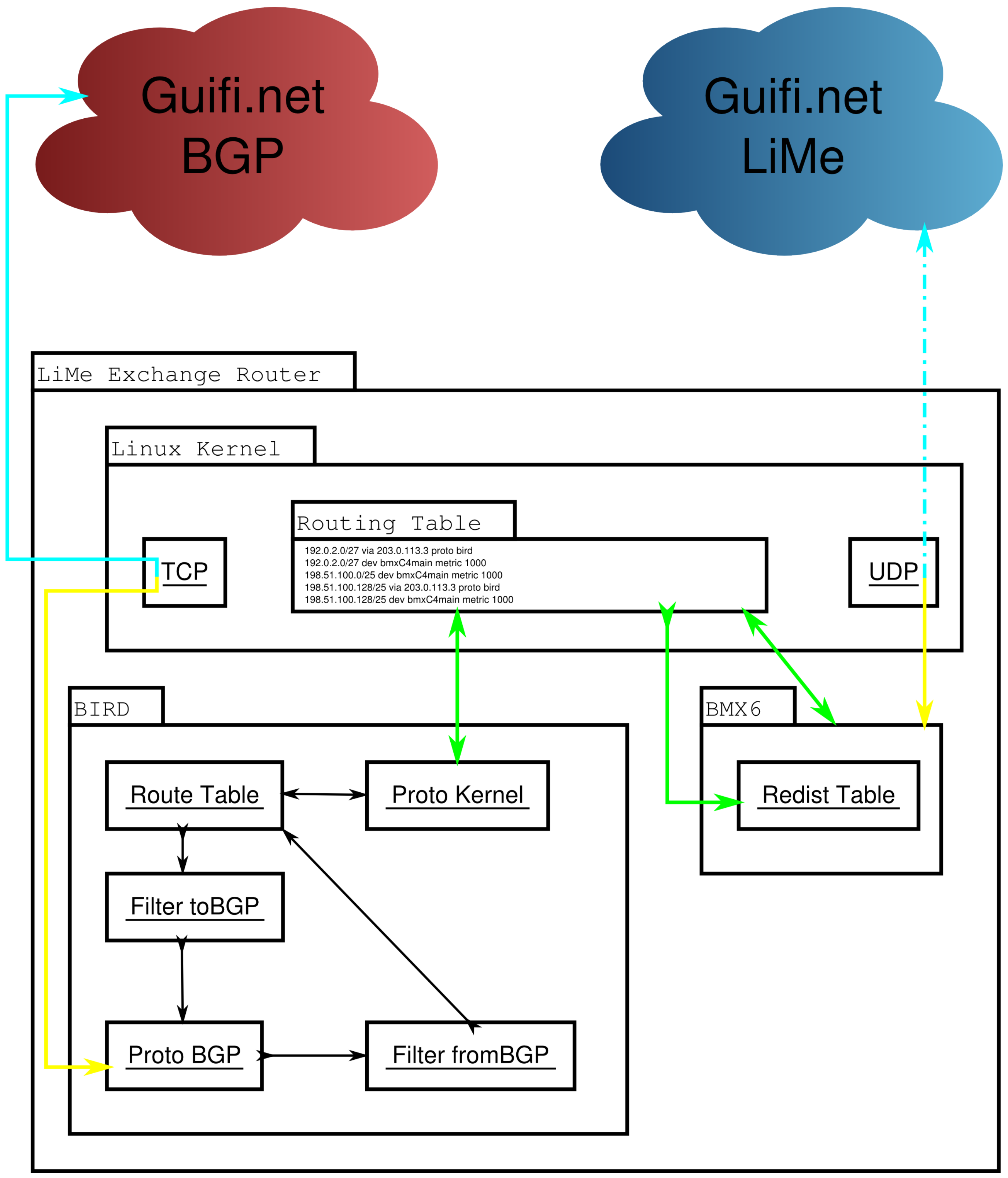
While BIRD has quite powerful route filters based on a C like language [8], in BMX6 `redistTable` plugin is the mechanism of choice to learn routes. Some filter capability were already present in `redistTable` but those were not enough for interoperability with BGP. In collaboration with BMX6 developer the `redistTable` plugin has been extend to be capable of filtering routes by Linux kernel `proto` number [46]-[51]. BMX6 routing protocols associate a metric to describe the quality of a path to a route, BGP has not a similar concept it just know hop count, because there is no reliable way to convert BGP hop count to a comparable BMX6 metric, I have decided to take a conservative approach and try to give to BMX6 zones the false but always worse then reality information that all BGP paths are to avoid if possible. The basic idea is that if a pure BMX6 path exists it should be preferred to a path including BGP networks because BMX6 know the quality of a mesh path while it doesn't have enough information to know if a BGP path is good or maybe the worse path possible. To accomplish this self imposed requirement `redistTable` is instructed to announce learned routes with low bandwidth.

Importing BGP routes from a kernel table and announce them as low bandwidth.

```
-- Enable Routing Table Redistribution plugin
uci:set("bmx6", "table", "plugin")
uci:set("bmx6", "table", "plugin", "bmx6_table.so")

-- Redistribute proto bird routes
uci:set("bmx6", "fromBird", "redistTable")
uci:set("bmx6", "fromBird", "redistTable", "fromBird")
uci:set("bmx6", "fromBird", "table", "200")
uci:set("bmx6", "fromBird", "bandwidth", "100")
uci:set("bmx6", "fromBird", "proto", "12")
```

Using only words to describe the overall architecture resulting from the developed code would be long and of scarce effectiveness, in Figure 22 the active components and the flow of routing information are displayed in a simplified manner all together.



```
192.0.2.0/27 via 203.0.113.3 proto bird
192.0.2.0/27 dev bmxC4main metric 1000
198.51.100.0/25 dev bmxC4main metric 1000
198.51.100.128/25 via 203.0.113.3 proto bird
198.51.100.128/25 dev bmxC4main metric 1000
```

Figure 22: Flow of routing information between components.

Legenda

- Black arrows: Routing information flowing inside components of the same process.

- Green arrows: Routing information flowing between kernel routing table and processes via Netlink.
- Yellow arrows: Routing information flowing between kernel and process via socket API.
- Cyan arrows: Routing information flowing between different routers via IP.
- Documentation addresses conform to RFC 5737 [26].

4 Experimental results

4.1 Interoperability use-cases

Due to LiMe modular nature this development has many possible application that spread out of boundaries of this internship, in the following sections a brief analysis of more common use-cases on which the development and experimentation has been focused is exposed.

4.1.1 Mesh network as distribution

In this scenario a BGP backbone is used as transport network while the mesh network is used as last mile distribution.

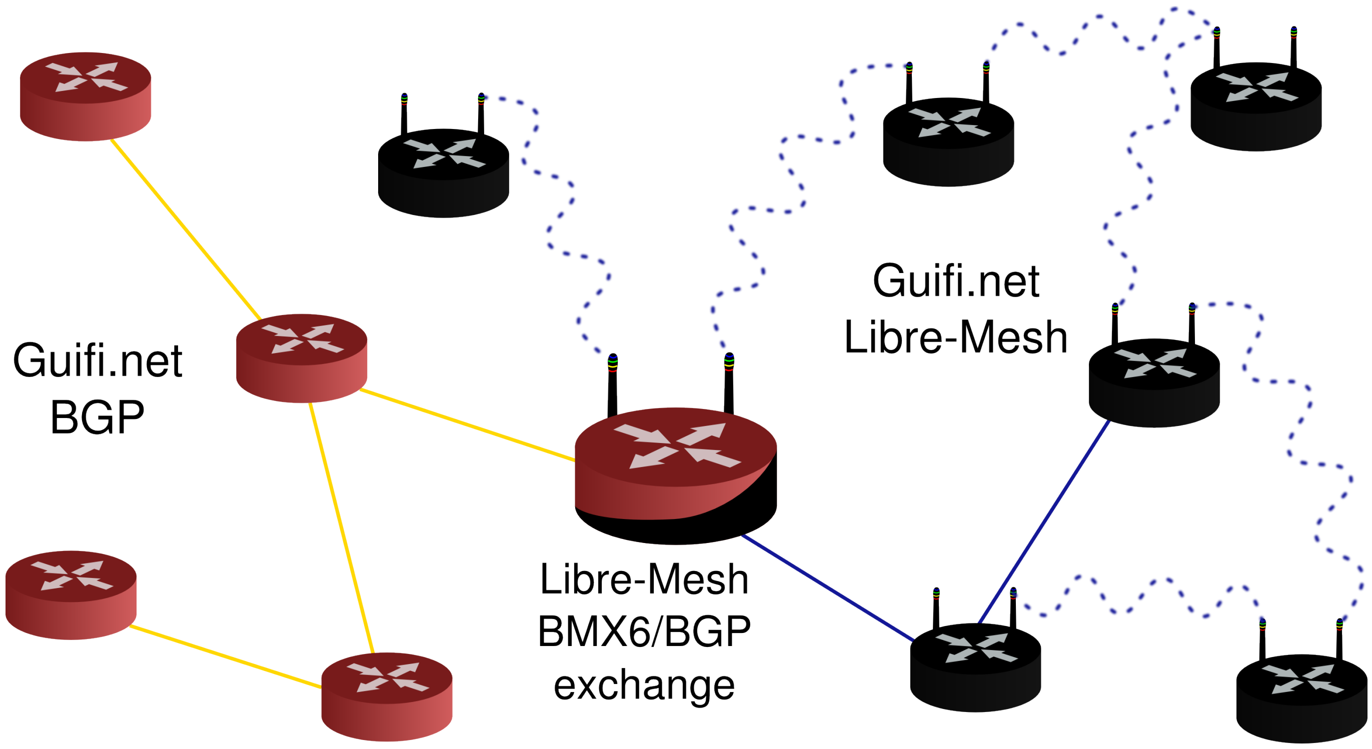


Figure 23: Simple exchange topology of a LiMe network used as last mile distribution.

Legenda

- Black and red routers: Routers running LiMe doing routing exchange between BGP and BMX6
- Black routers: Routers running LiMe with pure mesh setup.
- Red routers: Guifi.net routers configured as of *Modelo Híbrido*.
- Blue straight lines: Cabled mesh links.
- Blue curvy dashed lines: WiFi mesh links.
- Yellow straight lines: Guifi.net links as of *Modelo Híbrido*.

This use case is someway workable also with the pasts solution but with strong drawbacks:

- Instead of the “red and black” router we would need two devices one of the “red type” let’s call it `R` and one of the “black type” lets call it `B` connected, this mean a frontier node cost the double on average.

- The B router would announce statically the whole 10.0.0.0/8 to the mesh network and have R as gateway for the entire 10.0.0.0/8 net, this causes packets generated into the mesh directed to nonexistent hosts in that range will flow all across the mesh and even get into the BGP backbone, effectively reducing bandwidth available for legitimate packets.
- In the R router all the subnets used by the mesh network must be statically routed to B router and then written into the BGP daemon configuration to be announced to the rest of the “red network”, so each time a subnet is added to the the mesh network the R router need to be reconfigured manually, consuming time of some skilled Guifi.net operator.

All those limitations are solved using the software developed during this internship, moreover other more complex and even more common interoperability scenarios gets solved.

4.1.2 Mesh network with multiple exchanges

This scenario is very common in high population density zones, small neighborhood community mesh network grow beyond the district boundaries gaining geographical proximity to multiple backbone nodes, as the number of user usually grow too as the network expand having just an exchange point become a bottleneck, in this case having multiple exchange point can improve the access bandwidth to upstream networks.

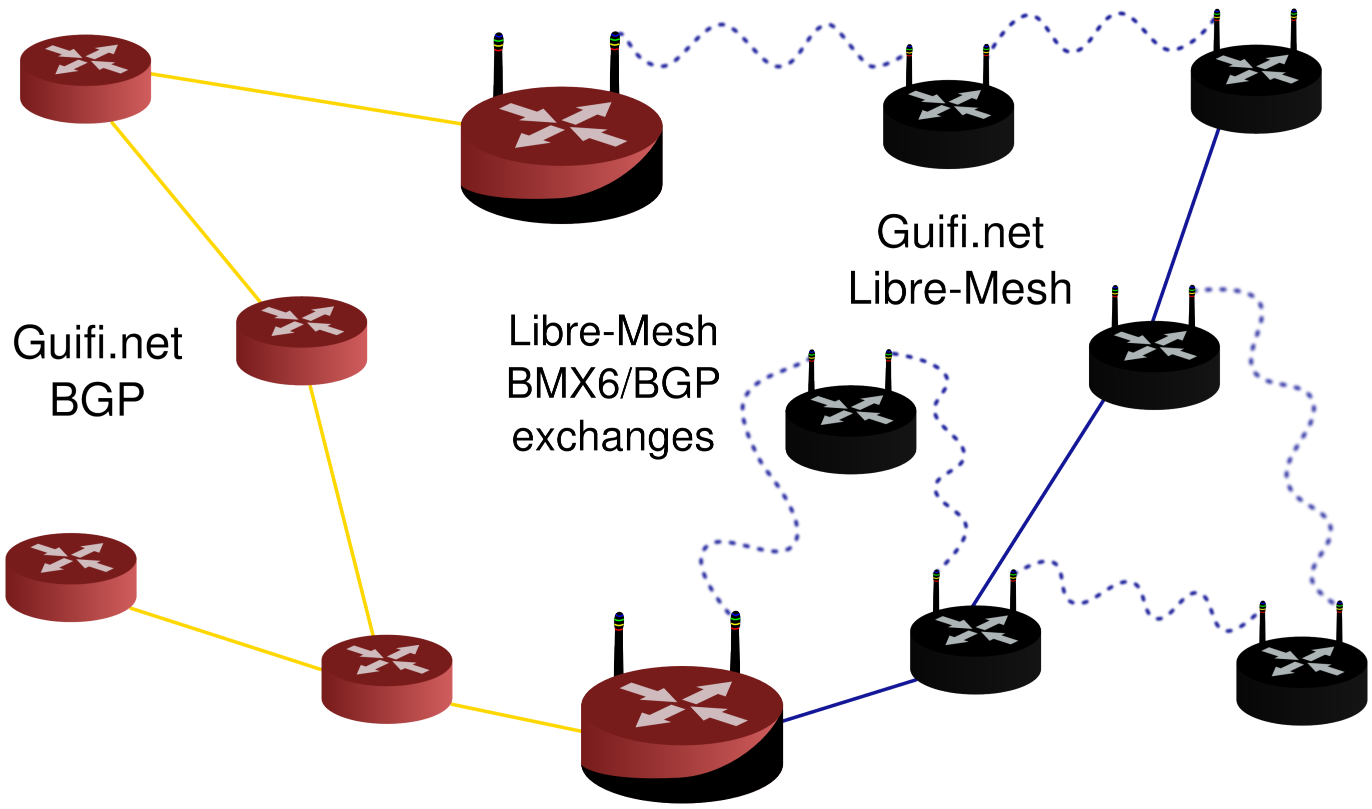


Figure 24: Multiple exchange between classic Guifi.net infrastructure and LiMe network example topology.

This scenario in not practically manageable with past technology, the more exchange points are added the trickier it gets to manually write a configuration that works stably. During the internship an automatic solution has been developed although it hasn’t been trivial, the usual Guifi.net eBGP setup is not enough as it causes routing loop [52] in this situation. When an host like `2001:DB8:9::4` from the LiMe network send a packet to an host like `2001:DB8:1::3` into BGP network the packet make his way to the destination without problems. Unfortunately is not the same when `2001:DB8:1::3` send a packet to his friend `2001:DB8:9::4`. The packet is correctly routed toward the exchange `BGPx2` but there because routes learned via BGP are preferred it’s sent back to `BGP4` expecting it have a better path for those packets. `BGP4` again thinks that `BGPx2` is the best next hop to reach `2001:DB8:9::4` and forward the packet to `BGPx2` and so on till the packet TTL is exhausted and one of the routers discard it.

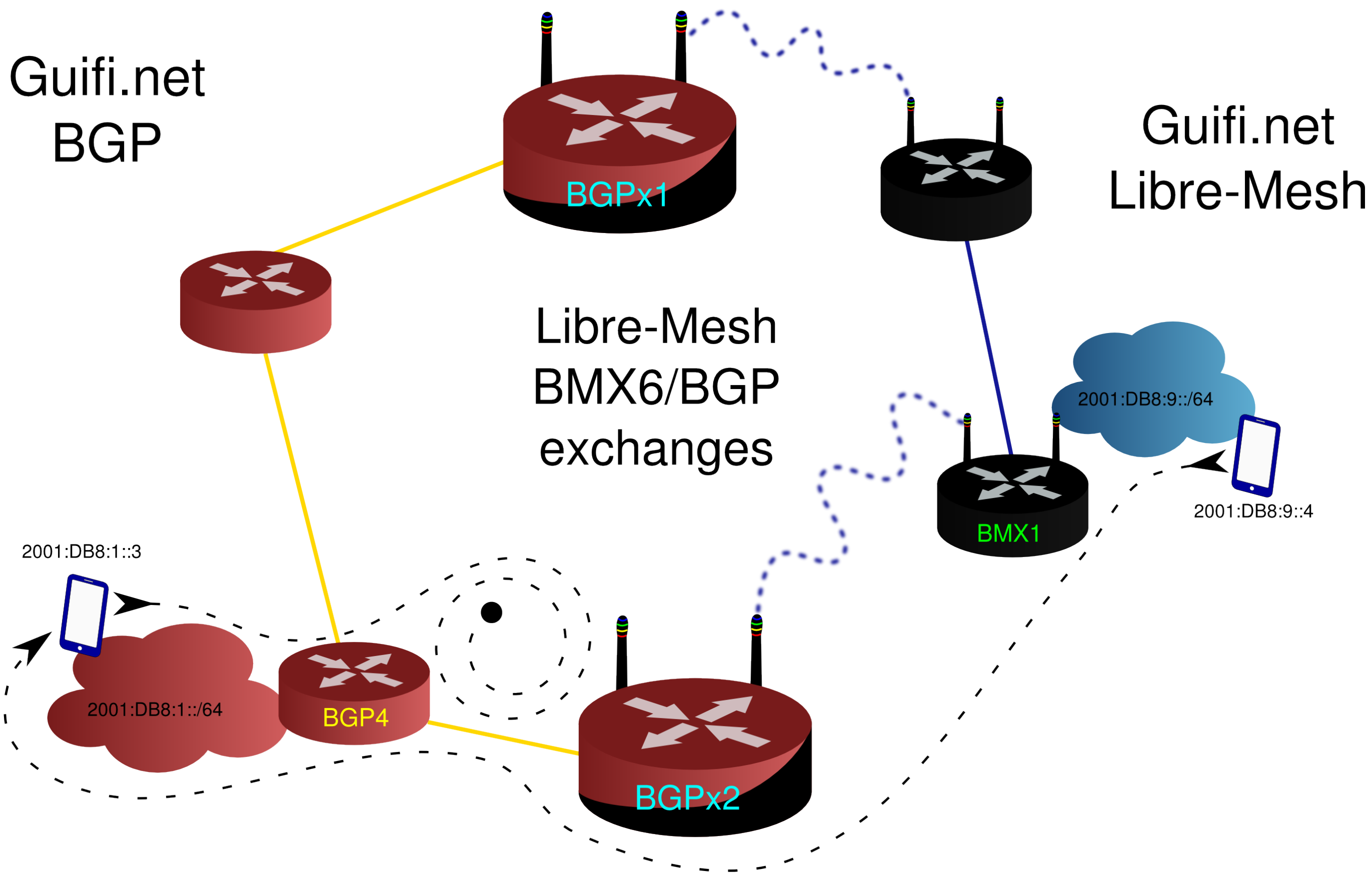


Figure 25: Routing loop discarding packets in naive eBGP multiple exchange setup.

Legenda

- Directed black dashed line: packets flowing in the direction of the arrow.

But what in detail causes this unwanted behavior? We can find the answer is in `BGPx2` routing table of which the relevant snippet is reported in the following lines.

```
2001:DB8:1::/64 via $BGP4 proto bird metric 0
2001:DB8:1::/64 dev bmxOut_BGPx1 proto static metric 1024
2001:DB8:9::/64 via 2001:DB8:8::1 proto bird metric 0
2001:DB8:9::/64 dev bmxOut_BMX1 proto static metric 1024
```

We can observe there are two routes towards `2001:DB8:9::/64` one learned via BGP (the `proto bird` ones) that have lower metric and one learned by BMX6 (the ones associated to `dev bmx*`) that have higher metric, thus the BIRD routes are always preferred. One could think that the problem could be fixed just by giving to BMX6 more preference but this tentative would just fail because it solves the loop in the BGP→BMX6 direction but it creates the exact opposite loop in BMX6→BGP direction. The classic way to solve this problem on BGP networks usually is to adopt the iBGP setup. In a iBGP setup routers share the same ASN so the exchanges can recognize internal routes when they learn them form other eBGP routers and avoid re-adding them to the kernel routing table. To adopt iBGP solve the problem but with an unbearable cost for an auto-organized mesh network that is that each iBGP router must be configured to establish peering connections to each other router in a setup called *full-mesh* where every router speaks to all other routers directly [53]. At this point, no other solutions then iBGP has been found. I started investigating how iBGP works internally and I discovered that the *full-mesh* peering is needed because every iBGP router need share the same routing information with all other iBGP routers [54]. In our case as BMX6 is a proactive distance-vector routing daemon it already spreads all routes to all routes so I supposed that in our case the iBGP *full-mesh* setup may be avoided, and empirical tests proved this supposition true. In fact loop is avoided and every host of both BGP and BMX6 networks can successfully reach every host of the whole Guifi.net (both BMX6 and BGP parts).

Relevant snippet of the resulting routing table using the “smart” iBGP-like setup.

```
2001:DB8:1::/64 via $BGP4 proto bird metric 0
2001:DB8:1::/64 dev bmxOut_BGPx1 proto static metric 1024
2001:DB8:9::/64 dev bmxOut_BMX1 proto static metric 1024
```

4.1.3 Mesh network as transport

The most important technical factor that slowed down LiMe adoption into Guifi.net is that the majority of preexistent network infrastructure is built following the *Modelo Híbrido* so when someone decide to extend Guifi.net infrastructure need to be compatible with *Modelo Híbrido*, without the development done during this internship the only possible interoperability setup was having the mesh as a leaf distribution network, while now it is possible to build Guifi.net backbone with LiMe being compatible both with mesh and BGP networks, it is even possible to use LiMe to fill the gap between isolated BGP network islands.

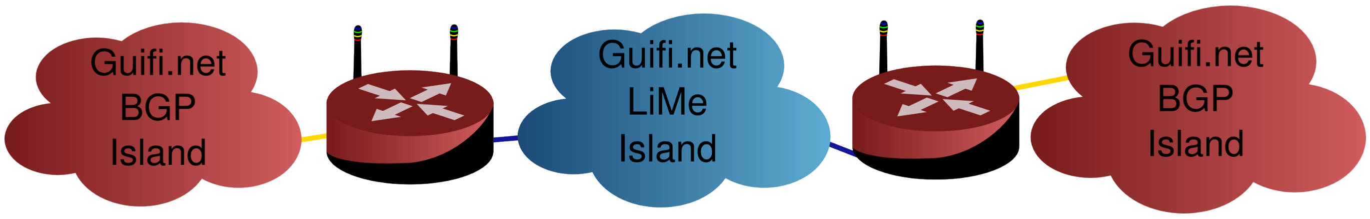


Figure 26: Mesh used as transport between two isolated BGP networks.

This will certain encourage LiMe adoption and maybe finally obsolete *Modelo Híbrido*, giving the possibility to Guifi.net to jump into another phase of horizontal growth with less dependency from manual configuration work, skilled people will have time to work on more interesting and not repetitive configuration tasks, with consequent work quality improvement, costs reduction and increased network accessibility for the masses.

4.2 Solving the route count limitation

BMX6 store distributed routes in a shared structure called “node description” for performance reasons the size of this structure is limited and it’s not enough to contain the whole Guifi.net BGP routing table, a first approach to solve this problem was to enable route aggregation in BMX6 `redistTable` plug-in, but despite the high CPU usage the size of the table was reduced only by half and that was not enough to fit inside a “node description”, moreover the high CPU load caused timeouts in communication between BMX6 and Linux kernel causing BMX6 to crash. To solve this problem in the new version of BMX6 the “node description” has been restructured like a linked list so the node description can contain a pointer to another structure if just one is not enough to store all the needed informations, the only drawback is that this is not compatible with past versions of BMX6 but it doesn’t represents a blocking problem because BMX6 users are used to incompatible changes between versions, this solution has been discussed at BattlemeshV8 [55] with BMX6 developer which has implemented it and is now in production at our offices in Barcelona.

4.3 Reducing the performance footprint

BMX6 `redistTable` as it was before this internship was not suitable to our needs in fact because it was not ready to handle a big routing table it was using 100% CPU all the time. The high CPU load do not only was stressing our little routers but causing them to have high response time and that was interpreted by other routers as if the link was performing really bad, this caused routers loose connectivity each others. In those condition to further analyze performances wasn't possible. After some performances fixes in upstream BMX6 code I have been able to see BMX6 running stably. So after setup the performance testbed Figure 27 with multiple connections to both BGP based portion of Guifi.net and Libre-Mesh based portion of Guifi.net I have proceeded optimizing performance tweaking BMX6 configurations.

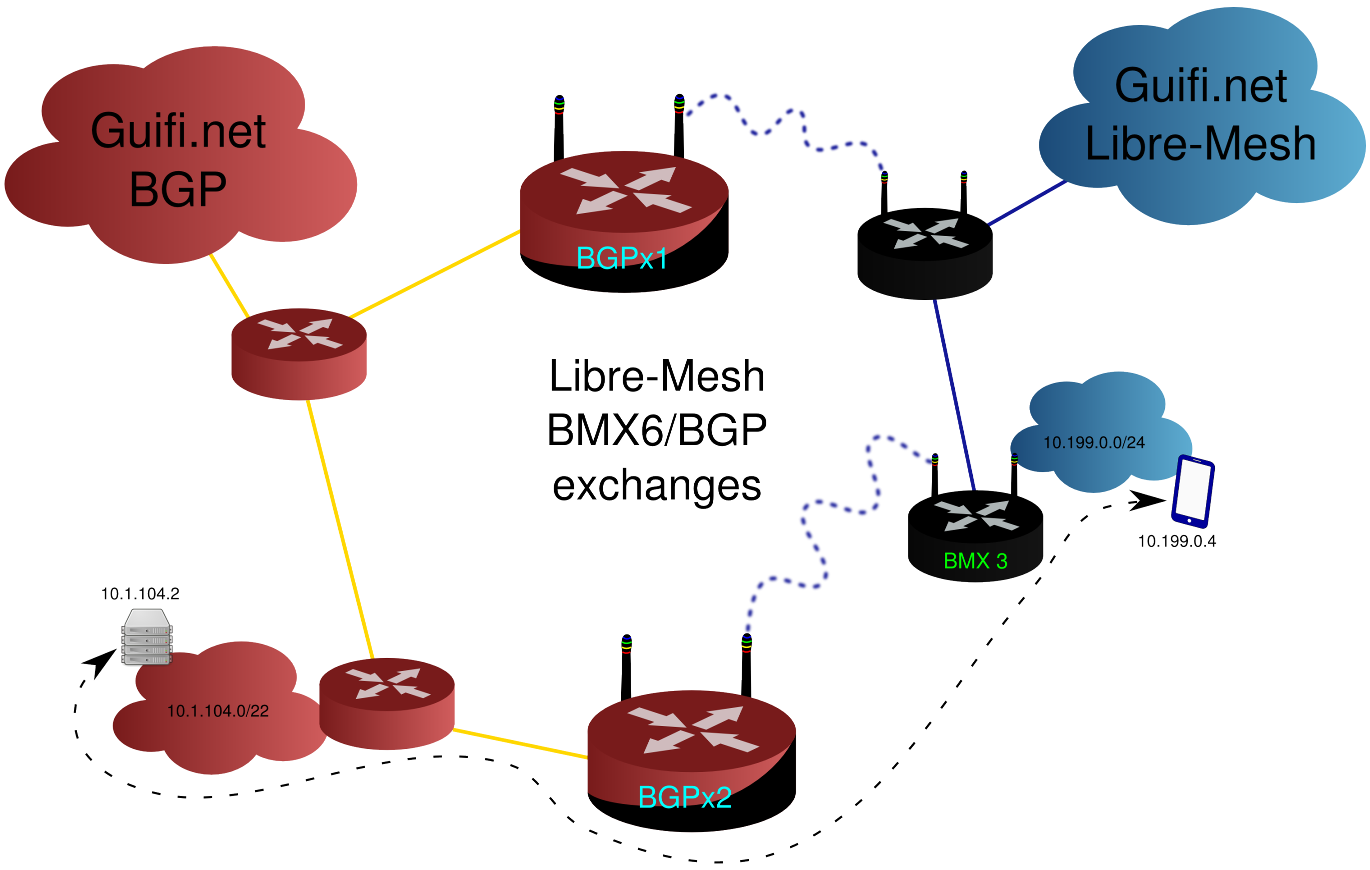


Figure 27: Performance analysis testbed.

Legenda

- Black routers: Routers running LiMe with pure mesh setup.
- Black and red routers: Routers running LiMe doing routing exchange between BGP and BMX6
- Red routers: Guifi.net routers configured as of *Modelo Híbrido*.
- Blue straight lines: Cabled mesh links.
- Blue curvy dashed lines: WiFi mesh links.
- Yellow straight lines: Guifi.net links as of *Modelo Híbrido*.
- Server and smart-phone pinging each other.

4.3.1 Default configuration

Even if BMX6 run stably its performance impact in the default configuration is not acceptable, as we can observe in Figure 28 the system load on BGP$\leftrightarrow$BMX6 exchanges too often exceed 1, and when it is under it still use more then half system resources living too few CPU time for other tasks such as packet forwarding.

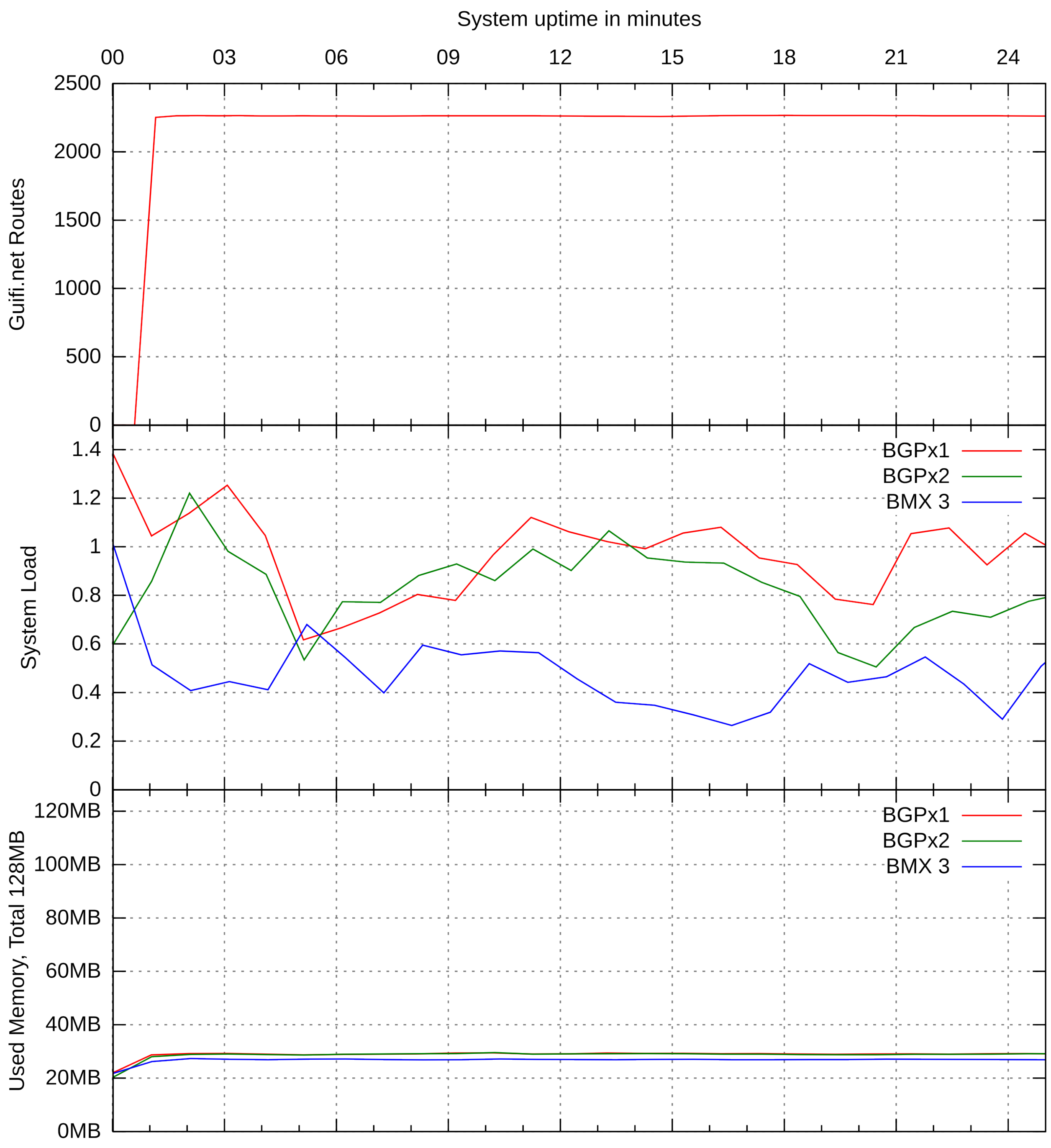


Figure 28: Performance graph of default configuration.

4.3.2 Route aggregation

By default `redistTable` do routes aggregation that is a CPU and memory intensive work. In our case as Guifi.net has thousands of routes it may have a big impact on the performances, disabling route aggregation is done directly into `lime-proto-bmx6` code.

```
uci:set("bmx6", "fromBird", "aggregatePrefixLen", "128")
```

As we can see in Figure 29 disabling route aggregation has a sensible positive impact on performances, system load is consistently reduced both on `BGPx1` and `BGPx2`, while a non critical slight increase in memory usage is noticeable on `BMX 3`. Due to disabled aggregation `BMX 3` receive same amounts of route announcements of BGP↔BMX6 exchanges so it's memory usage align to them.

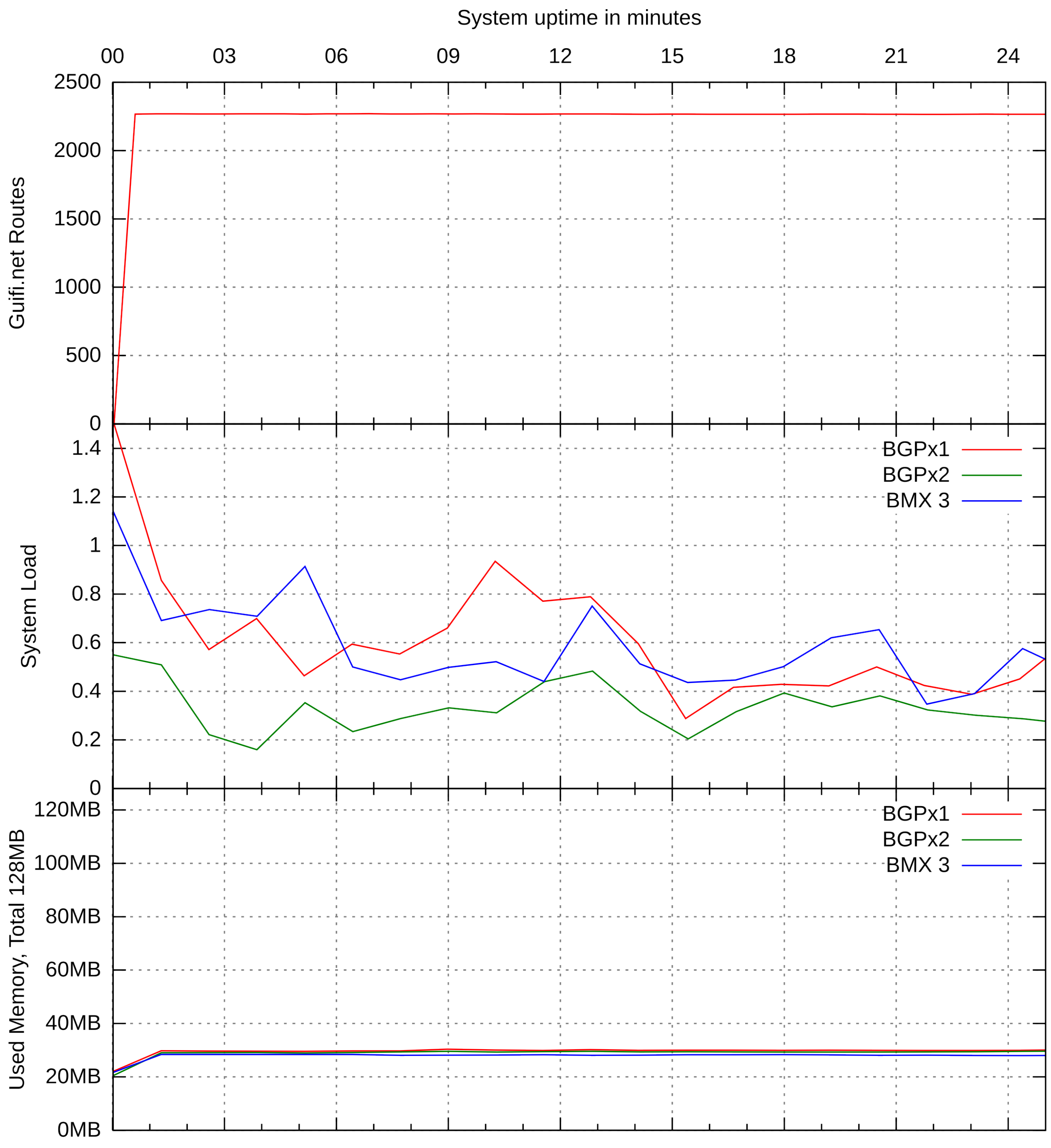


Figure 29: Performance graph without route aggregation.

4.3.3 Proactive tunnels

By default BMX6 react to every single change in the routing table. As Guifi.net is a quite big network lot of route changes get propagated and sometime some change is reverted before the change itself get propagated to the whole network. This two conditions together were caught causing lot of recalculation and propagation also of temporary changes, if a route was deleted by BIRD and then re-added in a fraction of a second this caused BMX6 recalculating announcements and propagating both changes.

To avoid it more options to regulate the BMX6 behavior has been added directly in BMX6 code [56][57] and then configured by `lime-proto-bmx6` only in cases where default configuration wasn't appropriated to Guifi.net needs.

```
uci:set("bmx6", "general", "proactiveTunRoutes", "0")
```

While this behavior depends on network conditions too, in Figure 30 we can observe a fair system load reduction and a more noticeable reduction in system load oscillation due to solved hypersensitivity to temporary changes.

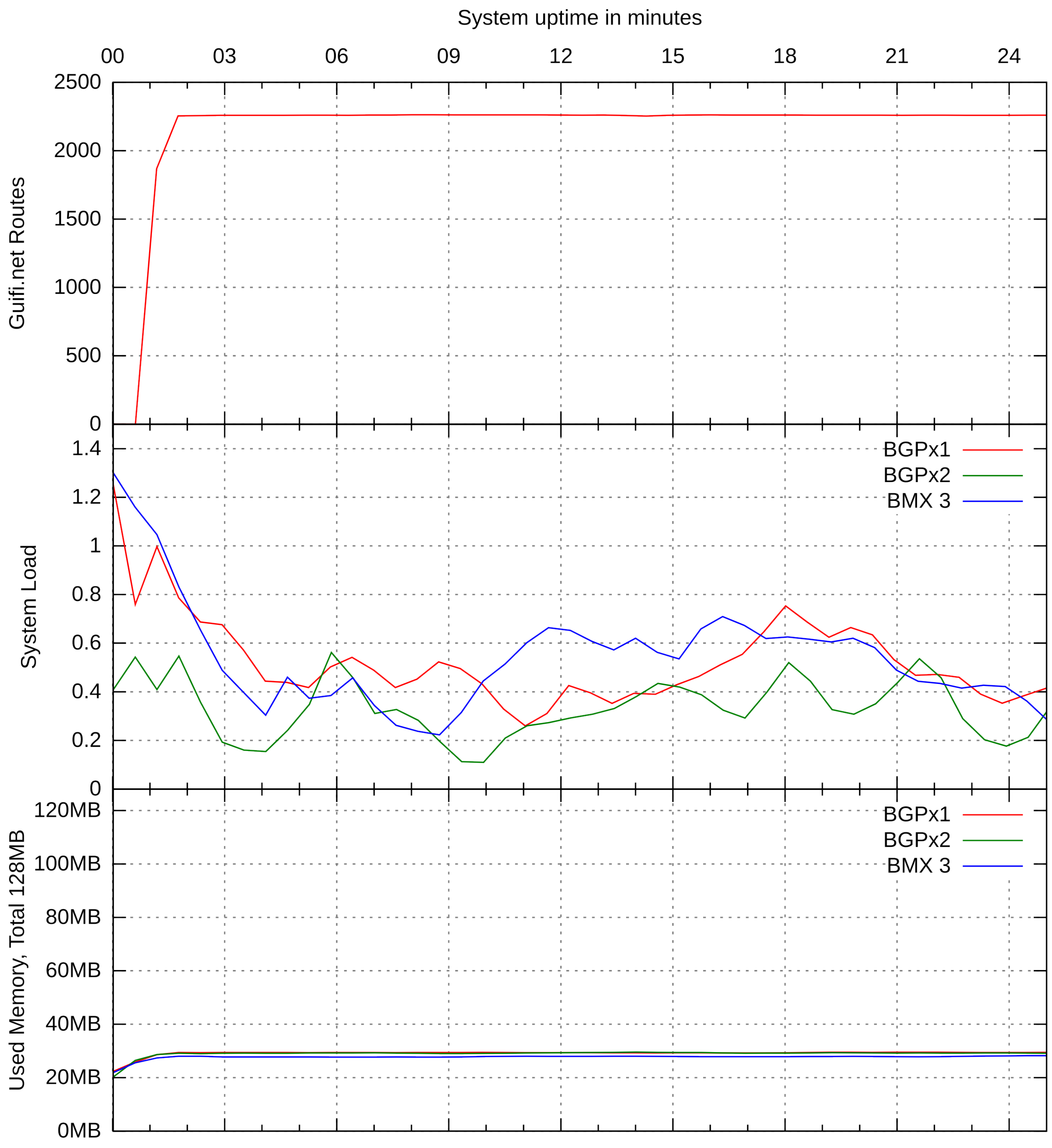


Figure 30: Performance graph with route aggregation and proactive tunnels both disabled.

4.3.4 Link Signature

Another aspect that may impact performances are the trusted networking features of BMX6 based on cryptography, those features, as examples, permits to filter out spoofed routing informations or to prefer paths made only of trusted nodes [58][59]. While those features are enable by default there was no option to disable them and Guifi.net LiMe setup doesn't take advantage of them so an option to permit to tweak them has been introduced in BMX6 [60], and disabled in `lime-proto-bmx6` code.

```
uci:set("bmx6", "general", "linkSignatureLen", "0")
```

As we can observe in Figure 31 this change caused a consistent CPU usage reduction, lowering system load to usual values even if the routing table is quite big.

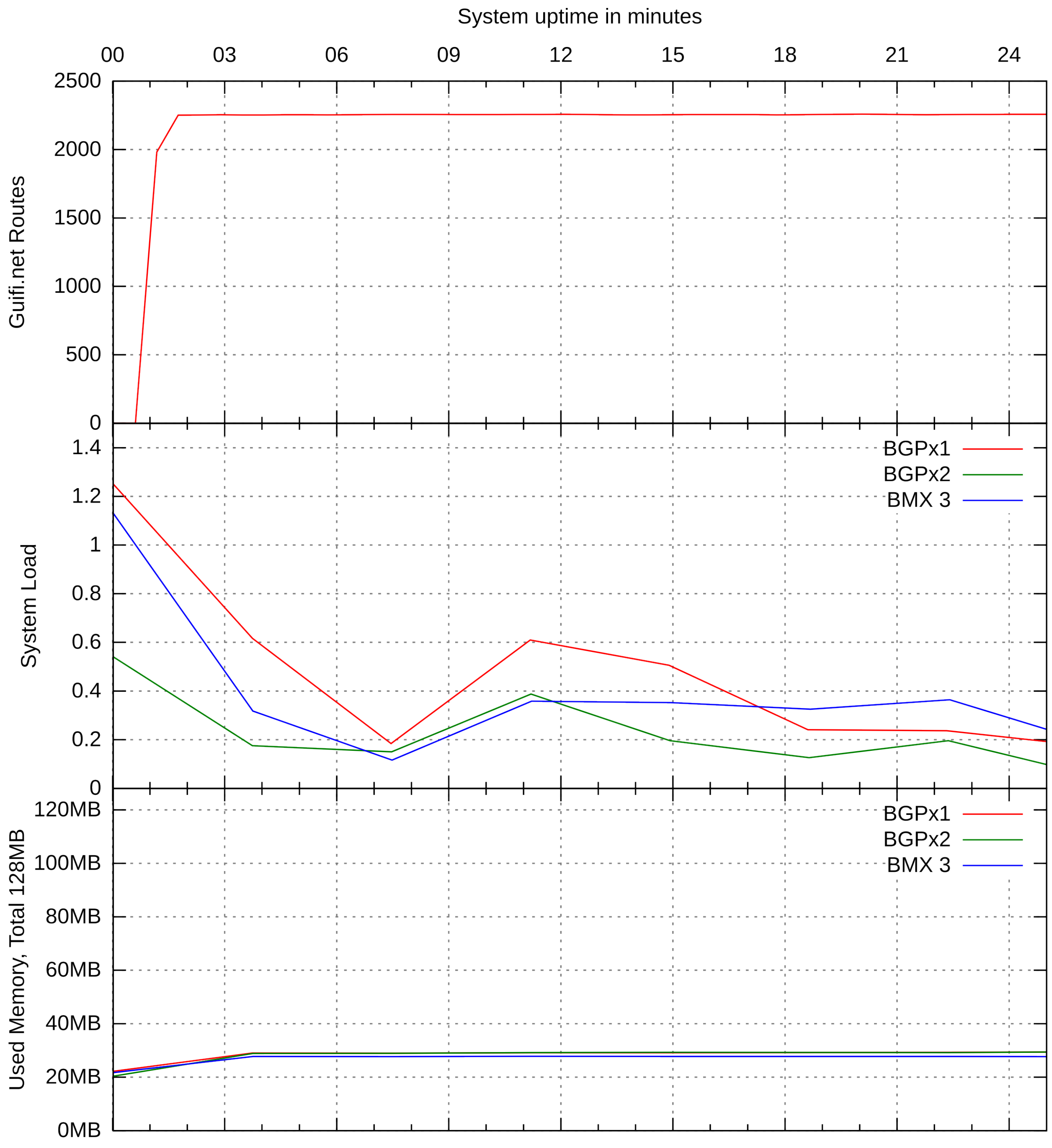


Figure 31: Performance graph with route aggregation, proactive tunnels and link signature all disabled.

5 Conclusions and future developments

During this internship, seamless interoperability between classic infrastructure and Libre-Mesh networks in Guifi.net has been achieved with satisfactory results. For Guifi.net, that at time of writing counts more then 10 years of activity and nearly 30000 active nodes, this potentially opens new horizons of sustainable growth. Building networks inside Guifi.net with Libre-Mesh doesn't mean anymore to be relegated to last mile distribution, but to build at same time affordable and auto-configuring Guifi.net carrier networks and doing it with completely free as in freedom software [61]. The possibility of using free software have big impact, as an example till today Guifi.net has experienced difficulty even to support IPv6 on carrier links because of proprietary routers requiring costly licenses upgrade to enable IPv6 support on their software, or alternatively to completely renew the infrastructure resulting in an awkward situation well known as collective vendor lock-in. This will never happen with free as in freedom routers, but this fact doesn't mean only IPv6 support. Being a free software solution, combined with the fact that LiMe architecture is modular guarantees an high level of future-proofness. Community participants can eventually extend and modify the software and improve it to satisfy unforeseen needs without high costs. LiMe networks requires significantly less human intervention then *Modelo Híbrido* based networks, as node installation will require less work, lot of skilled Guifi.net enthusiasts and professionals will have more time to install more nodes or to do more non automatable work like research, development and community networks promotion, this will probably foster even more Guifi.net growth. LiMe decentralized network model does allow more independence from centralized configuration provisioning tools, that are in contrast with the idea of a non-hierarchic collaborative network managed directly by its users, like Guifi.net try to be. This works has also benefited other projects aside of LiMe and Guifi.net, multiple BMX6 bugs and performance bottlenecks have been discovered during the experimental phase and isolated in collaboration with BMX6 main developer Axel Neumann who has fixed them in upstream code [46]-[51] [56] [57] [60] [65]-[103]. Mesh networks in general now benefits of a maturer software stressfully tested with real world huge routing table. Last but not least interoperability between BGP and BMX6 have been achieved in a very modular way, it doesn't really depends on specific features of the second protocol, so future implementations of route exchange with other protocols like Babel or OLSRv2 will require minimal effort and will benefit of same loop avoidance guarantees explored during this work, making easy to handle even more complex exchange topologies [2], with multiple different protocols and multiple independents peers.

[2] In respect to the ones analyzed in the scope of this work, see Section 4 for more information.

6 Appendix: Code

All the code code written for Libre-Mesh has been published on its public repository [104] in references you can find the list of specific commits [105]-[121]. All needed modification of BMX6 are already published on its public repository [122], in references you can find the list of specific commits [46]-[51] [56] [57] [60] [65]-[103] and on OpendWrt routing feed [123] in specific commits [62]-[64]. The following reported source code is from the files changed into Libre-Mesh packages.

packages/lime-proto-anygw/src/anygw.lua

```
#!/usr/bin/lua

local fs = require("nixio.fs")
local network = require("lime.network")
local libuci = require "uci"

anygw = {}

anygw.configured = false

function anygw.configure(args)
        if anygw.configured then return end
        anygw.configured = true

        local ipv4, ipv6 = network.primary_address()

        -- anygw macvlan interface
        print("Adding macvlan interface to uci network...")
        local anygw_mac = "aa:aa:aa:aa:aa:aa"
        local anygw_ipv6 = ipv6:minhost()
        local anygw_ipv4 = ipv4:minhost()
        anygw_ipv6[3] = 64 -- SLAAC only works with a /64, per RFC
        anygw_ipv4[3] = ipv4:prefix()

        local pfr = network.limeIfNamePrefix

        local uci = libuci:cursor()
        uci:set("network", pfr.."anygw_dev", "device")
        uci:set("network", pfr.."anygw_dev", "type", "macvlan")
        uci:set("network", pfr.."anygw_dev", "name", "anygw")
        uci:set("network", pfr.."anygw_dev", "ifname", "@lan")
        uci:set("network", pfr.."anygw_dev", "macaddr", anygw_mac)

        uci:set("network", pfr.."anygw_if", "interface")
        uci:set("network", pfr.."anygw_if", "proto", "static")
        uci:set("network", pfr.."anygw_if", "ifname", "anygw")
        uci:set("network", pfr.."anygw_if", "ip6addr", anygw_ipv6:string())
        uci:set("network", pfr.."anygw_if", "ipaddr", anygw_ipv4:host():string())
        uci:set("network", pfr.."anygw_if", "netmask", anygw_ipv4:mask():string())

        uci:set("network", pfr.."anygw_rule6", "rule6")
        uci:set("network", pfr.."anygw_rule6", "src", anygw_ipv6:host():string().."/128")
        uci:set("network", pfr.."anygw_rule6", "lookup", "170") -- 0xaa in decimal

        uci:set("network", pfr.."anygw_route6", "route6")
        uci:set("network", pfr.."anygw_route6", "interface", pfr.."anygw_if")
        uci:set("network", pfr.."anygw_route6", "target", anygw_ipv6:network():string().."/ ↩
            "..anygw_ipv6:prefix())
        uci:set("network", pfr.."anygw_route6", "table", "170")

        uci:set("network", pfr.."anygw_rule4", "rule")
        uci:set("network", pfr.."anygw_rule4", "src", anygw_ipv4:host():string().."/32")
        uci:set("network", pfr.."anygw_rule4", "lookup", "170")
        uci:set("network", pfr.."anygw_route4", "route")
        uci:set("network", pfr.."anygw_route4", "interface", pfr.."anygw_if")
        uci:set("network", pfr.."anygw_route4", "target", anygw_ipv4:network():string())
        uci:set("network", pfr.."anygw_route4", "netmask", anygw_ipv4:mask():string())
        uci:set("network", pfr.."anygw_route4", "table", "170")
        uci:save("network")

        fs.mkdir("/etc/firewall.user.d")
        fs.writefile(
                "/etc/firewall.user.d/20-anygw-ebtables",
                "\n" ..
                "ebtables -D FORWARD -j DROP -d " .. anygw_mac .. "\n" ..
                "ebtables -A FORWARD -j DROP -d " .. anygw_mac .. "\n" ..
                "ebtables -t nat -D POSTROUTING -o bat0 -j DROP -s " .. anygw_mac .. "\n"  ↩
                    ..
                "ebtables -t nat -A POSTROUTING -o bat0 -j DROP -s " .. anygw_mac .. "\n"
        )

        local content = { }
        table.insert(content, "interface=anygw")
        table.insert(content, "except-interface=br-lan")
        fs.writefile("/etc/dnsmasq.d/lime-proto-anygw-00-interfaces.conf", table.concat( ↩
            content, "\n").."\n")

        content = { }
        table.insert(content, "dhcp-range=tag:anygw,"..anygw_ipv4:add(1):host():string().." ↩
            ,"..ipv4:maxhost():string())
        table.insert(content, "dhcp-option=tag:anygw,option:router,"..anygw_ipv4:host(): ↩
            string())
        table.insert(content, "dhcp-option=tag:anygw,option:dns-server,"..anygw_ipv4:host() ↩
            :string())
        table.insert(content, "dhcp-option=tag:anygw,option:domain-name,lan")
        table.insert(content, "dhcp-option=tag:anygw,option:domain-search,lan")
        table.insert(content, "dhcp-option-force=tag:anygw,option:mtu,1350")
        table.insert(content, "dhcp-broadcast=tag:anygw")
        table.insert(content, "address=/anygw/"..anygw_ipv4:host():string())
        fs.writefile("/etc/dnsmasq.d/lime-proto-anygw-10-ipv4.conf", table.concat(content,  ↩
            "\n").."\n")

        content = { }
        table.insert(content, "enable-ra")
        table.insert(content, "dhcp-range=tag:anygw,"..ipv6:network():string()..",ra-names" ↩
            )
        table.insert(content, "dhcp-option=tag:anygw,option6:domain-search,lan")
        table.insert(content, "dhcp-option=tag:anygw,option6:dns-server,"..anygw_ipv6:host ↩
            ():string())
        table.insert(content, "address=/anygw/"..anygw_ipv6:host():string())
        fs.writefile("/etc/dnsmasq.d/lime-proto-anygw-20-ipv6.conf", table.concat(content,  ↩
            "\n").."\n")

        io.popen("/etc/init.d/dnsmasq enable || true"):close()
end

function anygw.setup_interface(ifname, args) end

function anygw.bgp_conf(templateVarsIPv4, templateVarsIPv6)
        local base_conf = [[
protocol direct {
        interface "anygw";
}
]]
```

```
        return base_conf
end

return anygw
```

packages/lime-proto-bgp/Makefile

```
#
# Copyright (C) 2006-2014 OpenWrt.org
#
# This is free software, licensed under the GNU General Public License v3.
#

include $(TOPDIR)/rules.mk

LIME_BUILDDATE:=$(shell date +%Y%m%d_%H%M)
LIME_CODENAME:=bigbang

GIT_COMMIT_DATE:=$(shell git log -n 1 --pretty=%ad --date=short . )
GIT_COMMIT_TSTAMP:=$(shell git log -n 1 --pretty=%at . )

PKG_NAME:=lime-proto-bgp
PKG_VERSION=$(GIT_COMMIT_DATE)-$(GIT_COMMIT_TSTAMP)

include $(INCLUDE_DIR)/package.mk

define Package/$(PKG_NAME)
  TITLE:=LiMe BGP proto support
  CATEGORY:=LiMe
  MAINTAINER:=Gioacchino Mazzurco <gio@diveni.re>
  URL:=http://libre-mesh.org
  DEPENDS:=+bird4 +bird6 +lime-system +lua
endef

define Build/Compile
        @rm -rf ./build || true
        @cp -r ./src ./build
        @sed -i '/^--!.*/d' build/*.lua
endef

define Package/$(PKG_NAME)/install
        @mkdir -p $(1)/usr/lib/lua/lime/proto || true
        $(CP) ./build/bgp.lua $(1)/usr/lib/lua/lime/proto/
endef

$(eval $(call BuildPackage,$(PKG_NAME)))
```

packages/lime-proto-bgp/src/bgp.lua

```
#!/usr/bin/lua

local network = require("lime.network")
local config = require("lime.config")
local fs = require("nixio.fs")
local utils = require("lime.utils")


proto = {}

proto.configured = false

function proto.configure(args)
```

```
        if proto.configured then return end
        proto.configured = true

        local ipv4, ipv6 = network.primary_address()
        local localAS = args[2] or 64496
        local bgp_exchanges = args[3]
        if bgp_exchanges then bgp_exchanges = utils.split(bgp_exchanges,",")
        else bgp_exchanges = {} end
        local meshPenalty = args[4] or 8

        local mp = "bgp_path.prepend("..localAS..");\n"
        for i=1,meshPenalty do
                mp = mp .. "\t\t\tbgp_path.prepend("..localAS..");\n"
        end

        local templateVarsIPv4 = { localIp=ipv4:host():string(),
                localAS=localAS, acceptedNet="10.0.0.0/8", meshPenalty=mp }
        local templateVarsIPv6 = { localIp=ipv6:host():string(),
                localAS=localAS, acceptedNet="2000::0/3", meshPenalty=mp }

        local base_template = [[
router id $localIp;

protocol device {
        scan time 10;
}

filter toBgp {
        if net ~ $acceptedNet then {
                if proto ~ "kernel*" then {
                        $meshPenalty
                }
                accept;
        }
        reject;
}

filter fromBgp {
        if net ~ $acceptedNet then accept;
        reject;
}

protocol kernel {
        learn;
        scan time 20;
        export all;
}
]]

        for _,protocol in pairs(bgp_exchanges) do
                local protoModule = "lime.proto."..protocol
                if utils.isModuleAvailable(protoModule) then
                        local proto = require(protoModule)
                        local snippet = nil
                        xpcall( function() snippet = proto.bgp_conf(templateVarsIPv4,  ↩
                            templateVarsIPv6) end,
                                function(errmsg) print(errmsg) ; print(debug.traceback()) ;  ↩
                                    snippet = nil end)
                        if snippet then base_template = base_template .. snippet end
                end
        end
        local bird4_config = utils.expandVars(base_template, templateVarsIPv4)
        local bird6_config = utils.expandVars(base_template, templateVarsIPv6)

        local peer_template = [[
protocol bgp {
        import filter fromBgp;
        export filter toBgp;

        local as $localAS;
        neighbor $remoteIP as $remoteAS;
}
]]

        local function apply_peer_template(s)
                s.localAS = localAS
                if string.find(s.remoteIP, ":", 1, true) then
                        bird6_config = bird6_config .. utils.expandVars(peer_template, s)
                elseif string.find(s.remoteIP, ".", 1, true) then
                        bird4_config = bird4_config .. utils.expandVars(peer_template, s)
                end
        end
        config.foreach("bgp_peer", apply_peer_template)

        fs.writefile("/etc/bird4.conf", bird4_config)
        fs.writefile("/etc/bird6.conf", bird6_config)
end

function proto.setup_interface(ifname, args)
end

function proto.apply()
    os.execute("/etc/init.d/bird4 restart")
    os.execute("/etc/init.d/bird6 restart")
end

return proto
```

packages/lime-proto-bmx6/Makefile

```
#
# Copyright (C) 2006-2014 OpenWrt.org
#
# This is free software, licensed under the GNU General Public License v3.
#

include $(TOPDIR)/rules.mk

LIME_BUILDDATE:=$(shell date +%Y%m%d_%H%M)
LIME_CODENAME:=bigbang

GIT_COMMIT_DATE:=$(shell git log -n 1 --pretty=%ad --date=short . )
GIT_COMMIT_TSTAMP:=$(shell git log -n 1 --pretty=%at . )

PKG_NAME:=lime-proto-bmx6
PKG_VERSION=$(GIT_COMMIT_DATE)-$(GIT_COMMIT_TSTAMP)

include $(INCLUDE_DIR)/package.mk

define Package/$(PKG_NAME)
  TITLE:=LiMe Bmx6 proto support
  CATEGORY:=LiMe
  MAINTAINER:=Gioacchino Mazzurco <gio@eigenlab.org>
  URL:=http://libre-mesh.org
  DEPENDS:=+bmx7 +bmx7-json +bmx7-sms +bmx7-table +bmx7-uci-config +iptables +lime-system + ↩
      lua +libuci-lua
endef

define Build/Compile
        @rm -rf ./build || true
        @cp -r ./src ./build
        @sed -i '/^--!.*/d' build/*.lua
endef

define Package/$(PKG_NAME)/install
        @mkdir -p $(1)/usr/lib/lua/lime/proto || true
        $(CP) ./build/bmx6.lua $(1)/usr/lib/lua/lime/proto/
endef

$(eval $(call BuildPackage,$(PKG_NAME)))
```

packages/lime-proto-bmx6/src/bmx6.lua

```
#!/usr/bin/lua

local network = require("lime.network")
local config = require("lime.config")
local fs = require("nixio.fs")
local libuci = require("uci")
local wireless = require("lime.wireless")

bmx6 = {}

bmx6.configured = false

function bmx6.configure(args)
        if bmx6.configured then return end
        bmx6.configured = true

        local uci = libuci:cursor()
        local ipv4, ipv6 = network.primary_address()

        fs.writefile("/etc/config/bmx6", "")

        uci:set("bmx6", "general", "bmx6")
        uci:set("bmx6", "general", "dbgMuteTimeout", "1000000")

        uci:set("bmx6", "main", "tunDev")
        uci:set("bmx6", "main", "tunDev", "main")
        uci:set("bmx6", "main", "tun4Address", ipv4:host():string().."/32")
        uci:set("bmx6", "main", "tun6Address", ipv6:host():string().."/128")

        -- Enable bmx6 uci config plugin
        uci:set("bmx6", "config", "plugin")
        uci:set("bmx6", "config", "plugin", "bmx6_config.so")

        -- Enable JSON plugin to get bmx6 information in json format
        uci:set("bmx6", "json", "plugin")
        uci:set("bmx6", "json", "plugin", "bmx6_json.so")

        -- Disable ThrowRules because they are broken in IPv6 with current Linux Kernel
        uci:set("bmx6", "ipVersion", "ipVersion")
        uci:set("bmx6", "ipVersion", "ipVersion", "6")

        -- Search for networks in 172.16.0.0/12
        uci:set("bmx6", "nodes", "tunOut")
        uci:set("bmx6", "nodes", "tunOut", "nodes")
        uci:set("bmx6", "nodes", "network", "172.16.0.0/12")

        -- Search for networks in 192.0.2.0/24 (for testing purpose)
        uci:set("bmx6", "nodes", "tunOut")
        uci:set("bmx6", "nodes", "tunOut", "dummynodes")
        uci:set("bmx6", "nodes", "network", "192.0.2.0/24")

        -- Search for networks in 10.0.0.0/8
        uci:set("bmx6", "clouds", "tunOut")
        uci:set("bmx6", "clouds", "tunOut", "clouds")
        uci:set("bmx6", "clouds", "network", "10.0.0.0/8")

        -- Search for internet in the mesh cloud
        uci:set("bmx6", "inet4", "tunOut")
        uci:set("bmx6", "inet4", "tunOut", "inet4")
        uci:set("bmx6", "inet4", "network", "0.0.0.0/0")
        uci:set("bmx6", "inet4", "maxPrefixLen", "0")

        -- Search for internet IPv6 gateways in the mesh cloud
        uci:set("bmx6", "inet6", "tunOut")
        uci:set("bmx6", "inet6", "tunOut", "inet6")
        uci:set("bmx6", "inet6", "network", "::/0")
        uci:set("bmx6", "inet6", "maxPrefixLen", "0")

        -- Search for other mesh cloud announcements that have public ipv6
        uci:set("bmx6", "publicv6", "tunOut")
        uci:set("bmx6", "publicv6", "tunOut", "publicv6")
        uci:set("bmx6", "publicv6", "network", "2000::/3")
        uci:set("bmx6", "publicv6", "maxPrefixLen", "64")

        -- Announce local ipv4 cloud
        uci:set("bmx6", "local4", "tunIn")
        uci:set("bmx6", "local4", "tunIn", "local4")
        uci:set("bmx6", "local4", "network", ipv4:network():string().."/"..ipv4:prefix())

        -- Announce local ipv6 cloud
        uci:set("bmx6", "local6", "tunIn")
        uci:set("bmx6", "local6", "tunIn", "local6")
        uci:set("bmx6", "local6", "network", ipv6:network():string().."/"..ipv6:prefix())

        if config.get_bool("network", "bmx6_over_batman") then
                for _,protoArgs in pairs(config.get("network", "protocols")) do
                        if(utils.split(protoArgs, network.protoParamsSeparator)[1] == " ↩
                            batadv") then bmx6.setup_interface("bat0", args) end
                end
        end

        uci:save("bmx6")


        uci:delete("firewall", "bmxtun")

        uci:set("firewall", "bmxtun", "zone")
        uci:set("firewall", "bmxtun", "name", "bmxtun")
        uci:set("firewall", "bmxtun", "input", "ACCEPT")
        uci:set("firewall", "bmxtun", "output", "ACCEPT")
        uci:set("firewall", "bmxtun", "forward", "ACCEPT")
        uci:set("firewall", "bmxtun", "mtu_fix", "1")
        uci:set("firewall", "bmxtun", "device", "bmx+")
        uci:set("firewall", "bmxtun", "family", "ipv4")
        uci:save("firewall")
end

function bmx6.setup_interface(ifname, args)
        if ifname:match("^wlan%d+_ap") then return end
        vlanId = args[2] or 13
        vlanProto = args[3] or "8021ad"
        nameSuffix = args[4] or "_bmx6"

        local owrtInterfaceName, linux802adIfName, owrtDeviceName = network.createVlanIface ↩
            (ifname, vlanId, nameSuffix, vlanProto)

        local uci = libuci:cursor()
        uci:set("network", owrtDeviceName, "mtu", "1398")

        -- BEGIN [Workaround issue 38]
        if ifname:match("^wlan%d+") then
                local macAddr = wireless.get_phy_mac("phy"..ifname:match("%d+"))
                local vlanIp = { 169, 254, tonumber(macAddr[5], 16), tonumber(macAddr[6],  ↩
                    16) }
                uci:set("network", owrtInterfaceName, "proto", "static")
                uci:set("network", owrtInterfaceName, "ipaddr", table.concat(vlanIp, "."))
                uci:set("network", owrtInterfaceName, "netmask", "255.255.255.255")
        end
        --- END [Workaround issue 38]

        uci:save("network")

        uci:set("bmx6", owrtInterfaceName, "dev")
        uci:set("bmx6", owrtInterfaceName, "dev", linux802adIfName)
        uci:save("bmx6")
end

function bmx6.apply()
    os.execute("killall bmx6 ; sleep 2 ; killall -9 bmx6")
    os.execute("bmx6")
end

function bmx6.bgp_conf(templateVarsIPv4, templateVarsIPv6)
        local uci = libuci:cursor()

        -- Enable Routing Table Redistribution plugin
        uci:set("bmx6", "table", "plugin")
        uci:set("bmx6", "table", "plugin", "bmx6_table.so")

        -- Redistribute proto bird routes
        uci:set("bmx6", "fromBird", "redistTable")
        uci:set("bmx6", "fromBird", "redistTable", "fromBird")
        uci:set("bmx6", "fromBird", "table", "254")
        uci:set("bmx6", "fromBird", "bandwidth", "100")
        uci:set("bmx6", "fromBird", "proto", "12")

        -- Avoid aggregation as it use lot of CPU with huge number of routes
        uci:set("bmx6", "fromBird", "aggregatePrefixLen", "128")

        -- Disable proactive tunnels announcement as it use lot of CPU with
        -- huge number of routes
        uci:set("bmx6", "general", "proactiveTunRoutes", "0")

        -- BMX6 security features are at moment not used by LiMe, disable hop
        -- by hop links signature as it consume a lot of CPU expecially in
        -- setups with multiples interfaces  and lot of routes like LiMe
        uci:set("bmx6", "general", "linkSignatureLen", "0")

        uci:save("bmx6")

        local base_bgp_conf = [[
protocol direct {
        interface "bmx*";
}
]]

        return base_bgp_conf
end

return bmx6
```

packages/lime-system/files/etc/config/lime

```
# The options marked with "# Parametrizable with %Mn, %Nn, %H"
# can include %Mn templates that will be substituted
# with the n’th byte of the primary_interface MAC
# and %Nn templates that will be replaced by the n’th network-identifier byte,
# calculated from the hash of the ap_ssid value, so that all the nodes that
# form a mesh cloud (share the same ap_ssid) will produce the same value
# and %H template that will be replaced by hostname


### System options

#config lime system
#       option hostname ’LiMeNode-%M4%M5%M6’                                 # ↩
    Parametrizable with %Mn


### Network general option

#config lime network
#       option primary_interface eth0                                        # The mac  ↩
    address of this device will be used in different places
#       option bmx6_over_batman false                                        # Disables  ↩
    Bmx6 meshing on top of batman
#       option main_ipv4_address ’192.0.2.0/24’                              # ↩
    Parametrizable with %Mn, %Nn
#       option main_ipv6_address ’2001:db8::%M5:%M6/64’                      # ↩
    Parametrizable with %Mn, %Nn
#       list protocols adhoc                                                 # List of  ↩
    protocols configured by LiMe
#       list protocols lan
#       list protocols anygw
#       list protocols batadv:%N1                                            # ↩
    Parametrizable with %Nn
#       list protocols bmx6:13
#       list protocols bgp:65551                                             # BGP  ↩
    protocol take AS number as param
#       list resolvers 8.8.8.8                                               # DNS  ↩
    servers node will use
#       list resolvers 2001:4860:4860::8844


### WiFi general options

#config lime wifi
```

```
#       option channel_2ghz '11'
#       option channel_5ghz '48'
#       list modes 'ap'
#       list modes 'adhoc'
#       option ap_ssid 'LiMe'
#       option adhoc_ssid 'libre-mesh'                                  #  ↩
    Parametrizable with %M, %H
#       option adhoc_bssid 'ca:fe:00:c0:ff:ee'
#       option adhoc_mcast_rate_2ghz '24000'
#       option adhoc_mcast_rate_5ghz '6000'
#       option mesh_mesh_fwding '0'
#       option mesh_mesh_id 'LiMe'


### WiFi interface specific options ( override general option )

#config wifi radio11
#       list modes 'adhoc'
#       option channel_2ghz '1'
#       option channel_5ghz '48'
#       option adhoc_mcast_rate '6000'
#       option adhoc_ssid 'libre-mesh'
#       option adhoc_bssid 'ca:fe:00:c0:ff:ee'

#config wifi radio12
#       list modes 'manual'                                              # If you use ↩
     manual protocol you must not specify other protocol, or your configuration will be  ↩
    broken!


### Network interface specific options ( override general option )
### Available protocols: bmx6, batadv, wan, lan, manual
### proto:vlan_number works too ( something like bmx6:13 is supported )
### If you use manual do not specify other protocols, may result in an unpredictable  ↩
    behavior/configuration (likely you loose connection to the node)

#config net eth5
#       option linux_name 'eth5'                                         # Should use ↩
     this because interface name can contains dots like eth0.2 while uci section names  ↩
    cannot
#       list protocols 'manual'


### Ground routing specific sections
### One section for each ground routing link

#config hwd_gr link1
#       option net_dev 'eth0'                                         # Plain  ↩
    ethernet device on top of which 802.1q vlan will be constructed
#       option vlan '5'                                               # Vlan id to  ↩
    use for this ground routing link, use little one because cheap switch doesn't supports  ↩
    big ids, this will bi used also as 802.1q vid
#       option switch_dev 'switch0'                                   # If your  ↩
    ethernet device is connected to a switch chip you must specify it
#       option switch_cpu_port '0'                                    # Refer to  ↩
    switch port map of your device on openwrt wiki to know CPU port index
#       list switch_ports '4'                                         # List switch  ↩
    ports on with you want the vlan being passed


### Proto BGP specific sections
### One section for each BGP peer
```

```
#config bgp_peer peer1
#       option remoteIP '192.0.2.6'
#       option remoteAS '65550'

#config bgp_peer peer2
#       option remoteIP '2001:db8::c001'
#       option remoteAS '65549'
```

packages/lime-system/files/usr/lib/lua/lime/proto/lan.lua

```
#!/usr/bin/lua

lan = {}

local network = require("lime.network")
local libuci = require("uci")

lan.configured = false

function lan.configure(args)
        if lan.configured then return end
        lan.configured = true

        local ipv4, ipv6 = network.primary_address()
        local uci = libuci:cursor()
        uci:set("network", "lan", "ip6addr", ipv6:string())
        uci:set("network", "lan", "ipaddr", ipv4:host():string())
        uci:set("network", "lan", "netmask", ipv4:mask():string())
        uci:set("network", "lan", "proto", "static")
        uci:set("network", "lan", "mtu", "1500")
        uci:delete("network", "lan", "ifname")
        uci:save("network")
end

function lan.setup_interface(ifname, args)
        if args and args["nobridge"] then return end
        if ifname:match("^wlan") then return end
        if ifname:match(network.protoVlanSeparator.."%d+$") then return end

        local uci = libuci:cursor()
        local bridgedIfs = {}
        local oldIfs = uci:get("network", "lan", "ifname") or {}
        if type(oldIfs) == "string" then oldIfs = utils.split(oldIfs, " ") end
        for _,iface in pairs(oldIfs) do
                if iface ~= ifname then
                        table.insert(bridgedIfs, iface)
                end
        end
        table.insert(bridgedIfs, ifname)
        uci:set("network", "lan", "ifname", bridgedIfs)
        uci:save("network")
end

function lan.bgp_conf(templateVarsIPv4, templateVarsIPv6)
        local base_conf = [[
protocol direct {
        interface "br-lan";
}
]]
        return base_conf
end
```

```
return lan
```

packages/lime-system/files/usr/lib/lua/lime/utils.lua

```
#!/usr/bin/lua

utils = {}

local config = require("lime.config")


function utils.split(string, sep)
    local ret = {}
    for token in string.gmatch(string, "[^"..sep.."]+") do table.insert(ret, token) end
    return ret
end

function utils.stringStarts(string, start)
   return (string.sub(string, 1, string.len(start)) == start)
end

function utils.stringEnds(string, _end)
   return ( _end == '' or string.sub( string, -string.len(_end) ) == _end)
end


function utils.hex(x)
    return string.format("%02x", x)
end

function utils.printf(fmt, ...)
    print(string.format(fmt, ...))
end

function utils.isModuleAvailable(name)
        if package.loaded[name] then
                return true
        else
                for _, searcher in ipairs(package.searchers or package.loaders) do
                        local loader = searcher(name)
                        if type(loader) == 'function' then
                                package.preload[name] = loader
                                return true
                        end
                end
                return false
        end
end

function utils.applyMacTemplate16(template, mac)
        for i=1,6,1 do template = template:gsub("%%M"..i, mac[i]) end
        return template
end

function utils.applyMacTemplate10(template, mac)
        for i=1,6,1 do template = template:gsub("%%M"..i, tonumber(mac[i], 16)) end
        return template
end

function utils.applyHostnameTemplate(template)
        local system = require("lime.system")
```

```lua
        return template:gsub("%%H", system.get_hostname())
end

function utils.network_id()
    local network_essid = config.get("wifi", "ap_ssid")
    local netid = {}
    local fd = io.popen('echo "' .. network_essid .. '" | md5sum')
    if fd then
        local md5 = fd:read("*a")
        netid[1] = md5:match("^(..)")
        netid[2] = md5:match("^..(..)")
        netid[3] = md5:match("^....(..)")
        fd:close()
    end
    return netid
end

function utils.applyNetTemplate16(template)
        local netid = utils.network_id()
        for i=1,3,1 do template = template:gsub("%%N"..i, netid[i]) end
        return template
end

function utils.applyNetTemplate10(template)
        local netid = utils.network_id()
        for i=1,3,1 do template = template:gsub("%%N"..i, tonumber(netid[i], 16)) end
        return template
end


--! This function is inspired to http://lua-users.org/wiki/VarExpand
--! version: 0.0.1
--! code: Ketmar // Avalon Group
--! licence: public domain
--! expand $var and ${var} in string
--! ${var} can call Lua functions: ${string.rep(' ', 10)}
--! `$' can be screened with `\'
--! `...': args for $<number>
--! if `...' is just a one table -- take it as args
function utils.expandVars(s, ...)
        local args = {...}
        args = #args == 1 and type(args[1]) == "table" and args[1] or args;

        --! return true if there was an expansion
        local function DoExpand(iscode)
                local was = false
                local mask = iscode and "()%$(%b{})" or "()%$([%a%d_]*)"
                local drepl = iscode and "\\$" or "\\\\$"
                s = s:gsub(mask,
                        function(pos, code)
                                if s:sub(pos-1, pos-1) == "\\" then
                                        return "$"..code
                                else
                                        was = true
                                        local v, err
                                        if iscode then
                                                code = code:sub(2, -2)
                                        else
                                                local n = tonumber(code)
                                                if n then
                                                        v = args[n]
                                                else
```

```
                                                        v = args[code]
                                                end
                                        end
                                        if not v then
                                                v, err = loadstring("return "..code)
                                                if not v then error(err) end
                                                v = v()
                                        end
                                        if v == nil then v = "" end
                                        v = tostring(v):gsub("%$", drepl)
                                        return v
                                end
                end)
                if not (iscode or was) then s = s:gsub("\\%$", "$") end
                return was
        end
        repeat DoExpand(true); until not DoExpand(false)
        return s
end

return utils
```

packages/lime-system/files/usr/lib/lua/lime/wireless.lua

```
#!/usr/bin/lua

local config = require("lime.config")
local network = require("lime.network")
local utils = require("lime.utils")
local libuci = require("uci")
local fs = require("nixio.fs")

wireless = {}

wireless.modeParamsSeparator=":"
wireless.limeIfNamePrefix="lm_"
wireless.ifnameModeSeparator="_"

function wireless.get_phy_mac(phy)
        local path = "/sys/class/ieee80211/"..phy.."/macaddress"
        local mac = assert(fs.readfile(path), "wireless.get_phy_mac(..) failed reading: " ↩
            ..path):gsub("\n","")
        return utils.split(mac, ":")
end

function wireless.clean()
        print("Clearing wireless config...")
        local uci = libuci:cursor()
        uci:foreach("wireless", "wifi-iface", function(s) uci:delete("wireless", s[".name" ↩
            ]) end)
        uci:save("wireless")
end

function wireless.scandevices()
        local devices = {}
        local uci = libuci:cursor()
        uci:foreach("wireless", "wifi-device", function(dev) devices[dev[".name"]] = dev  ↩
            end)
        return devices
end

function wireless.is5Ghz(radio)
```

```lua
        local uci = libuci:cursor()
        local hwmode = uci:get("wireless", radio, "hwmode") or "11ng"
        if hwmode:find("a") then
                return true
        end
        return false
end

wireless.availableModes = { adhoc=true, ap=true }
function wireless.isMode(m)
        return wireless.availableModes[m]
end

function wireless.createBaseWirelessIface(radio, mode, extras)
--! checks("table", "string", "?table")
--! checks(...) come from http://lua-users.org/wiki/LuaTypeChecking -> https://github.com/ ↩
    fab13n/checks

        local radioName = radio[".name"]
        local phyIndex = radioName:match("%d+")
        local ifname = "wlan"..phyIndex..wireless.ifnameModeSeparator..mode
        local wirelessInterfaceName = wireless.limeIfNamePrefix..ifname.."_"..radioName
        local networkInterfaceName = network.limeIfNamePrefix..ifname

        local uci = libuci:cursor()

        uci:set("wireless", wirelessInterfaceName, "wifi-iface")
        uci:set("wireless", wirelessInterfaceName, "mode", mode)
        uci:set("wireless", wirelessInterfaceName, "device", radioName)
        uci:set("wireless", wirelessInterfaceName, "ifname", ifname)
        uci:set("wireless", wirelessInterfaceName, "network", networkInterfaceName)

        if extras then
                for key, value in pairs(extras) do
                        uci:set("wireless", wirelessInterfaceName, key, value)
                end
        end

        uci:save("wireless")

        return uci:get_all("wireless", wirelessInterfaceName)
end

function wireless.configure()
        local specificRadios = {}
        config.foreach("wifi", function(radio) specificRadios[radio[".name"]] = radio end)

        local allRadios = wireless.scandevices()
        for _,radio in pairs(allRadios) do
                local radioName = radio[".name"]
                local phyIndex = radioName:match("%d+")
                if wireless.is5Ghz(radioName) then
                        freqSuffix = "_5ghz"
                        ignoredSuffix = "_2ghz"
                else
                        freqSuffix = "_2ghz"
                        ignoredSuffix = "_5ghz"
                end
                local modes = config.get("wifi", "modes")
                local options = config.get_all("wifi")

                local specRadio = specificRadios[radioName]
                if specRadio then
                        modes = specRadio["modes"]
                        options = specRadio
                end

                local uci = libuci:cursor()
                uci:set("wireless", radioName, "disabled", 0)
                uci:set("wireless", radioName, "channel", options["channel"..freqSuffix])
                uci:save("wireless")

                for _,modeArgs in pairs(modes) do
                        local args = utils.split(modeArgs, wireless.modeParamsSeparator)
                        local modeName = args[1]

                        if modeName == "manual" then break end

                        local mode = require("lime.mode."..modeName)
                        local wirelessInterfaceName = mode.setup_radio(radio, args)[".name" ↩
                            ]

                        local uci = libuci:cursor()

                        for key,value in pairs(options) do
                                local keyPrefix = utils.split(key, "_")[1]
                                local isGoodOption = ( (key ~= "modes")
                                                   and (not key:match("^%."))
                                                   and (not key:match("channel"))
                                                   and (not (wireless.isMode(keyPrefix) and ↩
                                                        keyPrefix ~= modeName))
                                                   and (not key:match(ignoredSuffix)) )

                                if isGoodOption then
                                        local nk = key:gsub("^"..modeName.."_", ""):gsub( ↩
                                            freqSuffix.."$", "")
                                        if nk == "ssid" then
                                                value = utils.applyHostnameTemplate(value)
                                                value = utils.applyMacTemplate16(value,  ↩
                                                    network.primary_mac())
                                                value = string.sub(value, 1, 32)
                                        end

                                        uci:set("wireless", wirelessInterfaceName, nk,  ↩
                                            value)
                                end
                        end

                        uci:save("wireless")
                end
        end
end

return wireless
```

7 Appendix: License

8 Glossary

- **Ad Hoc**: The 802.11 standard specifies "ad hoc" mode, which allows the radio network interface card (NIC) to operate in what the standard refers to as an independent basic service set (IBSS) network configuration. With an IBSS, there are no access points. User devices communicate directly with each other in a peer-to-peer manner [124].
- **Almquist shell**: The Almquist shell (also known as A Shell, sh and ash) is a lightweight Unix shell originally written by Kenneth Almquist in the late 1980s [125].
- **Anycast**: Anycast is a network addressing and routing methodology in which datagrams from a single sender are routed to the topologically nearest node in a group of potential receivers, though it may be sent to several nodes, all identified by the same destination address [126].
- **AP**: In computer networking, a wireless access point (AP) is a device that allows wireless devices to connect to a wired network using Wi-Fi, or related standards. The AP usually connects to a router (via a wired network) as a standalone device, but it can also be an integral component of the router itself. An AP is differentiated from a hotspot, which is the physical space where the wireless service is provided [127].
- **API**: In computer programming, an application programming interface (API) is a set of routines, protocols, and tools for building software applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types. An API defines functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface [128].
- **ARP**: The Address Resolution Protocol (ARP) is a telecommunication protocol used for resolution of network layer addresses into link layer addresses, a critical function in multiple-access networks [129].
- **AS**: Within the Internet, an autonomous system (AS) is a collection of connected Internet Protocol (IP) routing prefixes under the control of one or more network operators on behalf of a single administrative entity or domain that presents a common, clearly defined routing policy to the Internet [130].
- **ASN**: An autonomous system number (ASN) is a unique number to identify an autonomous system (AS) and which enables that system to exchange exterior routing information with other neighboring autonomous systems [130].
- **Babel**: The Babel routing protocol is a distance-vector routing protocol for Internet Protocol packet-switched networks that is designed to be robust and efficient on both wireless mesh networks and wired networks [131].
- **BGP**: Border Gateway Protocol (BGP) is a protocol designed to exchange routing and reachability information between autonomous systems (AS) on the Internet [132].
- **BIRD**: BIRD Internet Routing Daemon is a network routing software providing implementations of Border Gateway Protocol (BGP), Open Shortest Path First (OSPF) and others routing protocols [6].
- **Broadcast**: In computer networking, broadcasting refers to transmitting a packet that will be received by every device on the network [133].
- **BSSID**: In computer networking, a service set is a set consisting of all the devices associated with a wireless LAN. Basic service set identification (BSSID) is the formal name always associated with only one wireless LAN [134].

- **C**: C is a general-purpose, imperative computer programming language [135].
- **CIDR**: Classless Inter-Domain Routing (CIDR) is a method for allocating IP addresses and routing Internet Protocol packets. The Internet Engineering Task Force introduced CIDR in 1993 to replace the previous addressing architecture of classful network design in the Internet. Its goal was to slow the growth of routing tables on routers across the Internet, and to help slow the rapid exhaustion of IPv4 addresses [136].
- **Community Network**: A community network is a computer-based system that is intended to help support geographical communities by supporting, augmenting, and extending already existing social networks.
- **Drupal**: Drupal is a free and open-source web content-management framework written in PHP and distributed under the GNU General Public License [137].
- **Dual stack**: To an Internet host being dual stack implies providing complete implementations of both versions of the Internet Protocol (IPv4 and IPv6) [138].
- **eBGP**: Short for External Border Gateway Protocol, eBGP is the protocol used to transport information to other BGP enabled systems in different autonomous systems (AS) [54].
- **Ethernet**: Ethernet is a family of computer networking technologies for local area networks (LANs) [139].
- **Firmware**: Firmware is a type of software that usually control low-level components of the device it is usually held in ROM. While this is the general accepted meaning of the term, in the context of embedded routing and in community networks it is common practice to call firmware the software running on the routers, while it is usually flashed like a proper firmware it is really a software providing a full operative system and higher levels tools such as web interface or command line interface [140].
- **Hotplug**: On Linux systems Hotplug lets you plug in new devices and use them immediately [38].
- **IP**: The Internet Protocol (IP) is the principal communications protocol in the Internet protocol suite for relaying datagrams across network boundaries. Its routing function enables inter-networking, and essentially establishes the Internet [141].
- **IPv4**: Internet Protocol version 4 (IPv4) is the fourth version in the development of the Internet Protocol (IP) [142].
- **IPv6**: Internet Protocol version 6 (IPv6) is the most recent version of the Internet Protocol (IP) [143].
- **LAN**: A local area network (LAN) is a computer network that interconnects computers within a limited area such as a residence, school, laboratory, or office building [144].
- **Last mile**: Last mile is the common colloquialism referring to that portion of the telecommunications network chain that physically reaches the end-user [145].
- **LiMe**: Common abbreviation for Libre-Mesh [146].
- **Ln**: Abbreviation of layer n, it refers to Internet layers, as an example L2 refer to Ethernet the layer 2 of internet stack.
- **Lua**: Lua is a lightweight multi-paradigm programming language designed as a scripting language with extensible semantics as a primary goal [147].
- **MAC address**: A media access control address (MAC address) is a unique identifier assigned to network interfaces for communications on the physical network segment [43].
- **Mesh Network**: A mesh network is a network topology in which each node relays data for the network. All mesh nodes cooperate in the distribution of data in the network [148].
- **NAT**: Network address translation (NAT) is a methodology of remapping one IP address space into another by modifying network address information in Internet Protocol (IP) datagram packet headers while they are in transit across a traffic routing device [24].
- **Netlink**: Netlink socket family is a Linux kernel interface used for inter-process communication (IPC) between both the kernel and userspace processes, and between different userspace processes [149].
- **NIC**: A network interface controller (NIC) is a computer hardware component that connects a computer to a computer network [150].

- **OLSR**: The Optimized Link State Routing Protocol (OLSR) is an IP routing protocol optimized for mobile ad hoc networks, which can also be used on other wireless ad hoc networks [151].
- **OpenWrt**: OpenWrt is an operating system based on the Linux kernel, primarily used on embedded devices to route network traffic. All components OpenWrt ships are optimized for size, to be small enough for fitting into the limited storage and memory available in home routers [152].
- **OSPF**: Open Shortest Path First (OSPF) is a routing protocol for Internet Protocol (IP) networks. It uses a link state routing algorithm and falls into the group of interior routing protocols, operating within a single autonomous system (AS) [153].
- **Peering**: In computer networking, peering is a voluntary interconnection of administratively separate Internet networks for the purpose of exchanging traffic between the users of each network [154].
- **Quagga**: Quagga is a network routing software suite providing implementations of Open Shortest Path First (OSPF), Routing Information Protocol (RIP), Border Gateway Protocol (BGP) and IS-IS for Unix-like platforms [155].
- **Radio**: In OpenWrt context radio refers to the pysical part of a WiFi NIC [156].
- **Router**: Routers are devices which forward packets between interconnected networks in order to allow hosts not connected directly to the same local area network to communicate with each other [157].
- **Routing Daemon**: A Routing Daemon is in UNIX terminology a non-interactive program running on background which does the dynamic part of Internet routing, that is it communicates with the other routers, calculates routing tables and sends them to the OS kernel which does the actual packet forwarding [158].
- **RFC**: A Request for Comments (RFC) is a publication of the Internet Engineering Task Force (IETF) and the Internet Society, the principal technical development and standards-setting bodies for the Internet. An RFC is authored by engineers and computer scientists in the form of a memorandum describing methods, behaviors, research, or innovations applicable to the working of the Internet and Internet-connected systems. It is submitted either for peer review or simply to convey new concepts, information, or (occasionally) engineering humor. The IETF adopts some of the proposals published as RFCs as Internet standards [159].
- **Socket**: A network socket is an endpoint of an inter-process communication across a computer network. Today, most communication between computers is based on the Internet Protocol; therefore most network sockets are Internet sockets. A socket API is an application programming interface (API), usually provided by the operating system, that allows application programs to control and use network sockets. Internet socket APIs are usually based on the Berkeley sockets standard [160].
- **SSID**: In computer networking, a service set is a set consisting of all the devices associated with a wireless LAN. Service sets have an associated identifier, the Service Set Identifier (SSID), which consists of 32 octets that frequently contains a human readable identifier of the network [161].
- **STA**: In IEEE 802.11 (Wi-Fi) terminology, a station (STA) is a device that has the capability to use the 802.11 protocol. Generally in wireless networking terminology, a station, wireless client and node are often used interchangeably, with no strict distinction existing between these terms [162].
- **Subnet** : A subnetwork, or subnet, is a logical, visible subdivision of an IP network. The practice of dividing a network into two or more networks is called subnetting. Computers that belong to a subnet are addressed with a common, identical, most-significant bit-group in their IP address. This results in the logical division of an IP address into two fields, a network or routing prefix and the rest field or host identifier. The rest field is an identifier for a specific host or network interface [163].
- **TCP**: The Transmission Control Protocol (TCP) is a core protocol of the Internet Protocol Suite. TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating over an IP network [164].
- **TTL**: Time to live (TTL) or hop limit is a mechanism that limits the lifespan or lifetime of data in a computer or network. TTL may be implemented as a counter or timestamp attached to or embedded in the data. Once the prescribed event count or timespan has elapsed, data is discarded. In computer networking, TTL prevents a data packet from circulating indefinitely [165].
- **UML**: The Unified Modeling Language (UML) is a general-purpose modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system [166].

- **Unicast**: In computer networking, unicast transmission is the sending of messages to a single network destination identified by a unique address [167].
- **USB**: USB, short for Universal Serial Bus, is an industry standard developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication, and power supply between computers and electronic devices [168].
- **Vendor lock-in**: Vendor lock-in makes a customer dependent on a vendor for products and services, unable to use another vendor without substantial switching costs. [169]
- **VLAN**: A virtual LAN (VLAN) is any broadcast domain that is partitioned and isolated in a computer network at the data link layer (L2) [170].
- **WAN**: A wide area network (WAN) is a telecommunications network or computer network that extends over a large geographical distance [171].
- **WDS**: A wireless distribution system (WDS) is a system enabling the wireless interconnection of access points in an IEEE 802.11 network. It allows a wireless network to be expanded using multiple access points without the traditional requirement for a wired backbone to link them. The notable advantage of WDS over other solutions is it preserves the MAC addresses of client frames across links between access points [172].
- **WiFi**: WiFi is a local area wireless computer networking technology that allows electronic devices to network, mainly using the 2.4 gigahertz (12 cm) UHF and 5 gigahertz (6 cm) SHF ISM radio bands [173].

9 References

- [1] Wikipedia. 2015. Community network. https://en.wikipedia.org/wiki/Community_network
- [2] Wiki Guifi.net. 2015. Supernodos híbridos. http://es.wiki.guifi.net/wiki/Supernodos_h%C3%ADbridos
- [3] Wiki Guifi.net. 2015. Modelo híbrido guifi.net. http://es.wiki.guifi.net/wiki/Modelo_h%C3%ADbrido_guifi.net
- [4] BIRD source code. 2015. BIRD FAQ. https://gitlab.labs.nic.cz/labs/bird/wikis/FAQ
- [5] Vincent Bernat. 2015. Network lab: site to site VPN. http://vincent.bernat.im/en/blog/2011-lab-site-to-site-vpn.html
- [6] Wikipedia. 2015. Bird Internet routing daemon. https://en.wikipedia.org/wiki/Bird_Internet_routing_daemon
- [7] OpenWrt source code. 2015. BIRD into OpenWrt routing feed. https://github.com/openwrt-routing/packages/tree/master/-bird
- [8] BIRD documentation. 2015. Filters. http://bird.network.cz/?get_doc&f=bird-5.html
- [9] Preston, Tim. 2010. BIRD Route Server at LINX. http://www.uknof.org.uk/uknof15/Preston-Routeserver.pdf
- [10] Glenn Daneels. 2015. Analysis of the BMX6 routing protocol. http://bmx6.net/attachments/download/134/Thesis_AnalysisOfTh
- [11] Wikipedia. 2015. Table-driven proactive routing. https://en.wikipedia.org/wiki/List_of_ad_hoc_routing_protocols#Table-driven_.28proactive.29_routing
- [12] Wikipedia. 2015. Distance-vector routing protocol. https://en.wikipedia.org/wiki/Distance-vector_routing_protocol
- [13] Wikipedia. 2015. Context switch. https://en.wikipedia.org/wiki/Context_switch
- [14] Open-mesh.org. 2015. B.A.T.M.A.N. advanced. http://www.open-mesh.org/projects/batman-adv/wiki/Wiki
- [15] Quick Mesh Project. 2015. Easy deployment of Mesh networks. http://qmp.cat/
- [16] Jorge L. Florit. 2014. Integration of MANET/Mesh networks with qMp in the Guifi.net community. http://dev.qmp.cat/-attachments/download/143/PFC_qMp-Guifi.pdf
- [17] Wiki Guifi.net. 2015. Node frontera amb qMp. http://ca.wiki.guifi.net/wiki/Node_frontera_amb_qMp

- [18] Wikipedia. 2015. Wireless mesh network. https://en.wikipedia.org/wiki/Wireless_mesh_network
- [19] Wikipedia. 2015. Internet protocol suite. https://en.wikipedia.org/wiki/Internet_protocol_suite
- [20] Wikipedia. 2015. Network switch. https://en.wikipedia.org/wiki/Network_switch
- [21] Wikipedia. 2015. Broadcast domain. https://en.wikipedia.org/wiki/Broadcast_domain
- [22] Wikipedia. 2015. ARP spoofing. https://en.wikipedia.org/wiki/ARP_spoofing
- [23] Wikipedia. 2015. Hole punching (networking). https://en.wikipedia.org/wiki/Hole_punching_%28networking%29
- [24] Wikipedia. 2015. Network address translation. https://en.wikipedia.org/wiki/Network_address_translation
- [25] Wikipedia. 2015. End-to-end principle. https://en.wikipedia.org/wiki/End-to-end_principle
- [26] The Internet Society. 2010. IPv4 Address Blocks Reserved for Documentation. https://tools.ietf.org/html/rfc5737
- [27] Sarabjeet Singh Chugh. 2003. Impact of Network Address Translation on Router Performance. http://scholar.lib.vt.edu/-theses/available/etd-10062003-170440/unrestricted/thesis.pdf
- [28] The Internet Society. 2004. IPv6 Address Prefix Reserved for Documentation. https://tools.ietf.org/html/rfc3849
- [29] Lua Users Wiki. 2015. Lua Versus Python. http://lua-users.org/wiki/LuaVersusPython
- [30] OpenWrt Wiki. 2015. Ethernet Network Switch. http://wiki.openwrt.org/doc/hardware/switch
- [31] OpenWrt Wiki. 2015. Switch Documentation. http://wiki.openwrt.org/doc/uci/network/switch
- [32] Ninux.org Wiki. 2015. Ground Routing. http://wiki.ninux.org/GroundRouting
- [33] Libre-Mesh Wiki. 2015. Ground Routing. http://libre-mesh.org/projects/libre-mesh/wiki/Ground_routing
- [34] OpenWrt Wiki. 2015. TP-Link TL-WDR3600. http://wiki.openwrt.org/toh/tp-link/tl-wdr3600
- [35] OpenWrt Developers. 2015. OpenWrt code, creation of default network configuration. https://github.com/openwrt-mirror/openwrt/blob/master/target/linux/ar71xx/base-files/etc/uci-defaults/02_network
- [36] LiMe Developers. 2015. Libre-mesh code, 99-lime-config. https://github.com/libre-mesh/lime-packages/blob/develop/-packages/lime-system/files/etc/uci-defaults/99-lime-config
- [37] OpenWrt Wiki. 2015. Hotplug. http://wiki.openwrt.org/doc/techref/hotplug
- [38] Linux Hotplugging Project. 2015. hotplug(8) - Linux man page. http://linux.die.net/man/8/hotplug
- [39] LiMe Developers. 2015. Libre-mesh code, USB radio hotplug hook. https://github.com/libre-mesh/lime-packages/blob/-develop/packages/lime-hwd-usbradio/src/hotplug-hook.sh
- [40] Wikipedia. 2015. Wireless LAN. https://en.wikipedia.org/wiki/Wireless_LAN
- [41] Waldner, Backreference.org. 2014. Some notes on macvlan and macvtap. http://backreference.org/2014/03/20/some-notes-on-macvlanmacvtap/
- [42] Toshiaki Makita, NTT Open Source Software Center. 2014. Virtual switching technologies and Linux bridge http://events.linuxfou sites/events/files/slides/LinuxConJapan2014_makita_0.pdf
- [43] Wikipedia. 2015. MAC Address. https://en.wikipedia.org/wiki/MAC_address
- [44] Cisco.net. 2015. BGP Attribute Filter and Enhanced Attribute Error Handling. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bgp/configuration/xe-3s/irg-xe-3s-book/bgp_attribute_filter_and_enhanced_attribute_error_handling.html
- [45] Ivan Pepelnjak. 2015. AS-path prepending (technical details). http://wiki.nil.com/AS-path_prepending_%28technical_details%29
- [46] Axel Neumann. 2015. --redistTable /all=1 /sys=int. https://github.com/axn/bmx6/commit/06c7bebecf94e463103bda02169fc7f93
- [47] Axel Neumann. 2015. ip table: fixing ERROR rtnl_rcv: No buffer space available. https://github.com/axn/bmx6/commit/-f0e8c3ea176ba442031ec858266c026ef52eaf13

- [48] Axel Neumann. 2015. table redist: CPU optimize redist_table_routes(). https://github.com/axn/bmx6/commit/7a825c2c74430903
- [49] Axel Neumann. 2015. table: less redist_table_routes debugging to syslog. https://github.com/axn/bmx6/commit/-1ffda19b4accd02272e8c3b9fc6c1f0b68327e4b
- [50] Axel Neumann. 2015. table --redistTableDelay=2200. https://github.com/axn/bmx6/commit/b6107880b5460625b7ce9359dbeac0
- [51] Axel Neumann. 2015. hna redist table quagga ip: export, redistribute, and import tunnel routes using --proto --advProto --setProto subparameters. https://github.com/axn/bmx6/commit/36feab16656cb3b3dd9af8346c3bf6f98ca9c380
- [52] Wikipedia. 2015. Routing loop problem. https://en.wikipedia.org/wiki/Routing_loop_problem
- [53] Wikipedia. 2015. Border Gateway Protocol. https://en.wikipedia.org/wiki/Border_Gateway_Protocol
- [54] Juniper Networks. 2015. IBGP and EBGP. https://www.juniper.net/techpubs/software/junos-es/junos-es93/junos-es-swconfig-interfaces-and-routing/ibgp-and-ebgp.html
- [55] Battlemesh Community. 2015. BattlemeshV8. http://battlemesh.org/BattleMeshV8
- [56] Axel Neumann. 2015. Table redist: filter out temporary table-route changes --redistTableInDelay --redistTableOutDelay. https://github.com/axn/bmx6/commit/59d09f44e7b9c04a9546a273fdc559e2d52cfaad
- [57] Axel Neumann. 2015. Hna: --proactiveTunRoutes=1. https://github.com/axn/bmx6/commit/56f1cb7a28e8eabd72c7500d4405290
- [58] Axel Neumann, Leandro Navarro, Roger Baig, Pau Escrich. 2012. http://bmx6.net/attachments/download/108/CNBuB2012_BMX
- [59] Axel Neumann. 2013. Advances of BMX6 Mesh Routing Protocol. http://bmx6.net/attachments/download/137/is4cwn2013-bmx6.pdf
- [60] Axel Neumann. 2015. Sec: --linkSignatureLen. . . . https://github.com/axn/bmx6/commit/48b8674c2a4f7ed0f01b20ab62a35ba2f
- [61] Free Software Foundation. 2015. What is free software?. http://www.gnu.org/philosophy/free-sw.en.html
- [62] OpenWrt source code. 2015. Update to latest bmx6-master branch. https://github.com/openwrt-routing/packages/-commit/cfefe9fccae2332d147fd9bae7c72d15a42d1982
- [63] OpenWrt source code. 2015. Update to latest semtor-branch version. https://github.com/openwrt-routing/packages/-commit/5bcc48d5d032dd53ae58d03014b59e879eec2c3f
- [64] OpenWrt source code. 2015. Performance fixes. https://github.com/openwrt-routing/packages/commit/70db9d7197c9b5f113b10
- [65] Axel Neumann. 2015. Ip table: detect and fix rule corruption on the fly. https://github.com/axn/bmx6/commit/c04a3bb6acca75ff8
- [66] Axel Neumann. 2015. Dump: rerender --traffic and add traffic=devs. https://github.com/axn/bmx6/commit/49833b2dedf799239e
- [67] Axel Neumann. 2015. Prof: define --cpu argument in cpu.h. https://github.com/axn/bmx6/commit/09ec79c4b4f7af424c4d26d9f8
- [68] Axel Neumann. 2015. Control: add debuglevel 6,7,9. https://github.com/axn/bmx6/commit/2e706662eec5a010a8407e0b5dd09fa
- [69] Axel Neumann. 2015. --show=X neigh→nb for shorter line lengths. https://github.com/axn/bmx6/commit/62d547d785ad2650efc
- [70] Axel Neumann. 2015. Shorten originators. https://github.com/axn/bmx6/commit/e5e70071ef7c74b04002c4c62cc689448cf4484c
- [71] Axel Neumann. 2015. Ip table: less error debug logs. https://github.com/axn/bmx6/commit/c893847b458294e4559b8523f3397fb
- [72] Axel Neumann. 2015. Prof: ignore profiling probe if out of range (likely due to critical system time drift). https://github.com/-axn/bmx6/commit/ecfb56ca08af8dbd8004b62b4c8b1966023c9a57
- [73] Axel Neumann. 2015. Let --tunnels show non-existing gwName as ---. https://github.com/axn/bmx6/commit/a5e7a17c6c578d30a
- [74] Axel Neumann. 2015. Link: debug --link mac. https://github.com/axn/bmx6/commit/f26814fdac1c66fcd88903452af902e6be52fe
- [75] Axel Neumann. 2015. Link: debug --link mac. https://github.com/axn/bmx6/commit/4f17779c11bc19dd3acb9265010d74ff1baf7
- [76] Axel Neumann. 2015. Crypt: do NOT syslog private keys!. https://github.com/axn/bmx6/commit/b2664572a1ed6b890476d1e172

- [77] Axel Neumann. 2015. Bmx desc: less verbose description-update syslogs. https://github.com/axn/bmx6/commit/-85a8f10ba865afa66816f032a2d469fd681dbb0b
- [78] Axel Neumann. 2015. Table: resync_routes(). https://github.com/axn/bmx6/commit/0b115dd0826193e5010c48deaed0acfb9d35b
- [79] Axel Neumann. 2015. Bmx desc sec ip: fix assertion error codes. https://github.com/axn/bmx6/commit/5f8fc62f650a305035a7ca
- [80] Axel Neumann. 2015. Prof: align --cpu output. https://github.com/axn/bmx6/commit/209ce12be0f2c608e731ce8822433d80de8f
- [81] Axel Neumann. 2015. Hna: cpu-profile eval_tun_bit_tree(). https://github.com/axn/bmx6/commit/ace691a520d2c1382b6710a87
- [82] Axel Neumann. 2015. Redist table: cpu-profile redistribution functions. https://github.com/axn/bmx6/commit/473738ed85d6b20
- [83] Axel Neumann. 2015. Node: fix neigh_create() crash due to disabled packet signatures. https://github.com/axn/bmx6/-commit/15ca5d54ba203bbd27a69c59bafb098ac294d5fb
- [84] Axel Neumann. 2015. Link: debug descKey and pktKey in --links. https://github.com/axn/bmx6/commit/0ed1d1be34223800846b
- [85] Axel Neumann. 2015. All: rename pktKey→linkKey and descKey→nodeKey and related stuff. https://github.com/axn/-bmx6/commit/200465889061dd134ceb3bb3fcf864f49748e175
- [86] Axel Neumann. 2015. Bmx content hna prof sec: align status outputs. https://github.com/axn/bmx6/commit/86f53178d5a7ac8425
- [87] Axel Neumann. 2015. Table let resync_routes() call filter_temporary_route_changes(NOW). https://github.com/axn/-bmx6/commit/b233d81659b186e8988e19a22166de5742b5da2f
- [88] Axel Neumann. 2015. Key: debugging destroy_orig_node() reasons... https://github.com/axn/bmx6/commit/b62e635c4ef5839f5
- [89] Axel Neumann. 2015. Bmx: CRITICAL_PURGE_TIME_DRIFT 20→60. https://github.com/axn/bmx6/commit/93bd64e9ec96b
- [90] Axel Neumann. 2015. Schedule: move keyNode_fixTimeouts() to after packet reception. https://github.com/axn/bmx6/-commit/ccffed7babaa587d03371d7f93445ed3e2a0d24a
- [91] Axel Neumann. 2015. Common.mk -DAVL_5XLINKED. https://github.com/axn/bmx6/commit/7e8134bf61fcc96812cd2972f19
- [92] Axel Neumann. 2015. Ip hna: speeding up iproute(del). https://github.com/axn/bmx6/commit/f9fb36eb0419dc94c5dee8ff6ec63b
- [93] Axel Neumann. 2015. Redist: cpu-profile update_tunXin6_net_adv_list(). https://github.com/axn/bmx6/commit/d7591be3a5fc5
- [94] Axel Neumann. 2015. Speedup redist_table_routes() → update_tunXin6_net_adv_list(). https://github.com/axn/bmx6/-commit/258b962408a75a399437606c03924425195751c4
- [95] Axel Neumann. 2015. Fix curr_rx_packet→i.verifiedLink=NULL during neigh_destroy(). https://github.com/axn/bmx6/-commit/b485293101dae8d339fef6cbed1aba698f376bf9
- [96] Axel Neumann. 2015. Node key: debugging destroy_orig_node() reasons... https://github.com/axn/bmx6/commit/-f8497cd282d9bb11aa9d70ee0bcce32f303edbb1
- [97] Axel Neumann. 2015. Control: debug_output() count all debug messages. https://github.com/axn/bmx6/commit/-69e345952916bd7b420fa5d4863b3abc33f1049f
- [98] Axel Neumann. 2015. Node: refNode_destroy() do NOT del own credits. https://github.com/axn/bmx6/commit/-22b67f8f1be3f4d83bc733aefbae5ad3d2b6a4a9
- [99] Axel Neumann. 2015. Msg: keyNodes_block_and_sync() during rx_packet(). https://github.com/axn/bmx6/commit/-e121db468c9aeac493907937d2b92915a660d382
- [100] Axel Neumann. 2015. Metrics: use link_purge_to instead of fixed RP/TP_ADV_DELAY_RANGE=20000 within timeaware_rx/tx_probe(). https://github.com/axn/bmx6/commit/1cc7298bb04de590fe8bd48d9ff5aa0a90629137
- [101] Axel Neumann. 2015. Ip table: --netlinkBuffSize=(4*266240). https://github.com/axn/bmx6/commit/96db293122f95862a607b
- [102] Axel Neumann. 2015. Sec: fix opt_linkSigning() to not crash when returning to defaults. https://github.com/axn/bmx6/-commit/4b21205c6873190b8e09e1dc3a4bd343b90ae174
- [103] Axel Neumann. 2015. table: allow redistTable from any table. https://github.com/axn/bmx6/commit/6b7f8690d9c679f9e14ebfe

- [104] Libre-Mesh developers. 2015. Libre-Mesh Source Code. https://github.com/libre-mesh/
- [105] Gioacchino Mazzurco. 2015. lime-proto-bmx6: Simplified route exchange as BMX7 support filtering by proto. https://gitlab.con
libre-mesh/lime-packages/commit/1c24a88011ef32c1c7ba703f2d44970c48291374
- [106] Gioacchino Mazzurco. 2015. lime-proto-anygw: implement bgp_conf(. . .). https://gitlab.com/libre-mesh/lime-packages/-
commit/9f7b4ef748ab7dbfb2c1498ee885cdd8247710ff
- [107] Gioacchino Mazzurco. 2015. Reduce bmx6 redistribution performance impact, Move bmx6 route sharing code in the right place. https://gitlab.com/libre-mesh/lime-packages/commit/01fd9d20b65562d106f08648cd348fdb2283cf88
- [108] Gioacchino Mazzurco. 2015. Distribute bgp peering configuration accross lime-proto-*. https://gitlab.com/libre-mesh/-
lime-packages/commit/df5f6b73bd18fa6ee692fc610e14706d3aac488f
- [109] Gioacchino Mazzurco. 2015. lime-proto-bgp: Import BMX6 route prepending long path. https://gitlab.com/libre-mesh/-
lime-packages/commit/77c3d9f0b7299c5d6bebacc799238ef5166f7594
- [110] Gioacchino Mazzurco. 2015. lime-proto-bmx6: enable flag all and filter bird route with sys number. https://gitlab.com/-
libre-mesh/lime-packages/commit/dc5162554dd18350b8038448d5f0fbb322ba97e6
- [111] Gioacchino Mazzurco. 2015. lime-proto-bmx6: load automatically table plugin. https://gitlab.com/libre-mesh/lime-
packages/commit/b210adb63e3bd55da304143c33102bc6bdc1c7a6
- [112] Gioacchino Mazzurco. 2015. Depends on bmx7 (experimental) instead of bmx6. https://gitlab.com/libre-mesh/lime-
packages/commit/ddefa264428c2e6a60f1666c7646c4fd00359641
- [113] Gioacchino Mazzurco. 2015. lime-system: radio name shouldn't have dot in name so it is not necessary additional option, TEST ME BEFORE MERGE. https://gitlab.com/libre-mesh/lime-packages/commit/87d6c436726f620ea4c936da534dba8669052540
- [114] Gioacchino Mazzurco. 2015. lime-proto-{bgp,bmx}: added route sharing from bird to bmx6. https://gitlab.com/libre-
mesh/lime-packages/commit/3d94505c04c0adb650df01a18323f0d440f0dbf8
- [115] Gioacchino Mazzurco. 2015. lime-proto-bgp: read configuration instead of hardcoded values. https://gitlab.com/libre-
mesh/lime-packages/commit/7a3ae47b62f23e1b8beac516617f2a3461503324
- [116] Gioacchino Mazzurco. 2015. lime-proto-bgp: added automatic lan subnet announcement. https://gitlab.com/libre-mesh/-
lime-packages/commit/a6e27ced5dc8b94ae0a7306c088ee64a37be551d
- [117] Gioacchino Mazzurco. 2015. lime-proto-bgp: add protocol device to bird configuration so it listen for connections. https://gitlab.com/libre-mesh/lime-packages/commit/cecfe5a39e46f34aadd65ba798be42ba3c9ea7bf
- [118] Gioacchino Mazzurco. 2015. Use named table field instead of positional for bgp_peer template. https://gitlab.com/libre-
mesh/lime-packages/commit/f382c72b8b54051b7391682d4c019eeb5b651bfe
- [119] Gioacchino Mazzurco. 2015. utils.expandVars now support literal table index as variable names. https://gitlab.com/-
libre-mesh/lime-packages/commit/55b06dd09b55ba5287443ca21b61523d78dfb4f3
- [120] Gioacchino Mazzurco. 2015. add lime-proto-bgp stub. https://gitlab.com/libre-mesh/lime-packages/commit/ba6d1e087357c305
- [121] Gioacchino Mazzurco. 2015. added utils.expandVars for easier custom config file filling. https://gitlab.com/libre-mesh/-
lime-packages/commit/50b0296bbef50d94282c28e0d0d89b213241f15a
- [122] BMX6 developers. 2015. BMX6 Source Code. https://github.com/axn/bmx6/
- [123] OpenWrt source code. 2015. OpenWrt routing feed. https://github.com/openwrt-routing/
- [124] Wifi Planet. 2015. Understanding Ad Hoc Mode. http://www.wi-fiplanet.com/tutorials/article.php/1451421
- [125] Wikipedia. 2015. Almquist shell. https://en.wikipedia.org/wiki/Almquist_shell
- [126] Wikipedia. 2015. Anycast. https://en.wikipedia.org/wiki/Anycast
- [127] Wikipedia. 2015. Wireless access point. https://en.wikipedia.org/wiki/Wireless_access_point
- [128] Wikipedia. 2015. Application programming interface. https://en.wikipedia.org/wiki/Application_programming_interface

- [129] Wikipedia. 2015. Address Resolution Protocol. https://en.wikipedia.org/wiki/Address_Resolution_Protocol
- [130] Wikipedia. 2015. Autonomous system (Internet). https://en.wikipedia.org/wiki/Autonomous_system_%28Internet%29
- [131] Wikipedia. 2015. Babel (protocol). https://en.wikipedia.org/wiki/Babel_%28protocol%29
- [132] The Internet Society. 2006. A Border Gateway Protocol 4 (BGP-4). https://tools.ietf.org/html/rfc4271
- [133] Wikipedia. 2015. Broadcasting (networking). https://en.wikipedia.org/wiki/Broadcasting_%28networking%29
- [134] Wikipedia. 2015. Basic service set identification (BSSID). https://en.wikipedia.org/wiki/Service_set_%28802.11_network%29#
- [135] Wikipedia. 2015. C (programming language). https://en.wikipedia.org/wiki/C_%28programming_language%29
- [136] Wikipedia. 2015. Classless Inter-Domain Routing. https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing
- [137] Drupal.org. 2015 Drupal - Open Source CMS. https://www.drupal.org/
- [138] The Internet Society. 2005. Basic Transition Mechanisms for IPv6 Hosts and Routers. https://tools.ietf.org/html/rfc4213
- [139] Wikipedia. 2015. Ethernet. https://en.wikipedia.org/wiki/Ethernet
- [140] Wikipedia. 2015. Firmware. https://en.wikipedia.org/wiki/Firmware
- [141] Wikipedia. 2015. Internet Protocol. https://en.wikipedia.org/wiki/Internet_Protocol
- [142] Wikipedia. 2015. IPv4. https://en.wikipedia.org/wiki/IPv4
- [143] Wikipedia. 2015. IPv6. https://en.wikipedia.org/wiki/IPv6
- [144] Wikipedia. 2015. Local area network. https://en.wikipedia.org/wiki/Local_area_network
- [145] Wikipedia. 2015. Last mile. https://en.wikipedia.org/wiki/Last_mile
- [146] Libre-Mesh community. 2015. Libre-Mesh. http://libre-mesh.org/
- [147] Lua.org. 2015. About Lua. http://www.lua.org/about.html
- [148] Wikipedia. 2015. Mesh networking. https://en.wikipedia.org/wiki/Mesh_networking
- [149] Wikipedia. 2015. Netlink. https://en.wikipedia.org/wiki/Netlink
- [150] Wikipedia. 2015. Network interface controller. https://en.wikipedia.org/wiki/Network_interface_controller
- [151] OLSR community. 2015. OLSR Web Site. http://www.olsr.org
- [152] OpenWrt community. 2015. OpenWrt. https://openwrt.org/
- [153] Wikipedia. 2015. Open Shortest Path First. https://en.wikipedia.org/wiki/Open_Shortest_Path_First
- [154] Wikipedia. 2015. Peering. https://en.wikipedia.org/wiki/Peering
- [155] Wikipedia. 2015. Quagga (software). https://en.wikipedia.org/wiki/Quagga_%28software%29
- [156] OpenWrt Wiki. 2015. Wireless configuration. http://wiki.openwrt.org/doc/uci/wireless
- [157] Wikipedia. 2015. Router (computing). https://en.wikipedia.org/wiki/Router_%28computing%29
- [158] BIRD documentation. 2015. BIRD Internet Routing Daemon. http://bird.network.cz/?get_doc&f=bird-1.html
- [159] Wikipedia. 2015. Request for Comments. https://en.wikipedia.org/wiki/Request_for_Comments
- [160] Wikipedia. 2015. Network socket. https://en.wikipedia.org/wiki/Network_socket
- [161] Wikipedia. 2015. Service set (802.11 network). https://en.wikipedia.org/wiki/Service_set_%28802.11_network%29
- [162] Wikipedia. 2015. Station (networking). https://en.wikipedia.org/wiki/Station_%28networking%29
- [163] Wikipedia. 2015. Subnetwork. https://en.wikipedia.org/wiki/Subnetwork

- [164] Wikipedia. 2015. Transmission Control Protocol. https://en.wikipedia.org/wiki/Transmission_Control_Protocol
- [165] Wikipedia. 2015. Time to live. https://en.wikipedia.org/wiki/Time_to_live
- [166] Wikipedia. 2015. Unified Modeling Language. https://en.wikipedia.org/wiki/Unified_Modeling_Language
- [167] Wikipedia. 2015. Unicast. https://en.wikipedia.org/wiki/Unicast
- [168] Wikipedia. 2015. USB. https://en.wikipedia.org/wiki/USB
- [169] Wikipedia. 2015. Vendor lock-in. https://en.wikipedia.org/wiki/Vendor_lock-in
- [170] Wikipedia. 2015. Virtual LAN. https://en.wikipedia.org/wiki/Virtual_LAN
- [171] Wikipedia. 2015. Wide area network. https://en.wikipedia.org/wiki/Wide_area_network
- [172] Wikipedia. 2015. Wireless distribution system. https://en.wikipedia.org/wiki/Wireless_distribution_system
- [173] Wikipedia. 2015. Wi-Fi. https://en.wikipedia.org/wiki/Wi-Fi