



Serval PROJECT

Documentation for serval test network

Author: Stephane IMBERT
Exchange student at Flinders University
Project supervisor: Dr Paul Gardner-Stephen

Submitted to the School of Computer Science, Engineering, and Mathematics in the Faculty of Science and Engineering in partial fulfilment of the requirements for the degree of Master of Computer Science at Flinders University – Adelaide Australia.

Flinders University at Tonsley
1284 South Road
Clovely Park SA 5042

Contents

1	general presentation of the test network	2
1.1	Goals	2
1.2	Components	2
1.2.1	Hardware components	2
1.2.2	software	4
1.3	Big picture	4
1.3.1	Big picture of the network	5
1.3.2	Big picture of the software	5
2	Install	7
2.1	Basic requirements	7
2.2	Network setup	8
2.2.1	Configuration of the main router	8
2.2.2	slave router configuration	12
2.2.3	remote router configuration	17
2.3	Software installation	17
2.3.1	client installation	17
2.3.2	server installation	18
3	How to use it?	22
4	Unfinished parts	24
4.1	Wi-Fi monitoring	24
4.2	Phone	24
4.2.1	How to install screpy	25
5	Resources used	27
5.1	Network	27
5.2	server	27
5.3	client	27
5.4	client_shell	27
5.5	Phone	28
5.6	Other useful references	28
	List of Figures	29
	List of Tables	30

Chapter 1

general presentation of the test network

1.1 Goals

The foal of the project was to build a test network for the Mesh Extenders. We want to have Mesh Extenders at different places in a building. The Mesh extenders are communicating together using UHF (they need to far enough from each other or have their Wi-Fi range limited by any way). Each Mesh extender is connected to a phone by Wi-Fi.

The test network is built around this basic setup. We want to be able to remotely supervise and control each site. To do that we are going to built a test network with:

- a software part using c programs
- a hardware part using small linux routers

1.2 Components

As previously said, the test network is made of a software and a hardware part.

1.2.1 Hardware components

First, for this test network, we will obviously need Mesh Extenders:

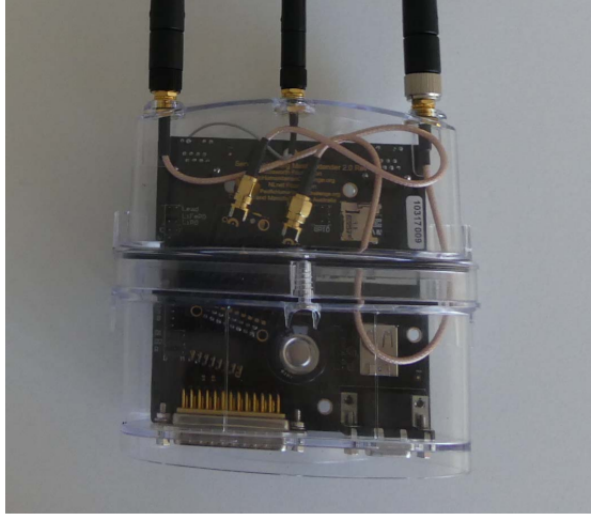


Figure 1.1: Mesh Extender

In this project we are also using small linux routers to built the network we will use to supervise the Mesh Extenders.

The small router we have chose to use are: the GL-AR150 and GL-AR750.



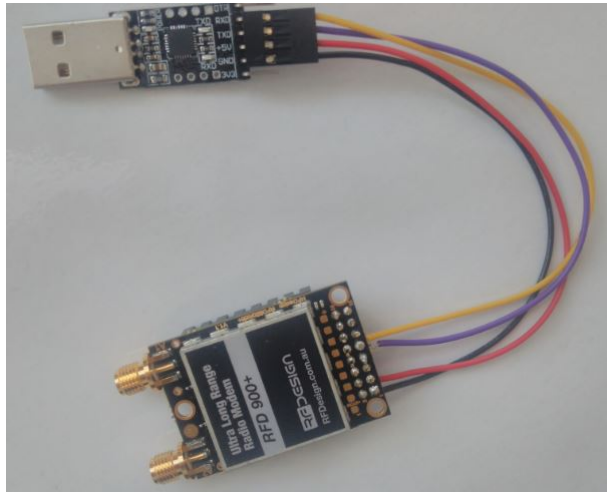
(a) GL-AR150



(b) GL-AR750

Figure 1.2: Routers used for the testbed

These two routers are great for the test network. Indeed, they are small, affordable and we have a lot of freedom since their OS is based on a Linux kernel (the OS used for the Mesh extenders can also be put inside these routers). The only issue we can have is the fact they do not have a lot of USB port and also no UHF radio. Therefore we need other devices:



(a) RFD900 with a serial to USB adapter



(b) USB hub

Figure 1.3: Other devices needed

To monitor the UHF packets of the Mesh Extenders, we need a RFD900+. Since the RFD900+ has a serial interface (pins), we need to create an adapter to plug it using USB. We are using a USB hub to increase the number of USB ports on the router. We now have all the needed hardware.

1.2.2 software

In this project, we are going to use c programs to remotely supervise the Mesh Extenders. C language was chosen because:

- it is a light and efficient language (which is great since our routers don't have a lot of power or memory)
- In the Serval project most of the coding was made in c. We can easily integrate other part of the serval project inside the test network.

I have decide to create a TCP client-server system composed of 3 files:

- server.c which will be on the small routers
- client_shell.c and client.c which will be on the computer used to supervise

I will explain what each file does in the next section.

A TCP client-server was not the only solution. However, I still opted for it because:

- I was more used to code TCP server and client than other solutions. Therefore I will be faster using this solution.
- The libraries for TCP are quite old and should work on most devices.

1.3 Big picture

We will now take a look at how the network and software work.

1.3.1 Big picture of the network

1.3.2 Big picture of the software

The software part corresponds to the communication between a few programs:

- the client_shell and the client
- the client and the server
- the server and the RFD900+

For the communication between the server and the RFD900+, we use the driver in c developed by Paul. This part use the code of the lbard git repository.

client to server communication

The communication between the client and the server is very basic. Most of the time the client is only receiving packet from the server and display them on the computer.

But the client can also send a few message to the server. This messages allow the client to close the communication with the server. Later they will also allow the client to change what is supervised.

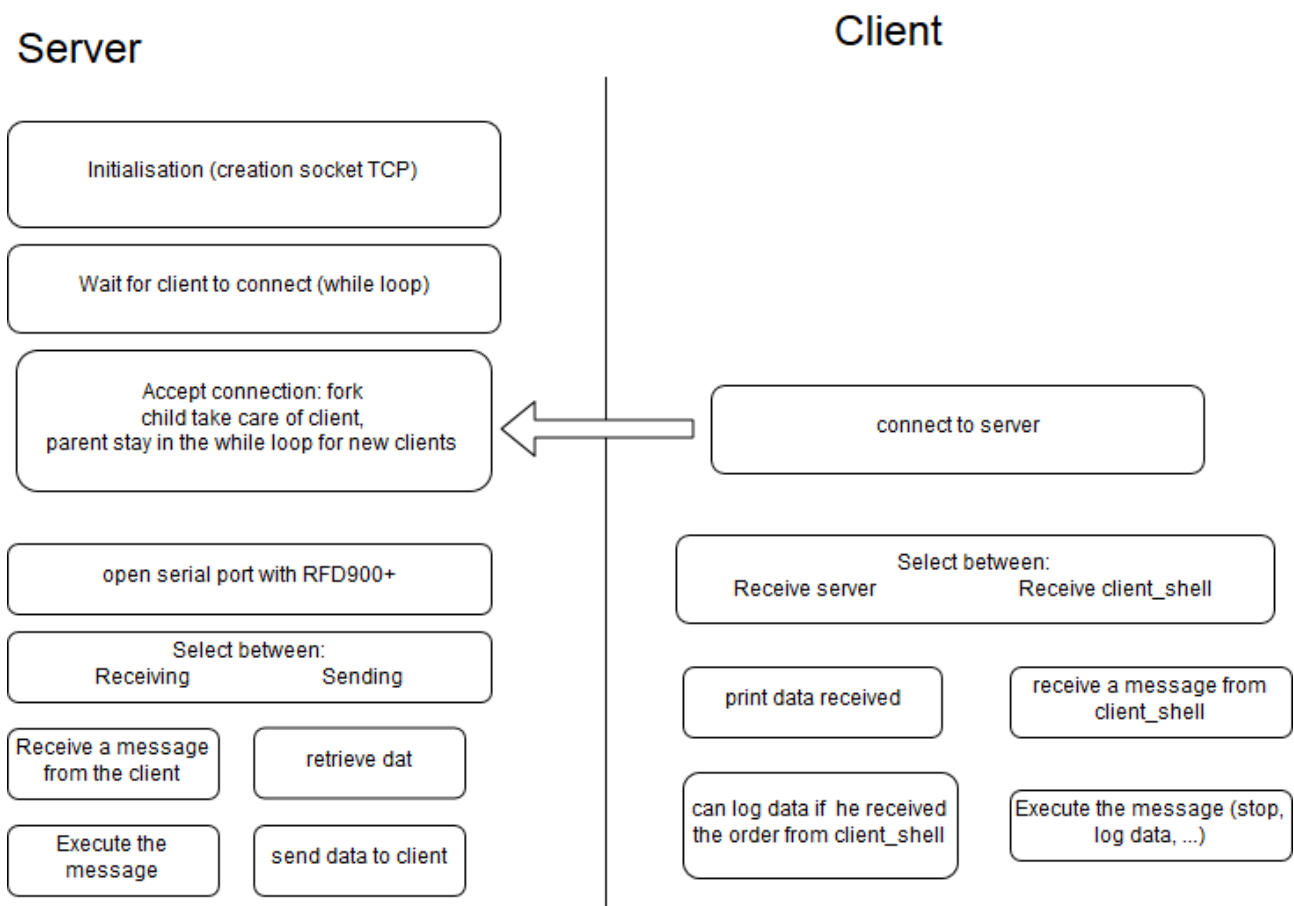


Figure 1.4: Client and server communication

client_shell and client interactions

The client_shell is a shell-like interface that gives access to all the available functionalities. It can:

- find the available servers using a file containing the hostnames of servers
- launch a client
- communicate with the client to send different orders such as
 - ”STOP” to close the connection
 - ”LOG Filename” to start logging the data
- keep a list of all the current client created

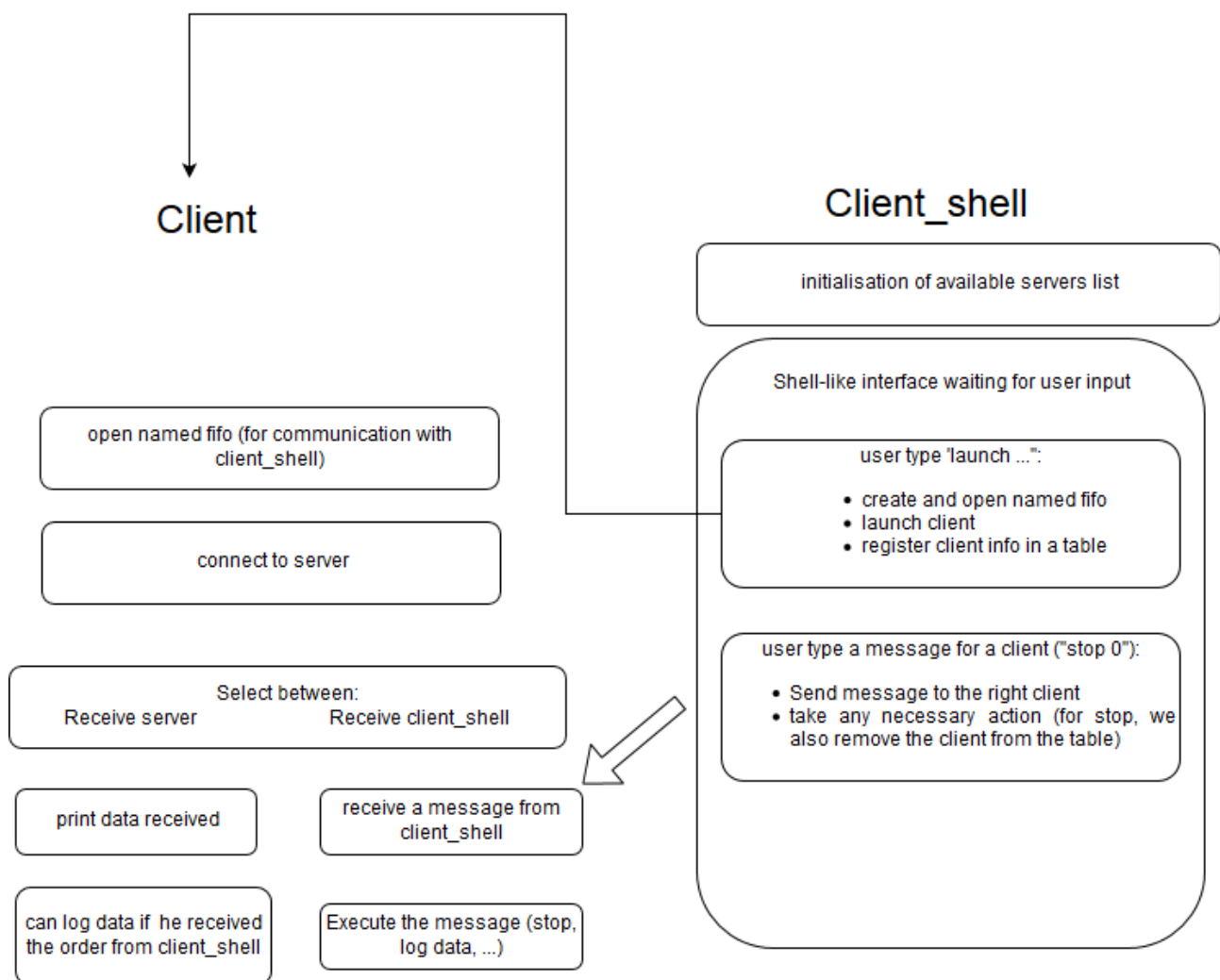


Figure 1.5: Client and Client_shell interaction

Chapter 2

Install

2.1 Basic requirements

To install this network, you will need a few devices. First you will need a computer with a linux OS (the development of the project was done on Ubuntu). But you also need a few devices:

- routers that support openwrt (I used GL-AR150 and GL-AR750)
- RFD900+ with serial to usb adaptor

Here is a list of all the package you need on your Linux machine:

- gcc
- binutils
- bzip2
- flex
- python
- perl
- make
- find
- grep
- diff
- unzip
- gawk
- getopt
- subversion
- libz-dev
- libc headers

2.2 Network setup

For this part, i will suppose you are using openwrt or Lede routers that support wireless communication.

First let's start by looking at the result we want. We have many remote sites and 1 main site. The main site is a Wi-Fi access to all the remote sites and it will be the main part of the configuration.

For this site I used GL-AR750. We will have a main router and a "slave" router. The main router will have a DHCP server and he will therefore give the addresses to all the devices.

The slave router will connect to the main router and act as a Wi-Fi bridge. The GL-AR750 does not allow us to "physically" bridge our local network (Wi-Fi + ethernet ports) and the connection with the main router. That's why we are going to use a packet called relayd that will allow us to simulate that bridge.

So the general steps are:

- Configure the main router by:
 - creating a Wi-Fi LAN
 - configuring fixed ip addresses for the small remote routers
- Configuring the slave router by:
 - installing relayd

We want to create a network like the one below:

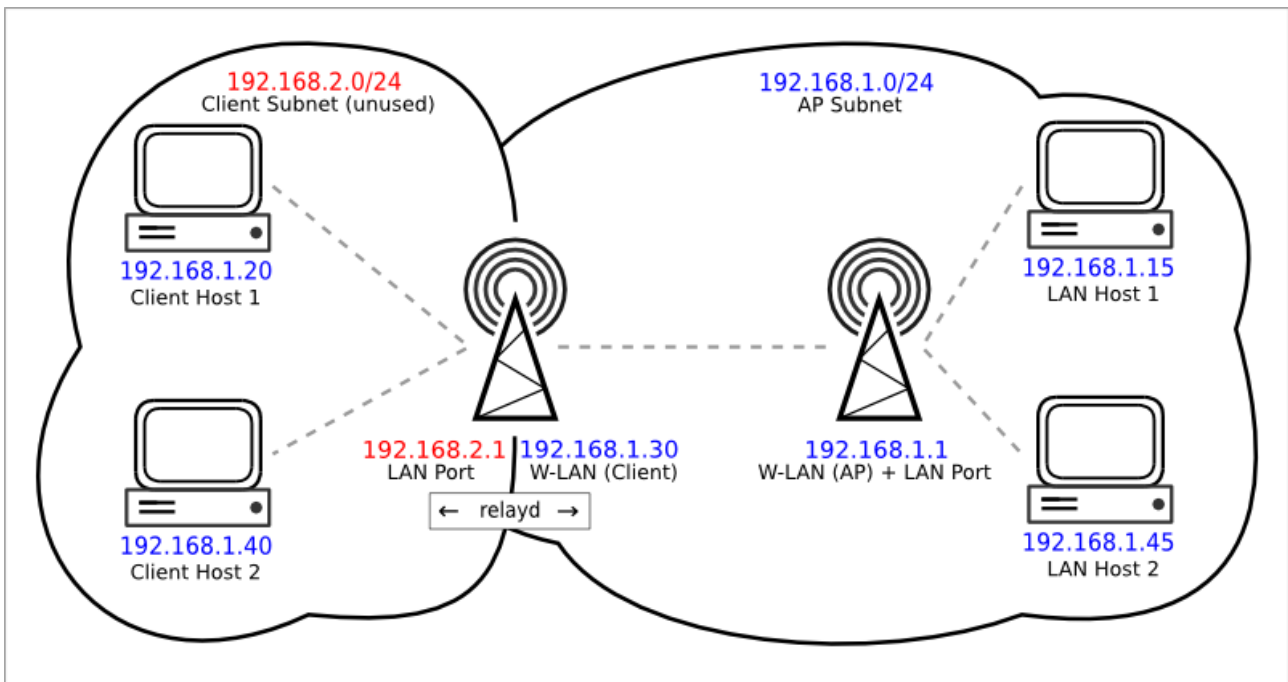


Figure 2.1: Bridge network using relayd (source:<https://wiki.openwrt.org/doc/recipes/relayclient>)

2.2.1 Configuration of the main router

All the configuration is done through the web interface at the address: `http:192.168.8.1cgi-bin/luci`. The IP address can change depending on the router you have. If you are using routers from GL-INET such as GL-AR750, it should be 192.168.8.1. Otherwise, you need to look at the documentation of your router.

Creating the Wi-Fi LAN

The first we want to do is to create the Wi-Fi and setup the LAN. So we first go to the Network → wireless tab. On this tab you should have two pre-configure networks (if you are using a GL-AR750).

We can remove all the networks here and we are going to create a new one. Once created, we will get the following page:

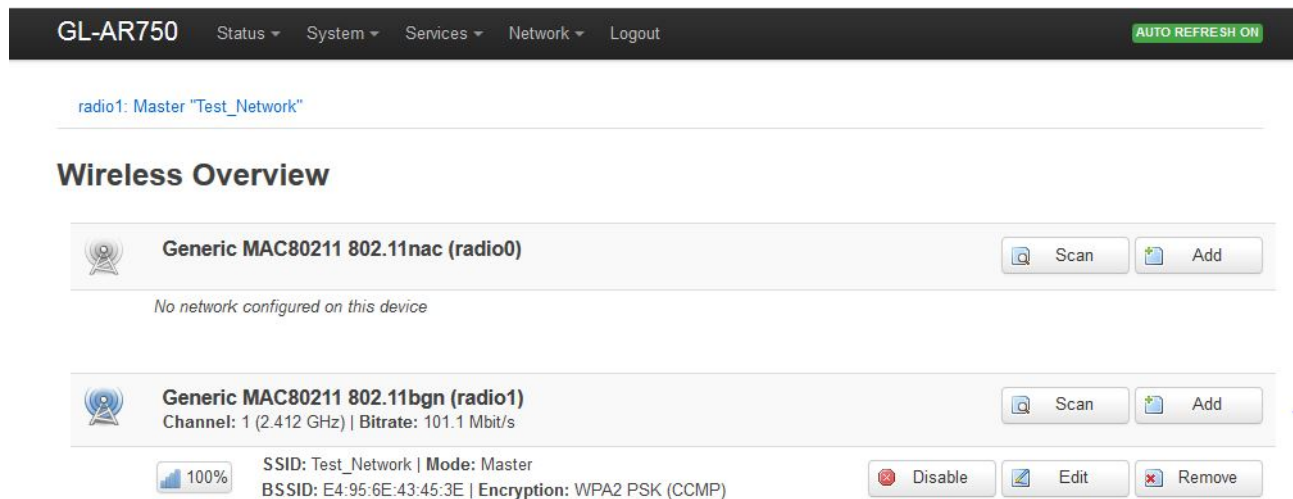


Figure 2.2: Wi-Fi network created

To create this network, you click on add. It will open a new page that you need to configure as such:

GL-AR750
Status
System
Services
Network
Logout
AUTO REFRESH ON

Wireless Network: Master "Test_Network" (wlan1)

The *Device Configuration* section covers physical settings of the radio hardware such as channel, transmit power or antenna selection which are shared among all defined wireless networks (if the radio hardware is multi-SSID capable). Per network settings like encryption or operation mode are grouped in the *Interface Configuration*.

Device Configuration

General Setup
Advanced Settings

Status
100%

Mode: Master | **SSID:** Test_Network
BSSID: E4:95:6E:43:45:3E | **Encryption:** WPA2 PSK (CCMP)
Channel: 1 (2.412 GHz) | **Tx-Power:** 23 dBm
Signal: -33 dBm | **Noise:** -95 dBm
Bitrate: 104.7 Mbit/s | **Country:** US

Wireless network is enabled

Operating frequency

Mode

Channel

Width

N
auto
20 MHz

Transmit Power

auto
dBm

General Setup
Wireless Security
MAC-Filter
Advanced Settings

ESSID

Test_Network

Mode

Access Point

Network

☒ lan:
☐ wan:
☐ wan6:
☐ create:

☒ Choose the network(s) you want to attach to this wireless interface or fill out the *create* field to define a new network.

Hide ESSID

☐

WMM Mode

☒

Figure 2.3: General configuration of the Wi-Fi network

On the device configuration, we want the router to automatically choose the channel. On the interface configuration, we want to:

- choose an SSID
- choose the mode Access Point
- the network is lan

We now go on the wireless security tab and set it up like below:

Interface Configuration

[General Setup](#) [Wireless Security](#) [MAC-Filter](#) [Advanced Settings](#)

Encryption

WPA2-PSK

Cipher

auto

Key

••••••••

802.11r Fast Transition

☐ Enables fast roaming among access points that belong to the same Mobility Domain

802.11w Management Frame Protection

Disabled (default)

Requires the 'full' version of wpa2/hostapd and support from the wifi driver (as of Feb 2017: ath9k and ath10k, in LEDE also mwifi and mt76)

Enable key reinstallation (KRACK) countermeasures

☐ Complicates key reinstallation attacks on the client side by disabling retransmission of EAPOL-Key frames that are used to install keys. This workaround might cause interoperability issues and reduced robustness of key negotiation especially in environments with heavy traffic load.

Figure 2.4: Security parameters

Here we want to create

We are now done and can save the changes. We have successfully created a new Wi-Fi network. We can move on the DHCP setup.

Static DHCP lease

For that part, we go to the Network → DHCP and DNS tab. We go at the bottom of the new page to find the section "Static Leases". In this section, we can add static DHCP leases. We configure it like that:

Static Leases

Static leases are used to assign fixed IP addresses and symbolic hostnames to DHCP clients. They are also required for non-dynamic interface configurations where only hosts with a corresponding lease are served.

Use the *Add* Button to add a new lease entry. The *MAC-Address* identifies the host, the *IPv4-Address* specifies to the fixed address to use and the *Hostname* is assigned as symbolic name to the requesting host. The optional *Lease time* can be used to set non-standard host-specific lease time, e.g. 12h, 3d or infinite.

Hostname	MAC-Address	IPv4-Address	Lease time	IPv6-Suffix (hex)
GLAR150-1	e4:95:6e:43:da:05	192.168.8.11	86400	
GLAR150-4	e4:95:6e:43:d9:24	192.168.8.41	86400	
BRIDGE	e4:95:6e:43:45:16 (192.168.8.2)	192.168.8.2	86400	
GLAR150-5	e4:95:6e:43:da:cd	192.168.8.51	86400	
GLAR150-LAB	e4:95:6E:43:D9:cf	192.168.8.121	86400	

Add

Save & Apply

Save

Reset

Figure 2.5: DHCP static leases

Pressing the button add, we can manually add a static lease. Static lease work using the mac address of the device.

2.2.2 slave router configuration

The slave router is a bit more complex to install. Indeed with a GL-AR750, by default, it is not possible to bridge a Wi-Fi Network to another Wi-Fi network. Thankfully, a linux package lets you simulate this behaviour. This package is relayd. So we are going to install and setup relayd.

Installation of relayd

To install relayd, we need to give the router an access to internet. To do that, you can simply plug the WLAN Ethernet interface to your internet access point or you can plug it to your computer and make your computer share its connection.

Then you connect to the router with ssh (on windows you can use putty). We want to install two packages:

- relayd
- luci-proto-relay

For that you can use the command:

```
opkg install package
```

Once the package installed we are going to set it up through the web interface. We can use this tutorial to help us: <https://wiki.openwrt.org/doc/recipes/relayclient>.

We can connect to the same web page as earlier (but for the slave router): <http://192.168.9.1/cgi-bin/luci/> The global steps are the following:

- Join the network of the main router created earlier
- Set up the Wi-Fi interface
- create the bridge interface with relayd
- update the LAN interface
- Set up the firewall

Joining the network of the main router

The slave server is going to be a client of the main router. For that we head up to Network → wireless. We remove the existing network and we create a new one:

GL-AR750
Status ▾
System ▾
Services ▾
Network ▾
Logout
AUTO REFRESH ON

radio1: Client "Test_Network"
radio1: Master "Test_Network_repeat"

Wireless Network: Client "Test_Network" (wlan1)

The *Device Configuration* section covers physical settings of the radio hardware such as channel, transmit power or antenna selection which are shared among all defined wireless networks (if the radio hardware is multi-SSID capable). Per network settings like encryption or operation mode are grouped in the *Interface Configuration*.

Device Configuration

General Setup
Advanced Settings

Status

Mode: Client | **SSID:** Test_Network
100% **BSSID:** E4:95:6E:43:45:3E | **Encryption:** WPA2 PSK (CCMP)
Channel: 1 (2.412 GHz) | **Tx-Power:** 23 dBm
Signal: -24 dBm | **Noise:** -95 dBm
Bitrate: 130.0 Mbit/s | **Country:** US

Wireless network is enabled

Disable

Operating frequency

Mode

Channel

Width

N
auto
20 MHz

Transmit Power

auto
dBm

Interface Configuration

General Setup
Wireless Security
Advanced Settings

ESSID

Test_Network

Mode

Client

BSSID

E4:95:6E:43:45:3E

Network

☐ lan:
☐ wan:
☐ wan6:
☒ wwan:
☐ create:

Figure 2.6: Joining the main router network

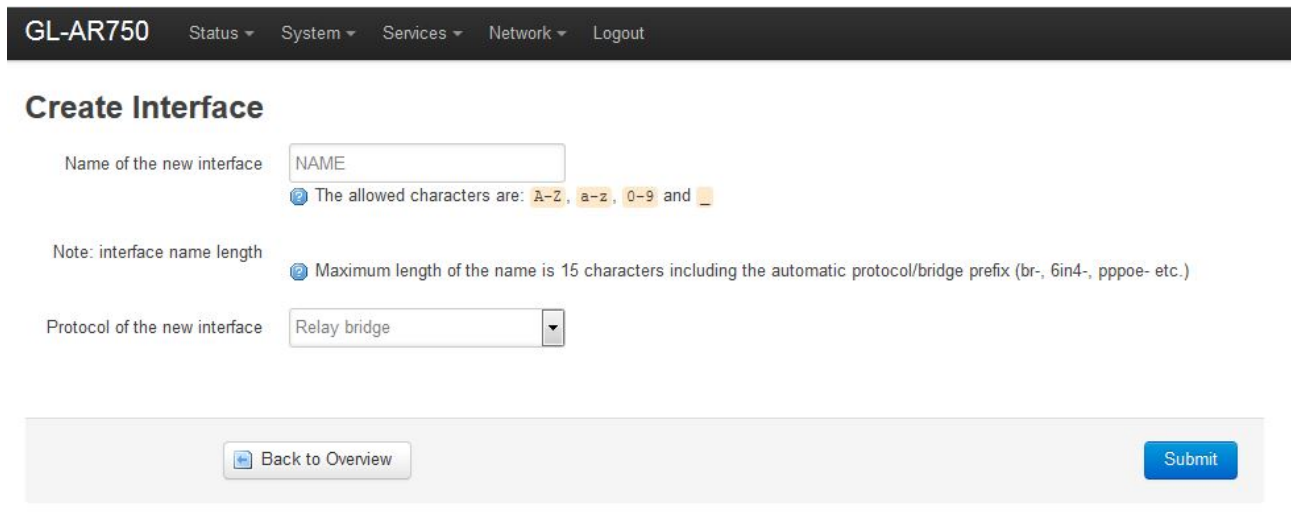
We configure this network in the same way as the network on the main router. The only difference is the mode and the network. Instead of choosing access point, we choose Client. For the network, we create a new one and call it wwan (it will not exist at first, you need to enter it in the create field). We save and apply theses changes.

Create a new Wi-Fi network

We are going to create a new Wi-Fi network. We go to the Network → interfaces and click on edit for the "WWAN" interface. In the "General Setup" we change the protocol to "DHCP Client" (this means that the slave router will automatically get its IP address from the main router).

Create the bridge interface

We go back to the Network → interface tab. Here, we create a new interface by clicking "Add new interface". We set it up like that:



The screenshot shows the 'Create Interface' page in the GL-AR750 web interface. At the top is a navigation bar with 'GL-AR750' and menu items: 'Status', 'System', 'Services', 'Network', and 'Logout'. The main heading is 'Create Interface'. Below it, there are three form fields: 1. 'Name of the new interface' with a text input containing 'NAME' and a tooltip stating 'The allowed characters are: A-Z, a-z, 0-9 and _'. 2. 'Note: interface name length' with a tooltip stating 'Maximum length of the name is 15 characters including the automatic protocol/bridge prefix (br-, 6in4-, pppoe- etc.)'. 3. 'Protocol of the new interface' with a dropdown menu currently set to 'Relay bridge'. At the bottom of the form are two buttons: 'Back to Overview' and 'Submit'.

Figure 2.7: Creation of the bridge interface

You can choose the name you want for this new interface. I personally chose REPEATER_TEST. The important setting is protocol. We want to choose "Relay bridge". This option will only appear if we have installed the package `luci-proto-relay` and the package `relayd`. We validate the creation by submitting it.

Now we go back to the interfaces. A new interface, bearing the name we chose, is present. We are going to edit it like the figure below:

REPEAT_TEST
WAN
LAN
WWAN
WAN6

Interfaces - REPEAT_TEST

On this page you can configure the network interfaces. You can bridge several interfaces by ticking the "bridge interfaces" field and enter the names of several network interfaces separated by spaces. You can also use VLAN notation `INTERFACE.VLANNR` (e.g.: `eth0.1`).

Common Configuration

General Setup
Advanced Settings
Firewall Settings

Status

Relay "repeat_test"

Uptime: 2h 8m 6s
MAC-Address: E4:95:6E:43:45:17
RX: 1.55 MB (11890 Pkts.)
TX: 2.80 MB (14161 Pkts.)

Protocol

Relay bridge

Local IPv4 address

? Address to access local relay bridge

Relay between networks

☒ lan:
☐ wan:
☐ wan6:
☒ wwan:

Back to Overview

Save & Apply
Save
Reset

Figure 2.8: General set up the bridge interface

Here we want to choose the network we want to bridge. We want to bridge the LAN network to the WWAN one. Therefore, every client that connect to the LAN of this router will be bridged to the WWAN network and therefore the main router.

In the "Advanced Settings", we want to make sure that "Forward DHCP traffic" is ticked (it should be ticked by default):

REPEAT_TEST
WAN
LAN
WWAN
WAN6

Interfaces - REPEAT_TEST

On this page you can configure the network interfaces. You can bridge several interfaces by ticking the "bridge interfaces" field and enter the names of several network interfaces separated by spaces. You can also use VLAN notation `INTERFACE.VLANNR` (e.g.: `eth0.1`).

Common Configuration

General Setup
Advanced Settings
Firewall Settings

Bring up on boot ☒

Use builtin IPv6-management ☒

Forward broadcast traffic ☒

Forward DHCP traffic ☒

Use DHCP gateway

Override the gateway in DHCP responses

Host expiry timeout

Specifies the maximum amount of seconds after which hosts are presumed to be dead

ARP retry threshold

Specifies the maximum amount of failed ARP requests until hosts are presumed to be dead

Use routing table

Override the table used for internal routes

Back to Overview

Save & Apply

Save

Reset

Figure 2.9: Advanced settings of the bridge interface

We are now done with the bridge interface and we need to modify the LAN interface.

Update the LAN interface

We go to the Network → Interfaces tab. We want to edit the LAN interface like below:

REPEAT_TEST
WAN
LAN
WWAN
WAN6

Interfaces - REPEAT_TEST

On this page you can configure the network interfaces. You can bridge several interfaces by ticking the "bridge interfaces" field and enter the names of several network interfaces separated by spaces. You can also use VLAN notation `INTERFACE.VLANNR` (e.g.: `eth0.1`).

Common Configuration

General Setup
Advanced Settings
Firewall Settings

Create / Assign firewall-zone

☒ lan: lan: wwan: repeat_test:

☐ wan: wan: wan6:

☐ unspecified -or- create:

Choose the firewall zone you want to assign to this interface. Select *unspecified* to remove the interface from the associated zone or fill out the *create* field to define a new zone and attach the interface to it.

Back to Overview
Save & Apply
Save
Reset

Figure 2.10: General settings of the LAN interface

Here we are going to use a fix address and no DHCP. We want a different address than the one used by the main router.

Set up the firewall

Set up the firewall

2.2.3 remote router configuration

The configuration of the remote router is quite easy. It has 2 steps:

- modify the traffic rules of the router
- modify the configuration of the LAN interface

2.3 Software installation

The software part consist at downloading the sources files and installing them. We want to install 2 differents part:

- a client part for linux
- a server part for openwrt

2.3.1 client installation

The client part is pretty straightforward as you just need to use the makefile by tiping:

```
###: make client
```

It will create two executable files:

- the main one is the `client_shell`. It is the program the user is going to launch.
- the other one is the `client` which is used by the `client_shell` to connect to the server

2.3.2 server installation

For the installation of the server, it's a bit more tricky. Indeed, our servers (GL-AR150) does not use a similar architecture as a computer. Therefore, we cannot directly compile it on the Linux machine. Furthermore, the memory of the GL-AR150 is too small to allow us to install a compiler on it and use the GL-AR150 to compile the server. Hence, we need to do a cross-compilation. It means we are going to install a new compiler on the Linux machine to tell it how to compile the server for a GL-AR150.

I will now explain how to do so for a GL-AR150. However, the following instructions are only valid for a GL-AR150 (or maybe any router using a AR71XXX architecture).e Depending on your router you will need to find the adequate cross-compiler.

Cross-compilation of the server

For the cross-compilation we first need to download the right SDK (Software Development Kit) for our router. In our case we want the SDK for a LEDE router with an ar71xxx architecture. We have 2 ways of getting the cross-compiler:

- build it using the git repository for lede
- download a pre-built SDK

I have used the first method. I will now explain how to build the cross-compiler.

First, clone the lede github project using the following command:

```
###: git clone https://github.com/lede-project/source
```

Once you have download the repository, go inside it and type:

```
###: make menuconfig
```

This command will open a terminal (see figure below) where you can configure what make will do.

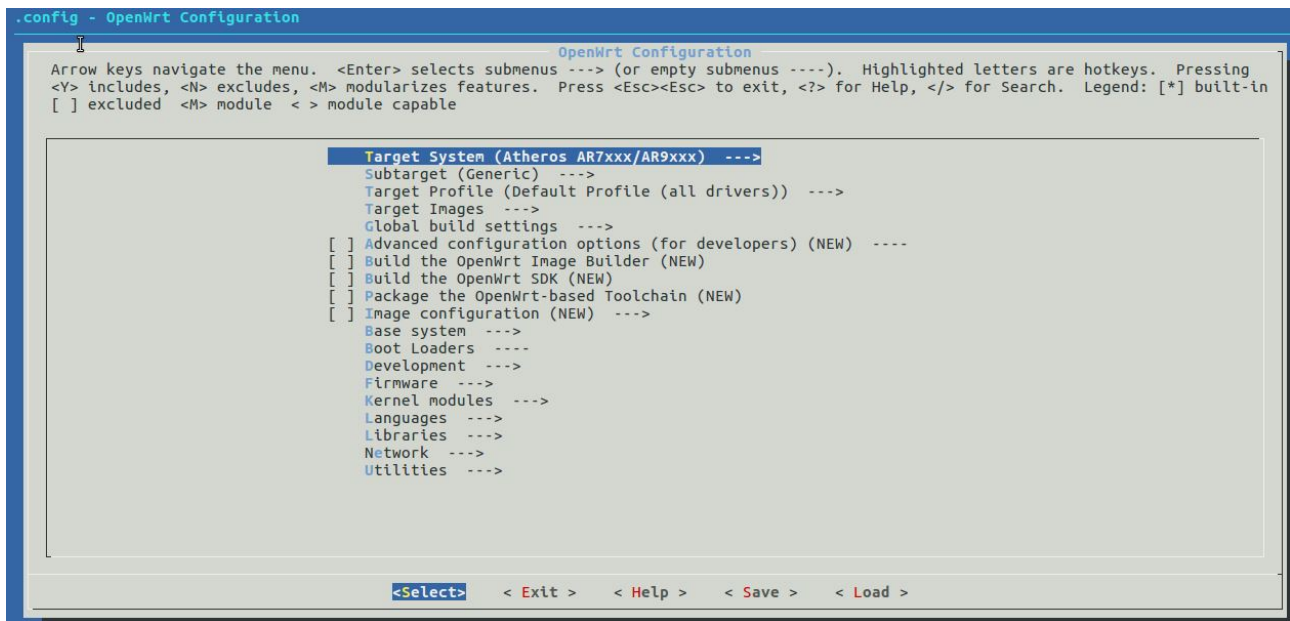


Figure 2.11: Menuconfig interface

We are going to change a few things with this menu: the target system, the target profile, and the SDK. First, we are going to put the target system to its right value. In our case, we are using a GL-AR150. Therefore, we need to put:

```
'Atheros AR71xxx/AR9xxx'
```

If you are using a different router, you may have to choose another option.

Then, we are going to select GL-AR150 in the target profile.

Finally, we want to tick the box for 'Build the LEDE SDK' (or openwrt if your router is not using LEDE as an os).

We know save these changes and exit the menuconfig.

Now, we can finally build the SDK using the command:

```
###: make
```

This command may take a long time depending the power of your computer. For instance, on my Linux virtual machine (with 4GB of RAM and a laptop cpu) it took several hours to finish. Once done, you will have a lot of new files. We are only interested in one repository:

```
staging_dir/toolchain-mips-24kc-gcc-5.4.0-musl-1.1.16
```

The name of the directory may change a bit but it will always start by toolchain. So if you go to the 'bin' directory inside the toolchain, you should find the following file:

```
staging_dir/toolchain...bin/mips-openwrt-linux-musl-gcc
```

This executable is a c compiler for our router. We can use it on the linux computer to compile a c file. It will create an executable that can run on the router.

We know have our compiler. To use it we need to do the following steps:

- export the bin directory in the PATH environment variable
- export a new environment variable called STAGING_DIR (it will be used by the compiler
- precise to the makefile of the server where to find the compiler

Let's start by creating the environment variables. In your main repository, there is a file called .bashrc. This file is executed each time you start a shell. We are going to add two lines at the end of this file:

```
PATH=$PATH:~/staging_dir/toolchain.../bin/
export STAGING_DIR=~/staging_dir/toolchain.../
```

Now we only need to slightly change the makefile. Go to the repository containing the sources. In the makefile, there is at the top of the file a variable called `SERVERCC`. This variable precise the compiler used for the `server.c` file. To compile it for the router, we just need to change its value to the correct gcc: `mips-openwrt-linux-musl-gcc`. If we later want to compile the `server.c` file for the linux computer (to do some testing), we only need to change back this variable to `gcc`.

Now we can create the server exec file by doing:

```
###: make server
```

WARNING:

The server and the client have two mode `DEBUG` and `normal`. To change the mode of functioning, you have to change the value of a line in their respective `c` files. The line to change is:

```
#define DEBUG 1
```

If the value of this line is 0, it means you are in `DEBUG` mode. Make sure to put any other number than 0 before doing make.

Adding the server to the router

We now have an executable called 'server'. In this part, we will:

- add this file to the router
- automatize its start

To add the file to the router, we just use `scp` (which is a copy function over a `ssh` connection) to copy the executable anywhere in the router.

Now we are going to automatize the server. We want two things:

- the server start during the boot
- if the server crash, a script is launching it again

First, let's start by writing the script that will start the server at the boot:

```
#!/bin/sh /etc/rc.common
```

```
# The order you want these scripts to be run.
```

```
# e.g Start=6 will run after scripts with Start=5 but before Sta
START=6
```

```
STOP=12
```

```
# The command(s) you want to run on start
```

```
start() {
```

```

        echo starting
        PATH_of_server_executable
    }

# The command(s) you want to run on stop
stop() {
    echo stopping
}

```

Now we want to restart the server if he crash. To do achieve that, we are going to use crontab and the following script.

```

#!/bin/sh
result=`netstat -an|grep 8800| grep LISTEN`
if [ -z "$result" ]
then
    /path_to_server
fi

```

We will execute this script every 10 minutes. By default cron is disable on lede. We can enable it with the following command:

```
/etc/init.d/cron enable
```

We can now run cron:

```
crontab -e
```

It will open a file with vi. On this file we want to add the following line

```
*/10 * * * * path_to_script
```

If you want to adjust how often it repeat, here is how crontab work. The line is composed of the following elements minute hour day(month) year day(week) command_to_execute

For the first 5 value, there are special characters:

- * represents any value
- , is a value line separator
- / is used for step values
- - is a range of value

Chapter 3

How to use it?

Once the installation steps are done, using the client is quite easy. You simply need to launch the `client_shell` in a terminal using the command:

```
###: ./client_shell
```

It will launch a shell-like interface. In this interface, you can input some commands to control the client.

We have two type of command:

- built in functions: functions written in c to control the system
- classic shell command

We are mainly interested in the built in functions. The available commands are:

- `help`: list the available built in functions
- `exit`: close the `client_shell`
- `launch`: launch a client
- `findServers`: update the list of available servers
- `displayServers`: display the available servers
- `close`: close a client
- `list`: list the clients currently running
- `log`: start or stop logging the data for one client

If you type a function but do not put the right parameters, the shell will display a short message explaining how to use the function properly.

Here is an example of process you want to do when starting the `client_shell` for the first time: First we want to find the available servers. So we type:

```
displayServers
```

We get a list of the available servers. Each server has a number in front of it. We can use this number to start a client who will communicate with this server.

```
launch 0
```

A new window pops up. This is the client we just created. We can start another client on another server.

```
launch 1
```

Now we want to list the clients running. So we type:

```
list
```

We get a list of the clients running. Each client has a number in front of it. We can use this number to send messages to the client.

For instance, we may want to log the data of the first client in a file called "Log_UHF".

```
log 0 Log_UHF
```

The client with the index 0 will start to log the data in the file Log_UHF.

After a while, we think we have enough data and we want to stop logging them.

```
log 0 stop
```

With the same command log we can stop it. However, it means stop can't be the filename.

Finally we want to shutdown all the clients.

```
close all
```

This will close all the clients and their connections. If we wanted to shutdown only one client, we just have to replace "all" by the number of the client.

Chapter 4

Unfinished parts

4.1 Wi-Fi monitoring

I wanted to integrate some c code for this part but I unfortunately didn't have the time to do it. Right now, we have to use a temporary solution with the package tcpdump and wireshark (a graphic network tool to analyse messages going through the network).

tcpdump is a Linux package that can be installed on the routers. This package allow an user to scan all the messages going through an interface. Therefore, using this command, we can analyse the Wi-Fi communication going around the router. We can install this package on the remote router.

Then on the computer in the lab, you simply have to use the following command:

```
ssh root@192.168.8.11 tcpdump -i wlan0 -U -s0 -w - 'not port 22'
... sudo wireshark -k -i -
```

With this command, we launch the command tcpdump on the remote router. It will analyse the Wi-Fi interface called wlan0. Then the result of the tcpdump command is sent to our computer and displayed through wireshark.

4.2 Phone

I managed to remotely control a phone using virtual machines. I had the phone plugged into one virtual machine, and I was controlling the phone in the other virtual machine.

All the component used for this solution were supposed to be available on the small routers. I was hoping it would work smoothly. However, when I tried it on the small routers it didn't work. I still have a proof of concept on virtual machines. It should be possible to make it work on the small router even if it needs to be a little tweaked.

The solution is based on:

- adb (android debug bridge) and scrcpy on the computer

- usbip on the small router

scrcpy is a free open software that use adb to give the user a full control of his phone through a computer. When you launch scrcpy, it will reproduce the screen of your phone on the computer and you can use it. There is only one issue with scrcpy. It is a very heavy software and can therefore not run on the small routers. The solution I came up with was to make scrcpy work on the computer. scrcpy will display the screen of your phone as long as adb can reach it. adb has two ways to reach a phone: through a usb connection or through the network. So I wanted to use the usb connection since the phone will be connected by Wi-Fi to the Mesh Extender and by usb to the remote router. On Linux, there is a package called usbip. This package can make a usb connection work over the network. So we have the phone connected to the remote router using usb. The router transfers this connection to the computer using the network. The computer can use scrcpy to control the phone.

4.2.1 How to install scrcpy

First, let's start by installing the required package:

```
apt install ffmpeg libsdl2-2.0.0
apt install make gcc pkg-config meson libavcodec-dev
        libavformat-dev libavutil-dev libsdl2-dev
apt install openjdk-8-jdk
```

Now we need to install android studio:

- Download the compressed file on their website
- Extract the files
- Goto bin folder inside the extracted files
- Launch the studio.sh file: ./studio.sh
- It will open the installer, follow the instruction

Activate sdk licenses:

- Go to /Android/Sdk/tools/bin
- ./sdkmanager --licenses

Create the Android SDK environment variable:

```
export ANDROID_HOME=/Android/Sdk
```

Watch out, the path could be /android/sdk instead of /Android/Sdk .

Clone and install scrcpy:

```
...:# git clone https://github.com/Genymobile/scrcpy

...:# cd scrcpy/
...:scrcpy# mkdir x
...:scrcpy# meson x --buildtype release --strip -Db_lto=true
The Meson build system
Version: 0.45.1
Source dir: .../scrcpy
Build dir: .../scrcpy/x
Build type: native build
Project name: scrcpy
Native C compiler: cc (...)
Build machine cpu family: x86_64
Build machine cpu: x86_64
Found pkg-config: /usr/bin/pkg-config (0.29.1)
Native dependency libavformat found: YES 57.83.100
Native dependency libavcodec found: YES 57.107.100
Native dependency libavutil found: YES 55.78.100
Native dependency sdl2 found: YES 2.0.8
Configuring config.h using configuration
Program ./scripts/build-wrapper.sh found: YES (...)
Build targets in project: 6
Found ninja-1.8.2 at /usr/bin/ninja
```

Chapter 5

Resources used

In this part, you will find every useful references I used to create the test network. If you want to have a better understanding on the functioning of the solution, you can take a look at them

5.1 Network

Source for relayd: <https://wiki.openwrt.org/doc/recipes/relayclient>
DNS and DHCP: <https://wiki.openwrt.org/doc/uci/dhcp>

5.2 server

function poll: https://www.ibm.com/support/knowledgecenter/en/ssw\i5_54/rzab6/poll.htm
cross-compilation: <https://manoftoday.wordpress.com/2007/10/11/writing-and-compiling-a-simple-program-for-openwrt/> cross-compilation: <https://github.com/airplug/airplug/wiki/Cross-compiling-for-OpenWRT>

5.3 client

signal handling: <https://stackoverflow.com/questions/5546223/signals-received-by-bash-when-terminal-is-closed>

5.4 client_shell

tutorial to write a shell in C: <https://brennan.io/2015/01/16/write-a-shell-in-c/> customise xterm: <https://scarygliders.net/2011/12/01/customize-xterm-the-original-and-best-terminal/>

5.5 Phone

usb over ip: <https://wiki.openwrt.org/doc/howto/usb.ip tunnel> scrcpy: <https://github.com/Genymobile/scrcpy> FIFO: <https://www.tldp.org/LDP/lpg/node11.html>

5.6 Other useful references

Boot structure of openwrt: <https://medium.com/openwrt-iot/lede-openwrt-boot-structure-e689c4ddea91> Scheduling tasks with cron on openwrt: <https://medium.com/openwrt-iot/openwrt-scheduling-tasks-6e19d507ae45>

List of Figures

1.1	Mesh Extender	3
1.2	Routers used for the testbed	3
1.3	Other devices needed	4
1.4	Client and server communication	5
1.5	Client and Client_shell interaction	6
2.1	Bridge network using relayd (source: https://wiki.openwrt.org/doc/recipes/relayd)	9
2.2	Wi-Fi network created	10
2.3	General configuration of the Wi-Fi network	11
2.4	Security parameters	11
2.5	DHCP static leases	13
2.6	Joining the main router network	14
2.7	Creation of the bridge interface	15
2.8	General set up the bridge interface	16
2.9	Advanced settings of the bridge interface	17
2.10	General settings of the LAN interface	19
2.11	Menuconfig interface	