UPC CN-A testbed mesh network deployment, monitoring and validation

Barcelona, June 2014

Major: Computer Science
Minor: Information Technologies

Author:
Eloi Carbó Solé

Director:
Leandro Navarro Moldes
Department of Computer Architecture

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya - BarcelonaTech
2014

Abstract

Testbeds are used in computer networking research to perform research experiments. Virtualization is sometimes used to simplify the experiment setup and reduce costs needed for large scale experimental testbeds. As a result, the experimental results include artefacts from these simplifications due to emulated components that can compromise its validity.

This project introduces WiBeb, a solution developed in order to deploy testbeds as close as possible to a real environment and, at the same time, share the features of virtualised experimental environments. Since this project has been developed as a part of the CONFINE project, all the software, hardware and documentation has been created with the goal of standardising the platform and make it open and affordable for researchers and to the community of interest in general.

Catalan abstract

En el món de la recerca de les xarxes de computadors, els testbeds són les plataformes on s'executen els experiments. Aquestes plataformes exigeixen unes infraestructures i software necessaris per regir-les molt costoses i estàtiques, a la vegada que generalment privatius. Per altre banda, per tal d'experimentar evitant aquests costos la tendència que s'està seguint és la virtualització, la qual simplifica i redueix els costos, però alhora requereix la concienciació de que al cap i a la fi els resultats seran fruit d'una simulació.

Aquest projecte presenta WiBed, una solució desenvolupada per obtenir un testbed el més proper possible als entorns reals, a la vegada que aportant les característiques comunes dels entorns virutalitzats. Donat que aquest projecte ha estat desenvolupat com a part del projecte europeu CONFINE, tot el software, hardware i documentació ha estat generat concienciadament per tal d'estandaritzar la plataforma i fer-la lliure i assequible per als investigadors i la comunitat en general.

Acknowledgements

First of all, I want to thank Leandro Navarro for your supervision in this bachelor's degree thesis and for inspiring me to better myself personally and professionally. Roger Baig as a mentor and promoter of WiBed and the community network philosophy. To Pau Escrich for your unconditional support, mentorship and your inspiring strong convictions in community networks. To Emmanouil Dimogerontakis, for your support and contribution during all the project and being there always I needed help. To the rest of LabE104 colleagues Roger Pueyo, Agustí Moll, Jorge, Ferran and Dani, because a healthy, and geek, work environment and companionship like what you have ends into achieving greater and passionate projects. To Axel Neumann for your support during the project, in our short stay in Berlin and for the hard work performing the Battlemesh experiment and posterior data gathering. To the Battlemesh community, for such amount of knowledge and wishes to make a better world, and for showing that change begins in our own choices and actions.

And finally to Sílvia, for being there always supporting me and encouraging me to grow.

Index

List of Figures

List of Tables

Glossary

- **Testbed**: Large-scale projects experimentation platforms.
- **Internet Service Provider**: Companies who bring Internet connectivity and other services to the final users.
- **Community Network**: Networks cooperatively built, owned, managed and used by its own users.
- **Script**: Programs written in particular languages widely used to perform tasks that will be interpreted instead of compiled.
- **Dynamic Routing Protocols**: Routing protocols that works equal to static ones, but with a more robust and adaptable capabilities for mobile or unstable environments and networks. These protocols are widely used in Community Networks.
- **Node**: Synonym of router device.
- **REST**: Representational state transfer. The system used by the controller to communicate and manage the testbed. It uses HTTP operations to handle requests and responses.
- **JSON**: JavaScript Object Notation. Is an object representational notation allowing to transmit data in a human-readable manner.
- **u-boot**: Non-writable partition used for initializing the node. Is equivalent to the BIOS system.
- **LZMA**: Lempel–Ziv–Markov chain algorithm. Loss-less data compression algorithm used to zip files.
- **Trunk**: Applied in branches of development, means a branch being tested and under revision of the developers.
- **HTML5, CSS and JavaScript**: Web development languages.
- **API**: Application Programming Interface. Toolkit used to intercommunicate two, or more, different software systems.

- **Firmware**: Embedded systems OS installed in the ROM (flash) memory of these devices.
- **To flash**: The process of overwrite the firmware in the ROM memory of embedded systems.
- **Repository**: Refereed to source code, a repository is a toolkit used to manage and centralize code's storage and historical revision information.
- **AP**: Access Point. Refereed to a wireless connection point with the purpose of bringing network access and services.
- **Round Trip Time**: The time needed to achieve a path between a source and destination (send the packet and receive the acknowledgement).

Chapter 1

Introduction

Experimenting with *testbeds* is a common practice when researching and working with networks. Testbeds are the stage that lays between simulation and real production environments when performing experiments or tests, using both simulation and production scenario benefits. Benefits of simulation are fault tolerance, ease of deployment, testing and data collection, reproduction, environment configuration, etc. And those of production environment are real hardware, on the field deployments, physical and direct access and manipulation of the hardware, etc.

The Wireless testBed (WiBed) project aims to improve testbed's use by giving the community a platform to perform quick and cost-efficient network deployments, using commodity (standard 802.11) routers and also giving to the researchers the possibility to access directly to low-level layers (transport and physical) to perform, not only network and transport protocol experiments, but also having complete access to the resources when p an experiment. Furthermore, WiBed goals are to automate the configuration and operation management process of the mesh testbed and to be able to work with most of the unstable and dynamic topologies without human interaction. This collides with the known perception of experimentation in networking: to reproduce relevant characteristics or behaviours in existing testbeds, being an static or hard to change topologies (unless in virtualization), because WiBed topologies and testbeds can be low-cost, temporary, even mobile, and heterogeneous.

1.1 Structure of the document

This dissertation is organized as follows. Chapter 1 introduces the WiBed platform background and main points regarding this project's development. Chapter 2 the general idea of the platform, its components and operation are described. Chapter 3 describes more specifically all the procedures and particularities as well as the characteristics of all platform parts. Chapter 4

brings a exhaustive description of the all the UPC CN-A deployment process. Chapter 5 presents all the procedures made to perform experiments that will validate WiBed as a trustworthy platform. Finally, chapter 6 concludes with a summarize of the platform goals achieved and future work.

1.2 Brief description of the problem

In a world ruled by a few Internet Service Providers (ISP) that own most of the major worldwide networks, study, access or modify these networks is a non-affordable work. This contrasts with the philosophy of having a free, open and neutral network which is the network promoted mainly by Community networks. These networks aim to be a real alternative to those restrictive ones and provide one to be used and owned by its own users. Furthermore, all these communities have their own rules to maintain and judge a neutral use of the network owing not to allow bad minded activities or abuse of service over the network. The commons of community networks might also forbid experimentation in the network so as not to affect the network normal behaviour and its users. However, community networks fits as the perfect scenario for experimentation due to its capability to adapt from strong to unstable or poorly connected networks (using dynamic routing protocols) but, again, experimentation must be done in isolated and controlled networks or, otherwise, in a different way such as virtualization or research testbeds.

This bachelor's degree thesis main goals are to improve the WiBed platform in order to allow research and experimentation of these community networks avoiding virtualization tools but also keeping its main benefits, such as fault tolerance, ease of deployment, testing and data collection, in an as close as possible to production environments using real hardware in real deployments. Moreover, to deploy a testbed mesh network in the UPC university in Campus Nord (Barcelona) which will be used to bring resources for experimentation to worldwide researchers.

1.3 Scope of the project

In order to satisfy the requirements and considerations described in "*WiBed Off-the-shelf Wireless Networks Research Testbed*"[1] thesis and the minimal goals to achieve agreed with the CONFINE project, this project ambition is to develop the necessary tools and packages to platform needs. Furthermore, due to the scale of this project, it counts with the previous hard work of some members of the CONFINE project team[1][2][3]. Initial parts working thanks to

[1] Nico Echániz: server side.
[2] Alexandre Fonseca: API development and server side.
[3] Guido Iribarren: node side.

team members in the beginning of the project were:

- Experimentation server's part in an advanced implementation state.
- 5 nodes and one gateway deployed in the lowest floors of the A5 building.
- OpenWRT firmware with packages to configure the platform and behaviour of the router.
- Wiki page of the project.
- A centralised software versions controller of the CONFINE project (Redmine server).

Using this information as the starting point of this project, the goals wanted to be scored are to deploy the theoretical network *UPC CN-A*, to improve not only the behaviour of the tools but also developing new ones to supply needs that may appear during the project's development and, last but not least, to bring support to researchers who use the platform as testers.

1.3.1 Methodology and communication

The WiBed project has followed the CONFINE work methodology and communication. It consists on a weekly work schedule based on goals and with a fast meeting on Wednesday. With that planning, the meetings are used to expose how the project is going, which goals have been achieved, which not, and also, to have a feedback and support of the rest of the team members. Moreover, these meetings allow other project members to know colleagues' work and merge ideas and tools willing to help to improve the overall CONFINE projects.

1.4 Background information

1.4.1 OpenWRT

Embedded systems lack of the high specifications found in desktops, laptops or even mobile phones, as well as the work devised for them need not much power. In the commodity routers topic, there are some operative systems that excels due to its community and support for a great amount of router architectures (at least 83 different manufacturers). Minimum requirements of OpenWRT to work with are having at least 4MB of flash to install the firmware and 16MB of RAM, bringing the possibility to install a powerful OS to low specs. routers and to have support from its community.

1.4.2 Testbed

As the WiBed paper[2] attests:
"*Testbeds are a stage between the simulation and the production stages. To this end they must be as close as possible to production environments (i.e. real hardware, on the field deployments) while also keeping the traits of experimentation facilities (i.e. fault tolerance, ease of deployment, testing and data collection)*".

1.4.3 The CONFINE Project

Community Networks Testbed for the Future Internet (CONFINE)[4] is an European Large-Scale Integrating Project (IP) scheduled from October 2011 to September 2015. It is included in the Future Internet Research and Experimentation Initiative (FIRE) of the European Community Framework Programme (FP7) and with a budget of 4.942.000€. This project is mainly executed in the UPC in Barcelona and coordinated by professor Leandro Navarro Moldes.

The main goal of the CONFINE project is to research new Internet models and find out if they are sustainable, not only economically and technically, but also socially. Its target research is focused on community networks.

1.4.4 Commodity routers

The WiBed platform aims to bring testbeds to general public as an standardised platform with standardised hardware and, for that reason, *off-the-shelf* routers are used. These routers must fit some characteristics owing to be supported by the WiBed platform:

- They must be supported by the OpenWRT community, that means that these routers will have the support of hackers (OpenWRT community), which are maintaining the firmware (and future updates) compatible with them.
- They must have most of the basic features but with a non-expensive cost (less than 100€).
- They must have sufficient hardware specifications to be able to work fluently.

Most of the routers that fit this requirements are from manufacturers that prioritise innovation and low prices against long-term support and production. That means that these manufacturers have a lot of models with similar or equal specifications that just have few improvements. In the end,

[4]CONFINE Project: `http://wiki.confine-project.eu/`

those models which have hackers' support, are the most common to have long-term support from OpenWRT community due to its popularity. Actually, current main supported by WiBed team router is TP-Link WDR-4300, a widely used by OpenWRT community router which current hardware revision is V1.7 (released in January 2014). WiBed also counts with some TP-Link WDR-4900 routers, which have better features but is still being considered on account of lower community support.

1.4.5 Wireless Battle of the Mesh

The Wireless Battle of the Mesh (WBM) is an annual event focused on the community networks, its routing protocols and also other related with the community networking stuff. WBM is a social event that encourages from enthusiasts to experts in the community network topic, to meet for a week in one different spot of the community every year. The main goal of the WBM is to develop and improve the routing protocols and tools that will benefit the community networks. Furthermore, it is an event to present ideas and projects owing not only to show it to the community, but also to obtain feedback and improvements. In addition, there is a healthy competitive tournament between the different routing protocols that are working and being developed by the community for the community. These routing protocols are: BABEL, BATMAN-ADV, BMX, OLSR and 802.11s. In section 5.1 this event and its relation with WiBed will be further described.

This year 2014, the 7th version of the Battlemesh has been performed in Leipzig (Germany)[5]. It has been chosen due to its high dense distribution of social communities and its autonomous and collaborative working ecosystem. One of these collaborators is Freifunk, who is sharing connectivity with these communities. During the social events focussed to communities and its interaction, there were two trips around the city, visiting the different communities and how they are sharing resources in a non-profit but proactive way.

Another important point of the WBMv7 is that it is the first Battlemesh using WiBed as the platform to develop and perform the experiments. As will be further explained in the next section 5.1.1, WiBed has been a successful platform to perform the experiments of the event and the researchers have agreed to use the platform in future editions. Finally, following the work that Roger Baig, with the assistance of Pau Escrich, did in the theoretical dissertation of WiBed[1], I included in the Appendix C.2 a table with updated information of Battlemesh main editions.

[5]The WBMv7 was performed in Leipzig, Germany. More information about this edition can be seen in Appendix C.2

1.4.6 UCI configuration

The Unified Configuration Interfaces is a centralized powerful configuration tool-kit. Its main point is to make OpenWRT tools as user-friendly as possible. The UCI configuration uses the folder */etc/config* to keep all configurations from the different programs in just one site. As an instance, we can see in the WiBed configuration script:

```
config wibed 'general'
    option coordx '41.38953'
    option coordy '2.11306'
    option coordz '2'
    option api\_url 'http://wibed.confine-project.eu/'
    option node\_id 'wibed-3e9dae'
    option status '1'
```

This code shows how UCI separates the configuration in sections, and also, that each section is compound by some field-value options, which will be checked and used to decide the behaviour of UCI scripts for the corresponding tool.

1.4.7 Dynamic Routing Protocols

In this section, WiBed most relevant routing protocols will be described, as the reader will find lots of references mainly in the validation chapter 5 and in the Appendix C.

BMX6

BATMAN-EXPERIMENTAL 6[3] is a distance-vector routing protocol forked from original BATMAN routing protocol and focused in IPv6 mesh networks. Its main efforts are first, in improving IPv6 capabilities, and second, in automating and improving node-related configuration and announcement. Its routing methodology consists on table-driven, so each node decides the best next hope for the package transmission.

BATMAN-ADV

Better Approach To Mobile Adhoc Networking protocol[4] is a layer 2 routing protocol. It encapsulates not only the routing information, but also the data traffic in ethernet frames with a virtual network switch and, as it operates in 2nd layer, it uses the same collision domain. Its main point is the decentralization of the network knowledge as willing to have a network of *collective intelligence*.

OLSR

Optimized Link-State Routing Protocol[5] is a proactive link-state protocol that, instead of flooding the network with control packages, it distributes them through selected MultiPoint Relays (MPR) reducing the overhead of the process. While OLSRv1 was not much efficient in mesh networks due to the MPR procedure and the nature of the networks, OLSRv2 has shown an impressive improvement during the tests in the WBMv75.2.

BABEL

Babel[6] is a distance-vector routing protocol specialized in loop-avoiding techniques and designed to work well in a wide amount of scenarios: it works regardless if the network uses IPv4 or v6 and also works correctly in wired and mesh networks. It is based and is using ideas from different systems: DSDV[6], AODV[7] and also EIGPR[8]. Its main point is to use advanced techniques to avoid *absence of routing pathologies*.

1.5 State of the art

This section will present a brief description of current research testbeds and will serve as contrast of what WiBed pretends to be. While an extended description of these testbeds can be found in the project *Off-the-shelf Wireless Networks Research Testbed*, here we will only enumerate the most interesting points of each research testbeds and, finally, present what WiBed does and why.

1.5.1 Research testbeds

ORBIT Testbed

The Open-Access Research Testbed for Next-Generation Wireless Network[7] is an academic, industrial and governmental partners research testbed compound by 400 radio nodes (1GHz processor, 512MB RAM, 20GB HDD, two Wireless Radios a/b/g and two 100BaseT Ethernet ports). One interesting point of ORBIT is that counts with a sandbox system consisting in some smaller node grids used to perform experimental or unstable tests before moving them to the main grid. To sum up, this testbed is not only expensive due to its components and infrastructure needs, but also is too complex to maintain and replicate.

[6] Destination-Sequenced Distance Vector Routing: `http://www.cs.virginia.edu/~cl7v/cs851-papers/dsdv-sigcomm94.pdf`

[7] Adhoc On-Demand Distance Vector Routing `https://tools.ietf.org/html/rfc3561`

[8] Enhanced Interior Gateway Routing Protocol `http://www.ietf.org/staging/draft-savage-eigrp-00.txt`

NITOS testbed

Network Implementation Testbed using Open Source code[8] is an experimental outdoor testbed owned by CERTH in association with NITLab in Greece. This testbed main points are to use nodes based on open source software. Actually, NITOS uses Alix nodes with commercial Wifi cards and Linux open source drivers controlled using the OMF (cOntrol and Management Framework) tool.

QuRiNet

Quail Ridge Wireless Mesh Network[9] is a wide-area outdoor testbed deployed in the Quail Ridge national park in US. This testbed main point is to be deployed in a spot completely free of interferences from other networks, being a noise free environment. Routers used in the testbed are Soekris net4826 using a custom distribution with the 2.6.26 Kernel version and modified Wireless drivers.

DES-testbed

Distributed Embedded System testbed[10] is a research testbed owned by Freie Universität focused in "*real world wireless multi-hop networks (WMHN), the performance and applicability of abstract algorithms, as well as the common assumptions of simulations and analytical methods by testbed-based research*". The mesh network is supported by Alix2/3 embedded PC boards with at least three network a/b/g interfaces. The main point of this testbed is being hybrid due to the use of sensors nodes as well as the Alix nodes, bringing some interesting scenarios to experiment with. Finally, the testbed is available for educational purposes being used in some subjects of the university.

BOWL

Berlin Open Wireless Lab[11] is a testbed of the INET group at Deutsche Telekom Laboratories (TU-Berlin). The main point of this outdoors testbed is to bring access, not only to research in community wireless networking but also giving internet access to students of the Technishche Universität, this is how BOWL's research wants not only to contribute in researching, but also to contribute with the testbed's community as can be seen in their paper: "*Experiences with BOWL: Managing an Outdoor WiFi Network (or How to Keep Both Internet Users and Researchers Happy?)*"

1.5.2 Main differences between other research testbeds and WiBed

Having a brief description of the most important testbeds in the world will help to understand why WiBed is, possibly, the most important platform,

not only for networking research testbeds, but also for educational purposes testbeds:

- Quick and possible to be temporary deployments in target experimentation environments with the possibility to modify and improve the topology with instantly response.

  Main testbeds are static and pre-configured to the case of study and modifications may be hard to adopt due to testbed's nature and cost. For instance, we could see the ORBIT testbed as a good example of that complexity, with a hughe and powerfull infraestructure.

- Possibility of federation with other large-scale (global-scale) testbeds.

  WiBed platform is under the scope of CONFINE project, which means the possibility to federate with European large-scale testbeds. On the contrary, most of the testbeds are not able to federate with other testbeds as they are planned to be stand-alone systems.

- Framework and tools to enable external researchers to use the testbed resources.

  Again, being part of CONFINE project allows WiBed to put efforts in develop collaborative tools to give user-friendly tools to bring resources for interested researchers.

- Open and free software project, with low cost supported hardware and standardised owing to be an easy to replicate and customise platform.

  The project main efforts are to standardise all its procedures and to use cheap hardware owing to open the project to the community for educational or research purposes.

1.6 Temporal planning

Due to project's complexity, it has been developed mainly by two persons[9], one in the controller's side and, another, in the router's side, collaborating and contributing in each others work, but here only will **have into account** the tasks done in router/network's side. In addition, this project work started a year before with Roger Baig and CONFINE team's work. However, the temporal planning that will be used for this project lasts 6 months: planning starts on February 2014 and, although the team actually will continue working in the platform for at least two years from july 2014, planning finishes on June 2014. Look figure 1.1 and the description that follows in next sections[10].

[9] Part of the platform's work was already done before the beginning of this project.

[10] In the Appendix B a bigger Gantt image is shown.

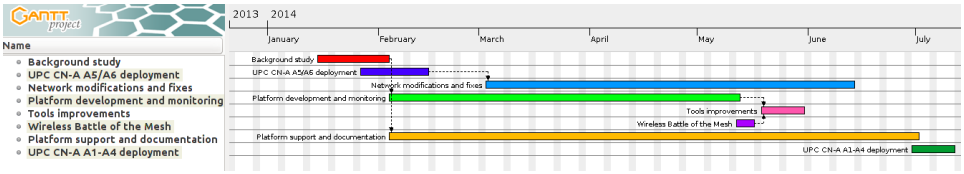


Figure 1.1: Tasks planning schedule.

1.6.1 Tasks

In this section the tasks done in this project's development will be presented as blocks, or goals, achieved during the implementation process.

Project's background

WiBed platform is a complex and ambitious project conceived from an ideology and methodology of work with a strong background and rich environment that needs to be seen thoroughly before you start trying to understand its main goals. Otherwise, is not possible to get the main points of using this type of technology and environment and someone may miss how important this platform is for the community networks and its users.

Deployment and network adjustments

As can be seen in *The deployment* section, the network UPC CN-A was defined in WiBed theoretical project and one of the main goals of this project has been to deploy it. Moreover, this project explains how the deployment has been modified due to structural and connectivity related problems and how has been studied and dealt to deploy the final network.

Platform management tools' development

Developing tools to improve, maintain and control the platform is one of the goals of this project. The initial state of the platform, apart from the network deployment state, was not stable enough to be used as a trustworthy platform for research purposes due to some issues and its early development status and, for that reason, one of the most important goals for WiBed was to receive improvements, not only in the robustness of its firmware and behaviour, but also adding new powerful tools to make WiBed a trustworthy platform for networking research.

Platform's validation

Once the deployment is finished and the tools are mature enough to make WiBed an stable platform to experiment with, the final goal of this project

is to validate WiBed as an reliable platform to research on community networks.

Document and support

Being under the scope of CONFINE project allowed WiBed to be developed using the same methodology that its projects so there are some important questions that this project achieved to work with:

- Being able to develop the project under a free and open source licence.
- Using source repositories open to worldwide developers and being able to have their feedback and support.
- Improving the CONFINE's wiki page with the information regarding the WiBed's development process.
- Giving at least two years of support for the UPC CN-A testbed mesh network in UPC and the platform.

1.6.2 Deviations and modifications in the planning

The project's original planning had into account some deviations and risks that actually had become true:

First and most important risk, was to be able to deploy the network in the given time (4 months) following Roger's project instructions. This fast deployment became impossible as the buildings' structure hardened the connectivity of the network and made to discuss and change the emplacement to the upper floors of the buildings, which actually was another emplacement already described. Besides the emplacement's change, another risk that occurred was that the maintenance department could not afford the cost of that new installation, so a budget to supply this cost was needed. Finally, as a consequence of being under the scope of CONFINE project, which is an European FP7 project directed by the Professor Leandro Navarro Moldes from the UPC, WiBed project's human and budget resources were enough to solve any unforeseen deviation or risk.

1.7 Budget

WiBed project's budget is compound by two main parts: the first one is the cost of hardware and human resources, described in the table X, and the second part, described in the table Y, is the budget agreed with the maintenance department to supply the costs of the final deployment described in *The testbed* section. Finally, the Z table shows the final cost of the WiBed platform.

- UPC Assistant: cost of the scholarship to develop the platform.
- Fungible: cost calculated assuming an overcost of 15% of the Hardware costs.

Description	**Unit**	**Quantity**	**Price (€/unit)**	**Total (€)**
UPC Assistant	20h/month	6	600.00	3600.00
Routers (TL-WDR4300)	unit	40	41.50	1660.00
Kingston USB 16GB	unit	40	6.74	270.00
USB Radio (TL-WN722N)	unit	30	7.82	235.00
Fungible		1	324.75	324.75
Total direct costs				5765.00
Total indirect costs				3805.00
Total				9570.00

Table 1.1: WiBed hardware and human resources costs

Description	**Unit**	**Quantity**	**Price (€/unit)**	**Total (€)**
Power cable	Meters	100	0.84	83.75
Aerial gum power base	Units	20	8.69	173.75
Installation work	Units	1	1031.20	1031.20
Total				1560.00

Table 1.2: Installation cost for the final UPC CN-A network

Subject	**Price (€)**
WiBed budget	9570.00
Installation budget	1560.00
Total project's cost	11130.00

Table 1.3: WiBed budget merging resources' cost and installations' cost

Chapter 2

Architecture

2.1 Design

2.1.1 Platform operation

The WiBed platform is compound by a controller that manages a group of nodes, which works as a testbed mesh network as can be seen in figure 2.1. These nodes have two wifi radios: one for the management (node-controller communication) and the other to be used for experimentation purposes. WiBed follows the client-server model, being each node the one who will contact the server by a pull methodology. In the controller side, there are two type of users: the first one manages the network and the second one will use network's resources. In addition, it is important to take into account that external interactions are not allowed so all the communications and requests must pass through the controller.

On WiBed development beginning users were not meant to access directly into the nodes so all the interactions must go through the controller. That philosophy changed thanks to the WBM contribution to the project and the result was that, owing to improve the feedback that researchers may need and due to some experiment's nature, in future improvements of the controller the researchers will be allowed to log into the nodes via VPN services.

Back again into the client-server model topic, WiBed nodes are the ones who will ask the controller for orders and commands, pulling periodically (UPC CN-A routers pulls every 15 seconds). The controller will answer each node request separately so this brings a more efficient method to manage the testbed than wasting controller resources using a push model.

2.1.2 Software and hardware

The hardware and software needed to deploy the full WiBed platform will be described in the next sections. Specific models and tools will be mentioned but, thanks to WiBed initial goal of make an open platform, main procedures

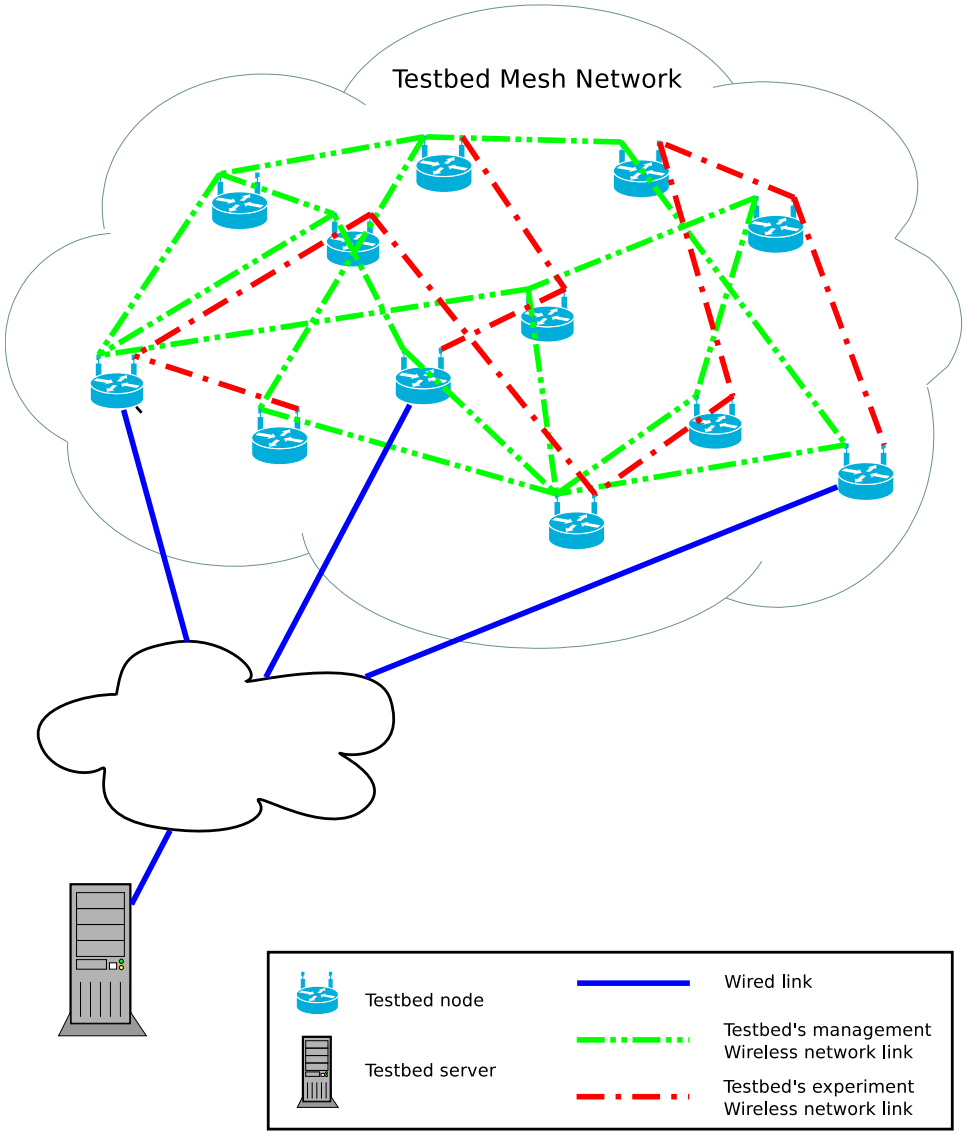


Figure 2.1: WiBed platform design

and hardware can be easily developed and switched, due to the efforts put in the standardisation of the platform. Moreover, initial specifications were described in the *Off-the-shelf Wireless Networks Research Testbed*, attached in the Appendix A.

2.1.3 Testbed mesh network collocation

WiBed's mesh network works very similar to a real community network. Unlike most of the existing testbeds, WiBed capabilities encourages the re-

searchers to deploy the network as a temporary solution and, even more, to deploy it with mobile devices. Using the mesh topology and, thanks to be using BATMAN-ADV as the management network's dynamic routing protocol, all the connections may fluctuate and change but the network will still work and adapt to each topology change. Is for that reason that WiBed is an excellent solution for hard environment or with low budget deployments, allowing to use different types of nodes and also being able to modify its emplacement.

2.2 The Overlay File System

The Overlay File System (OFS) is a file system focused on embedded needs (also able to be used in other solutions due to its inclusion in the Kernel of Linux in the version 3.11). The basic distribution of this file system can be seen in figure 2.2. OFS summarised operation is to protect the *core* OS files, the base directory, but also allowing the modification of a mirrored copy of them, the storage directory. In less abstract terms, this file system envelopes two other file systems and is able to merge them as if they were one. Moreover, as its name mentions, there is an over and a below file system, so the first one will be hidden by the second in those files or directories that match. The main idea is, when using this enveloping file system, both enveloped FS are merged willing that the files that users will see, are the ones in the upper one and, once finished, all the changes will be still in the modifiable directory, maintaining the base directory clean (in its default state). That transparent to the user process makes the user think that is modifying the FS directly but redirecting these changes to the modifiable directory. This improves not only the security of the files in the base directory, but also allows to have a clean state to go back when needed. Some examples of the kind of projects where OFS could fit in:

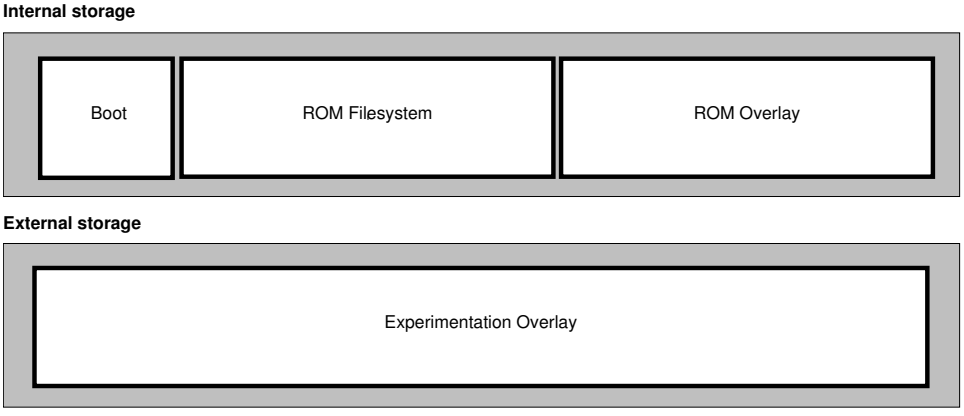


Figure 2.2: Overlay File System default state.

1. Low flash and RAM systems that need an extra storage to work correctly.
2. Systems that do not need to store the changes made during its operation.
3. Critical systems that need to protect the OS from external modifications.
4. As a recovery system to bring a 'Back to default state' method.

Looking these four examples of possible uses of OFS, we can see that all them fits in WiBed platform needs, keeping in mind not only the use of commodity routers (1st and 2nd), but also the experimentation's process needs (3rd and 4th).

2.3 Experiment operation

Performing experiments in the WiBed platform means to apply an overlay to the default base system (the Overlay File System), owing to bring packages, configuration files, scripts and, in summary, any necessary changes to perform the experiment. Moreover, each node can only be in one experiment at the same time but in the testbed can coexist (in parallel) as experiments as combinations of nodes.

Each experiment, and the resources involved, are isolated from other experiments and the unique way to access them is through the controller and the testbed's management network. And is thanks to that management network (not allowed to be manipulated by the researchers), that the communication with the controller is never lost and this allows the researcher to send commands and receive feedback from the nodes.

The experiment server-side works as a controller for the commands and monitoring the outputs of them. In addition, the controller also works as a centralised storage system, collecting the data that each node in the experiment must save in a special folder (*/save*). With it, researchers can find not only the outputs of each command sent to the experiment but also the results of the procedures done.

Chapter 3

Implementation

In this chapter we will explain from the router-side WiBed platform main functionalities as well as a brief description of the procedures used.

3.1 The WiBed node

3.1.1 Original testbed router requirements

WiBed minimal initial hardware and software requirements were widely defined in the theoretical beginning of WiBed *Off-the-shelf Wireless Networks Research Testbed* project[1], here there is just a summary of these requirements:

- Has to be reasonable cheap.
- Must have at least two independent NICs[1].
- Must have an Ethernet port.

Also, using OpenWRT as the platform firmware supposed some extra requirements:

- At least 4MB of flash (recommended 8MB).
- At least 16MB of RAM (recommended 32MB).

The software and package requirements to work properly are:

- Runs the open source operating system OpenWRT.
- Runs Dropbear as SSH client.

[1]Network Interface Controller

- Runs Busybox built in time client configured to use to server's NTP server.
- Runs cURL as HTTP client.
- The Ethernet interface and the management wireless interface are bridged.
- Runs Tinc VPN client for tunnelling the management network.
- Has the BATMAN-ADV kernel module activated and runs on the management bridge.
- Uses the server as the Internet gateway.
- Uses the server as DNS server.
- Has the block-mount kernel module enabled (to allow USB sticks to be mounted)
- Has the Ath9k kernel driver enabled (to allow Atheros Radio management and use).
- Uses Busybox cron utility to periodically check for updates in the server.

3.1.2 WiBed supported router

Following all these requirements, the WiBed team chose one node to fit them and be the project's main supported router, the TP-Link WDR4300[2]. This router specifications are:

- CPU: Atheros AR9344 560MHz (MIPS Instruction set)
- Flash: 8MB
- RAM: 128MB
- Network Ethernet ports: 4 1Gigabit Ethernet
- WAN ports: 1Gigabit Ethernet
- 2.4GHz Wireless radio: Integrated Atheros radio 802.11b/g/n
- 5GHz Wireless radio: Separated Atheros AR9580 chip 802.11a/n
- USB ports: 2 USB2.0
- Serial ports: Yes
- Power supply: 12V / 1.5A

[2] `http://wiki.openwrt.org/toh/tp-link/tl-wdr4300`

3.1.3 Extensions and modifications in the WiBed router requirements

While developing the WiBed platform and more specifically in the WBM event, we realised that some of the researchers asked us for an image of WiBed and support for devices with lower specifications than WiBed minimal requirements. After studying to add support for these devices, we could agree with a new set of requirements that could fit in WiBed testbeds as well as researchers that use this devices should take into account that its uses could differ from original router ones.

The modifications on the default requirements are:

- One Radio for experimentation
- One Ethernet port for network management (this restriction will only apply to those routers without 2 wireless radios)

These modifications widened the variety of capable routers and gave to WiBed the possibility of using devices from 30€ to make the research testbed platform even more accessible, cheaper and mobile-capable. Moreover, the second main goal to use this kind of devices is to use both radio and Ethernet cable as the management network, bringing to the administrator or researcher an easier way to access to the mesh network management and connect to nodes for administration or experimentation purposes.

The nodes that will be supported from the WiBed platform with these new requirements are TP-Link MR3020[3], MR3040[4] and WR703N[5], with few hardware differences (like the 5 hours battery of the MR3040):

- CPU: Atheros AR7240@400MHz
- Flash: 4MB
- RAM: 32MB
- 2.4GHz Wireless radio: Atheros AR9330 (MR3020) and Atheros AR9331 (MR3040 and WR703N)
- USB port: Yes, 1 USB2.0
- Ethernet port: Yes, 1x100Megabit port

[3] `http://wiki.openwrt.org/toh/tp-link/tl-mr3020`
[4] `http://wiki.openwrt.org/toh/tp-link/tl-mr3040`
[5] `http://wiki.openwrt.org/toh/tp-link/tl-wr703n`

3.2 The WiBed controller

As important as the mesh network and its routers capabilities, the server that will control the platform needs also to fit in some requirements to work correctly. In contrast on the router requirements, the controller has just few recommendations not to be a bottle neck to the mesh network (in those that are large-scale). In the theoretical WiBed project, the WiBed controller would be an Alix2d2 embedded system (as an usual node of CONFINE projects) but, the current controller is working on a Virtual Machine, in UPC-Pangea's department, with these specifications:

- CPU: 1GHz
- RAM: 2GB
- HDD: 2GB
- OS: Debian 7.1

3.2.1 Software requirements

The WiBed server is designed as a web application. In order to run the WiBed server platform in the controller, there is some software needed:

- Flask: lightweight modular framework for web development made in python.
- SQLite: lightweight relational database.
- SQLAlchemy: toolkit for database management and a Object-Relational mapper made in python.
- Tornado: web application server focused on serving asynchronously and in a non-blocking manner the network requests in large-scale environments.

3.2.2 Main functions of the controller

Controller's main functions consist on the management of the nodes and information gathering of its status and experiment's process, but also the endpoint for users to reach the testbed. Here we are going to focus in the features created for the users. Currently WiBed has into consideration two types of users: administrators and researchers, both have most of the privileges to control the testbed and check its status, although researchers have experiment and result gathering functionalities and administrators have routers' firmware management and debug functionalities.

Functionalities

- Admin tab:

  This page allow users to send commands to the routers in the testbed by selecting them manually. More specifically, the server will create as many entries in the database as routers will receive this command. These commands added to be executed will be showed in a list in the bottom of the page showing, not only the information regarding the command and how many routers have executed it already, but also being a link to a command page with the execution information of each router.

- Nodes tab:

  This page has a list of all the available nodes in the different testbeds working in the controller showing its identification name, its testbed name, its status and the time since the last pull to the server[6]. Each node in the list is also a link to a information page of the router.

- Node's page:

  Each node in the test has an information page with information regarding the node, such as the model, firmware version, last success pull to the server, its status, if its performing an experiment, a list of previous experiments where it was involved and a description[7][8]. Furthermore, there is an OpenStreetMap section available showing node's current location and being able to be modified if necessary (by clicking its position in the map or with its coordinates).

- Repo tab:

  This page has a list of firmwares available to be downloaded. First, they are split by the development branch [master (stable version) and last_trunk (testing version)]. Once in the development branch, there is another list of router architectures available (WiBed's current supported architecture is *ar71xx*, but **mpc85xx** architecture will be supported soon). Finally, each architecture folder has its own different models' firmware and also packages already compiled for them.

- Errors tab:

  This page has a list of router's ID which has gone to the Error state during its operation. The available options are to watch the log files online (a list of log files) or to download them as an *id.tar.gz* file.

[6] Administrator users are able to show/hide nodes to the research users.

[7] Administrator users are able to hide/show the node to the researchers and also to delete it from the server.

[8] Administrators are able to modify the description of the nodes

Researcher's functionalities

- Experiments tab:

  This page shows the information regarding the experiments being performed in the testbed and the ones finished, as well as it brings to the researcher the possibility to create new experiments.

- Add Experiment page:

  This page allows the user to name the experiment, select or add an overlay and the nodes involved in the experiment.

- Experiment information page:

  This page allows the user to manage the experiment process. The researcher can start or stop the experiment, check experiment's general information, check the nodes and its status and finally add commands that will be sent to all the involved nodes.

- Results tab:

  This page shows a list of finished experiments and its time stamp. Each experiment contains an expanded list with the results of each node involved in the experiment and, as in the error's tab, the results can be downloaded or checked online.

Administrator's functionalities

- Firmwares tab:

  This page allows the administrator to add new firmwares to install them in the nodes automatically. Also it shows information about existing firmwares in the server and in which routers are installed.

- dbDebug tab:

  This page shows the information contained in the database of the server in a sorted manner. This information is informative only and it is not available to be modified.

3.3 Communication between controller and the testbed

In order to communicate the controller with the network it was required not to be the controller who asks to each node its commands, as in a push request system, but to reverse it and use a pull request system, where the nodes ask for commands to execute. Thanks to this pull procedure and the scalable Flask RESTful features, the controller can serve the petition with few resource requirements, as can be seen in *The WiBed controller* section.

3.3.1 The management network

The management network is an 802.11 ad-hoc wifi network used to access remotely to the testbed and send them commands and monitoring the nodes. In addition, this network is provided by one of the node's radios using the Batman-adv routing protocol (see section 1.4.7). The resulting meshed network uses the same collision domain and is automatically configured using IPv6 addresses. Is thanks to that auto-configuration that the management and administration of the testbed is easily possible, even when performing an experiment.

Although this management network does not require any wired link, the network will have at least one wired node acting as the gateway of the testbed. Gateways are the nodes in charge of bringing connectivity through the Internet and, mainly, to connect the testbed with the controller.

One of the hardest problems to solve is the disconnection or lack of connectivity in the testbed due to a bad configuration or the nature of the experiment. Hence, security actuations are needed to avoid isolation or node's misconfiguration. There are three main scenarios where the recovery system should act:

1. The node is performing an experiment and the configuration of the management network has been changed.

   In this case, the node will not be able to connect to the Internet or the controller. Current solution is that, giving a predefined time interval, the node will unmount the overlay, finish the experiment and going back to the default state.

2. The node is performing an experiment and the controller is not responding the pull requests.

   In this case, the node will not be able to receive any of the researcher's requests, even having Internet connection. Current solution is to wait **N** pull requests not correctly processed and then unmount the overlay, finish the experiment and go back to the default state.

3. The node is working in its default state and the controller is not responding its pull requests or it has no connection to the Internet.

   In this case, the node is not performing an experiment but, for some reason, its configuration has been modified or the node has lost Internet connectivity. Current solution is to wait a predefined interval of time and go back to the default state. Consequently, if the node is isolated a long time, it will be going back to default state continuously.

It is important to keep in mind that in some experimental scenarios that may have special disconnection needs, that recovery system must be

reconfigured willing to allow bigger time intervals or pull request or, in some cases, to totally stop it.

3.3.2 Controller acknowledgement system

One of the most important troubles that the WiBed team has faced is that, when working in a pull-based system, the controller could receive a repeated request due to connectivity issues or because the latency between the node and the server. Keeping that in mind, the solution proposed was to use an acknowledgement system between node and controller in commands, experiments and its results. The experiment part just adds information regarding the experiment being performed by the node so, focusing in the command's part, which is the important one, the controller has a auto-increased list of executed commands, and which nodes were involved, willing to save an order of execution and also not to repeat commands already finished.

The information that is currently keeped in each node is:

- **exp_id**: This variable shows the last or current experiment in which the node was involved.
- **commandsAck**: This variable shows the last successful command received by the controller and executed in the node.
- **resultsAck**: This variable shows the last successful result sent to the controller by the node.

The information kept for each node in the controller is:

- **commandId**: This variable shows the ID of the last sent command to the node.
- **executionId**: This variable shows the ID of the last executed command in the node that its results have been correctly received.

To summarise, the point of the acknowledgement system relays in the synchronitzation of the commandsAck-commandId and resultsAck-executionId. The procedure of executing commands follows this sequence:

1. A researcher introduces a command in the controller.
2. The controller checks the last command's ID (commandId) and sets the new command as the number *ID+1*.
3. The node executes a pull request: It sends its status, the commandAck of the last successfully executed command and a JSON formmatted list with the results pending to be sent to the controller.

4. The controller receives all the information regarding node's state and responds with the new command to be executed and the executionId of the last succesfully received result.

5. The node receives the new command ID and, if is greater than its current commandAck, it executes the command, stores the results and sets the commandAck to new command's ID.

6. In the pull request after the command has been executed and the results have been saved, the node sends the information regarding the status, commandAck (updated) and the command's results (its ID and the outputs).

7. The controller receives the request, synchronises its variables with the commandAck and its results and sends back to the node the updated executionId, which is the successfully completed new command's ID.

3.4 Node management system

In order to improve testbed's management system and to show the information regarding nodes in an easy way, WiBed platform nodes always have a status related to the system's current state. This status methodology is used as much in the node side as in the controller side[9], not only to interact and respond researchers orders, but also to make the testbed easier to be monitored and also, with the tools available, *self-fixable*. Currently there are 8 valid statuses:

The status in WiBed is a variable found in */etc/config/wibed* in the *general* section. Most of the scripts that the testbed has use this value willing to adapt its behaviour to the final action that will be applied. In addition, the controller acts in the same manner using the status of the node as a control of the functionalities available in the webpages.

Some status-behaviour examples:

- From status 2 to 6 means that the node is in an experiment, so it can not be shown as an available resource in the testbed.
- ERROR status prevents the controller to take any other action with a node except for sending commands.
- INIT status occurs when the node is recently upgraded or when it is registered in the controller. Also it is the resulting status when resetting the node due to any issue.
- Any different status from 0 to 8 will be treated as an invalid status and the controller will not accept it.

[9]Controller's API examples: `https://wiki.confine-project.eu/wibed:unified-api`

Status	Name	Description
0	INIT	Initial state of the node. In this state, the node is not configured yet.
1	IDLE	Main status. The node is working correctly and is waiting for orders.
2	PREPARING	The node has received an experiment command and it is downloading the overlay.
3	READY	The node has correctly received the overlay and it is waiting to install it.
4	DEPLOYING	The node has received the order to being the experiment. The overlay is installed and mounted and the node reboots.
5	RUNNING	The node is in an experiment and waiting for orders.
6	RESETTING	The node unmounts the overlay and reboots to the default state.
7	UPGRADING	The node has received an upgrade order. It will download and upgrade the firmware and finally reboot.
8	ERROR	The node has detected an error and sends logs to the controller. The node waits until the administrator fixes the issue.

Table 3.1: WiBed status table

3.4.1 Status operation

As can be seen in figure 3.1, the WiBed status system operation has a logical flow provided by the wibed-node script. As its description is already explained in table 3.4, this is a summarization of statuses operation:

- INIT: This status can change only to idle state and this happens when the node is registered to the controller.
- IDLE: As the default state, it can change to all the status except from READY and RUNNING, which are subsequent status of PREPARING.
- PREPARING: It can change to error if the overlay is not correctly downloaded or if the controller does not respond.
- READY: It can change to ERROR if the controller does not respond or to DEPLOYING if all is correct.
- DEPLOYING: This is a *hidden* to the controller status that comes from READY and goes to RUNNING. This is a bypass status that occurs when the node installs and mounts the experiment's overlay internally. If the overlay's installation is correct, it sets the status to RUNNING and reboots.
- RUNNING: In an experiment, this status can go to ERROR if something wrong happens or to RESETTING state.

- RESETTING: This is a *hidden* to the controller status that comes from RUNNING when the controller sends a Finish order. Then, after the node has sent successfully the results to the controller, it dismounts the experiment overlay and mounts the default one. Finally, it reboots in order to recover into the IDLE state.
- ERROR: This status can come from every status and can go to each status depending on the situation and action taken by the researcher or administrator.

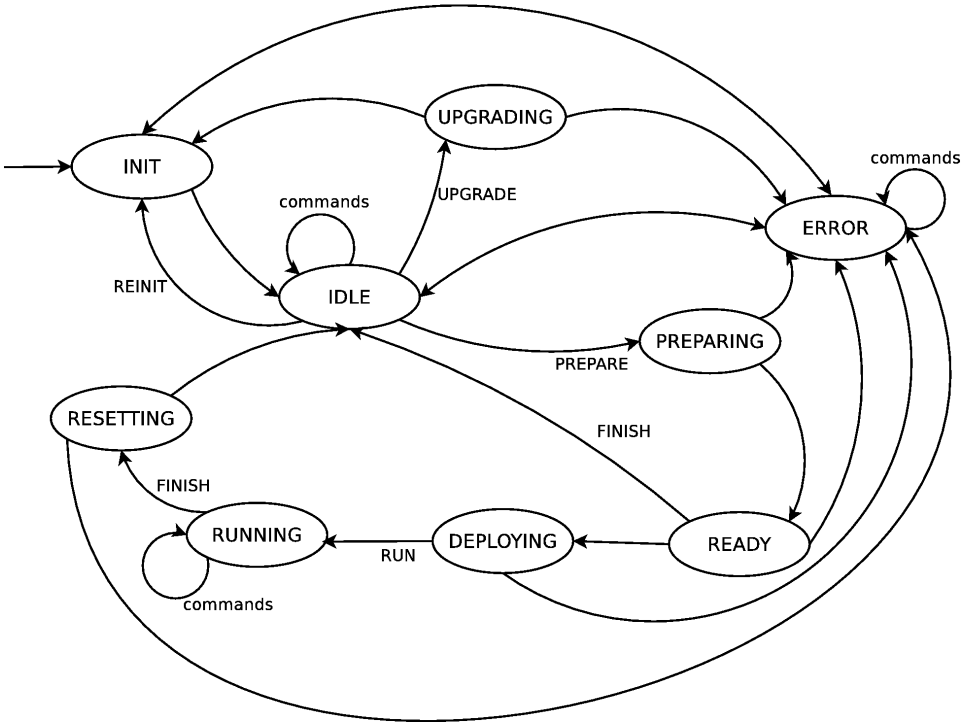


Figure 3.1: WiBed node-side status system operation

3.5 The Overlay FS and experiments

As the main idea of what OFS is has been further explained in the The Overlay File System section 2.2, this section wills to explain more practically how it is applied and managed in the WiBed supported node TL-WDR4300 seen in the section 3.1.2 and how the experiments take in advantage the OFS features.

As described in the section 3.1, the WDR4300 router has 8MB of flash memory. The OFS is distributed as can be seen in table 3.5 and described in this manner:

- mtdblock0: The u-boot partition. It contains the analogous version of computer's BIOS. It is read only.
- mtdblock5: The *rootfs* partition. It is the partition which contains the firmware separated in: the Kernel, and the other firmware parts.
  - mtdblock1: The Kernel partition. It contains the Kernel of the OpenWRT firmware. It is installed by OpenWRT but protected as read only.
  - mtdblock2: The base file system partition. Here will be mounted the Overlay FS as the / folder.
    - Unnamed partition: This partition is the one that will act as the core system. It is mounted in the /rom folder, uses a SquashFS and it is read only.
    - mtdblock3: The data partition. This partition is the one that will act as the internal overlay file system. It is a writable JFFS2 partition where the merged overlay will save all the changes.
- mtdblock4: The art partition. It contains the Atheros tools that calibrates and manages the radios. It is read only.

<table>
<tr><td colspan="5">OpenWRT overlay FS in the WDR-4300
[8192KiB]</td></tr>
<tr><td>u-boot (0)
[128KB]</td><td colspan="3">The firmware (5)
[8000KB]</td><td>art (4)
[64KiB]</td></tr>
<tr><td>Read Only</td><td>Kernel (1)
[1280KiB]</td><td colspan="2">OverlayFS: / (2)
[6720KiB]</td><td>Read Only</td></tr>
<tr><td></td><td>Read Only</td><td>SquashFS: /rom
[1536KiB]</td><td>JFFS2: /overlay (3)
[5184KiB]</td><td></td></tr>
</table>

Table 3.2: OpenWRT distribution of the File Systems using OFS

.

The SquashFS is a read only file system compressed in LZMA focused in being as reduced as possible. Actually, as it is a read only system and do not need to be modified, it is from 20 to 30% more compressed than the writable JFFS2 file system.

The JFFS2 is a writable file system which counts with a journaling system, is compressed in LZMA and also counts with a wear leveling technique.

The journaling file system is a technique consisting on keep the log of file changes in a file system. Journaling techniques track all the changes owing to prevent the information lost or corrupted if the system crashes.

The wear leveling is a technique used in most of Flash, SSD and USB drives focused in the management of the data distribution when is saved in the device, owing to homogenise the accesses to the memory blocks. This technique is still being developed due to the SSD future expectations and because the current management being performed in HDD shortens the utile life of this type of devices.

3.5.1 The default overlay

Keeping in mind the default state of the FS seen in figure 2.2, when being in the default state[10] (the nodes are not performing any experiment), the storage system that is installed as the overlay of the *core* FS is the internal, as can be seen in figure 3.2. All the changes performed into the system, will remain in the internal *modifiable* partition.

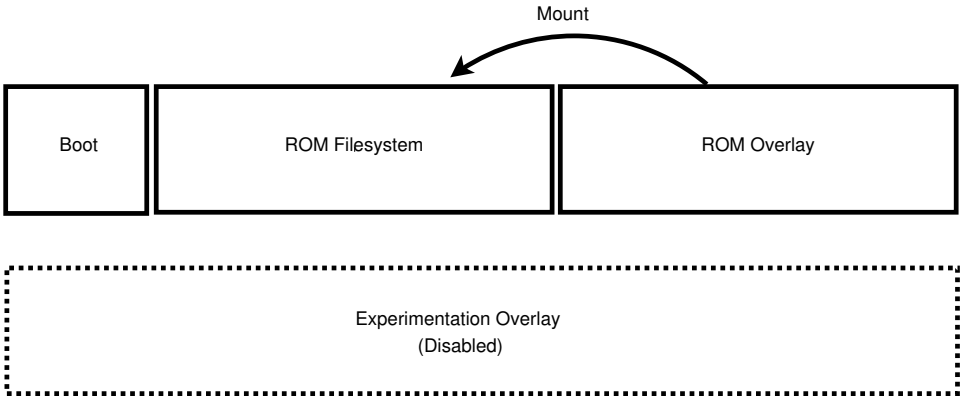


Figure 3.2: The FS is using the internal overlay.

3.5.2 The experiment overlay

Keeping in mind the default state of the FS seen in figure 2.2, when performing an experiment the controller will send the overlay to the node, and then, the node will install it into the external storage overlay[11]. Then, the internal modifiable FS is synchronized with the external[12] and mounted as

[10] In the default state the nodes waits in IDLE.
[11] The external storage is an USB of 16GB using the ext4 File System.
[12] All the changes performed during the previous default state remains in the experiment.

the overlay FS to the *core* FS[13] as can be seen in figure 3.3.

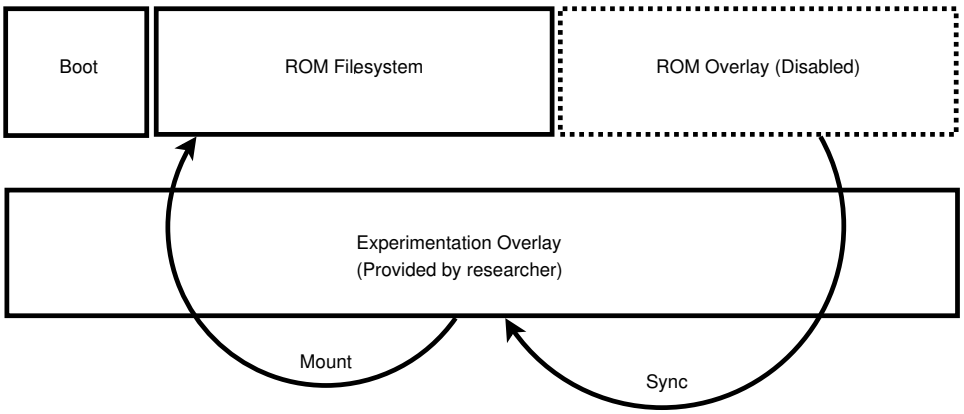


Figure 3.3: The FS is using the external overlay.

3.6 The WiBed firmware: main scripts and configurations

This section aims to give a brief description of how WiBed firmware is made and how its most important scripts work. Finally, in the Appendix section E, there is a copy of these scripts.

3.6.1 The WiBed firmware

WiBed node's firmware is compound basically by an OpenWRT as the base system and repositories providing all the extra packages needed to work with the platform. The OpenWRT used is a frozen commit version of the trunk repository of the OS. This version is an already tested and approved by the WiBed team. Versions scheduled to be maintained are 'master' as the one with a stable and contrasted correct behaviour and the 'last_trunk', which is using a newer commit, bringing updates to the platform as well as it is also less tested. Moreover, OpenWRT supports 44 different architectures and WiBed firmware and packages are currently compiled just for *ar71xx* as it is current WiBed supported node architecture (the TL-WDR4300). In addition, the WiBed team has already tested *mpc85xx* (TL-WDR4900) and *x86* (ALIX2d2) architectures and are currently under development. Finally, current WiBed firmware fits on 8MB or more devices but future improvements will reduce the firmware to be included in devices up to 4MB of flash.

[13]When the experiment finishes, the external overlay is disabled and the internal restored.

3.6.2 wibed-node

The wibed-node script is the one in charge of making the pull requests to the controller and also to manage node's behaviour according to the status of the node in each request and the response received. The script is scheduled each 15 seconds and its output is redirected to a log file willing to keep any issues during the operation of the nodes. To sum up, its main behaviour is:

Request information according to each state:

- If status is INIT, then send to the controller all the information needed to register the node in the database.
- If status is IDLE or RUNNING or ERROR, then just send information regarding the commandAck and pending results of commands not acknowledged yet by the controller.

Once received the response of the server, the script should act consequently not only with the response received, but also with the status which receives that order.

- If the node was in INIT state, a successful response puts the node into the IDLE status as the controller acknowledges it as a valid node.
- If the node is in INIT state and comes from a failed upgrade, it goes to ERROR state.
- If the node is in IDLE state, the script checks the response and acts consequently:
  - If the response is a REINIT order, the node must execute the wibed-reset script owing to restore default status.
  - If the response is an UPGRADE order, the node must perform a firmware upgrade and go to UPGRADING state.
  - If the response is a PREPARE order, the node will start downloading the experiment's overlay and from PREPARING to READY state if the overlay is successfully downloaded.
- If the node is IDLE or RUNNING or ERROR state and receives a command order, the node will execute it and send its results in the next pull request.
- When performing an experiment (status from PREPARING to RUNNING), the node must check experiment related orders in the response of the controller such as:
  - If the response is a RUN order, the node will go from READY to DEPLOYING state, then it will install the overlay and, if it succeeds, the node will reboot and change to RUNNING status.

– If the response is a FINISH order, the node will change its status to RESETTING, then will send to the controller the files in the /save folder in a *tar.gz* file and, if all succeeds, it will unmount the overlay, change its status to IDLE and reboot.

3.6.3 wibed-config

The wibed-config script is an all-in-one configuration tool that is able to set all required settings but keeping in mind that it will not apply any of these configurations as it is not its purpose. The area of actuation of wibed-config script is:

1. Reset and clean the network and wifi previous configuration files.
2. Configure the network and wifi:
   (a) Configure the management networks (LAN and wifi)
   (b) Configure the wireless radio settings
   (c) Configure the BATMAN-ADV settings
3. Configure basic node information such as its hostname, model or identification name.
4. Configure node's location information.

3.6.4 wibed-location and libremap.net

The location of the nodes is a really interesting and hard to face problem that has been discussed and studied since the beginning of the project. The main problem that is manifested when trying to locate the nodes is the complexity of developing a 3D-map of the emplacement and network. Most of the current solutions use 2D maps, such as GoogleMaps or OpenStreetMaps, that fits in a wide-extended network as could be Guifi.net, which has near to 25.000 operating nodes in an extension of 45.000Km wide. That solution is not enough when working in testbeds deployed only in a single or few buildings and locating the nodes in different floors. As can be seen in the figure 3.4, a 2D mapping tool is not clear enough to show node's location. In the opposite side there are solutions as the one used by Freifunk, using GoogleEarth view to represent the network in 3D using KML representational language.

The WiBed final solution is still in development and the current one is using two 2D maps to represent: in the first one an approximate location of the nodes in each building and, in the second one, the exact location of the nodes, in each floor of each building. It must be taken into account that, while the first solution is to use an external open source worldwide map visualisation tool, the second one is a web page built in HTML5, CSS and

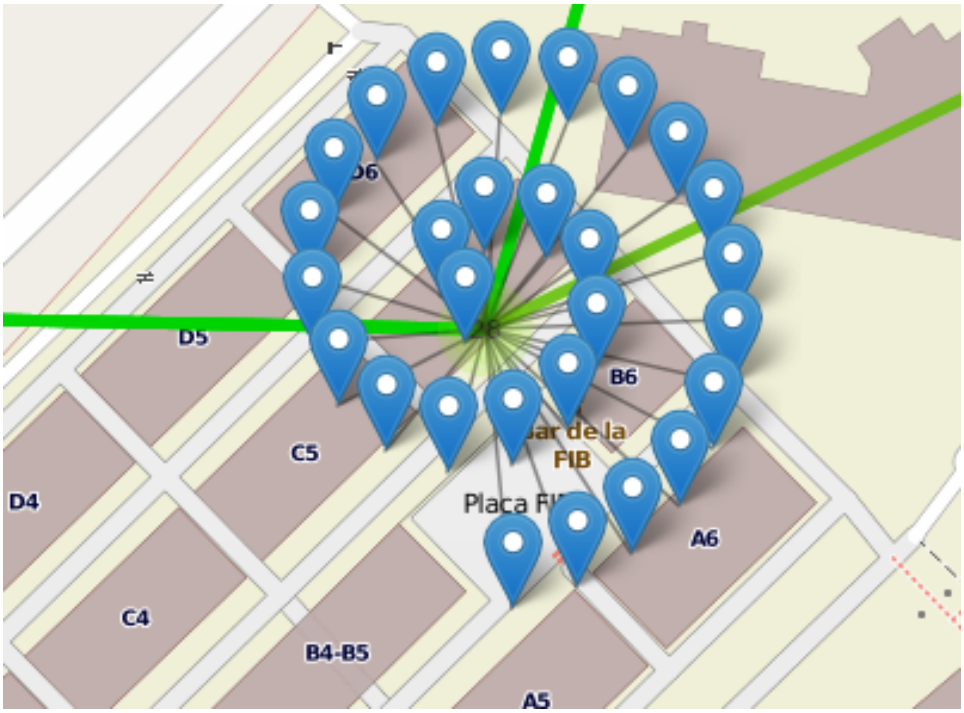


Figure 3.4: Representation of a large group of nodes close to each other.

JavaScript with the hard-coded representation of each node in its location showing node's basic information.

In order to keep both maps updated and monitored, a script to manage location's information was conceived: the *wibed-location* script. This script allows to change the coordinates and to send them to the *libremap-agent* script, which will communicate with the Libremap.net API owing to print our testbed nodes in the map.

Future improvements of WiBed location tools are to design a 3d map of the Campus Nord using Three.js[14], a JavaScript interactive framework, and to document all the map design process willing to ease the process for coming developers or researchers. Future wibed-location script improvements will allow the administrators to configure not only the coordinates, but also the description of the node spot and also to export these configurations and descriptions to both map tools.

3.6.5 wibed-upgrade

The upgrading system is a powerful tool recently improved that allows the testbed administrator to flash a specific firmware to a selected group of nodes through the web administration page. Moreover, in order to flash a firmware,

[14] `http://threejs.org/`

the administrator needs to download or compile a WiBed firmware, obtain its MD5sum and upload it to the server. Server's requirements to accept a firmware are:

1. The firmware name must be the version of the WiBed firmware file.

   WiBed firmware name example: *WiBed.master.**927d4d87**-ar71xx-generic-tl-wdr4300-v1-squashfs-sysupgrade.bin*

   - master: Branch of the development repository.
   - 927d4d87: Current version of the firmware (based on the wibed-packages repository version)
   - ar71xx: Target architecture.
   - generic-tl-wdr4300-v1: Target router model.
   - squashfs: Target filesystem.
   - sysupgrade: Type of firmware installation.

2. The MD5sum value must be correct. Otherwise, the firmware will be rejected.

It is important to keep in mind that if a researcher compiles a firmware version, so he/she is not using one of the tested images in the WiBed repository, the controller will not be able to check its correctness because it only protects both controller and nodes from corrupted files due to the transmission process, but it will not be able to check if the firmware file has bugs or compilation issues and, for that reason, the process of upgrading nodes may brick the nodes.

3.6.6 Spread the word script

Due to the issues found in the upgrading system before the WBM, I envisaged an "*Spread the word*" system to flood the mesh network from the gateway to each node with the firmware owing to update each node automatically. This system was interesting to develop but also unnecessary complex, so after a discussion with the WiBed team, we decided to put efforts on remake the existing upgrading system (see the Appendix E.4) instead of continuing this script development[15].

A brief summarize of the "*Spread the word*" script:

Gateway's side

- Get the firmware and checks its MD5.

[15]Spread-the-Word code repository: `https://github.com/eloicaso/wibed-testing-scripts`

- Spreads the firmware and the script, performs security checks and mark neighbour nodes.
- Make neighbours execute the script and ends.

Node's side (all files involved are supposed to be checked)

- Check if neighbours have been marked.
- Check if neighbours have the same version as the upgrade one.
- Send the firmware and the script, perform security checks to neighbours and mark them.
- Make neighbours execute the script.
- Wait some random time and upgrade.

3.7 WiBed's source repositories

The WiBed platform is an open source solution counting with a project related documentation wiki page. The current system has changed due to the Battlemesh and will be further explained in the section 5.1 but, regarding this section, Redmine system will be explained.

Following the CONFINE project methodology, WiBed source is hosted in the Redmine server [16]. Redmine is an open source (GPL2) web solution to manage projects. Its main functionalities are: Wiki management, source repository and version handler, scheduler, issue and task manager, forum service, etc. To sum up, it is a complete project manager helping to simplify project's control

WiBed project current source repositories are:

1. *wibed-controller*: This repository stores the development source of the server-side code. This source consists on the web server and database management.
2. *wibed-openwrt*: This repository stores the base necessary files to compile the WiBed firmware. This base system consists on a frozen snapshot of the OpenWRT, stable and further tested, plus the extra repositories needed for WiBed firmware linked to the compilation process.
3. *wibed-openwrt-routing*: This repository stores a frozen snapshot of the OpenWRT routing repository. It is used as a tested and stable source for the routing protocols used in WiBed.

[16] WiBed Redmine project page: `http://redmine.confine-project.eu/projects/wibed`

4. *wibed-owrt-packages*: This repository stores a frozen snapshot of the OpenWRT packages repository. It is used as a stable and tested source for the packages used in WiBed.

5. *wibed-packages*: This is the base repository of the WiBed packages which stores the two branches of packages development that compounds the firmware.

Chapter 4

Deployment

In this chapter, the testbed mesh network in the Campus Nord of the UPC, its components and the process of the deployment will be presented as well as the problems found and decisions taken to avoid them.

4.1 UPC CN-A testbed mesh network

UPC CN-A testbed network deployment has changed from its first theoretical steps. All the deployment process will be explained in this section, focusing on all the issues that altered the network and the decisions taken owing to solve these issues. From deploying a network with about 60 nodes in the buildings from A1 to A6 and floors from -1 to 2nd, putting the idea to practise has cost more than expected.

First problems came with the reinforced concrete in the walls on the lower floors classrooms. Later, the deployment faced the difficulty to throw connectivity to upper floors, needing to deploy some nodes outdoors in ventilation holes -some of them were not usable because of the hot air extraction when air conditioners are working- and the problematic with the insufficient space inside the dropped ceilings. Finally, the deployment changed to be just in upper floors -first and second- where the conditions improved connectivity and accessibility and the inclusion of some nodes in the roof to obtain an heterogeneous network -as similar as possible to a real testbed-.

4.1.1 Network previous state: first and dismissed deployment

The first nodes of the UPC CN-A testbed network were deployed before this project began (and after theoretical project finished). This section aims to explain how the deployment was planned -theoretically-, how this deployment started in the beginning of this final degree project -A5 and A6 buildings in a vertical deployment-, why this deployment was modified -A1

to A6, in upper floors and horizontal deployment- and how the deployment issues have been solved.

Original deployment

The original idea of the UPC CN-A network was presented at Roger Baig's Final Degree Thesis in Computer Science on the UOC University and in collaboration with the CONFINE Project of the UPC University. The theoretical deployment was planned to be in the buildings of the Campus Nord (UPC) in Barcelona, using the buildings from A1 to A6 as a 260 meters long, 20 meters wide and 5 storeys tall buildings. This deployment (theoretical map in figure X) started with a gateway in the -1 floor in the operators' network rack and 4 nodes: 2 in the maintenance zone and 2 on the ground floor (A5001 and A5002).

In order to understand how we dealt with the problems we will see in this section, it is necessary to kno w how the network was thought to be deployed. WiBed nodes have 2 wifi radios: one internal 2.4GHz and one external 5Ghz. One of the two radios should be used as the management network to control and interconnect all the nodes in the mesh. This means that we will only be able to use the other radio to experiment with. For this reason, an extra 2.4GHz USB Radio was added in order to free both node radios and use the USB as the management network.

First steps

Main deployment was composed by one node as a gateway in the -1 floor of the A5 building, two nodes in the -1 floor of the A6 building in maintenance warehouse and once the deployed nodes were checked (checking the hardware status and updating the update), new nodes were added to the testbed: two nodes in the A6 ground floor and one more in the A6E floor (on an unused disabled's bathroom). With a prototype of the mesh (about 8 nodes) we start facing the first connectivity issues. The main connectivity problem of the A buildings was that all main walls are made of reinforced concrete and that all the walls, lands and roofs on floors 0 and -1 are made of this material too. Furthermore, the structure of the classrooms -amphitheatre- supposed an extra inconvenient because the triangular form of the land -made with reinforced concrete that isolated the nodes.

Classroom wardrobes

Focusing on the amphitheatre classrooms, this isolation problem supposed one of the two main reasons to leave lower floors -and the reason why these floors were rejected in the final network distribution- and deploy the network in the 1st and 2nd floors. In figure X can be seen that all the ground floor classrooms forms a series of pronounced concave and convex walls of

nontransferable wifi waves -in addition of the between-buildings main walls-. In order to surpass these handicaps, we added extra nodes in wardrobes that were inside the classrooms and on the opposite side of the classrooms. Adding more nodes allowed us to have a better connection between nodes that were hidden from neighbour classroom nodes but it actually didn't solve the problem. In addition, these wardrobes were not secure and any student could open it and steal the nodes so this solution was only temporary.

Adding a 2.4GHz USB Radio

In spite of improving the connectivity adding nodes in the wardrobes, connectivity problems remain, so, the idea of adding a dedicated USB radio owing to be the 2,4GHz management network and releasing the internal radio for experimenting was put into practice -the radio selected was a TP-Link TL-WN722N consequently of being supported by OpenWRT community-. At this moment, the WiBed testbed was composed by 8 nodes and the USB radio was set as the management radio increasing the connectivity -from a power of 17dBm to 20dBm-. After that, the testbed expanded adding two nodes in the -1 floor and two nodes in the E floor, increasing testbed nodes to 12 nodes. Suddenly, some nodes started to be unreachable by the management network and needed to be checked manually so we had to find the issue and solve it. After some days of study and some stress tests, we found that the USB Radio capabilities in Ad-hoc mode were unstable and it could not bear the network in that mode. As a result, the USB radio was set to be only for experimenting and not for be used as an Ad-hoc management network.

Growing to upper floors

With the previous experience and a testbed of 12 nodes (gateway not used in experiments), an INESC researcher[1] started to test the network doing some experiments while, in parallel, he started also a new deployment in his university campus. Thought the experiments done in the WiBed server worked well, his feedback about UPC CN-A testbed was that the connectivity was weak, so we started to study upper floors with some mobile nodes. That gives us a really important information regarding the 1st and 2nd floors: while lower floors are made in major part of reinforced concrete, upper floors just have few mainstays and walls made of that material. Thus, we started deploying 4 nodes in the 1st floor -in maintenance wardrobes in each side of the buildings- owing to check its behaviour against experimenting situations.

Despite we found that the A6 1st floor right wardrobe node had really bad connectivity, the other nodes had great connectivity between them, so we deployed 4 nodes in 2nd floors of the A5 and A6 buildings. The problem

[1]INESC Porto: `http://www.inescporto.pt/`

with connectivity of the righter node was found also in the 2nd floor and, studying the walls behind the wardrobes, we find out that were made of reinforced concrete and that they were also mainstays. As a result of that study, moving the testbed network deployment to upper floors, gave the project the perfect situation for the main deployment.

Both 1st and 2nd floors have reinforced concrete external walls but most of the internal separation walls are made of bricks and plaster. As a result, with weaker internal barriers and nearer nodes in external reinforced concrete barriers these floors are a good option to deploy the network. Furthermore, the location of these nodes was the same as planned in the theoretical deployment so the collocation of the nodes was easy and secure (closed maintenance wardrobes) and previously agreed with UPC.

With the prototype deployment done, we started to study the behaviour, strong and weak points of upper floors. The conclusions of our study and the feedback given by the researcher was:

- The connectivity between 1st floor and 2nd floor nodes is greater than the connectivity in lower floors as a result of weaker indoor walls and closer nodes in the between-buildings reinforced concrete walls.
- There are elevated metallic trays inside the building that can hold nodes in both sides of the building and inside classrooms of the building which gives a secure place to deploy the nodes.
- The resulting mesh network using this distribution will have 34 nodes (and 2 gateways).
- Using the wardrobes to place nodes lacks the connectivity of the nodes.

In figure 4.1 the status of the current deployment can be seen. Nodes ID colour represents the feasibility of the controller to connect with it using the management network, white meaning connectivty and black no connectivity. Moreover, the background colour of the nodes ID represents the quality of the average connection with the controller from green (excellent) to red (poor).

4.1.2 Network current state: final deployment

From previous to final network

Once studied and having decided to deploy the network only on upper floors, we needed to arrange a budget with the maintenance department owing to supply the costs of installing new power lines to plug the nodes and also arrange a budget with UPC-Net, who is in charge of adding 2 internet connections (for 2 gateways) and to revoke the actual one in the A5 (-1) floor.

Main points of this deployment arranged with the maintenance department are:

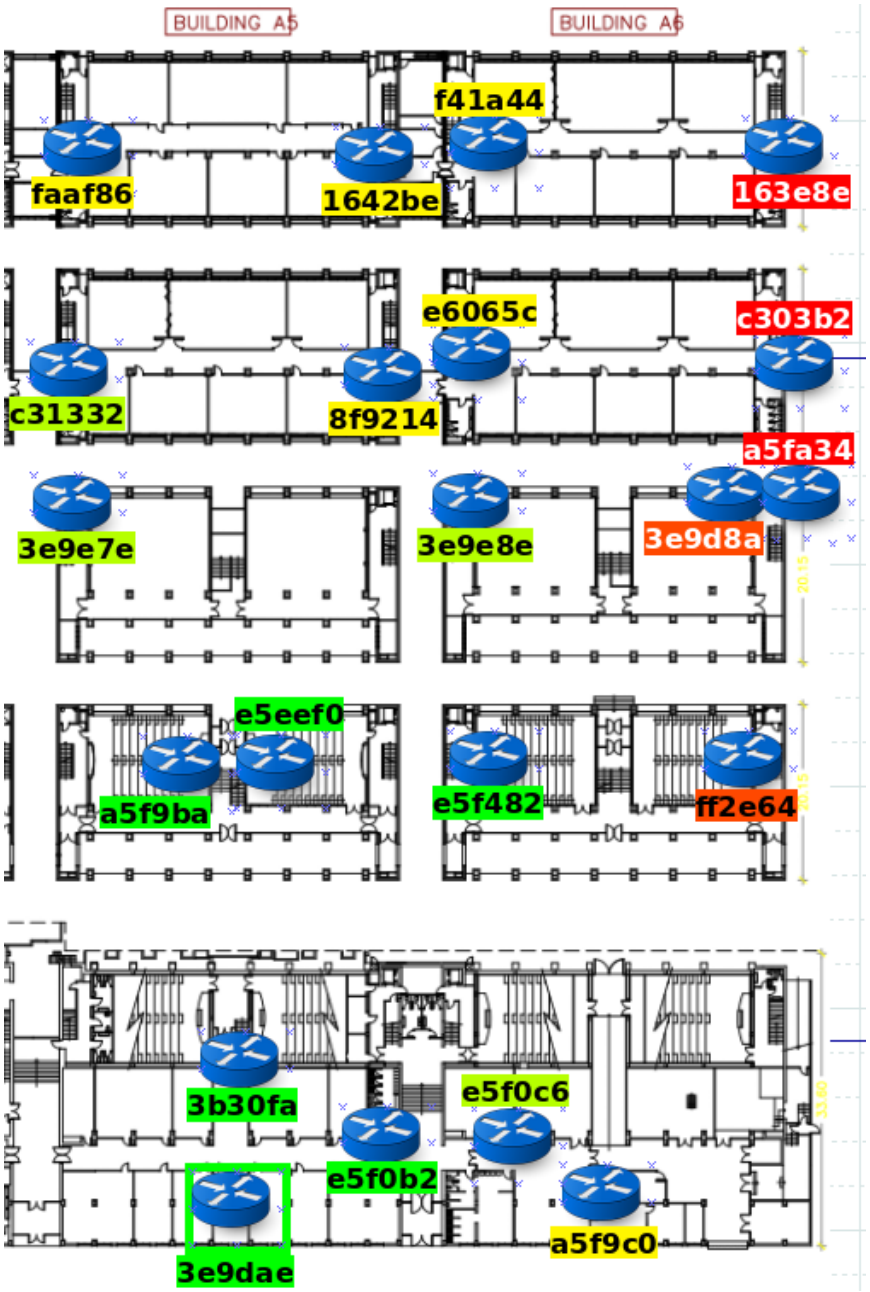


Figure 4.1: UPC CN-A current deployment state.

- Nodes will be placed from A1 to A6
- Nodes will be placed in 1st and 2nd floors
- Nodes will not use any separated electrical line

- All nodes will be placed in elevated trays and secured with flanges
- There will be one node in each side of the floor and another one inside the middle classroom

Final deployment

UPC CN-A final network was going to be deployed before may ends or at early June but, at the end of writing this dissertation, we just received maintenance's budget confirmation so power supplies' installation is being done. From now to the end of this section, the information of the deployment is expected to be done during early July. Furthermore, the two gateways are already installed in buildings A2 and A4 (2nd floor) that are actually working. Figure 4.2 presents this final state of the network when the deployment will be finished.

Final network will fit on arranged requirements shown in previous section. Using upper floors following the described deployment will make the mesh testbed network a robust and reliable network to use the WiBed platform, as well as it will improve and solve connectivity problems found in lower floor reinforced concrete walls. Moreover, the network will gain in accessibility while leaving just 12 nodes inside classrooms (making harder to access them) and keeping in mind that 2 of these 12 are gateways which are assumed not to be accessed or manipulated (only for network administration tasks).

The main point of this deployment is the improvement on the reliability and strength of the connectivity between the nodes, being far enough not to see all them but to be close enough not only to have a strong 2.4GHz connectivity, but 5GHz also, to enable experimentation and management with both of them indistinctly. Concretely, there are three nodes in each floor and building (from A1 to A6 building and from 1st to 2nd floor), located in elevated trays already used to transport the Ethernet/optical fiber cables together with power ones by the floor. Each node is placed above the tray, fixed with a strong flange to prevent movements or thefts. Furthermore, in every floor there is one node near the right corridor door, one more near the left corridor door and the last one inside the middle class of the bathrooms side of the building. Comparing this new location against the previous one (in maintenance wardrobes), the nodes may seem more exposed to students but, actually, this new location improved the connectivity and also it is a more correct place for research nodes than a cleaning and maintenance stuff wardrobe.

WiBed future extensions

As the UPC CN-A mesh network is using the entire usable part from A1 to A6 in Campus Nord (understanding usable part as 1st and 2nd floors and with good wireless connectivity signal), the network should not be able

to grow in the future. Actually, that topic was already discussed, studied and solved being the next step in WiBed project goal. Moreover, this next step consists on adding support to ALIX2 routers and include them into the deployment and change the minimal requirements of the nodes allowed in the platform:

First, adding support to ALIX2 routers, WiBed platform improves its testbed heterogeneity as well as the possibility to use this powerful routers as roof bridges to interconnect not only the closest buildings but also the far ones, bringing the possibility to interconnect testbeds that initially could be isolated and increasing experimentation possibilities besides of having a closer to production environment testbed.

Finally, as explained in the components section (not only the specifications but also the router type), changing the minimal requirements to fit in simpler nodes will allow to low, even more, the cost of replicate the project but keeping in mind that this will add some complexity to fit the firmware in these simpler routers. Furthermore, it is important to take into account that these nodes will have a special firmware version without unnecessary packages, willing not to waste any MB of their little Flash capacity.

4.2 Battlemesh network

The Battlemesh network was deployed during the WBMv7 in Leipzig as already seen in section 1.4.5. This deployment information is important in order to understand, not only the results of the main experiment seen in section 5.2, but also the conclusions and reasonings described in section 5.2.6 and chapter 6.

The situation:

- The Sublab[2], this year's Battlemesh emplacement, is an old factory (walls, roof and floor are reinforced) converted into a hacking space and maintained by its own community.
- The testbed network was deployed among the two upper floors of this factory and in the upper floor of the building in front of the factory (3 floors below).
- Both factories upper floors were correctly connected by the testbed network thanks to WiBed nodes in the entrance of each floor.
- The factory and building networks were correctly connected via wireless due to the nearby location of both buildings and nodes.
- During the week, due to experiments needs, some nodes were moved from its emplacement, so an exact and final deployment map is not

[2]Sublab[Ger]: `http://sublab.org/`

available (Figure C.1 in Appendix C.3 shows nodes first emplacement in the upper factory floor[3]).

Environment of the network during the Battlemesh:

- The default environment of the sublab is to have about 15 different wireless APs in both 2.4Ghz and 5GHz bands.
- The first day, about 20 new different wireless APs were count.
- About 50 different laptops were transmitting into the wireless environment a time.
- There were different experiments being performed by the experimenters and using the wireless environment.

The WiBed testbed:

- The WiBed testbed was compound by 20 WDR4300 nodes deployed in the 2 factory floors and in the office building.
  - 3 nodes in the front building.
  - 17 nodes in the factory.
    - 1 mobile node (battery Power Gorylla 21.000mAh[4]).
    - 2 GW in the main hall.
    - 3 nodes on the lower floor deployed throughout the hall.
    - 11 nodes deployed on the upper factory floor. These nodes changed its location as the experiment changed.

[3]Maps of the other floors and the position of the nodes are pending to be received

[4]`https://www.powertraveller.com/en/shop/portable-chargers/professional/powergorilla/`

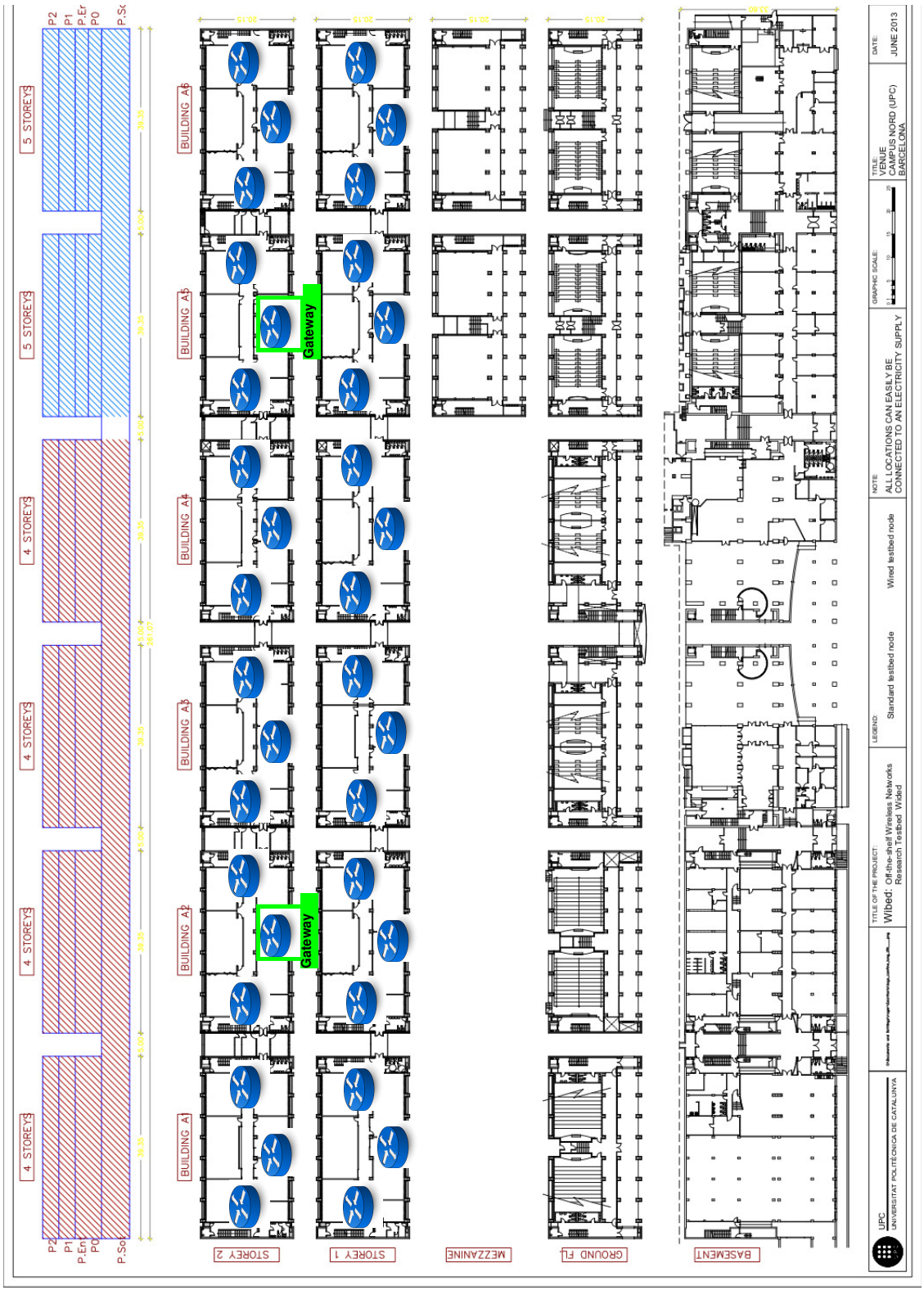


Figure 4.2: Final deployment of the UPC CN-A network.

Chapter 5

Validation of the platform and network

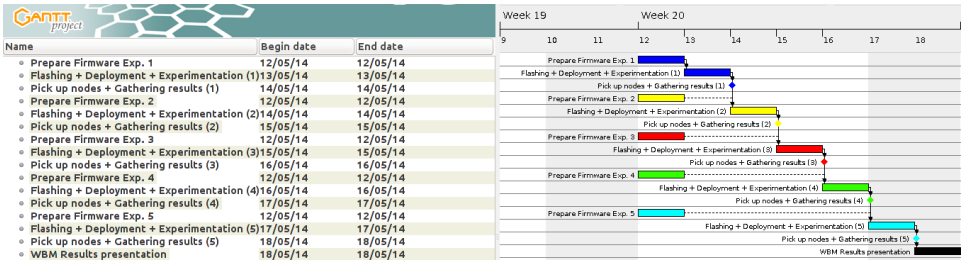
5.1 WBMv7 and experimentation environment

5.1.1 Relation between WiBed and the WBM

As we already mentioned, the WBM is an event strongly connected with WiBed project since it was originally an idea to improve and speed up the WBM procedure. After the Battlemesh, and with the results of the experiments and researchers opinion, we could assure that the WiBed platform was a step forward the Battlemesh experimenting procedure. While on one hand, the WBMv6[1] process of performing an experiment was to setup the firmware manually, flash the firmware in each node, deploy the nodes in the choosen spot, perform the experiments and get the nodes back because the next experiment must follow the same process; on the other hand, the WBMv7 (done in Leipzig, Germany) used our platform, which took some time to adapt the firmware from the UPC to the Battlemesh needs (from Monday to Wednesday), but once that was done, the firmware was flashed in each node (Wednesday), nodes were deployed in the chosen spots and, finally, researchers could perform their experiments one after another by distributing the resources (researchers could experiment with the nodes from Wednesday to Saturday). Again, we can assure that WiBed was an step over because the Battlemesh was following a procedure of start and repeat the loops described to perform experiments in Monday and finishing the **N**th experiment on Saturday, starting to process of the results of this experiment on Sunday, so most of WBM events didn't finished the experimentation nor the results gathering due to this long process.

As can be seen in Figure 5.1) the procedure of perform several experi-

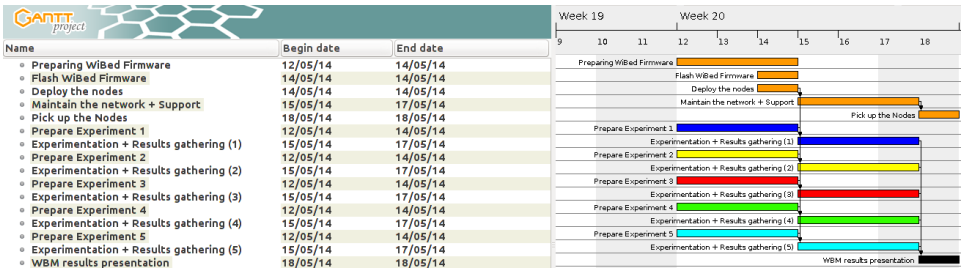
[1]The WBMv6 was performed in Aalborg, Denmark. More information about this edition can be seen in Appendix C.2



Name	Begin date	End date
Prepare Firmware Exp. 1	12/05/14	12/05/14
Flashing + Deployment + Experimentation (1)	13/05/14	13/05/14
Pick up nodes + Gathering results (1)	14/05/14	14/05/14
Prepare Firmware Exp. 2	12/05/14	12/05/14
Flashing + Deployment + Experimentation (2)	14/05/14	14/05/14
Pick up nodes + Gathering results (2)	15/05/14	15/05/14
Prepare Firmware Exp. 3	12/05/14	12/05/14
Flashing + Deployment + Experimentation (3)	15/05/14	15/05/14
Pick up nodes + Gathering results (3)	16/05/14	16/05/14
Prepare Firmware Exp. 4	12/05/14	12/05/14
Flashing + Deployment + Experimentation (4)	16/05/14	16/05/14
Pick up nodes + Gathering results (4)	17/05/14	17/05/14
Prepare Firmware Exp. 5	12/05/14	12/05/14
Flashing + Deployment + Experimentation (5)	17/05/14	17/05/14
Pick up nodes + Gathering results (5)	18/05/14	18/05/14
WBM Results presentation	18/05/14	18/05/14

Figure 5.1: Gantt of WBM without the WiBed Project

ments without the WiBed platform supposed to have a bottleneck that disabled parallel work in different experiments unless these experiments were using different resources and keeping in mind that changing the experiment means to change the firmware[2]. That changed with the application of the WiBed platform, where the experiments can be performed sequentially, or in parallel if the resources are different, during all week using always the same firmware and changing only the particular components that the experiment may need.



Name	Begin date	End date
Preparing WiBed Firmware	12/05/14	14/05/14
Flash WiBed Firmware	14/05/14	14/05/14
Deploy the nodes	14/05/14	14/05/14
Maintain the network + Support	15/05/14	17/05/14
Pick up the Nodes	18/05/14	18/05/14
Prepare Experiment 1	12/05/14	14/05/14
Experimentation + Results gathering (1)	15/05/14	17/05/14
Prepare Experiment 2	12/05/14	14/05/14
Experimentation + Results gathering (2)	15/05/14	17/05/14
Prepare Experiment 3	12/05/14	14/05/14
Experimentation + Results gathering (3)	15/05/14	17/05/14
Prepare Experiment 4	12/05/14	14/05/14
Experimentation + Results gathering (4)	15/05/14	17/05/14
Prepare Experiment 5	12/05/14	14/05/14
Experimentation + Results gathering (5)	15/05/14	17/05/14
WBM results presentation	18/05/14	18/05/14

Figure 5.2: Gantt of WBM with the WiBed Project

WiBed was presented during this year's Battlemesh so experimenters were there first introduced to the platform and they did not have the necessary knowledge regarding its usage and operation. This also meant for them to have training in the experiment's operation so first experiments were slower than they really could be. As can be seen in Figure 5.2, three days were needed to prepare and adapt the firmware to fit in the event's needs[3]. This time was mainly needed to fix some bugs and to add improvements that were previously scheduled as we knew that in the participants list, main developers or experts in the topic were inscribed to the event. Thus, allowed us to arrange meetings with them and discuss a solution to improve the platform behaviour. To summarize, the most important points that delayed the deployment were:

[2]A bigger image of the figure 5.1 can be seen in Appendix B.2.
[3]A bigger image of the figure 5.2 can be seen in Appendix B.3.

- WiBed firmware had some bugs related with the region domain and the driver of the USB 2.4GHz Radio. These problems reduced network behaviour (not having the possibility of configuring some network aspects like radio's tx-power or the region).
- Automated firmware upgrading tool was outdated and insufficiently tested due to the bad connectivity of initial UPC CN-A network. In the UPC initial network nodes have not enough coverage so that made impossible to download files of about 10MB, consequently, downloading the firmware was not possible in such conditions. In the WBM (on Tuesday) we solved and remake this functionality.
- The WiBed platform was configured to be developed and used in the UPC by project team. When bringing it to the WBM, we needed not only to adapt the configuration to the spot needs but also allow WBM researchers to improve the platform (bringing WiBed to GitHub repositories).

Keeping that information in mind and using public code repositories (seen in section 5.1.3), the time and resources spent to prepare the Battlemesh will be shorten to just one day fitting the firmware to the emplacement needs and deploy the nodes and then spend the rest of the Battlemesh performing experiments and just needing only to monitor and maintain the network's behaviour during these experiments.

5.1.2 Minor Battlemesh experiments

In this section the different minor experiments performed during the Battlemesh will be explained. They were proposed to be performed during the Battlemesh and, even while some teams were working successfully with them, at the moment of finish this dissertation there are no uploaded results or conclusions yet.

The Hidden Node Problem

As one of the most well-known problems in wireless networks, the hidden node problem consists on avoiding collissions when transmitting data on shared environments.

The test was planned to be performed in 4 UBNT PicostationM2[4] using airMAX proprietary TDMA solution against the OpenWRT RTC/CTS implementation in 4 TL-WDR4300 devices.

[4] `http://www.ubnt.com/picostation`

The Mesh of Death Adversity

In a scenario where mesh nodes are really close to each other, as close as all of them see each other in good conditions, dynamic routing protocol algorithms that choose the path to transport the packages may suffer lots of modifications due to the fluctuation of the controlled metrics and the congestion of the network because of the flooding technique.

The test was planned to be performed in 7 TL-WDR4300 devices in a closed room and with 5 meters of distance between each node. The second requirement could not be achieved because the room was about 5x5 meters wide so the nodes were closer than the requirements.

The Convergence Time Relay

The scenario found in that experiment is a device L1 connected to a mesh network M1 that connects with a second device L2. Later, L1 connects to a new mesh network M2 that also connects with the device L2. M2 has better metrics to connect L1 to L2 so, while this mesh appears, the path from L1 to L2 must switch from M1 to M2. The goal of this experiment is to analyse how long it takes the routing protocol to change its path.

The test was planned to be performed with 20 TL-WDR4300 and 2 laptops. The protocols used are BMX6, OLSR, OLSRv2 and batman-adv.

5.1.3 Adapting the WiBed platform

As the general idea of how WiBed improved and helped the Battlemesh has been already explained in previous section, this section will explain the concrete changes and improvements that have been performed in order to face the problems and accommodate experiments like the ones described above.

Changes in the firmware

Main changes in the WiBed firmware were focused on improving the stability and power of the platform as well as to involve the whole Battlemesh community to adopt it as an useful platform to experiment with. Here it is a list of the most important achieved points of this Battlemesh's edition:

- Creation of a WiBed firmware version in the last_trunk OpenWRT version and adapt it to the emplacement needs

  Upgrading WiBed firmware to the last source version of the OpenWRT was possible because WiBed had already a development branch, *testing*, with generated WiBed firmware images with the last version of the packages already tested, otherwise, there would not be any guarantee that the resulting image worked correctly. The use of the last version

of OpenWRT was necessary because some of the experimenters were also maintainers of OpenWRT and they needed the improvements and fixes found in the last version.

- Packages installed in the firmware were changed as well as the configuration settings of the compilation system.

  Due to the concrete needs of the Battlemesh, some concrete aspects of the configuration of the WiBed firmware were changed to fit in the requirements of the emplacement. As an example of these changes:

  Changing the 2.4GHz radio channel from 11th to 6th due to the high occupancy of the 11th channel.

  ```
  -  option channel2 '11'
  +  option channel2 '6'
  ```

  Once fixed the problem with the RegDomain, recover the maximum power transmission of the antenna.

  ```
  -  option txpower2 '22'
  +  option txpower2 '17'

  -  option txpower5 '14'
  -  option txpower5 '18'
  ```

  Add high Adhoc Multicast Rates

  ```
  +  option mrate '12000'
  ```

  Add an greater originator interval (less broadcast messages)

  ```
  +  x:set("batman-adv","bat0","orig_interval","5000")
  ```

- Add oficial repositories for OpenWRT Routing protocols, OLSRv2 specific repository and the new repository created for the Battlemesh experiments (see section 5.1.3).

  As some experimenters were package developers or routing protocols maintainers, they were asked for the version of their software to be used in the experiments. For instance, OLSR main developer asked for using its newest OLSRv2 instead of official version. Furthermore, WiBed management network had a bug regarding that, under concrete circumstances, the wifi interface was not restarted and was keeped down. WiBed was using an older version of the routing protocol for that reason and, once the bug was fixed during the Battlemesh, the new version was used.

- Update and apply patch to fix Reghack[5] package wrong behaviour.

  As it was an urgent scheduled issue that was lacking the WiBed network performance, in the Battlemesh WiBed team could met one of the maintainers of OpenWRT and also the developer of the Reghack tool. After some patches, the problem found in the RegDomain using Reghack utility was fixed. Furthermore, another RegDomain related issue regarding the USB Radio used in the WiBed platform was discussed and, as explained in the section 4.1.1, it was decided not to put more efforts in this problem and better put them in improving network's connectivity in 5GHz band owing to be able to use the 5GHz band as the management network.

- Improve controller's representation of the nodes.

  WiBed controller is able to bring service to large-scale testbeds as well as to allocate a big amount of them. That brings the controller a problem regarding the distribution of that information and how is presented to the user. This topic was widely discussed during the Battlemesh and, as a provisional solution of that, the controller is currently showing each node's testbed.

  Add descriptive information regarding the location and testbed of the node and send it to the controller in the registration pull request (INIT state).

```
Configuration:
+  config wibed 'location'
+    option testbed 'Wibed-UPC'# <string> Testbed description.
+    option building 'C6'# <string> Building short information
+    option floor 'E1'# <string> Floor short information
+    option room '104'# <string> Room short information

Pull system script:

+  testbed = readVariable("location.testbed")
+  is_gw = readVariable("management.is_gw")
[...]
+  request["testbed"] = testbed
+  request["gateway"] = is_gw
```

- Rewrite the wibed-upgrade script.

  As further described in section 3.6.6, the wibed-upgrade script was not working correctly and it was rewriten.

[5]Reghack is a regulatory domain modifier for OpenWRT. `http://luci.subsignal.org/~jow/reghack/README.txt`

Changes in the minimal requirements and new supported nodes

Battlemesh experimenters gave feedback not only in the firmware related aspects, but also in the hardware topic. The most important point to improve was to reduce the minimal requirements owing to be able to add less powerfull devices. This, was widely discussed in the Battlemesh and the resolution was to accept these changes and allow new devices already shown in section 3.1.3. Another accepted point was to add official support for arquitectures *mpc85xx* and *x86*, which actually were already being tested but as a future work to be done. With these new arquitectures, the devices currently being tested (Alix2d2[6] and OpenWRT4900[7]) will receive support and increase experimentation possibilities. As a summarization, the most diversity the testbed has, the most close to real environment the testbed will be (as described in section 4.1.2).

The Battlemesh repository system

The first day of the Battlemesh, WiBed was presented and experimenters were asked to contribute with the project giving feedback and contributing in the platform's source. Willing to allow them to do that and as WiBed was made for improving the Battlemesh, all the sources in the Redmine[8] were added to the Battlemesh Git repositories[9]. These repositories are: wibed[10], wibed-openwrt-packages[11] and wibed-packages[12].

Finally, one more repository was added owing to store all the code regarding the main experiment of the Battlemesh that will be explained in the section 5.2:

- wibed-battlemesh-experiment[13]: All the protocols, configurations and settings of the tools used in the experiments are in this repository. Moreover, it has the necessary scripts to manage and execute the experiment. Compiling this repository contents, the experimenters will obtain a package that will join the protocols packages to be zipped in a .tar.gz as the overlay content.

5.2 The Battle of the mesh experiment

One of the main goals of the Battlemesh is to face the different networking protocols owing to see how they have improved during the past year and

[6] Alix2d2: `http://www.pcengines.ch/alix2d2.htm`
[7] OpenWRT4900: `http://wiki.openwrt.org/toh/tp-link/tl-wdr4900`
[8] `http://redmine.confine-project.eu/projects/wibed`
[9] `https://github.com/battlemesh`
[10] `https://github.com/battlemesh/wibed`
[11] `https://github.com/battlemesh/wibed-openwrt-packages`
[12] `https://github.com/battlemesh/wibed-packages`
[13] `https://github.com/battlemesh/wibed-battlemesh-experiment`

raise the one with the better behaviour and performance as the Battlemesh winner. Furthermore, this Battlemesh is the first one that gathered network data as well as added resource's consumption[14]. At the end, most of the years there is no protocol winner but, at the same time, it allows to see how much the protocols improved in a healthy competition. This section explains all the background information, the experiment itself and, finally, the results' interpretation.

5.2.1 Battlemesh overlay contents

The experiment of the Battlemesh[15] repository contains all the necessary components to perform the experiment. The overlay contents are scripts and configurations for all the involved protocols and the pre-compiled packages of the protocols. The scripts used are:

1. wbm-manage: This script allows the experimenters to set up the protocols and initialize or finish them.
2. wbm-test: This script starts the experiment[16].
3. wbm-config: This is the same wibed-config script with new settings related with the protocols added.

5.2.2 Environment considerations

As it has been seen in the previous section 4.2, the testbed environment has some particularities that distances this experiment of the ones that somewho could perform in a laboratory or precast and fixed testbed. The high density of wireless interferences, seen in the section 5.1.3, forced us to change, not only wireless propagation aspects, but also other ones related to the routing protocol like the interval of non-user traffic packages.

To sum up, the main aspects to take into consideration when checking the results are:

- The environment was highly charged with interferences of non-experiment related issuers. These interferences were in 2.4GHz as well as in the 5GHz band.
- Most of this experiment repetitions (at least 20 repetitions of it), used 16 nodes.
- The experiment was performed in the 5GHz band (140th channel).

[14]CPU and memory consumption

[15]Battlemesh experiment source: `https://github.com/battlemesh/wibed-battlemesh-experiment`

[16]In the Appendix C.1 its code is shown

- The environment could be equated to city conditions, although the distance between the transmitters was lower.

5.2.3 Protocol related considerations

Keeping in mind all the issues regarding the environment, to ensure the reliability and equality of all the protocols against this hard conditions, all the protocols ran in parallel as well as they are monitored in parallel too (overheads and anomalous behaviour are monitored for all the protocols). With such conditions, the protocols face the same environment behaviour and *battles* in a fairer way. Furthermore, the overhead caused by the monitoring tools is dismissed because the traffic and resource consumption generated is tiny.

Finally, more focused in the protocols' configuration:

- All the protocols will work using IPv6 addresses.
- Each protocol will have its own ULA[17] that will identify it when checking the results and monitoring logs.
- The protocols' are setted by default and no changes or improvements are done in their configuration.
- To prevent any topology change or any update related with the start up of the network, a previous to the experiment idle time of 200 seconds is set.
- After an experiment is performed and, for the same reasons that previous point, another 200 seconds are waited until the next experiment could be started.
- The experiments will consist on source to destination actions. This will help to understand how the protocol works in this scenario.

Unforeseen issues:

- Babel configuration was not correctly configured (more settings that the default ones were needed) so the results were not correctly gathered. For that reason, Babel is not present in the results and conclusions of the experiment.
- Batman-adv is a Kernel module working in Kernel space. For that reason, it is harder for the default utilities to monitor its resource consumption.

[17]ULA: Unique Local Address

Well-known Batman-Adv particularities:

As a L2[18] routing protocol, Batman-adv required special rules for its traffic monitoring. Moreover, when gathering the information of the hops required to reach the destination, the behaviour of this data link protocol is to use just one hop, as they are interconnected to all other nodes through the virtual switch created.

5.2.4 Configuring the experiment

Once the assumptions of the environment and state of the protocol are known, the next step is to understand the tools and how they are used in the experiment:

Network monitoring

- tcpdump: With this tool the overhead created by each protocol during all the experiment is executed. That means that this tool will be working all the time[19].

Network tools

- mtr: This tool tracks the source-destiny path information in an internval of 1 second. The results obtained with this tool were too complex and hard to be represented, so the results are available in the Battlemesh repositories but not used.
- ping6[20]: This tool sends 1.000 bytes of icmp packages (user data packages) from source to destination in a 1 second interval. Ping6 measures are the RTT[21], the packages lost and the hops to destination count.
- netperf: This tool is the most agressive among the used. Its operation consists on checking the maximum network bandwidth between the source and destination nodes. As this tool consumes as bandwith as possible, its operation is not done parallely but in this manner:
  - 4 rounds executing **sequentially** the netperf tool for each protocol.
  - Each execution lasts 10 seconds.
  - The duration of all the process is:

$$4procotols \times 4rounds \times 10seconds = 160seconds.$$

Resources monitoring

[18] L2: the data link OSI level layer. Its units are data frames.

[19] Experiment's time: 200 seconds.

[20] Ping6 is the adapted for IPv6 version of the ping utility.

[21] Round-trip time: is the time needed for a package to go to its destination and return the acknowledgement to the source.

- top: This tool is used for monitoring resources consumption (CPU and memory) for each protocol. Its information gathering has an interval of 1 second.

5.2.5 Considered scenarios

The experiments will have static and mobile scenarios focused on checking protocols' overhead and performance. These experiments were repeated about 22 times and the first 5 of them were used to test and improve the experiments, so its results are not used in the conclusions. Finally, the protocols used were BMX6, Batman-Adv, OLSR and OLSRv2 (experimental).

Static scenario

In this scenario the nodes were not moved. The nodes selected to be the source and destination were wisely choosen owing not to use neighbours and being representative. The experiments were sequentially executed by pairs of nodes and successed until the delay time.

Mobile scenario

In this scenario the conditions are the same as in the static scenario exceot from that there is one mobile node[22] acting as destination node for different source nodes. The mobile node is carried by one experimenter[23] carring the mobile node and walking slowly (about 1 meter/second) from the main hall, going down to the lower floor, walk through the lower floor and then return to the initial point (walking during the 200 seconds that the experiment lasts).

5.2.6 Experiments' results and conclusions

This section explain the results in different subsections. The results shown in these sections comes from the experiments **16** and **19**. They have been selected because they suit as a representative results of the common behaviour seen in the whole experiments and present the tendence used to take the experiment's conclusions.

Resource consumption and overhead

The figure 5.3 shows in continuous lines the protocols' memory consumption limit.

One of the most significative point of resources results is that the protocols never changed its memory limit and, moreover, as can be seen in figure

[22]Node c24174 + 21.000mAh Power Gorilla battery

[23]The same experimenter done all the routes.

5.3. These limits are close to be a 2% of the total memory available in the devices[24]. To sum up, we can see that the most consumer protocol is OLSRv2 with about 1.7MB of RAM.

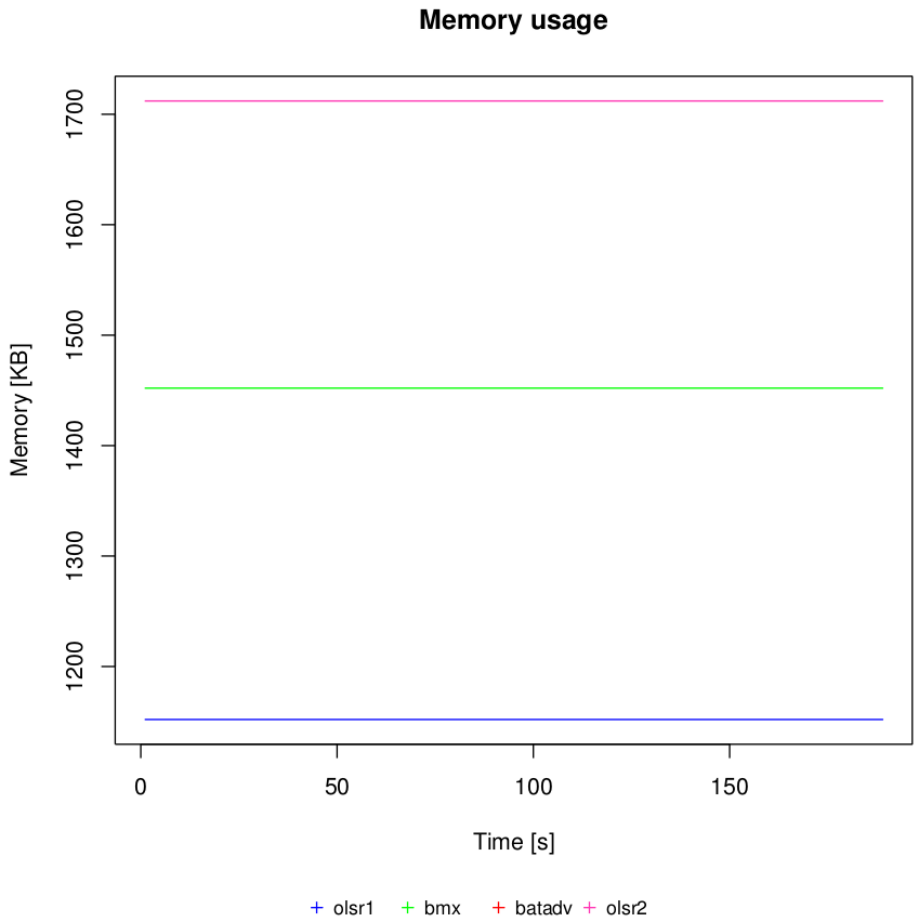


Figure 5.3: Protocols memory consumption

The figure 5.4 shows in continuous lines the CPU consumption of each protocol calculated each 1 second interval. The dashed line shows the average CPU consumption over all the experiment.

In the CPU resource consumption we can see that most of the protocols uses from 0 to 5% of the CPU capacity. That fact guides us to the other significative point that can be seen in the resources consumption, the OLSRv2[25] anomalous behaviour: its CPU consumption varies from 2% to 20% and an average of 10%. That enormous consumption compared to the rest of protocols has a reason:

[24]TL-WDR4300: 128MB of RAM.
[25]OLSRv2 is in an early experimental state.

- The WiBed platform is OLSRv2 first real and diverse testbed environment test.
- The protocol has debugging and logging options enabled that reduced its performance as well as increased its overhead.
- It is a new branch of OLSR being under active development and it is far from being a stable version.

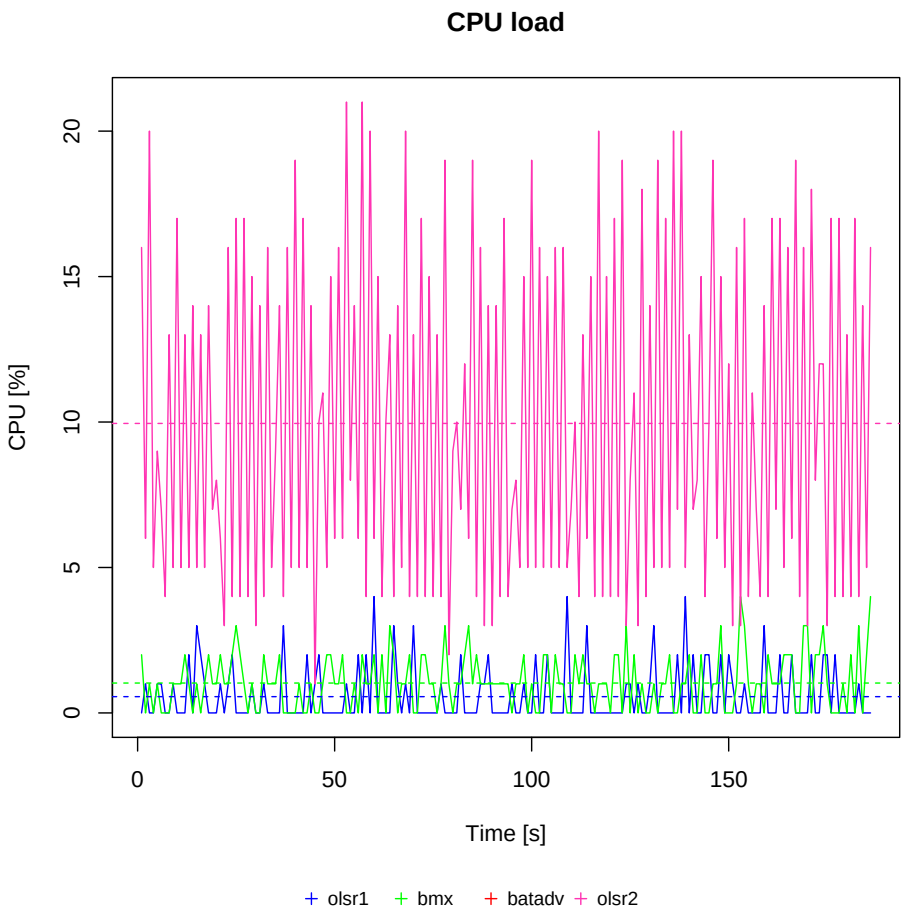


Figure 5.4: Protocols CPU consumption

Traffic protocol-related overhead

The figure 5.5 shows the network overhead in bytes per second (B/s) of each protocol in bytes per second calculated over all the experiment in 1 second intervals. The dashed lines represents the average overhead during

all the experiment. The second figure 5.6, shows also the network overhead but, this time, counting the overhead generated in Packets per second (P/s). The continuous lines shows the number of packets/second calculed in 1 second intervals and the dashed lines the average packets/second all over the experiment.

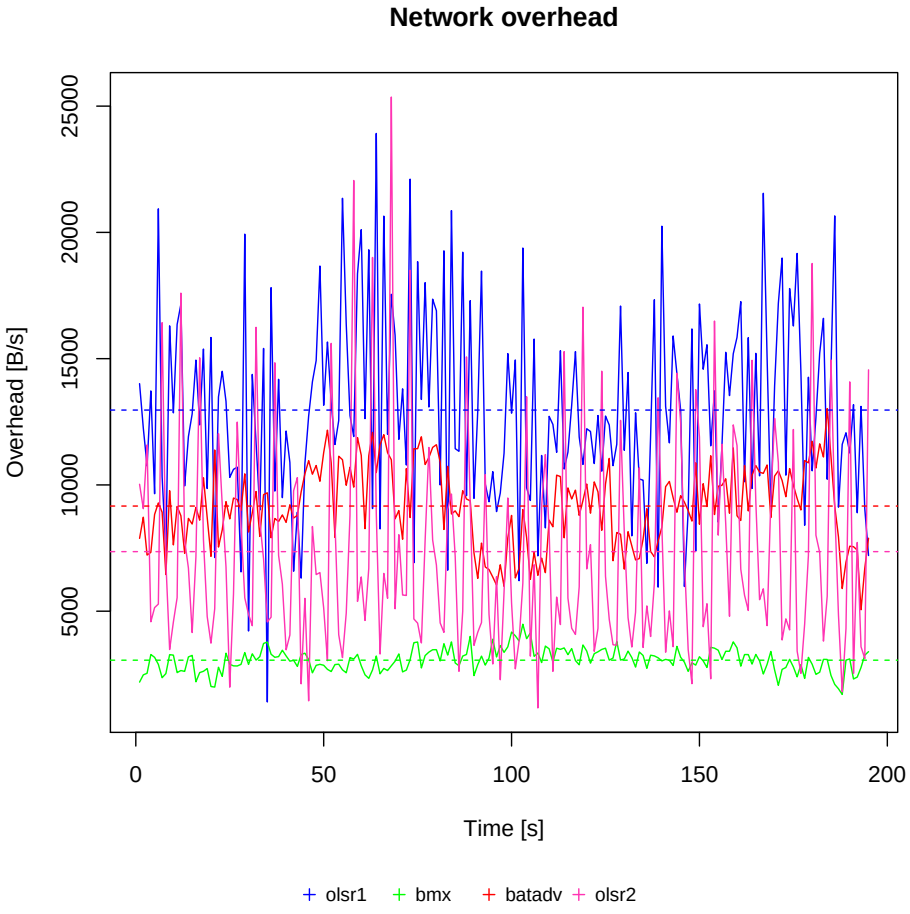


Figure 5.5: Protocol-related overhead in B/s

To understand the results of this section, it is important too keep in mind that the traffic being calculated here is the protocol related and not the user one. That means that data transferences between the nodes are not included as they will be explained in the *TCP throughput* section.

As can be seen in figure 5.5, BMX6 is showing the most stable and with less overhead in B/s behaviour among the other protocols. The rest of them are quite more unstable in overhead but with a really close average overhead of about 1 KB/s. In figure 5.6 we can see that Batman-adv is the most unstable and with more overhead with an average of 120 P/s and the rest

is showing an stable behaviour with a low overhead of about 20 P/s so, in general terms, the difference between the protocols is really small.

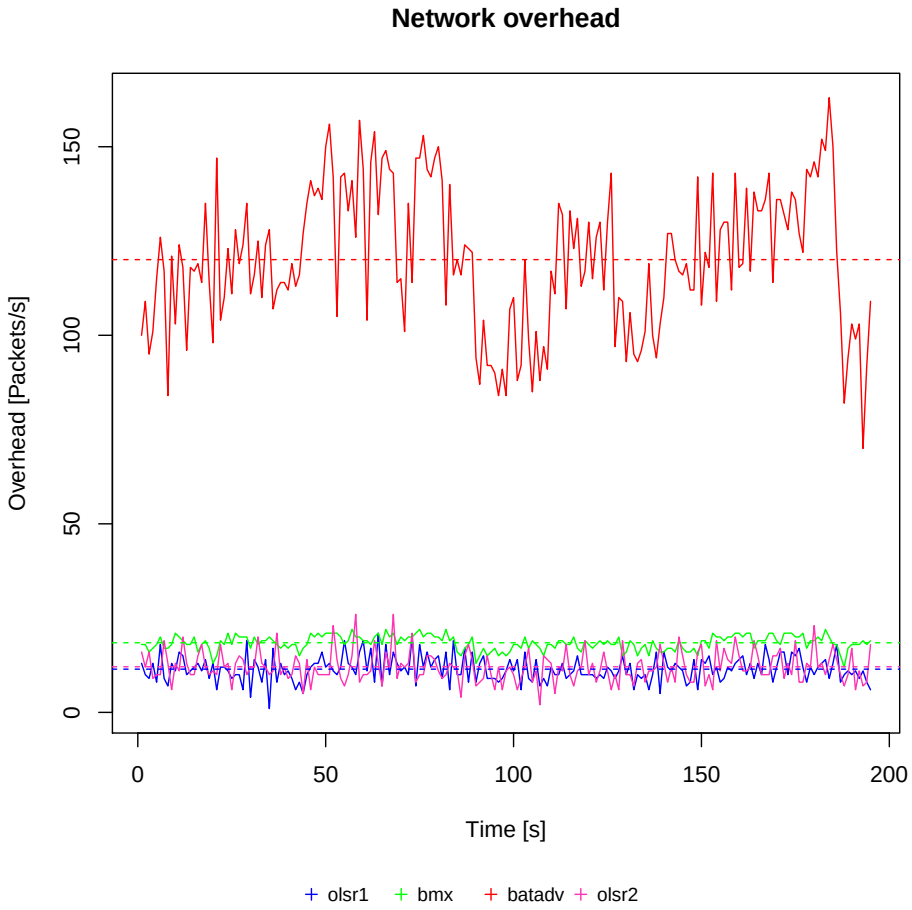


Figure 5.6: Protocol-related overhead in P/s

Once we join both figures we can see that, with a similar overhead of 3.000 B/s, OLSR/v2 have an overhead of 3 5 P/s against the 40 P/s of Batman-adv. The conclusion extracted from this information is that, while they obtain an equal B/s throughput, its difference remains in the size of the traffic packages that they send. Batman-adv is sending smaller packages continuously and OLSR/v2 is sending bigger packages but less often.

Network-related behaviour: RTTS, hops and delivery rate

In this section it is included all the network-related metrics of the experiment regarding the user and not the protocol. Four figures showing the network information in different manners will be explained in this section and then,

a final combined conclusion extracted of the experiment results.

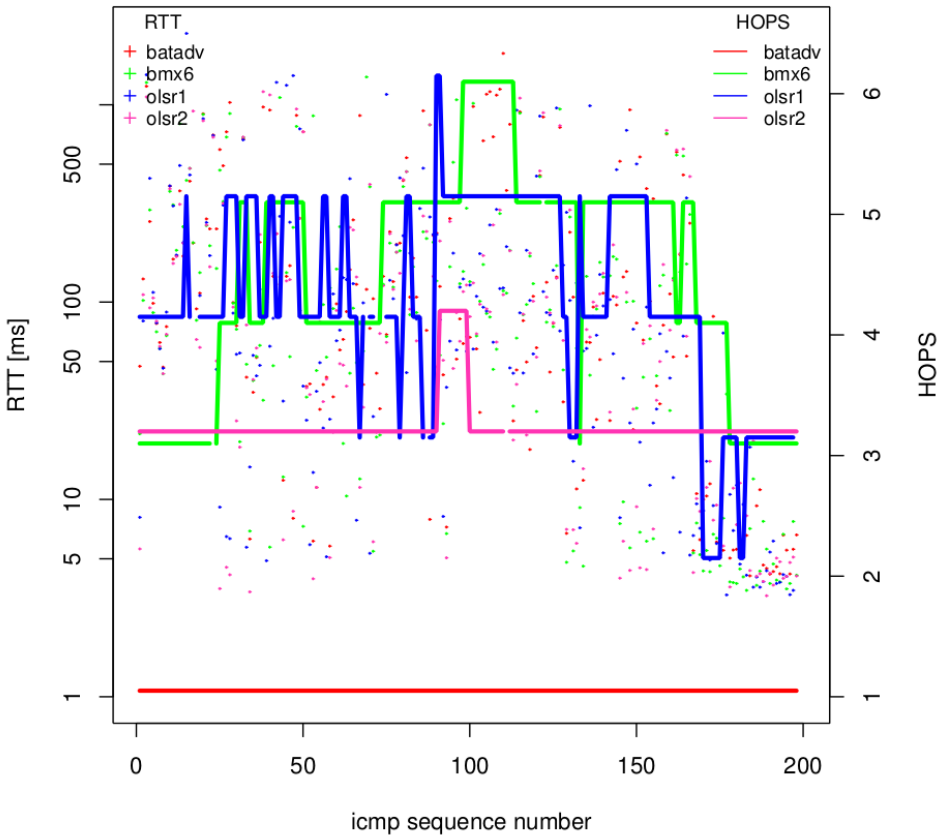


Figure 5.7: Number of hops and RTTs

Figure 5.7 shows the number of hops from source to destination during the experiment in a continuous line, as well as the RTT[26] time in miliseconds of each protocol as coloured crosses during all the experiment time.

Figure 5.8 shows the occurance and the RTT in miliseconds according the hops of the source to destination path. The coloured bar shows the average RTT time and the coloured crosses the latency of each occurance for each protocol and hop number.

Figure 5.9 shows the ECDF[27] of the success paths and its RTT. The coloured crosses show the average distribution of the RTT and the countinuous line its tendency.

To understand the conclusions extracted, it is needed to know some particularities of the results obtained. First of all, the RTT is the time needed to achieve a path and return to the source with the acknowledgement that it

[26]RTT: Round Trip Time.
[27]Empyrical Cummulative Distribution Function

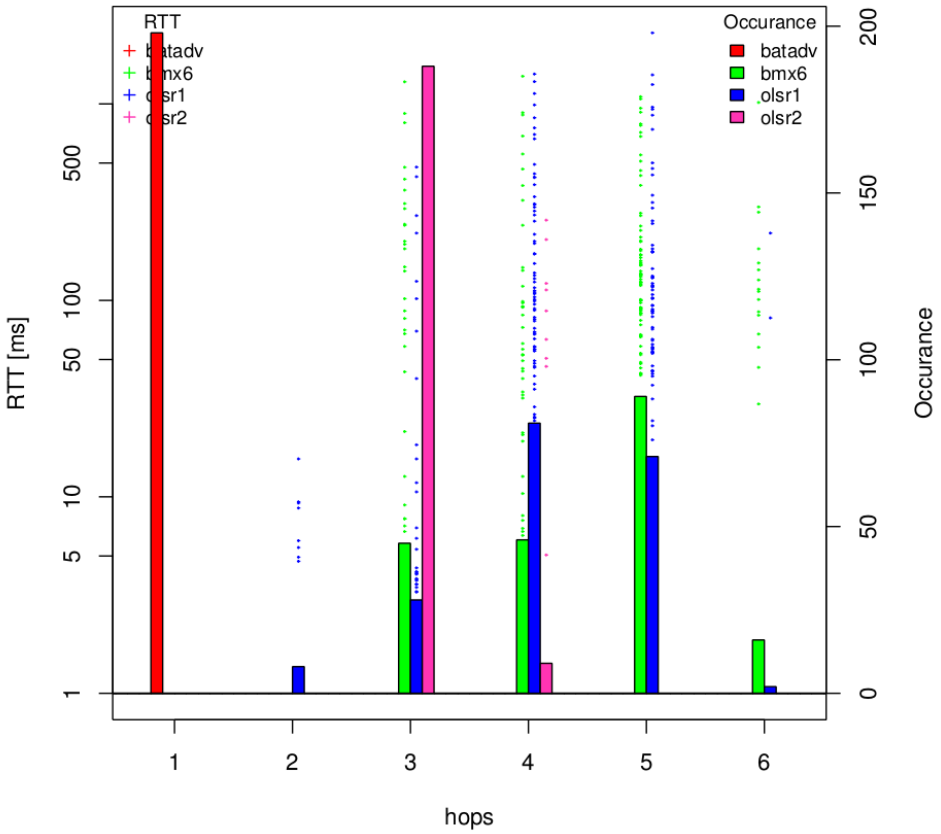


Figure 5.8: Path RTTs and occurances

is a valid path. Second, the number of hops of the Batman-adv **is correct** as is a L2 routing protocol connected by a virtual switch, so the tools used in the experiment will see their paths as if they were directly connected (1 hop). And third, that the fact BMX6 and OLSR fluctuates more than OLSRv2 is a tendency shown in all the experiments so is also a correct behaviour.

The conclusions that we have seen are:

1. The RTT values grow according the number of hops needed to achieve the destination, so no anomalous behaviour were observed.

2. The tendency of BMX6 and OLSR number of hops inestability seems to have relation with the interferences generated by the netperf execution (160 seconds of high data transmission) in paralel with the other tools' execution. The two protocols seem to apply shorter and stronger links instead of far and weaker ones. By the other way, OLSRv2 is more stable increasing its hops' number only under critic circumstances and Batman-adv hops number is not ilustrative.

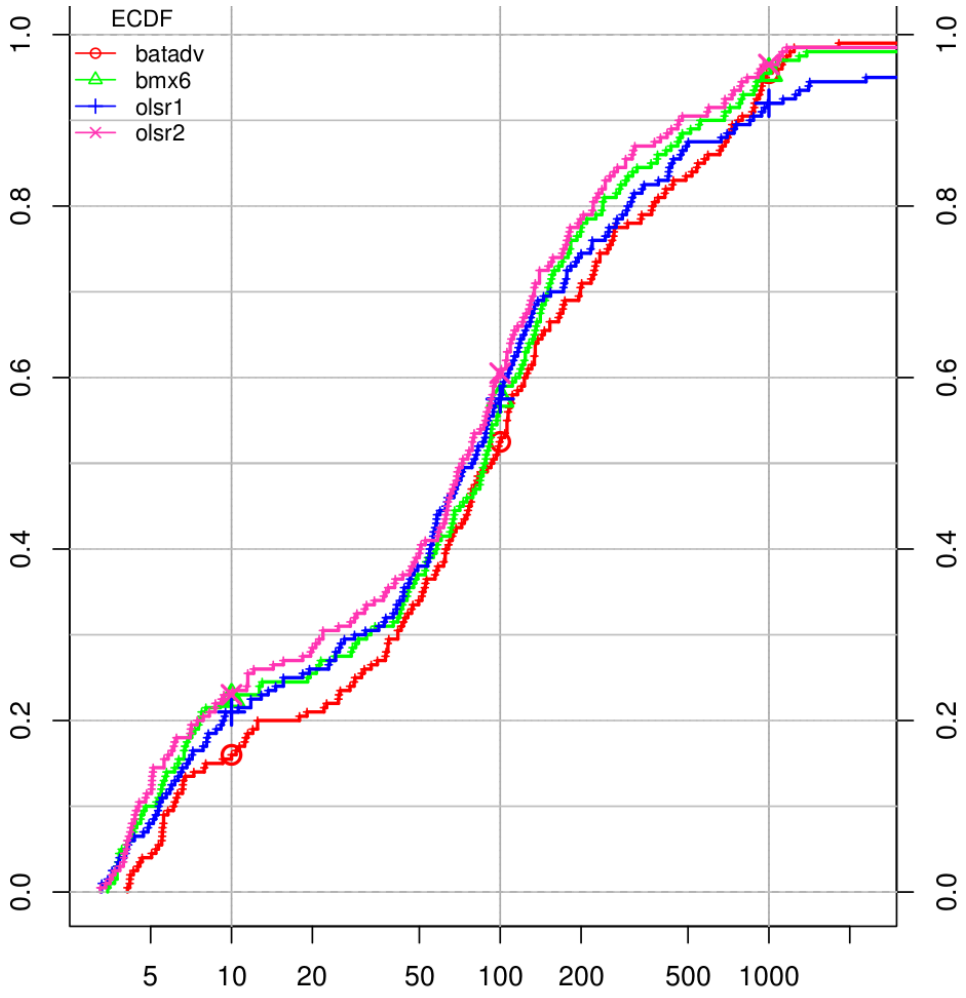


Figure 5.9: Ping sucess rates

3. According to the figure 5.9, in the interval from 10ms to 100ms, the ping time that is interesting for our conclusions, the most successful protocol is OLSRv2 followed by BMX6 and OLSR. In most of the experiments performed the performance of the three protocols is similar and higher than in Batman-adv.

TCP Throughput

This section shows the results of the netperf executions using a good example and also describing the behaviour of other interesting experiments that are not shown.

Figure 5.10 shows the throughput of the protocols in each round. Each bar is the number of KiloBytes per second (KB/s) showing the maximum

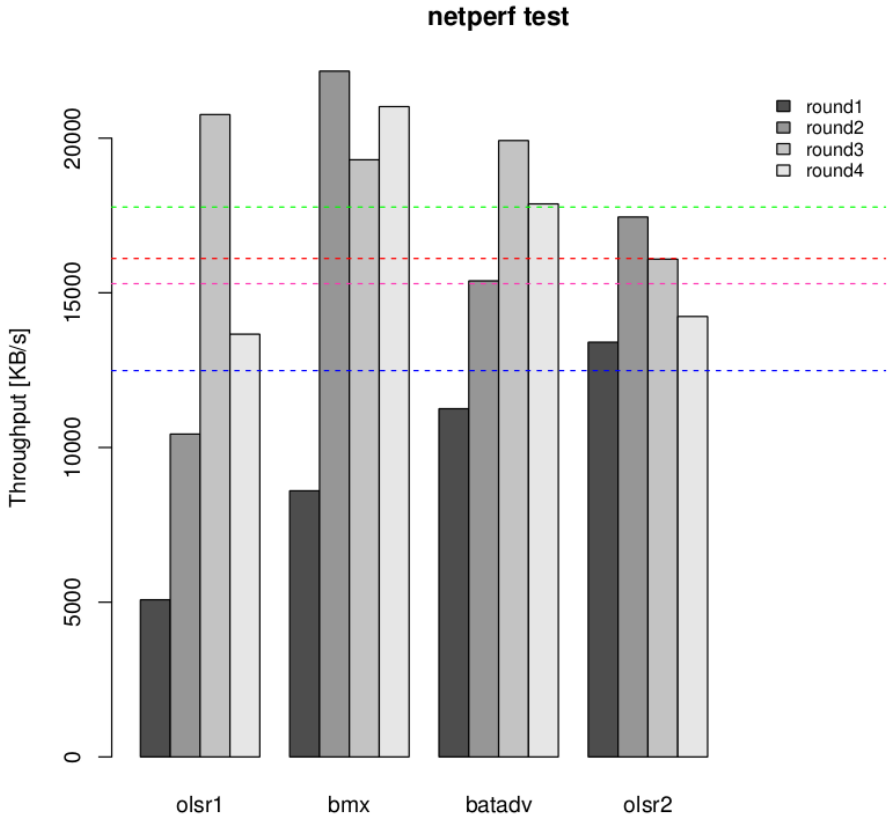


Figure 5.10: Netperf throughput rounds

bandwith obtained and the slashed line the average in the 4 rounds.

The conclusions extracted of this figure are that the intelligence behind the routing of the protocol is really important as can be seen in the maximum value of each round: in the first round all the protocols obtain low KB/s capacity, and this is related with the fact that the network was in an stable state and not expecting a high traffic charge; in the next two rounds the protocols adapts its routes to this new congested situation adopting stronger and nearer links and increasing the value; finally, in the last round can be seen that most of the protocols' throughput decreases a bit but showing that the tendence is a sign of stabilization of the situation and not a deterioration of the network capacity.

These conclusions have been contrasted with the other throughput results and show that the tendence is to shook the network in the first instance and to be stabilized by itself thanks to protocols' intelligence. It is important to take into account that the OLSRv2 is under development and unstable yet as it was found that in some experiments, its performance was about 60%

worse than the rest of the protocols.

Personal experiment conclusions

In the Battlemesh the results were presented as *nobody won, we can only see tendencies, but every protocol improved since last Battlemesh.* As a closed and isolated experiment for this dissertation, my conclusions are:

1. There are two clear winners, not only in resource consumption, but also in network performance and adaptability to environment circumstances: BMX6 and OLSR.

2. Batman-adv competes with routing protocols working in a different layer, so maybe it is not exactly fair to diminish its great potential (for that reason we are using this protocol as our management network).

3. OLSRv2 is not yet as stable as it will be in the next Battlemesh so, having seen its high potential in its early development state, the other protocols will need to improve seriously owing not to be overcome by it.

4. The resource consumption of the protocols shows that even 4MB of flash and 16MB of RAM devices are able to work in medium-scale mesh networks without real resource problems.

5.3 Validation of the platform

The validation of the platform is one of the pillars of this project viability as it attests that all the work done together with the WiBed team has born fruits or if the project needs some adjustments to fulfil the requirements. During the early project's development, our *fixation* was to validate the network owing to have a strong and reliable network in the Campus Nord for researchers. As our main goal, following the philosophy and methodology of the CONFINE project, was to develop a standardised project, this *fixation* changed to want to validate the platform, prepare it for as scenarios and environments as possible and also bringing a stable network for the researchers in the University. This motivated the changes applied to fit even more devices and scenarios to the testbed, see section 3.1.3. Even more, the hole behaviour was quite unstable at the beginning of the development and it has became a trustful and capable platform to work with.

Initially, the validation process was thought to be a series of experiments reproducible and comparable in other real environments to join these results and finally see if the platform was trustworthy and stable or not. This began to change as the Battlemesh approached and the use of the Battlemesh main experiment took force. Finally, after being in the Battlemesh, with the

results already analysed and the acceptance of the Battlemesh experimenters and community to continue using[28] and improving the project[29], WiBed has became the platform we expected to. Moreover, during all the platform's development, INESC[30] workgroup was performing experiments to test and improve its networking tools.

Though, there are yet lot of work to be done, new features to add and to bring support for new devices and architectures so, WiBed will not stop its development now that this project finishes, but starts its real journey as a standard, versatile, cheap and pedagogic platform for network's studies.

[28]The WBMv8 will use WiBed as the main experimentation platform.

[29]WiBed Battlemesh repositories currently have 6 active contributors.

[30]INESC TEC: Laboratório Associado de Tecnologia e Ciência a la Universidade do Porto.

Chapter 6

Conclusions

In this document the WiBed platform development and UPC CN-A testbed mesh network deployment have been presented. Although it started as a platform in early development state, WiBed has improved and has became a stable, reliable and accepted to be used in experimentation platform. Furthermore, the presentation of WiBed to the Wireless Battle of the Mesh event, helped it to improve even more and fixed some known issues thanks to the community efforts. The Battlemesh also got benefit from the platform as experimenters could perform its experiments faster and easier. Moreover, thanks to the Battlemesh experiments and results analysis, we state that the platform is trustworthy and stable enough to state that it is a valid research tool. Finally, even the UPC CN-A network's deployment is not finished yet, it is expected to complete the deployment in the next weeks and the prototype network deployed in the buildings A5 and A6 has been successfully performing experiments during the project's development.

In a more personal approach, this project helped me to grow personally and professionally. Being together with network communities, its contributors and its environment gave me a priceless knowledge and the personal satisfaction of having been part of the project and the community, and having contributed to it. In addition, I will continue supporting the WiBed platform strongly during the 2 more years that the CONFINE project will be supporting the UPC CN-A network. And finally, being able to develop WiBed made me realise that I am exited about continuing working in open source projects.

6.1 Future work

At the end of this project, the platform has been proved to be stable and reliable enough to perform big experiments. However, improving even more its stability, adding new features and, still more important, adding new devices to experiment with is the future work that WiBed will need in order

to achieve a step forward.

1. Current firmware is compiled with a concrete Kernel version and, to change it, the researcher has to reflash the node. This forces the developers to adapt their tools and packages to that version in order to make it compatible. As this *issue* is not meant to be solved soon, the documentation and support given to experimenters must emphasize it.

2. The management configuration can be modified so this could interrupt the communication with the controller and provoke the node to go back to its original state. It is needed to protect the management network default configuration and, furthermore, to protect as well the rest of the critical settings of the nodes and the platform.

3. Start giving support to the architectures *mpc85xx* and *x86* as a previous step for enlarging UPC CN-A network to other campus buildings. Concretely, using the WDR4300 as the main experimentation nodes, the WDR4900 as the gateway and the Alix2d2 as the bridge that will connect between building roofs.

4. Continue working together with the Battlemesh community to improve even more the platform and be prepared for the new challenges of the next Battlemesh.

5. Some experimenters need more direct access to the nodes. Currently they can only work using the controller, so adding a VPN connection to grant them access to the nodes' terminal is already being developed.

Appendices

Appendix A

Off-the-shelf Wireless Networks Research Testbed

This chapter aims to show the reader the main characteristics presented in WiBed theoretical dissertation[1]. The content of this chapter is taken directly from the original dissertation of Roger Baig with no modifications.

A.1 WiBed Software

A.1.1 Testbed server

The server's software solution must:

- accept node identifiers announcements
- make specific announcement for each node
- implement the following announcements:
  - overlay upgrade (a complete filesystem is available to be installed)
  - base system upgrade (a complete filesystem is available to be installed)
  - command line execution (a command line is available to be executed)
  - timeframes for wireless experiments
- specify the execution delay time for each announcement
- have a time server
- have shell access from the Internet

A.1.2 Testbed nodes

The nodes' software solution must:

- have a unique invariable identifier
- have a time client synchronised with the testbed server time server
- announce their identifier to the server during the boot process
- understand the following announcements:
  - overlay upgrade (check if a new overlay is available; if so, download it and install it in the overlay area)
  - base system upgrade (check if a new base system is available; if so, download it and install it in the base system area)
  - command line execution (a command line is available to be executed)
  - timeframes for wireless experiments

A.2 WiBed Hardware

A.2.1 Testbed server

The server's hardware solution must:

- have a reasonable amount of storage capacity to store nodes' firmware images

A.2.2 Testbed nodes

The nodes hardware solution must:

- be a reasonable cheap
- must have at least two Atheros independent NICs
- have an Ethernet port

A.3 WiBed collocation

A.3.1 Testbed server

The server's collocation place must:

- have internet access

A.3.2 Testbed nodes

The nodes' collocation places must:

- guarantee the server can be reached. Thus, there must be at least one node connected to the server and the rest must be reachable over the management network.

Appendix B

WiBed schedule

In this section, a bigger version of the schedule figures is provided.

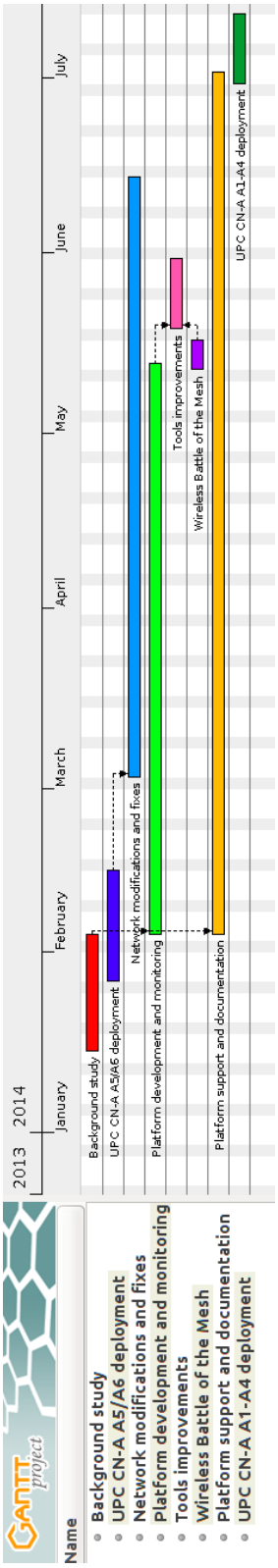


Figure B.1: Project's general planning.

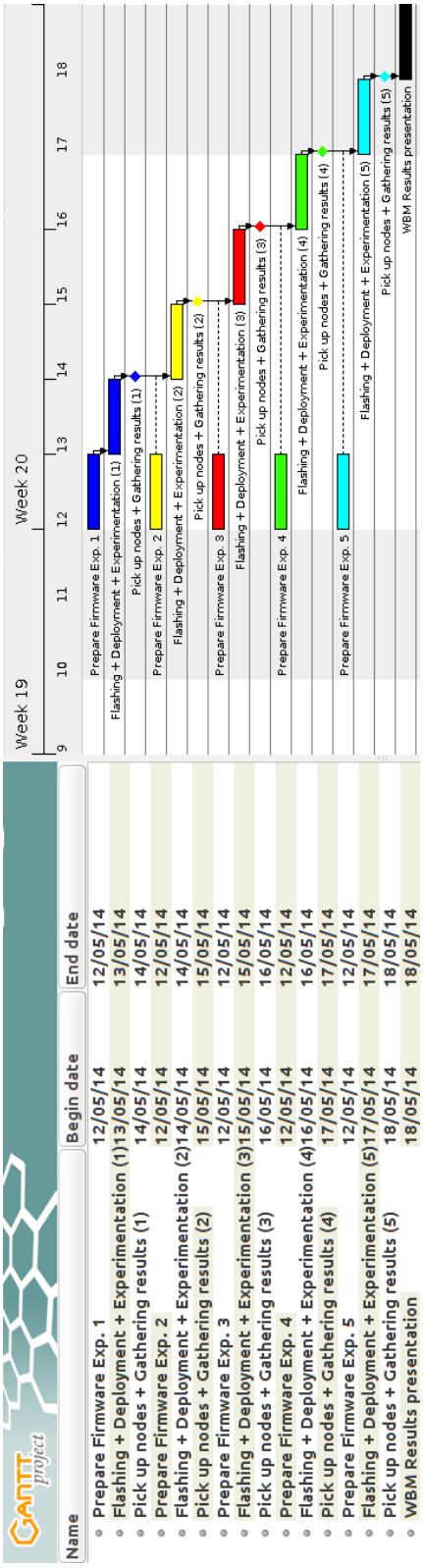


Figure B.2: Gantt of WBM without the WiBed Project

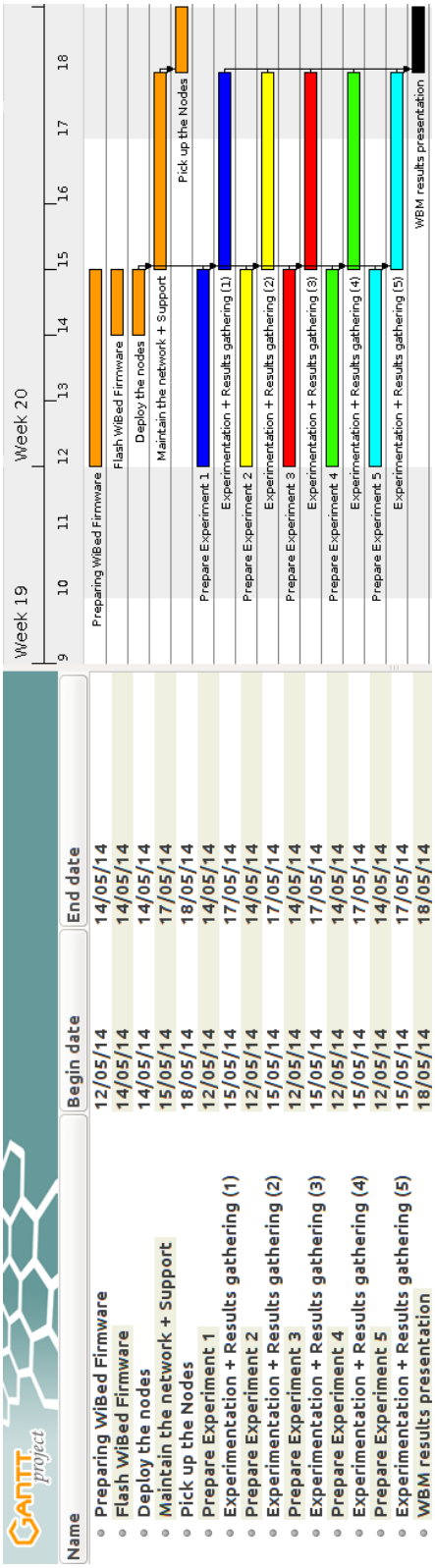


Figure B.3: Gantt of WBMv7 with the WiBed Project (1st time)

Appendix C

WBMv7 documents

This section provides the contents of the overlay of the Battlemesh main experiment, the table with the information regarding Battlemesh editions and the map of the initial deployment.

C.1 Battlemesh main experiment code

Listing C.1: Battlemesh experiment code

```
#!/bin/sh

SRCNAMEID=${1:-$(cat /proc/sys/kernel/hostname | grep -o "....$"
    )}
HOSTNAMEID=$(cat /proc/sys/kernel/hostname | grep -o "....$")
[ "$SRCNAMEID" == "$HOSTNAMEID" ] || exit
set -x

shift
DSTNAMEID=${1:-4174}

shift
NETPERF=${1:-YES}

shift
PROTOS=${1:-olsr1_bmx_bat_babel_olsr2}
OLSR1=$(echo $PROTOS | grep olsr1)
BMX=$(echo $PROTOS | grep bmx)
BAT=$(echo $PROTOS | grep bat)
BABEL=$(echo $PROTOS | grep babel)
OLSR2=$(echo $PROTOS | grep olsr2)
PROTOCOLS_NUM=$(( $([ "$OLSR1" ] && echo 1 || echo 0) + $([ "
    $BMX" ] && echo 1 || echo 0) + $([ "$BAT" ] && echo 1 || echo
     0) + $([ "$BABEL" ] && echo 1 || echo 0) + $([ "$OLSR2" ] &&
     echo 1 || echo 0) ))

shift
OUTDIR=${1:-/save/wbm-axn}
```

```
shift
DURATION=${1:-200}
TCPDUMP_CAPTURE_SIZE=${5:-100}
TCPDUMP_CAPTURE_DEV=${5:-wbm1}

PING6_INTERVAL="1"
PING6_SIZE="400"

NETPERF_DURATION="10"

DSTADDRID=$DSTNAMEID #$(bmx6 -c originators | grep "wibed-..
    $DSTNAMEID" | awk '{print $3}' | awk -F: '{print $3}' )
OLSR1_V6="fdba:11:$DSTADDRID::1"
BMX_V6="fdba:12:$DSTADDRID::1"
BAT_V6="fdbb::$DSTADDRID"
BABEL_V6="fdba:14:$DSTADDRID::1"
OLSR2_V6="fdba:15:$DSTADDRID::1"

mkdir -p $OUTDIR
rm -f $OUTDIR/*

ps                    | tee $OUTDIR/ps-begin.log
bmx6 -c status interfaces links originators tunnels | tee
    $OUTDIR/bmx-cd8-begin.log
ip -4 route           | tee $OUTDIR/ip4routes-begin.log
ip -6 route           | tee $OUTDIR/ip6routes-begin.log
ip -6 route ls t 60   | tee $OUTDIR/ip6routes60-begin.log
ip addr               | tee $OUTDIR/ipaddr-begin.log

echo Starting top...
timeout $DURATION top -b -d1 >> $OUTDIR/top.log &

echo Starting tcpdump...
timeout $DURATION tcpdump -nve -i $TCPDUMP_CAPTURE_DEV -s
    $TCPDUMP_CAPTURE_SIZE -w $OUTDIR/tcpdump.raw port 6240 or
    port 698 or port 269 or port 6696 or \(ether proto 0x4305 and
     \( ether[14]==0x00 or ether[14]==0x44 \) \) 2>/dev/null &

echo Starting pings...
[ $OLSR1 ] && timeout --signal=INT $DURATION sh -c "while true;
    do ping6 -i $PING6_INTERVAL -s $PING6_SIZE -n $OLSR1_V6 >>
    $OUTDIR/ping6-olsr1.log  2>&1 && break || sleep 1; done" &
[ $BMX ]   && timeout --signal=INT $DURATION sh -c "while true;
    do ping6 -i $PING6_INTERVAL -s $PING6_SIZE -n $BMX_V6   >>
    $OUTDIR/ping6-bmx.log    2>&1 && break || sleep 1; done" &
[ $BAT ]   && timeout --signal=INT $DURATION sh -c "while true;
    do ping6 -i $PING6_INTERVAL -s $PING6_SIZE -n $BAT_V6   >>
    $OUTDIR/ping6-batadv.log 2>&1 && break || sleep 1; done" &
[ $BABEL ] && timeout --signal=INT $DURATION sh -c "while true;
    do ping6 -i $PING6_INTERVAL -s $PING6_SIZE -n $BABEL_V6 >>
    $OUTDIR/ping6-babel.log  2>&1 && break || sleep 1; done" &
[ $OLSR2 ] && timeout --signal=INT $DURATION sh -c "while true;
    do ping6 -i $PING6_INTERVAL -s $PING6_SIZE -n $OLSR2_V6 >>
```

```
    $OUTDIR/ping6-olsr2.log  2>&1 && break || sleep 1; done" &

echo Starting mtr...
[ $OLSR1 ] && timeout --signal=INT $DURATION sh -c "while true;
    do date +%s.%N; mtr -nt -6 -r -c 1 $OLSR1_V6;  sleep 1; done"
     >> $OUTDIR/mtr-olsr1.log &
[ $BMX ]   && timeout --signal=INT $DURATION sh -c "while true;
    do date +%s.%N; mtr -nt -6 -r -c 1 $BMX_V6;    sleep 1; done"
     >> $OUTDIR/mtr-bmx.log &
[ $BAT ]   && timeout --signal=INT $DURATION sh -c "while true;
    do date +%s.%N; mtr -nt -6 -r -c 1 $BAT_V6;    sleep 1; done"
     >> $OUTDIR/mtr-batadv.log &
[ $BABEL ] && timeout --signal=INT $DURATION sh -c "while true;
    do date +%s.%N; mtr -nt -6 -r -c 1 $BABEL_V6;  sleep 1; done"
     >> $OUTDIR/mtr-babel.log &
[ $OLSR2 ] && timeout --signal=INT $DURATION sh -c "while true;
    do date +%s.%N; mtr -nt -6 -r -c 1 $OLSR2_V6;  sleep 1; done"
     >> $OUTDIR/mtr-olsr2.log &

if [ "$NETPERF" == "YES" ]; then
    echo Starting netperfs...
    NETPERF_ROUNDS=$(( ( $DURATION / $PROTOCOLS_NUM /
        $NETPERF_DURATION ) ))
    NETPERF_COUNT=1
    while [ $NETPERF_COUNT -le $NETPERF_ROUNDS ]; do

        [ $OLSR1 ] && timeout --signal=INT $NETPERF_DURATION
            netperf -f k -c -C -D 1 -j -H $OLSR1_V6 > $OUTDIR/
            netperf-olsr1-$NETPERF_COUNT.log
        [ $BMX ]   && timeout --signal=INT $NETPERF_DURATION
            netperf -f k -c -C -D 1 -j -H $BMX_V6   > $OUTDIR/
            netperf-bmx-$NETPERF_COUNT.log
        [ $BAT ]   && timeout --signal=INT $NETPERF_DURATION
            netperf -f k -c -C -D 1 -j -H $BAT_V6   > $OUTDIR/
            netperf-batadv-$NETPERF_COUNT.log
        [ $BABEL ] && timeout --signal=INT $NETPERF_DURATION
            netperf -f k -c -C -D 1 -j -H $BABEL_V6 > $OUTDIR/
            netperf-babel-$NETPERF_COUNT.log
        [ $OLSR2 ] && timeout --signal=INT $NETPERF_DURATION
            netperf -f k -c -C -D 1 -j -H $OLSR2_V6 > $OUTDIR/
            netperf-olsr2-$NETPERF_COUNT.log

        NETPERF_COUNT=$(( $NETPERF_COUNT + 1 ));
    done
fi

set +x

wait

ps                  | tee $OUTDIR/ps-end.log
bmx6 -c status interfaces links originators tunnels | tee
    $OUTDIR/bmx-cd8-end.log
ip -4 route         | tee $OUTDIR/ip4routes-end.log
```

```
ip -6 route          | tee $OUTDIR/ip6routes-end.log
ip -6 route ls t 60  | tee $OUTDIR/ip6routes60-end.log
ip addr              | tee $OUTDIR/ipaddr-end.log

echo Finished!
```

C.2 Battlemesh versions table

Edition	Dates	Location	Participants[1]	Host	Remarks
v1	Apr,11-12, 2009	Paris, France	12	/tmp/lab	1^{st} edition; few devs gather together to see what happens
v2	Oct, 17-18, 2009	Brussels, Belgium	20	HackerSpace Brussels	Due to the success of 1^{st} ed. they repeat 6 month later
v3	Jun, 2-6, 2010	Bracciano, Italy	25	Ninux.org	Mass upgrade system. Spontaneous talks. 10^{th} nodes deployed.
v4	Mar, 16-20, 2011	Sant Bartomeu del Grau, Catalonia	56	guifi.net	First one week long. Agenda pre-arranged. Results made available.
v5	Mar 23-Apr 1, 2012	Athens, Greece	60	AWMN	OpenWRT devs join the WBM. Warm up event.
v6	Apr, 12-21, 2013	Aalborg, Denmark	47	Aalborg University	Hosted by a university. WBM frim as OpenWRT feed.
v7	May 12-18 2014	Leipzig Germany	75	FreiFunk Leipzig	Hosted by the SubLab. First WiBed Battlemesh.
v8	??? 2015	Maribor Slovenia	??	Wlan Slovenija	Emplacement to be confirmed.

[1] Registered at the wiki. The real number of participants ends up being around 10% and 30% higher.

Table C.1: Summary of the WBM editions.

C.3 Battlemesh deployment

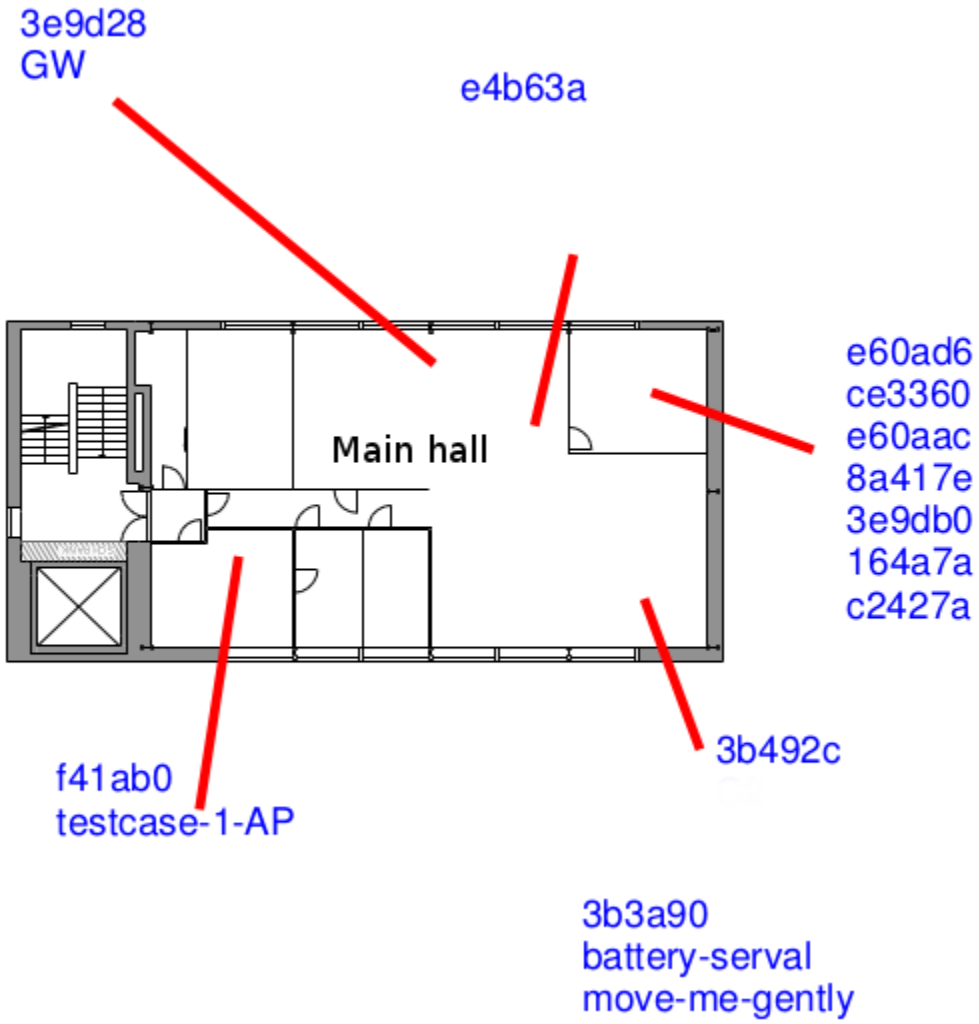


Figure C.1: Initial deployment of the Battlemesh

Appendix D

WiBed operation examples

D.1 Example of command's execution

Sending the command *ls -la /root*:

```
Previous state:
Node: wibed-e62b54
Current state: 1 (IDLE)
Command Ack: 72
Result Ack: 72


Pull request 1:
{
    "status":1,
    "commandAck":72,
    "results":[]
}


Controller response 1:
{
    "resultAck": 72
}
```

Researcher sends the command:

```
Pull request 2:
{
    "status":1,
    "commandAck":72,
    "results":[]
}


Controller response 2:
{
    "commands":
    {
```

```
        "73": "ls -la /root"
    },
    "resultAck": 72
}


Result:
Command Ack: 73
Result Ack: 72
```

The node executes the command and stores the outputs:

```
73
ls -la /root
Executing command 73 "ls -la /root"
0
nil
```

The node executes the next pull request:

```
Pull request 3:
{
    "status":1,
    "commandAck":73,
    "results":
    {
        "73":["0","EXECUTED COMMAND RESULTS",""]
    }
}


Controller reponse 3:
{
    "resultAck": 73
}


Result:
Command Ack: 73
Result Ack: 73
```

D.2 Example of experiment's execution

Performing and experiment with a dummy overlay:

This experiment will execute 2 scripts: the first one just prints a text in the terminal output and the second one puts a text in the /save/result.txt file owing to be uploaded to the controller as a result.

```
Previous state:
Node: wibed-e62b54
Current state: 1 (IDLE)
Experiment Id: 42
Command Ack: 72
```

```
Result Ack: 72
Flash is the overlay (default state)
```

The researcher adds the experiment in the controller:

```
Pull request 1:
{
    "status":1,
    "commandAck":73,
    "results":[]
}


Controller response 1:
{
    "experiment":
    {
        "action": "PREPARE",
        "id": 43,
        "overlay": "dummy.tar.gz"
    },
    "resultAck": 73
}


Result:
Experiment id: 43
Status: 2
```

The node is successful downloading the overlay:

```
Pull request 2:
{
    "status":3
}


Controller response 2:
{}
```

The researcher starts the experiment

```
Pull request 3:
{
    "status":3
}


Controller response 3:
{
    "experiment":
    {
        "action": "RUN"
    }
}
```

Result:
Status: 4
Previous files allocated in the overlay are deleted and the new overlay is installed.
The overlay will be mounted in the next boot. The node reboots after 60 seconds.

The experiment is correctly deployed in the node:

```
Node's state:
Status: 5
Experiment Id: 43
Command Ack: 73
Result Ack: 73
USB is the overlay (The experiment overlay is mounted)


Pull request 4:
{
    "status":5,
    "commandAck":73,
    "results":[]
}


Controller response 4:
{
    "resultAck": 73
}
```

The researcher executes a command in the experiment without saving its results:

```
Pull request 5:
{
    "status":5,
    "commandAck":73,
    "results":[]
}


Controller response 5:
{
    "commands":
    {
        "74": "./script.sh"
    },
    "resultAck": 73
}
```

The node executes the command:

```
74
./script.sh
```

```
Executing command 74 "./script.sh"
Command ID: 74
Command Str: ./script.sh
0
nil
```

The node executes the next pull request:

```
Pull response 6:
{
    "status":5,
    "commandAck":74,
    "results":
    {
        "74":["0","This script does not save any result, so it
        will only upload the outputs.",""]
    }
}


Controller response 6:
{
    "resultAck": 74
}


Result:
Command Ack: 74
Result Ack: 74
```

The research executes a command in the experiment saving its results:

```
Pull request 7:
{
    "status":5,
    "commandAck":74,
    "results":[]
}


Controller response 7:
{
    "commands":
    {
        "75": "./script2.sh"
    },
    "resultAck": 74
}
```

The node executes the command:

```
75
./script2.sh
Executing command 75 "./script2.sh"
0
```

```
nil
Command ID: 75
Command Str: ./script2.sh
```

The node executes the next pull request:

```
Pull request 8:
{
    "status":5,
    "commandAck":75,
    "results":
    {
        "75":["0","",""]
    }
}


Controller response 8:
{
    "resultAck": 75
}


Result:
Command Ack: 75
Result Ack: 75
```

The researcher finishes the experiment

```
Pull request 9:
{
    "status":5,
    "commandAck":75,
    "results":[]
}


Controller response 9:
{
     "experiment":
     {
         "action": "FINISH"
     },
    "resultAck": 75
}
Command ID: -1
Command Str: exit
Found exit signal
Command executer exiting
./
./result.txt
The results url is:
http://wibed.confine-project.eu/api/results/wibed-e62b54
Sending to server file: /root/43.tar.gz
```

```
{
    "exit": "success"
}
```

Result:
The experiment is finished and the */save* folder is scanned. Once all the files in the folder are gathered, a 'exp/id.tar.gz' zipped file is sent to the controller. If the uploading process is successful, the node starts going to the default state. Otherwise, it tries to upload it again.
Status:6

The node executes the next pull request:

```
Pull request 10:
{
    "status":6
}

Controller response 10:
{}


Result:
The node reboots, unmounts the overlay and returns to its
default state.
Status: 1
Experiment Id: 43
Command Ack: 75
Result Ack: 75
Flash is the overlay
```

Appendix E

WiBed scripts

In this section, the code of the main WiBed platform scripts is shown. As most of them have hundreds of lines, only the most important functions and its main procedure will be shown and if the reader wants to check the entire code, in the WiBed repositories section 3.7 the information regarding how to access them can be found.

E.1 wibed-node

```
#!/usr/bin/lua

io = require("io")
fs = require("nixio.fs")
http = require("socket.http")
ltn12 = require("ltn12")
json = require ("dkjson")
libuci = require ("uci")
cURL = require("cURL")

RESULTS_DIR = "/root/results"
COMMANDS_PIPE = "/var/run/command-executer.sock"
OVERLAY_DIR = "/overlay"
MNT_USB_OVERLAY = "/tmp/usb-overlay"
MNT_FLASH_OVERLAY = "/tmp/flash-overlay"
REBOOT_DELAY = 60
SAVE_DIR = "/save"

-- Status
INIT = 0
IDLE = 1
PREPARING = 2
READY = 3
DEPLOYING = 4
RUNNING = 5
RESETTING = 6
UPGRADING = 7
```

```
ERROR = 8

-- Overlay
NO = 0
FLASH = 1
USB = 2

-- Function: uciSetInOverlays
-- Write variable to the UCI database in flash and in the usb
    stick.
--
-- Params:
-- * config - Which config file to use.
-- * section - Which uci section.
-- * option - Which uci option.
-- * value - Value to write.
-- * overlay - Which overlay to write, options can be "both", "
    USB", "FLASH"
function uciSetInOverlays(config, section, option, value,
    overlay)

    local overlays = {}
    if overlay == "both" then
        overlays = {"/", MNT_USB_OVERLAY, MNT_FLASH_OVERLAY}
    elseif overlay == "USB" then
        if mountedOverlay == USB then
            overlays = {"/"}
        else
            overlays = {MNT_USB_OVERLAY}
        end
    elseif overlay == "FLASH" then
        if mountedOverlay == FLASH then
            overlays = {"/"}
        else
            overlays = {MNT_FLASH_OVERLAY}
        end
    else
        print("Something went terribly wrong")
        os.exit(1)
    end

    for _, dir in ipairs(overlays) do
        local confdir = dir .. "/etc/config"
        local savedir = "/tmp/.uci/" .. dir
        if folderExists(confdir) then
            if not folderExists(savedir) then fs.mkdirr(savedir)
                end
            local uci = libuci:cursor()
            uci:set_confdir(confdir)
            uci:set_savedir(savedir)
            uci:set(config, section, option, value)
            uci:save(config)
            uci:commit(config)
        end
```

```
    end
end


-- Function: getOverlay
-- -- Write variable to the UCI database in flash and in the usb
     stick.
--
-- -- Returns: if and which overlay exists
function getOverlay()

    _, overlay  = executeCommand(string.format("mount | grep \"
        /overlay\" | cut -d\" \" -f5"))
    if overlay == "" then
        print(string.format("No overlay"))
        return NO
    elseif overlay == "jffs2" then
        print(string.format("Flash is the overlay"))
        return FLASH
    elseif overlay == "ext4" then
        print(string.format("USB is the overlay"))
        return USB
    else
        print(string.format("Error executing mount!"))
        os.exit(1)
    end
end


-- Function: buildResults
-- Builds a trable containing information about all non-
    acknowledged
-- results.
--
-- Returns:
-- * Table with information about non-acked results.
function buildResults()
    local results = {}
    local resultAck = resultAck or 0

    if not folderExists(RESULTS_DIR) then
        return results
    end

    local _, commandIdsStr = executeCommand(string.format("ls -1
        \"%s\"",RESULTS_DIR))
    local commandIds = commandIdsStr:split("\n")

    if not commandIds then
        return
    end

    for _, commandId in ipairs(commandIds) do
        commandId = tonumber(commandId)
```

```
            cmdResultFolder=string.format("%s/%s",RESULTS_DIR,
                commandId)

            if commandId > resultAck then
                if not fileExists(string.format("%s/exitCode",
                    cmdResultFolder)) then
                    break
                end

                exitCode = readFile(string.format("%s/exitCode",
                    cmdResultFolder))
                stdout = readFile(string.format("%s/stdout",
                    cmdResultFolder))
                stderr = readFile(string.format("%s/stderr",
                    cmdResultFolder))

                results[tostring(commandId)] = {exitCode, stdout,
                    stderr}
            end
        end

        return results
end


-- Function: sendError
-- Send error info to the server
--
function sendError()
    -- Create tar.gz file from the result files
    _, exp = executeCommand(string.format("wibed-getlogs"))
    -- Transfer tar.gz file to the server and then delete it
    local c = cURL.easy_init()
    local api = readVariable("general.api_url")
    local nodeId = readVariable("general.node_id")
    local url = string.format("%sapi/error/%s",api,nodeId)
    print("The error url is: "..url)
    c:setopt_url(url)
    local filePath=string.format("/root/error.tar.gz")
    print("Sending to server file: "..filePath)
    local postdata = {
        file = {file=filePath,
                type="text/plain"}}
    c:post(postdata)
    c:perform()
    fs.remove(filePath)
end


-- Function: doPrepareFirmwareUpgrade
-- Prepare the firmware upgrade process.
--
-- Params:
```

```
-- * version - New firmware version.
-- * hash - The hash of the new firmware.
-- * upgradeTime - The time at which to make the upgrade.
function doPrepareFirmwareUpgrade(version, hash, upgradeTime)
    print(string.format("Fetching firmware upgrade at %s",
        upgradeTime))
    success, statusCode, _, _ = http.request{
        url = string.format("%s/static/firmwares/%s",apiUrl,
            version),
        sink = ltn12.sink.file(io.open("/tmp/wibed.bin", 'w'))
    }

    if success and statusCode == 200 then
            status = UPGRADING
            writeFile("/root/wibed.upgrade.lasthash",hash)
            writeFile("/root/wibed.upgrade.version",version)
            print(string.format("Start UPGRADING process in the
                next wibed-node call"))
    else
        print(string.format("Downloading of firmware failed: %s"
            , statusCode))
        status = ERROR
        sendError()
    end
end


-- Function: doFirmwareUpgrade
-- Start the firmware upgrade process.
--
function doFirmwareUpgrade()
        print(string.format("Attempting firmware upgrade"))
        local oldVersion = readVariable("upgrade.version") or
            nil
        local upVersion = readFile("/root/wibed.upgrade.version"
            ) or nil
        local hash = readFile("/root/wibed.upgrade.lasthash") or
             nil

        if hash == nil or #hash < 8 then
                status = ERROR
                print(string.format("Failed to upgrade to
                    version %s, hash file does not exist",
                    upVersion))
                sendError()
                return 1
        end

        -- Execute the upgrade script
        print(string.format("Executing the upgrade script"))
        local exitCode, _ = executeCommand(string.format("/usr/
            sbin/wibed-upgrade %s %s" , upVersion, hash ))

        if exitCode ~= "0" then
```

```
                status = IDLE
                writeVariable("general.status", status)
                print(string.format("Failed to upgrade to
                    version %s, going back to IDLE state",
                    upVersion))
                return 1
        end
end


-- Function: doPrepareExperiment
-- Prepares an experiment by downloading the respective overlay
    and
-- installing it.
--
-- Params:
-- * experimentId - The id of the experiment.
-- * overlayId - The id of the overlay used in the experiment.
-- * overlayHash - The hash of the overlay.
function doPrepareExperiment(experimentId, overlayId,
    overlayHash)
    status = PREPARING
    writeVariable("general.status", status)

    executeCommand(string.format("rm -rf %s/*", MNT_USB_OVERLAY)
        )

    success, statusCode, _, _ = http.request{
        url = string.format("%s/static/overlays/%s",apiUrl,
            overlayId),
        sink = ltn12.sink.file(io.open(string.format("%s/overlay
            .tar.gz", MNT_USB_OVERLAY), "w"))
    }
    if success and statusCode == 200 then
        executeCommand(string.format("mkdir %s/sbin 2>/dev/null"
            ,OVERLAY_DIR))
        executeCommand(string.format("cp -f /rom/sbin/block %s/
            sbin/block",OVERLAY_DIR))
        executeCommand(string.format("touch /etc/config/fstab"))
        executeCommand(string.format("cp -a %s/* %s/",
            OVERLAY_DIR, MNT_USB_OVERLAY))
        executeCommand(string.format("rm -f %s/etc/.extroot-uuid
             2>/dev/null",MNT_USB_OVERLAY))
        --executeCommand(string.format("rm -rf %s/etc/uci-
            defaults 2>/dev/null",MNT_USB_OVERLAY))
        executeCommand(string.format("tar -xhzf %s/overlay.tar.
            gz -C %s",MNT_USB_OVERLAY, MNT_USB_OVERLAY))

        status = READY
        writeVariable("general.status", status)
        writeVariable("experiment.exp_id", experimentId)
    else
        -- TODO: Report error
        print("Downloading of overlay failed: " .. statusCode)
```

```
        status = ERROR
        sendError()
    end
end

-- Function: doStartExperiment
-- Starts the experiment.
function doStartExperiment()
    status = DEPLOYING
    uciSetInOverlays("fstab", "usb_overlay", "target",
        OVERLAY_DIR, "both")
    -- Change status to flash overlay as DEPLOYING
    uciSetInOverlays("wibed", "general", "status", status, "
        FLASH")
    -- Change future status to usb overlay as RUNNING
    uciSetInOverlays("wibed", "general", "status", RUNNING, "USB
        ")
    executeCommand(string.format("sleep %d && reboot -f &",
        REBOOT_DELAY))
end


-- Function: saveResults
-- Send the results in /save folder to the server
--
function saveResults()
    -- Create tar.gz files from the result files
    _, exp = executeCommand(string.format("ID=`uci get wibed.
        experiment.exp_id` && tar -cvzf /root/${ID}.tar.gz -C /
        save/ . && rm -rf /save/* && echo $ID"))
    -- Transfer tar.gz file to the server and then delete it
    local c = cURL.easy_init()
    local saveUrl = readVariable("experiment.save_url")
    local nodeId = readVariable("general.node_id")
    local url = string.format("%s/%s",saveUrl,nodeId)
    print("The results url is: "..url)
    c:setopt_url(url)
    local filePath=string.format("%s/%s.%s","/root", exp,"tar.gz
        ")
    print("Sending to server file: "..filePath)
    local postdata = {
        file = {file=filePath,
                type="text/plain"}}
    c:post(postdata)
    c:perform()
    fs.remove(filePath)
end


-- Function: doFinishExperiment
-- Finishes an active experiment.
--
function doFinishExperiment()
```

```
    if mountedOverlay == USB then
        uciSetInOverlays("fstab", "usb_overlay", "target",
            MNT_USB_OVERLAY, "both")
        executeCommand(string.format("echo \"-1 exit\" > \"%s\""
            ,COMMANDS_PIPE), false)

        status = RESETTING
        -- Change status to usb overlay as RESETTING
        uciSetInOverlays("wibed", "general", "status", status, "
            USB")
        -- Change  future status to flash overlay as IDLE
        uciSetInOverlays("wibed", "general", "status", IDLE, "
            FLASH")

        saveResults()

        executeCommand(string.format("sleep %d && reboot -f &",
            REBOOT_DELAY))

    else
        -- Assuming that no error happened and the experiment
        -- was finished before even running
        status = IDLE
    end

end


-- Function: executeCommands
-- Sets up the commands provided as argument for execution.
--
-- Args:
-- * commands - Table of commands {<id1>=<cmd1>, <id2>=<cmd2>}
function executeCommands(commands)
    if not commands then
        return
    end

    local lastCommandId=commandAck
    local sanitizedCommands = {}

    for commandId, commandStr in pairs(commands) do
        table.insert(sanitizedCommands, {tonumber(commandId),
            commandStr})
    end
    table.sort(sanitizedCommands, function (a,b) return b[1] < a
        [1] end)

    for _ , pair in ipairs(sanitizedCommands) do
        commandId, commandStr = unpack(pair)
        print(commandId)
        print(commandStr)
        commandId = tonumber(commandId)
```

```
        if not pipeExists(COMMANDS_PIPE) then
            executeCommand("command-executer &")
            -- Give some time for named pipe to be created by
                executer
            executeCommand("sleep 1")
        end

        print(string.format("Executing command %d \"%s\"",
            commandId, commandStr))
        --This doesn't work so we have to hack a lil bit
        --writeFile(COMMANDS_PIPE, "%d %s" % {commandId,
            commandStr})
        exitCode,stdout = executeCommand(string.format("echo \"%
            d %s\" > \"%s\"", commandId, commandStr,
            COMMANDS_PIPE), false)
        print(exitCode)
        print(stdout)
        lastCommandId = commandId
    end

    writeVariable("general.commandAck", lastCommandId)
end

-- END OF FUNCTIONS.
-- START PULL PROCESS

apiUrl = assert(readVariable("general.api_url"), "API URL not
    defined")
print(string.format("API URL: %s", apiUrl))
id = assert(readVariable("general.node_id"), "Node ID not
    defined")
print(string.format("Id: %s", id))
status = tonumber(readVariable("general.status")) or INIT
print(string.format("Status: %d", status))
model = readVariable("upgrade.model")
print(string.format("Model: %s", model))
version = readVariable("upgrade.version")
print(string.format("Version: %s", version))
experimentId = readVariable("experiment.exp_id")
print(string.format(string.format("Experiment Id: %s", (
    experimentId or "None"))))
commandAck = tonumber(readVariable("general.commandAck"))
print(string.format("Command Ack: %s", (commandAck or "None")))
resultAck = tonumber(readVariable("general.resultAck"))
print(string.format("Result Ack: %s", (resultAck or "None")))
mountedOverlay = tonumber(getOverlay())
--print(string.format("The overlay used is: %s", mountedOverlay)
    )
coordx = readVariable("general.coordx")
coordy = readVariable("general.coordy")
coordz = readVariable("general.coordz")
testbed = readVariable("location.testbed")
is_gw = readVariable("management.is_gw")
```

```
request = {}
request["status"] = status

if status == INIT then

    -- Check the version (prevent to send a wrong version when
        failed to upgrade the firmware)
    local _, tailfirmv = executeCommand("tail -n 1 /etc/wibed.
        version",true)
    local firmv = string.sub(tailfirmv, 1, 8)

    if version ~= firmv then
        writeVariable("upgrade.version", firmv)
        version = firmv
    end

    request["model"] = model
    request["version"] = version
    request["coordx"] = coordx
    request["coordy"] = coordy
    request["coordz"] = coordz
    request["testbed"] = testbed
    request["gateway"] = is_gw

elseif status == IDLE or status == RUNNING or status == ERROR
    then
    if commandAck then
        request["results"] = buildResults()
        request["commandAck"] = commandAck
    end
end

jsonEncodedRequest = json.encode(request, {indent = true})

print(" ")
print("Request:")
print(jsonEncodedRequest)
print(" ")

responseBody, statusCode, _, _ = http.request(
    string.format("%s/api/wibednode/%s", apiUrl, id),
    jsonEncodedRequest)

if responseBody and statusCode == 200 then
    print("Communication with server successful")
    print(" ")
    print(responseBody)

    response, pos, err = json.decode(responseBody)

    if err then
        print(string.format("Error parsing json: %s", err))
        os.exit(1)
    elseif response["errors"] then
```

```
        print("Error sent by server:")
        for key, value in pairs(response["errors"]) do print(key
            , value) end
        os.exit(1)
    end

    -- Migration from init to idle is automatic upon receival of
    -- response by server
    if status == INIT then
        -- Except the case that an upgrade was performed
            incorrectly
        local upgrade = response["upgrade"]
        if upgrade then
            print(string.format("There was an error upgrading.")
                )
            print(string.format("Upgrade version and node
                version do not match"))
            status = ERROR
            sendError()
        end
        status = IDLE
    end

--if status == RESETTING then
--    local error, stdout = executeCommand(string.format("ps |
    grep reboot -c"), true)
--    local st = tonumber(stdout)
--   if st <= 2 then
--       status = IDLE
--       writeVariable("general.status", status)
--       executeCommand(string.format("sleep %d && reboot -f
    &", REBOOT_DELAY))
--    end
--end

    -- If IDLE
    if status == IDLE then
        local reinit = response["reinit"]
        if reinit then
                print("Doing REINIT")
            executeCommand(string.format("/usr/sbin/wibed-
                reset"))
            os.exit(0)
        end
        local upgrade = response["upgrade"]
        local experiment = response["experiment"]
        if upgrade then
            doPrepareFirmwareUpgrade(upgrade["version"],
                                     upgrade["hash"],
                                     upgrade["utime"])
        elseif experiment and experiment["action"] == "PREPARE"
            then
            doPrepareExperiment(experiment["id"],
                                  experiment["overlay"],
```

```
                                    experiment["hash"])
        end
    elseif response["experiment"] and status >= PREPARING
                                    and status <= RUNNING then
        if response["experiment"]["action"] == "FINISH" then
            doFinishExperiment()
        elseif response["experiment"]["action"] == "RUN" and
                status == READY then
            doStartExperiment()
        end
    elseif status == UPGRADING then
        print(string.format("Have to upgrade"))
        doFirmwareUpgrade()
    end

    if status == IDLE or status == RUNNING or status == ERROR
        then
        executeCommands(response["commands"])
    end
    local resultAck = response["resultAck"]
    if resultAck then
        writeVariable("general.resultAck", resultAck)
    end

    -- Write any changes that maybe took place upon the response
        only in the current overlay
    -- writeVariable("general.status", status)
    if mountedOverlay == FLASH then
        uciSetInOverlays("wibed", "general", "status", status, "
            FLASH")
    else
        uciSetInOverlays("wibed", "general", "status", status, "
            USB")
    end

else
    print(string.format("Communication with server unsuccessful:
        %s", statusCode))
end
```

E.2 wibed-config

```
#!/usr/bin/lua

local crc = require "crc16"
local fs  = require "nixio.fs"
local uci = require "uci"
local iw  = require "iwinfo"
local util = require "util"
local ucic = "wibed"
local ucil = "libremap"
local x = uci:cursor()
```

```
local function node_id()
        local dev = "eth0"
        local mac = assert(fs.readfile("/sys/class/net/"..dev.."
            /address"))
        local hash = crc.hash(mac)
        return math.floor(hash / 256), hash % 256
end

local function mac_id()
        local dev = "eth0"
        local mac = assert(fs.readfile("/sys/class/net/"..dev.."
            /address"))
        local id = string.format("%s%s%s",string.sub(mac,10,11),
            string.sub(mac, 13, 14),string.sub(mac, 16, 17))
        return id
end

local function get_hostname()
        local hostname = string.format("wibed-%s",mac_id())
        return hostname
end

local function get_model()
        local model = readfile("/tmp/sysinfo/board_name")
        if model == nil then
                model = shell("uname -m",true).."-"..shell("
                    uname -p",true)
        end
        return model
end

local function generate_address()
        local r1, r2 = node_id()
        local ipv4_template = assert(x:get(ucic, "management", "
            ipv4_net"))
        local ipv6_template = assert(x:get(ucic, "management", "
            ipv6_net"))

        return ipv4_template:gsub("R1", r1):gsub("R2", r2),
                ipv6_template:gsub("R1", hex(r1)):gsub("R2", hex(
                    r2))
-- XXX: id should be hex coded but for backwards compat keep it
    decimal
end

local function generate_mgmt_lan_address()
        local r1, r2 = node_id()
        local ipv4_template = assert(x:get(ucic, "management", "
            ipv4_lan_net"))
        return ipv4_template:gsub("R1", r1):gsub("R2", r2)
end

local function generate_ssid()
        local ssid = assert(x:get(ucic, "management", "ssid"))
```

```
        local id = mac_id()
        return string.format("%s-%s",ssid, id)
end

local function get_bssid()
        return x:get(ucic, "management","bssid") or "02:CA:FF:EE
            :BA:BE"
end
\end{verbbox}
\resizebox{\textwidth}{!}{\theverbbox}

\begin{verbbox}
local function is_net(d)
        -- get the physical ethernet device if dev is a vlan
        local dev = d:split_str("%.")
        local fdev
        if dev[1] == nil then
                fdev = d
        else
                fdev = dev[1]
        end
        local r = os.execute(string.format("ls /sys/class/net/%s
            >/dev/null 2>&1",fdev))
        return r==0
end

local function is_wifi(dev)
        local r1 = os.execute(string.format("ls /sys/class/net/%
            s/phy80211/ >/dev/null 2>&1",dev))
        local r2 = os.execute(string.format("cat /tmp/wireless.
            backup | egrep 'ifname|wifi-device' | grep %s >/dev/
            null 2>&1",dev))
        return (r1==0 or r2==0)
end

local function set_batadv(dev)
        printf("-> Configuring batman-adv for %s ",dev)
        local ifn = "bat"..dev
        x:set("batman-adv","bat0","mesh")
        x:set("batman-adv","bat0","bridge_loop_avoidance","1")
        x:set("batman-adv","bat0","orig_interval","5000")
        x:set("network",ifn,"interface")
--      x:set("network",ifn,"ifname",dev)
        x:set("network",ifn,"proto","batadv")
        x:set("network",ifn,"mesh","bat0")
        if is_wifi(dev) then
                x:set("network",ifn,"mtu","1532")
        end
end

local function set_mgmt_net()
        print("Configuring management ethernet devices")
        local ifaces,_ = split_ifaces(x:get(ucic, "management",
            "ifaces"))
```

```
        local _,i
        for _,i in ipairs(ifaces) do print("-> Interface "..i)
            end

        table.insert(ifaces,"bat0")
        local ipv4, ipv6 = generate_address()

        x:set("network", "mgmt", "interface")
        x:set("network", "mgmt", "type","bridge")
        x:set("network", "mgmt", "proto", "static")
        x:set("network", "mgmt", "ipaddr", ipv4)
        x:set("network", "mgmt", "netmask", "255.255.0.0")
        x:set("network", "mgmt", "ip6addr", ipv6)
        x:set("network", "mgmt", "ifname", ifaces)
end
\end{verbatim}
%\end{verbbox}
%\resizebox{\textwidth}{!}{\theverbbox}
%\begin{verbbox}
\begin{verbatim}
local function set_mgmt_lan()
        if x:get("wibed","management","is_gw") == "1" then
                print("Configuring node as Internet gateway...")
                local ipv4 = generate_mgmt_lan_address()
                 -- BATMANT-ADV GW CONFIGURATION:
                x:set("batman-adv", "bat0", "gw_mode", "server")

                x:set("network", "mgmt_lan", "interface")
                x:set("network", "mgmt_lan", "proto", "static")
                x:set("network", "mgmt_lan", "ipaddr", ipv4)
                x:set("network", "mgmt_lan", "netmask", "
                    255.255.255.0")
                x:set("network", "mgmt_lan", "ifname", "br-mgmt"
                    )

                x:set("dhcp","mgmt_lan","dhcp")
                x:set("dhcp","mgmt_lan","interface","mgmt_lan")
                x:set("dhcp","mgmt_lan","start","2")
                x:set("dhcp","mgmt_lan","limit","250")
                x:set("dhcp","mgmt_lan","leasetime","1h")

                x:foreach("firewall", "zone", function(s)
                        if x:get("firewall",s[".name"],"name")
                            == "lan" then
                                x:set("firewall",s[".name"],"
                                    network",{"mgmt_lan"})
                        end
                end)
        else
                -- BATMANT-ADV CLIENT CONFIGURATION:
                x:set("batman-adv", "bat0", "gw_mode", "client")

                x:set("network", "mgmt_lan", "interface")
                x:set("network", "mgmt_lan", "proto", "dhcp")
```

```
                x:set("network", "mgmt_lan", "ifname", "br-mgmt"
                    )
        end

        x:foreach("firewall", "zone", function(s) x:set("
            firewall", s[".name"],"input","ACCEPT") end)
        x:foreach("firewall", "zone", function(s) x:set("
            firewall", s[".name"],"output","ACCEPT") end)
        x:foreach("firewall", "zone", function(s) x:set("
            firewall", s[".name"],"forward","ACCEPT") end)
end

local function set_system()
        print("Configuring system...")
        -- Setting hostname
        local hostname = get_hostname()

        x:foreach("system", "system", function(s)
                x:set("system", s[".name"], "hostname", hostname
                    )
        end)

        fs.writefile("/proc/sys/kernel/hostname", hostname)

        -- Setting model name
        x:set(ucic,"upgrade","model",get_model())
        x:set(ucic,"general","node_id",hostname)
end

local function set_mgmt_wifi()
        print("Configuring management wifi devices")
        local channel5 = assert(x:get(ucic, "management", "
            channel5"))
        local txpower2 = x:get(ucic, "management", "txpower2")
            or "30"
        local txpower5 = x:get(ucic, "management", "txpower5")
            or "20"
        local mrate = x:get(ucic, "management", "mrate") or nil
        local countrycode = x:get(ucic, "management", "country")
             or nil
        local channel2 = assert(x:get(ucic, "management", "
            channel2"))
        local i,_
        local _,ifaces = split_ifaces(x:get(ucic, "management",
            "ifaces"))
        local wifi_num = 0

        for _,i in ipairs(ifaces) do
                print("-> Interface " .. i)
                local id = string.format("mgmt%d", wifi_num)
                local net = "bat"..id

                local t = iw.type(i)
                if t then
```

```
        local is_5ghz = iw[t].hwmodelist(i).a

        if is_5ghz then ch=channel5
        else  ch=channel2 end

        if not ch then
                printf("-> No channel defined
                    for %dGHz %s", is_5ghz and 5
                    or 2, i)
                return
        end

        local ht = ch:match("[-+]?$")

        printf("-> Using channel %s for %dGHz %s
            ", ch, is_5ghz and 5 or 2, i)
        x:set("wireless", i, "channel", (ch:gsub
            ("[-+]$", "")))

        if x:get("wireless", i, "ht_capab") then
                if ht == "+" or ht == "-" then
                        x:set("wireless", i, "
                            htmode", "HT40"..ht)
                else
                        x:set("wireless", i, "
                            htmode", "HT20")
                end
        end

    x:set("wireless", i, "disabled", 0)
        if is_5ghz then
                x:set("wireless", i, "txpower",
                    txpower5)
                printf("-> Using txpower %s for
                    5GHz", txpower5)
        else
                x:set("wireless", i, "txpower",
                    txpower2)
                printf("-> Using txpower %s for
                    2GHz", txpower2)
        end
        if countrycode ~= nil then
                x:set("wireless", i, "country",
                    countrycode)
                printf("-> Using countrycode %s"
                    , countrycode)
        end
        if mrate ~= nil then
                x:set("wireless", i, "mcast_rate
                    ", mrate)
                printf("-> Using multicast rate
                    %s", mrate)
        end
```

```
                        x:set("wireless", id, "wifi-iface")
                        x:set("wireless", id, "hidden", 1)
                        x:set("wireless", id, "ssid",
                            generate_ssid())
                        x:set("wireless", id, "bssid", get_bssid
                            ())
                        x:set("wireless", id, "device", i)
                        x:set("wireless", id, "network", net)
                        x:set("wireless", id, "mode", "adhoc")
                        x:set("wireless", id, "ifname", id)

                        set_batadv(id)

                        wifi_num = wifi_num + 1
                else
                        printf("-> Error, device %s not found as
                            WiFi interface", i)
                end
        end
end

function set_location()
        local coordx = x:get(ucic, "general", "coordx")
        local coordy = x:get(ucic, "general", "coordy")
        local coordz = x:get(ucic, "general", "coordz")

        x:set(ucil, "location", "latitude", coordx)
        x:set(ucil, "location", "longitude", coordy)
        x:set(ucil, "location", "elev", coordz)
end


function main()
        reset_wifi()
        clean()
        set_mgmt_wifi()
        set_mgmt_net()
        set_mgmt_lan()
        set_system()
        set_location()

        print("Committing config files...")
        x:save("network")
        x:save("batman-adv")
        x:save(ucic)
        x:save("system")
        x:save("wireless")
        x:commit("network")
        x:commit("batman-adv")
        x:commit("wireless")
        x:commit("firewall")
        x:commit("dhcp")
        x:commit(ucic)
        x:commit(ucil)
```

```
        x:commit("system")
end

main()
```

E.3 wibed-location

```
!/usr/bin/lua

local uci = require "uci"
local ucic = "wibed"
local x = uci:cursor()

function locate (lat, lon, elev)
        os.execute(string.format("uci set wibed.general.coordx=%
            s", lat))
        os.execute(string.format("uci set wibed.general.coordy=%
            s", lon))
        os.execute(string.format("uci set wibed.general.coordz=%
            s", elev))

        os.execute(string.format("uci set libremap.location.
            longitude=%s", lon))
        os.execute(string.format("uci set libremap.location.
            latitude=%s", lat))
        os.execute(string.format("uci set libremap.location.elev
            =%s", elev))

        os.execute("uci commit")
        os.execute("libremap-agent >/dev/null &")
end

-- EXECUTION OF THE SCRIPT
function main()

        if #arg == 1 and arg[1] == "-d" then
                default_location()
        elseif #arg ~= 3 then
                if arg[1] == "-s" then
                        if #arg ~= 4 then
                                usage()
                        else
                                os.execute("sleep 300")
                                locate(arg[2], arg[3], arg[4])
                        end
                else
                        usage()
                end
        else
                locate(arg[1], arg[2], arg[3])
        end
end
```

```
main()
```

E.4 wibed-upgrade

```
#!/bin/sh

VERSION="$1"
HASH="$2"
MYHASH="$(md5sum /tmp/wibed.bin | awk '{print $1}')"
if [ "$HASH" == "$MYHASH" ] ; then
        echo "UPGRADING"
        sysupgrade -n /tmp/wibed.bin &
        sleep 1
        return 0
fi

echo "FAILED TO UPGRADE"
rm -f /tmp/wibed.bin
return 1
```

Bibliography

[1] R. B. Viñas, "Wibed off-the-shelf wireless networks research testbed," bachelor thesis, Universitat Oberta de Catalunya, June 2013.

[2] P. Escrich, R. Baig, A. Neumann, A. Fonseca, F. Freitag, and L. Navarro, "Wibed, a platform for commodity wireless testbeds.," in *Wireless Days*, pp. 1–3, IEEE, 2013.

[3] "BMX6 – a loop-free routing protocol for IP-based mesh networks," June 2011. `http://www.bmx6.net`.

[4] A. Neumann and C. Aichele and M. Lindner and S. Wunderlich, "Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.)." Internet draft, work in progress, Mar. 2008.

[5] T. Clausen, P. Jacquet, "Optimized Link State Routing Protocol (OLSR)." RFC3626 (Experimental), 2003.

[6] Juliusz Chroboczek, "The Babel Routing Protocol." RFC 6126 (Experimental), 2011.

[7] D. Raychaudhuri, M. Ott, and I. Secker, "Orbit radio grid tested for evaluation of next-generation wireless network protocols," in *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, TRIDENTCOM '05, (Washington, DC, USA), pp. 308–309, IEEE Computer Society, 2005.

[8] "NITOS Wireless Testbed - Network Implementation Testbed Laboratory." `http://nitlab.inf.uth.gr/NITlab/index.php/testbed`.

[9] D. Wu, D. Gupta, and P. Mohapatra, "Qurinet: A wide-area wireless mesh testbed for research and experimental evaluations," *Ad Hoc Netw.*, vol. 9, pp. 1221–1237, Sept. 2011.

[10] M. Günes, B. Blywis, and F. Juraschek, "Concept and design of the hybrid distributed embedded systems testbed," no. TR-B-08-10, 2008.

[11] T. Fischer, T. Hühn, R. Kuck, R. Merz, J. Schulz-Zander, and C. Sengul, "Experiences with bowl: Managing an outdoor wifi network (or how to keep both internet users and researchers happy?)," in *Proceedings of the 25th Large Installation System Administration Conference (LISA'11)*, 2011.