WebLab-Deusto Documentation

Release 5.0

WebLab-Deusto authors

Oct 23, 2020

Contents

Table of Contents

CHAPTER 1

General

This section provides a general overview of the project.

1.1 Summary

WebLab-Deusto is an Open Source (BSD 2-clause license) remote laboratory management system developed in the University of Deusto. A remote laboratory is a software and hardware solution that enables students to access equipment which is physically located in a university, secondary school or research centre. There are many types of remote laboratories (for physics, chemistry, electronics…). What WebLab-Deusto does is:

1. provide a set of APIs to develop new remote laboratories.
2. maintain remote laboratories developed on top of WebLab-Deusto: manage users, permissions, user tracking, scheduling, etc.
3. share remote laboratories developed on top of WebLab-Deusto: let other universities or secondary schools use your laboratories.
4. use remote laboratories provided by other universities (such as the University of Deusto).

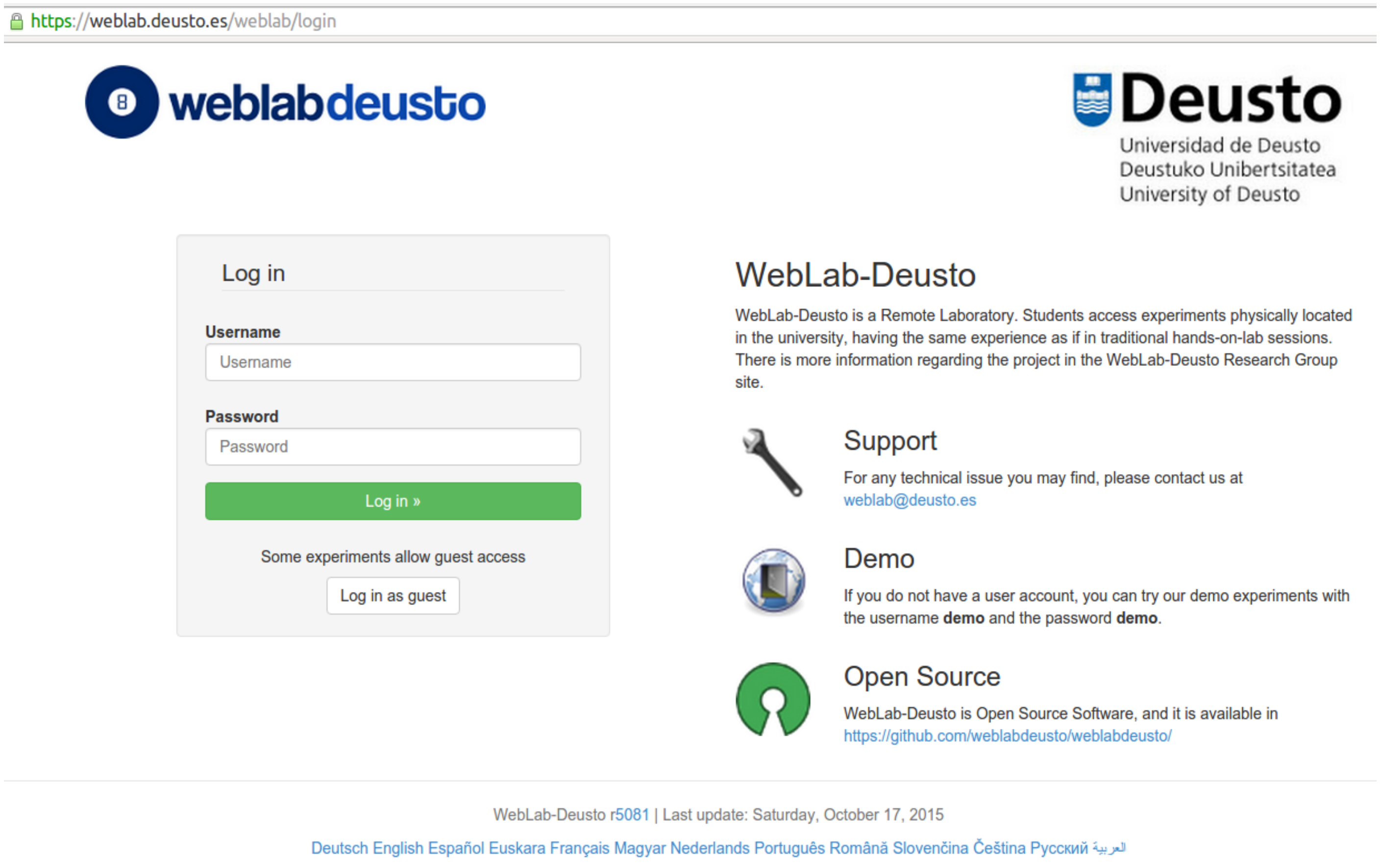
If you want to see examples of running laboratories, try the demo version at:

> https://weblab.deusto.es/weblab/

1.2 Screenshots

Table of Contents

1.2.1 User interface



https://weblab.deusto.es/weblab/login

weblabdeusto

Deusto
Universidad de Deusto
Deustuko Unibertsitatea
University of Deusto

Log in

Username
Username

Password
Password

Log in »

Some experiments allow guest access

Log in as guest

WebLab-Deusto

WebLab-Deusto is a Remote Laboratory. Students access experiments physically located in the university, having the same experience as if in traditional hands-on-lab sessions. There is more information regarding the project in the WebLab-Deusto Research Group site.

Support

For any technical issue you may find, please contact us at weblab@deusto.es

Demo

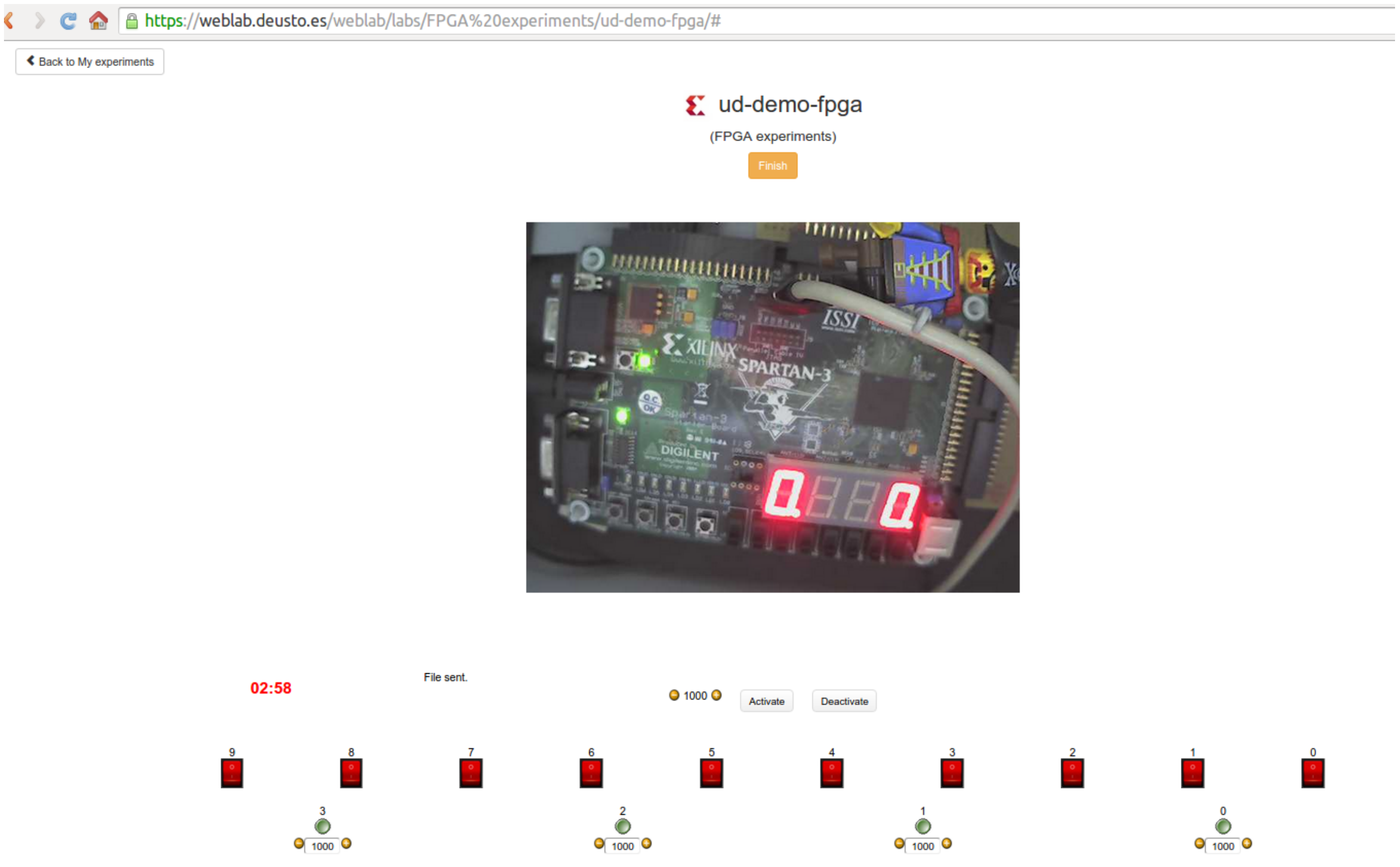
If you do not have a user account, you can try our demo experiments with the username **demo** and the password **demo**.

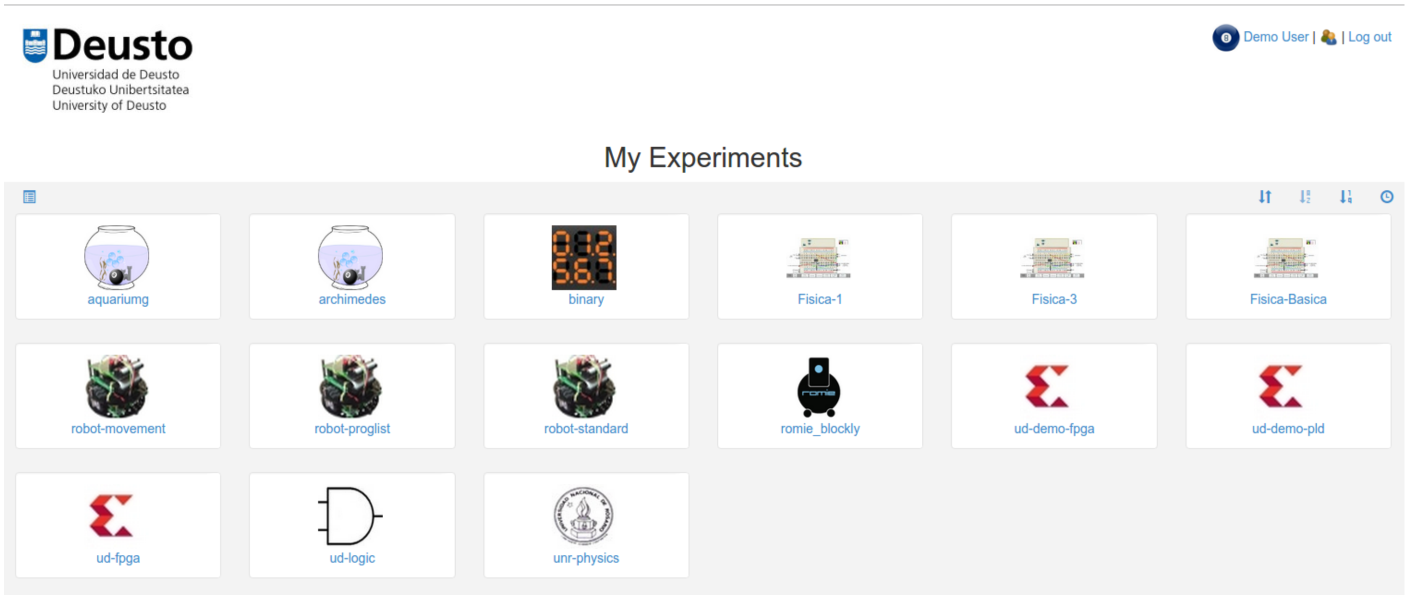
Open Source

WebLab-Deusto is Open Source Software, and it is available in https://github.com/weblabdeusto/weblabdeusto/

WebLab-Deusto r5081 | Last update: Saturday, October 17, 2015

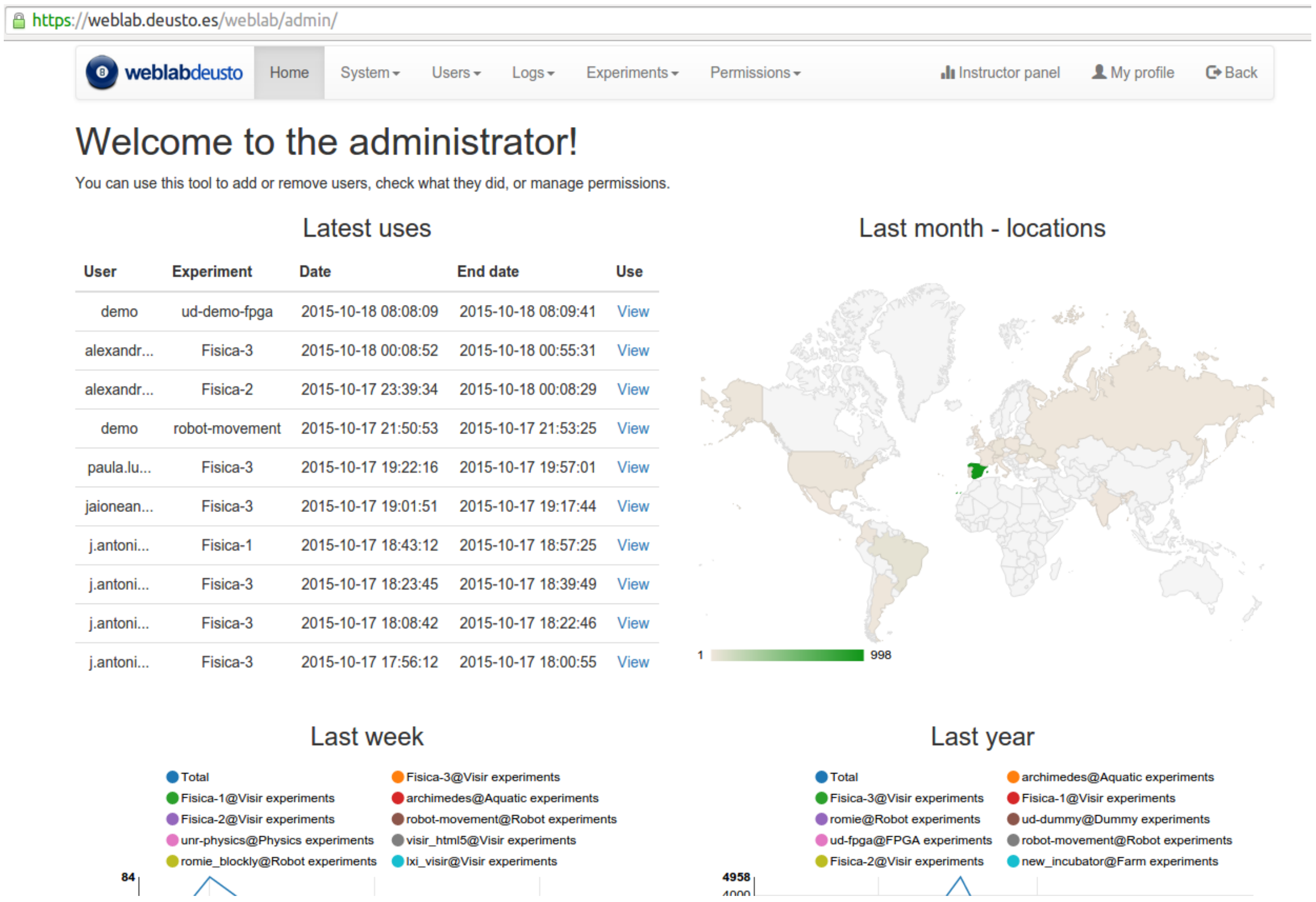
Deutsch English Español Euskara Français Magyar Nederlands Português Română Slovenčina Čeština Русский العربية


https://weblab.deusto.es/weblab/labs/FPGA%20experiments/ud-demo-fpga/#
Back to My experiments
ud-demo-fpga
(FPGA experiments)
Finish
02:58
File sent.
1000
Activate
Deactivate
9 8 7 6 5 4 3 2 1 0
3 2 1 0
1000


Deusto
Universidad de Deusto
Deustuko Unibertsitatea
University of Deusto
Demo User | | Log out
My Experiments
aquariumg
archimedes
binary
Fisica-1
Fisica-3
Fisica-Basica
robot-movement
robot-proglist
robot-standard
romie_blockly
ud-demo-fpga
ud-demo-pld
ud-fpga
ud-logic
unr-physics

1.2.2 Tools

Administration panel



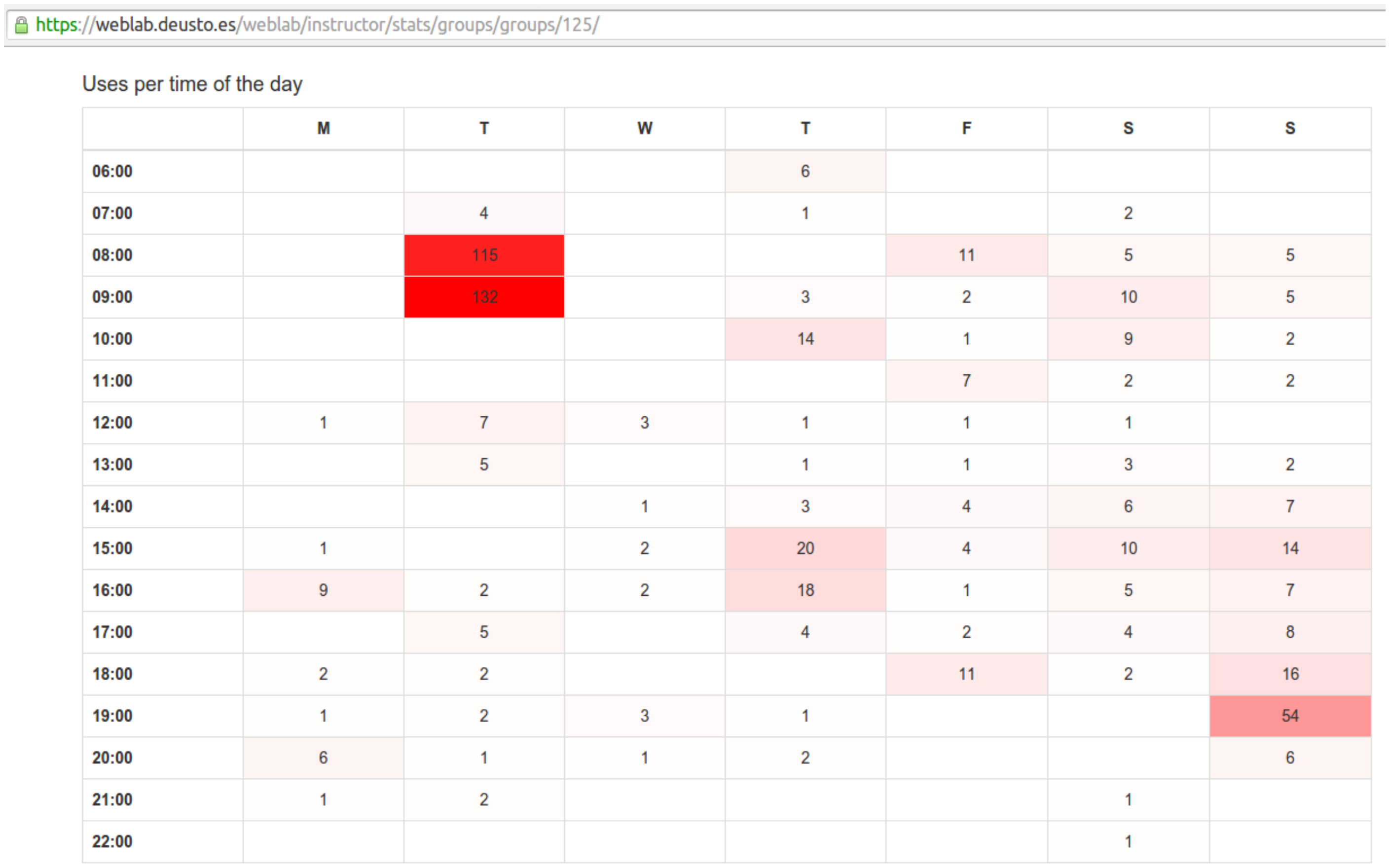
Welcome to the administrator!

You can use this tool to add or remove users, check what they did, or manage permissions.

Latest uses

User	Experiment	Date	End date	Use
demo	ud-demo-fpga	2015-10-18 08:08:09	2015-10-18 08:09:41	View
alexandr...	Fisica-3	2015-10-18 00:08:52	2015-10-18 00:55:31	View
alexandr...	Fisica-2	2015-10-17 23:39:34	2015-10-18 00:08:29	View
demo	robot-movement	2015-10-17 21:50:53	2015-10-17 21:53:25	View
paula.lu...	Fisica-3	2015-10-17 19:22:16	2015-10-17 19:57:01	View
jaionean...	Fisica-3	2015-10-17 19:01:51	2015-10-17 19:17:44	View
j.antoni...	Fisica-1	2015-10-17 18:43:12	2015-10-17 18:57:25	View
j.antoni...	Fisica-3	2015-10-17 18:23:45	2015-10-17 18:39:49	View
j.antoni...	Fisica-3	2015-10-17 18:08:42	2015-10-17 18:22:46	View
j.antoni...	Fisica-3	2015-10-17 17:56:12	2015-10-17 18:00:55	View

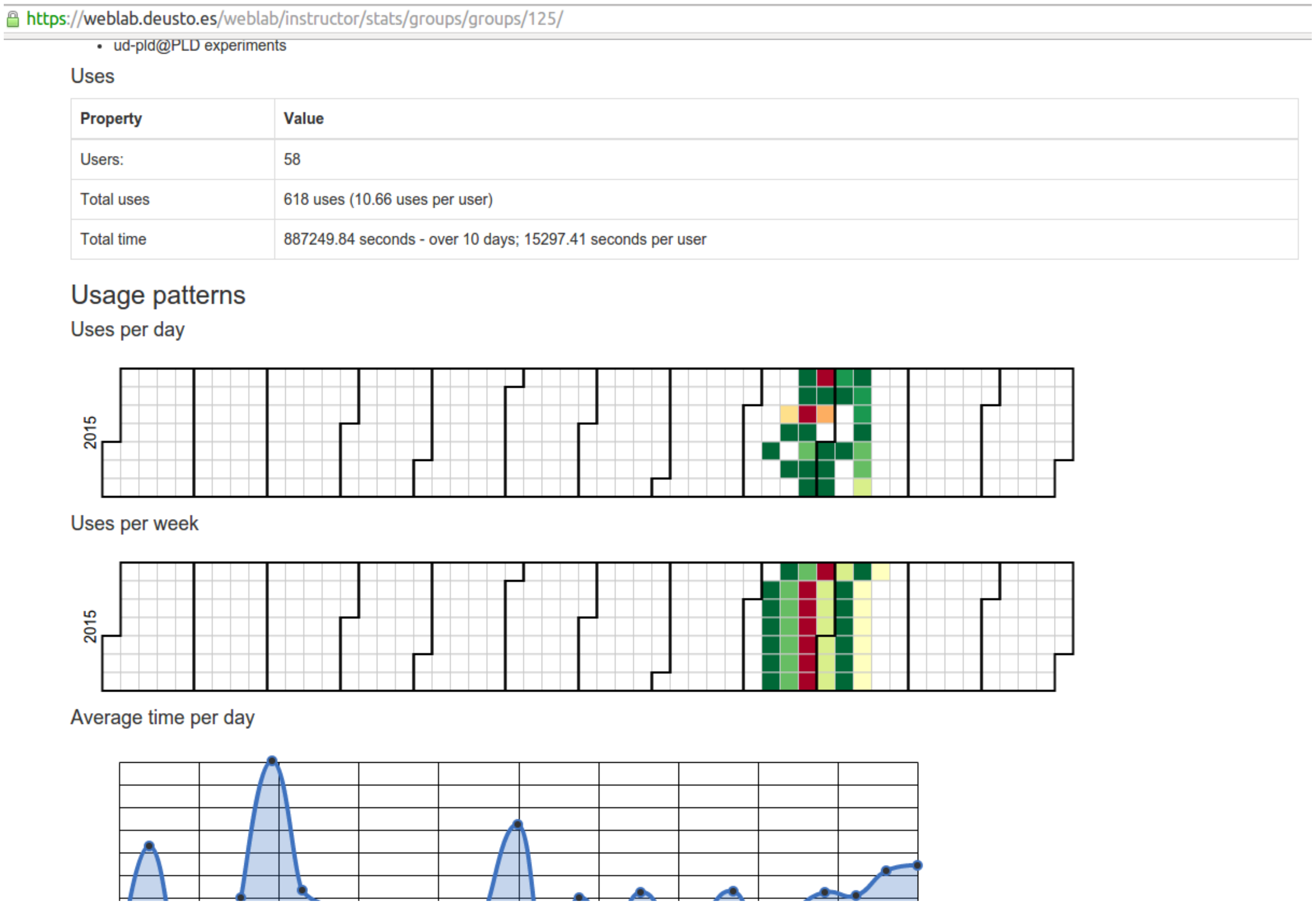
Last month - locations
1
998
Last week
Total
Fisica-3@Visir experiments
Fisica-1@Visir experiments
archimedes@Aquatic experiments
Fisica-2@Visir experiments
robot-movement@Robot experiments
unr-physics@Physics experiments
visir_html5@Visir experiments
romie_blockly@Robot experiments
lxi_visir@Visir experiments
84
Last year
Total
archimedes@Aquatic experiments
Fisica-3@Visir experiments
Fisica-1@Visir experiments
romie@Robot experiments
ud-dummy@Dummy experiments
ud-fpga@FPGA experiments
robot-movement@Robot experiments
Fisica-2@Visir experiments
new_incubator@Farm experiments
4958
4000



https://weblab.deusto.es/weblab/instructor/stats/groups/groups/125/

Uses per time of the day

	M	T	W	T	F	S	S
06:00				6			
07:00		4		1		2	
08:00		115			11	5	5
09:00		132		3	2	10	5
10:00				14	1	9	2
11:00					7	2	2
12:00	1	7	3	1	1	1	
13:00		5		1	1	3	2
14:00			1	3	4	6	7
15:00	1		2	20	4	10	14
16:00	9	2	2	18	1	5	7
17:00		5		4	2	4	8
18:00	2	2			11	2	16
19:00	1	2	3	1			54
20:00	6	1	1	2			6
21:00	1	2				1	
22:00						1	



https://weblab.deusto.es/weblab/instructor/stats/groups/groups/125/

- ud-pld@PLD experiments

Uses

Property	Value
Users:	58
Total uses	618 uses (10.66 uses per user)
Total time	887249.84 seconds - over 10 days; 15297.41 seconds per user

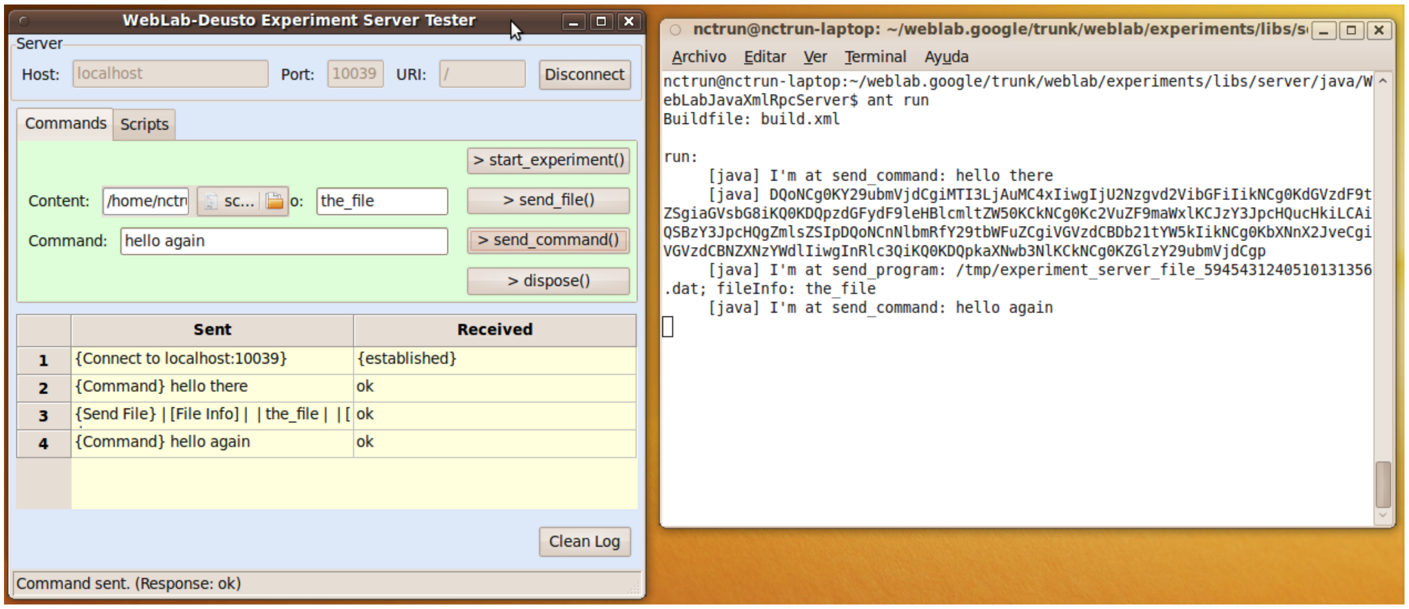
Usage patterns

Uses per day

Uses per week

Average time per day

Server tester



Content: /home/nctr sc... o: the_file

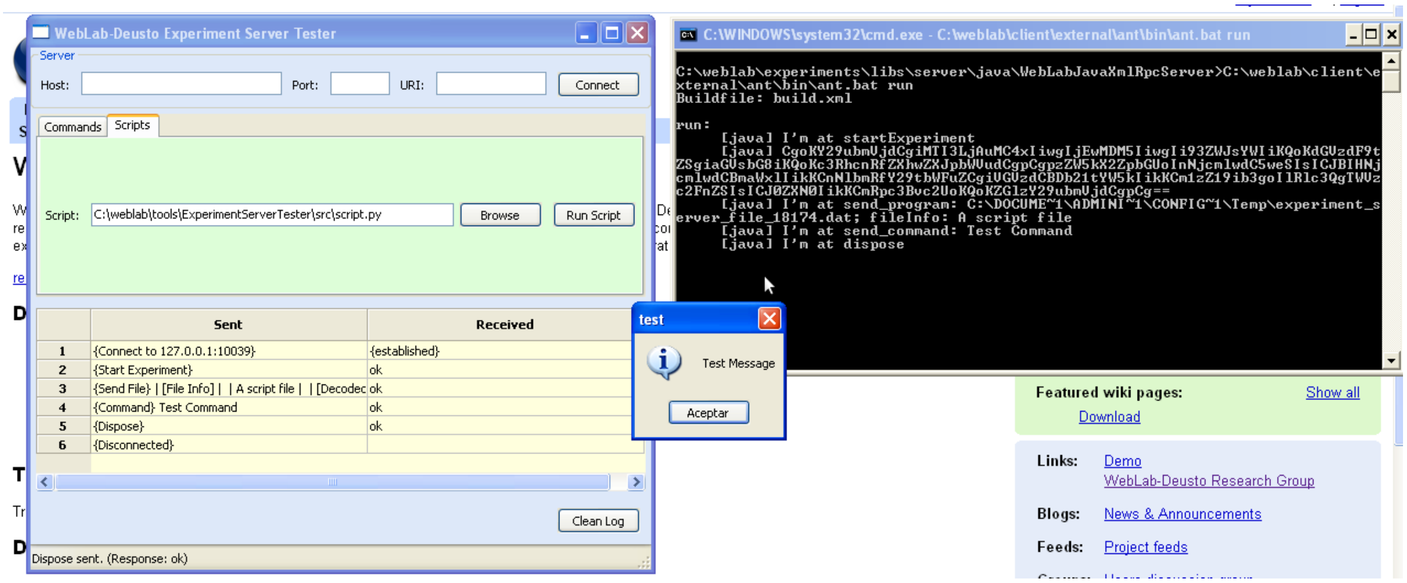
Command: hello again

	Sent	Received
1	{Connect to localhost:10039}	{established}
2	{Command} hello there	ok
3	{Send File} | [File Info] | | the_file | | [	ok
4	{Command} hello again	ok

Command sent. (Response: ok)

nctrun@nctrun-laptop: ~/weblab.google/trunk/weblab/experiments/libs/s

```
nctrun@nctrun-laptop:~/weblab.google/trunk/weblab/experiments/libs/server/java/W
ebLabJavaXmlRpcServer$ ant run
Buildfile: build.xml

run:
     [java] I'm at send_command: hello there
     [java] DQoNCg0KY29ubmVjdCgiMTI3LjAuMC4xIiwgIjU2Nzgvd2VibGFiIikNCg0KdGVzdF9t
ZSgiaGVsbG8iKQ0KDQpzdGFydF9leHBlcmltZW50KCkNCg0Kc2VuZF9maWxlKCJzY3JpcHQucHkiLCAi
QSBzY3JpcHQgZmlsZSIpDQoNCnNlbmRfY29tbWFuZCgiVGVzdCBDb21tYW5kIikNCg0KbXNnX2JveCgi
VGVzdCBNZXNzYWdlIiwgInRlc3QiKQ0KDQpkaXNwb3NlKCkNCg0KZGlzY29ubmVjdCgp
     [java] I'm at send_program: /tmp/experiment_server_file_594543124051013135б
.dat; fileInfo: the_file
     [java] I'm at send_command: hello again
```



WebLab-Deusto Experiment Server Tester
Server
Host:
Port:
URI:
Connect
Commands
Scripts
Script:
Browse
Run Script
Clean Log
test
Test Message
Aceptar
C:\WINDOWS\system32\cmd.exe - C:\weblab\client\external\ant\bin\ant.bat run

Script: C:\weblab\tools\ExperimentServerTester\src\script.py

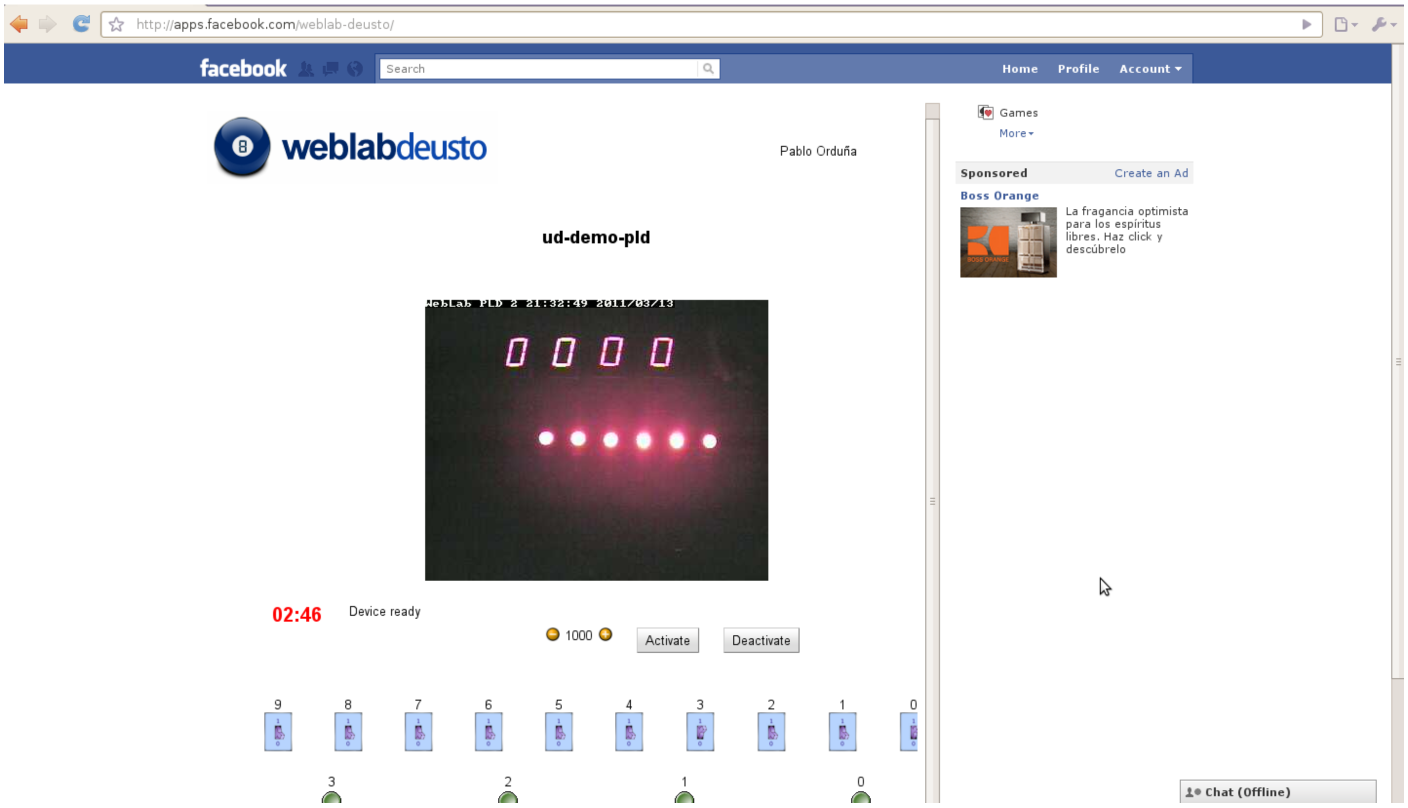
	Sent	Received
1	{Connect to 127.0.0.1:10039}	{established}
2	{Start Experiment}	ok
3	{Send File} | [File Info] | | A script file | | [Decodec	ok
4	{Command} Test Command	ok
5	{Dispose}	ok
6	{Disconnected}	

Dispose sent. (Response: ok)

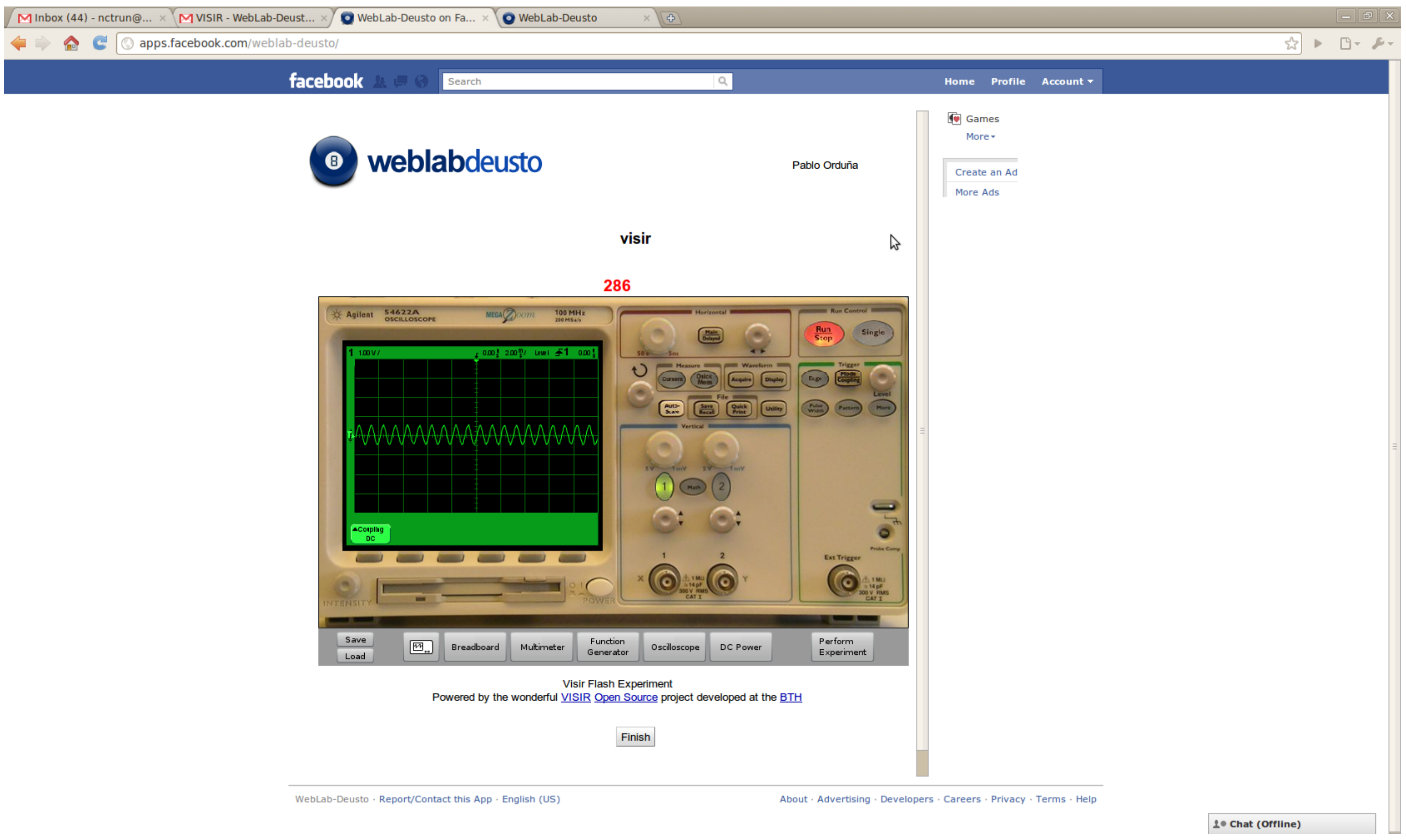
```
C:\weblab\experiments\libs\server\java\WebLabJavaXmlRpcServer>C:\weblab\client\e
xternal\ant\bin\ant.bat run
Buildfile: build.xml

run:
     [java] I'm at startExperiment
     [java] CgoKY29ubmVjdCgiMTI3LjAuMC4xIiwgIjEwMDM5IiwgIi93ZWJsYWIiKQoKdGVzdF9t
ZSgiaGVsbG8iKQoKc3RhcnRfZXhwZXJpbWVudCgpCgpzZW5kX2ZpbGUoInNjcmlwdC5weSIsICJBIHNj
cmlwdCBmaWxlIikKCnNlbmRfY29tbWFuZCgiVGVzdCBDb21tYW5kIikKCm1zZ19ib3goIlRlc3QgTWVz
c2FnZSIsICJ0ZXN0IikKCmRpc3Bvc2UoKQoKZGlzY29ubmVjdCgpCg==
     [java] I'm at send_program: C:\DOCUME~1\ADMINI~1\CONFIG~1\Temp\experiment_s
erver_file_18174.dat; fileInfo: A script file
     [java] I'm at send_command: Test Command
     [java] I'm at dispose
```

1.2.3 Integrations

Facebook (OAuth 2.0)



VISIR



Visir Flash Experiment
Powered by the wonderful VISIR Open Source project developed at the BTH

1.3 Sample laboratories

This section presents the a set of sample laboratories available in the University of Deusto.

Note: The terms *laboratory*, *experiment* or *rig* are a common problem in the remote laboratories literature. We will use laboratory or experiment identically in this document. But take into account that in the case of the CPLD there is a single laboratory but there are two copies (commonly called *rigs*) of them, and students are balanced among them. But on the Robot laboratory, there are three different *laboratories* running on the same single *rig*. This way, WebLab-Deusto separates *resources* (*rigs*) from *laboratories*.

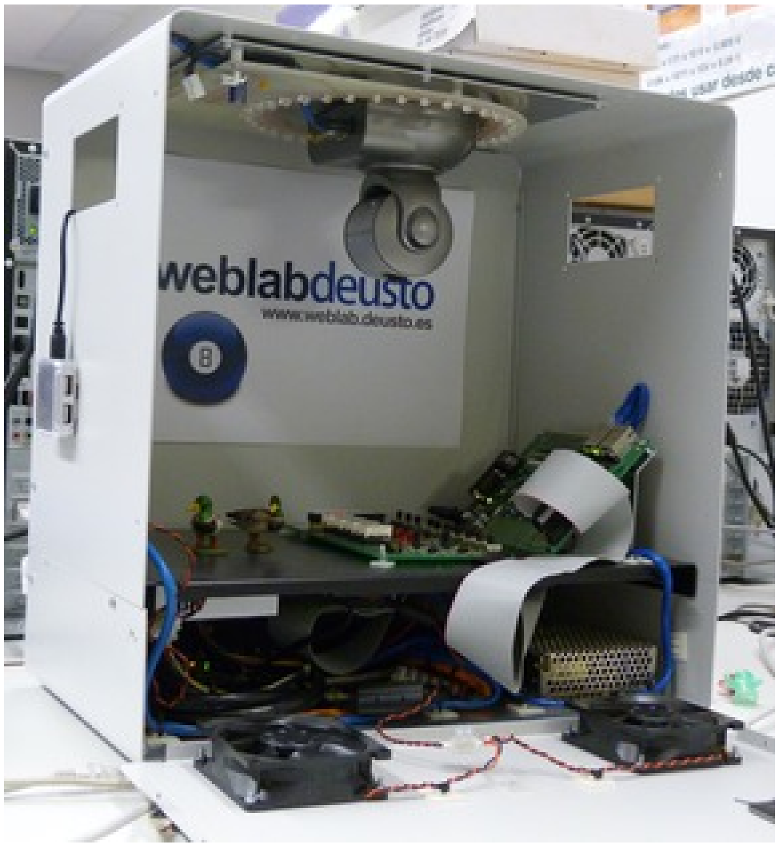
Table of Contents

1.3.1 FPGA

ud-fpga lets you remotely practise with a Field Programmable Gate Array. Through the Xilinx software, you can write a FPGA program locally as you normally would. Once the program is compiled, and ready to be tested, you should simply upload the binary ".bit" file through the experiment.

ud-fpga will automatically program the FPGA board with the binary you provided, and start running it. To see the results, a Webcam is of course provided. You may also interact with the board remotely, by using the provided widgets. Though the widgets themselves might appear artificial, they will send a signal to the board just like their physical counterparts would.

However, due to certain safety concerns, in the demo version you can't upload your own file for this demo. Instead, a specific demo program (which has already been uploaded) will be used. Everything else will work as in the standard FPGA experiment.



The FPGA laboratory, as other WebLab-Deusto laboratories (PIC or CPLD), is developed within the WebLab-Box. On the WebLab-Box, the device, as well as a fit-pc, a PIC microcontroller, a camera, lighting system and networking materials is installed, so as to make it easier to create and deploy new laboratories.

- **Target audience:** *Electronics Engineering students*.

1.3.2 CPLD

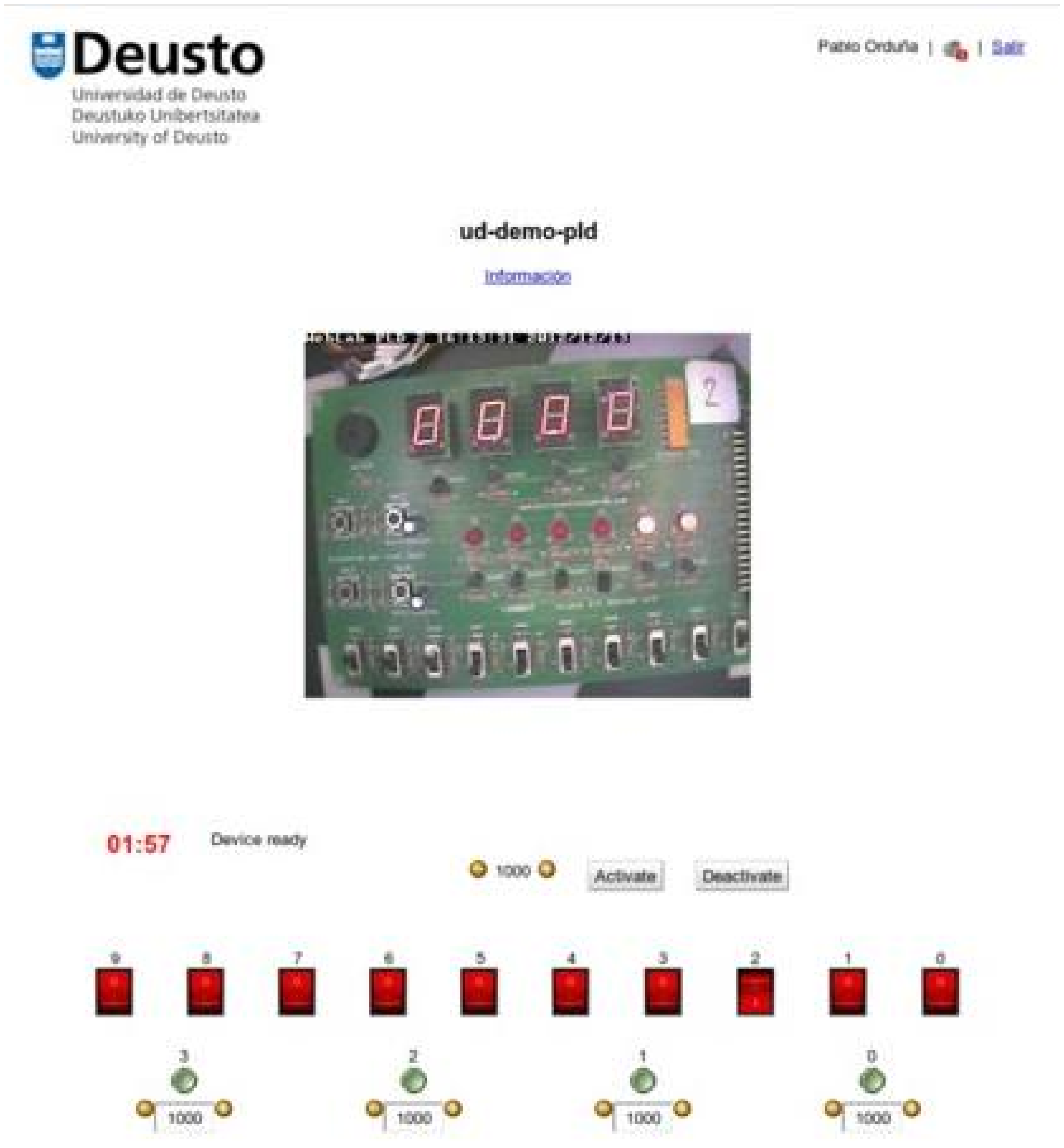
ud-demo-pld lets you remotely practise with a Programmable Logic Device.

With the standard PLD experiment, through the Xilinx software you can write a PLD program locally as you normally would. Once the program is compiled, and ready to be tested, you can simply upload the binary ".jed" file, and it will be programmed on the physical board and run.

However, due to certain safety concerns, you can't upload your own file for this demo. Instead, a specific demo program (which has already been uploaded) will be used. Everything else will work as in the standard FPGA experiment.

That binary file will be automatically programmed into the board, and it will start running. To see the results, a Webcam is provided. You may also interact with the board remotely, by using the provided widgets. Though the widgets themselves might appear artificial, they will send a signal to the board just like their physical counterparts would.

As the FPGA, the CPLD laboratory is running in the WebLab-Box. However, two different laboratories are available. The queue of students is balanced between both copies, so it goes twice faster.



- **Target audience:** *Electronics Engineering students*.
- **Video:** http://www.youtube.com/watch?v=zON7oYtssrw

1.3.3 Aquarium

The aquarium laboratory creates an access to a real aquarium located in the University of Deusto. On it, it is possible to feed the fish, turn on and off the lights, and, if the submarine is in the water and it is charged, control the submarine. The problem is that most of the time, the submarine is out of battery so we only put it in the fishtank certain days.

Regarding feeding the fish, it may seem dangerous, but it is not. The system feeds them automatically three times a day, every 8 hour. If a user feeds them, then it does not let any other user to feed them before the next shift, guaranteeing that they are only fed three times. So go ahead and try it!

The initial rationale behind this laboratory is that groups of primary school students are responsible of the life of these fish (even if they are not under a real danger). Teachers may know which groups of students have feed them correctly, which students didn't forget and which students coordinated correctly so no one overfed the fish.

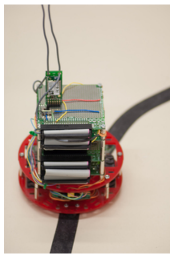
However, at the time of this writing ongoing work is being done for adding more sensors to this laboratory, so stay tuned ;-)

From a technical perspective, the whole laboratory is deployed in http://fishtank.weblab.deusto.es/, which uses a low cost ARM microprocessor called IGEPv2. So basically it is an example of *federated system*.

- **Target audience:** initially, *primary school students*. Right now the focus is changing to take into account physics principles with sensors.

1.3.4 Robot

The robot laboratory uses the commercial robot *Azkar-bot*, with an attached microcontroller. WebLab-Deusto manages to establish that three different learning activities are using the same equipment, so the scheduling system will queue other users internally.



- **Video:** http://www.youtube.com/watch?v=1WWAZVyuOBg

robot-proglist

robot-proglist lets you choose one among a few of predefined programs to program the bot with.

The programs currently available are the following:

Follow black line

The robot will first move randomly while avoiding obstacles (walls) until it finds the black line. It will then position itself on the line and follow it using its infrared sensors

Walk alone

Will simply walk around while avoiding any obstacles in its way.

Interactive Demo

Programs it with the same program that is used in the robot-movement. Doesn't really do much because there are no controls available in this mode.

Turn left & turn right

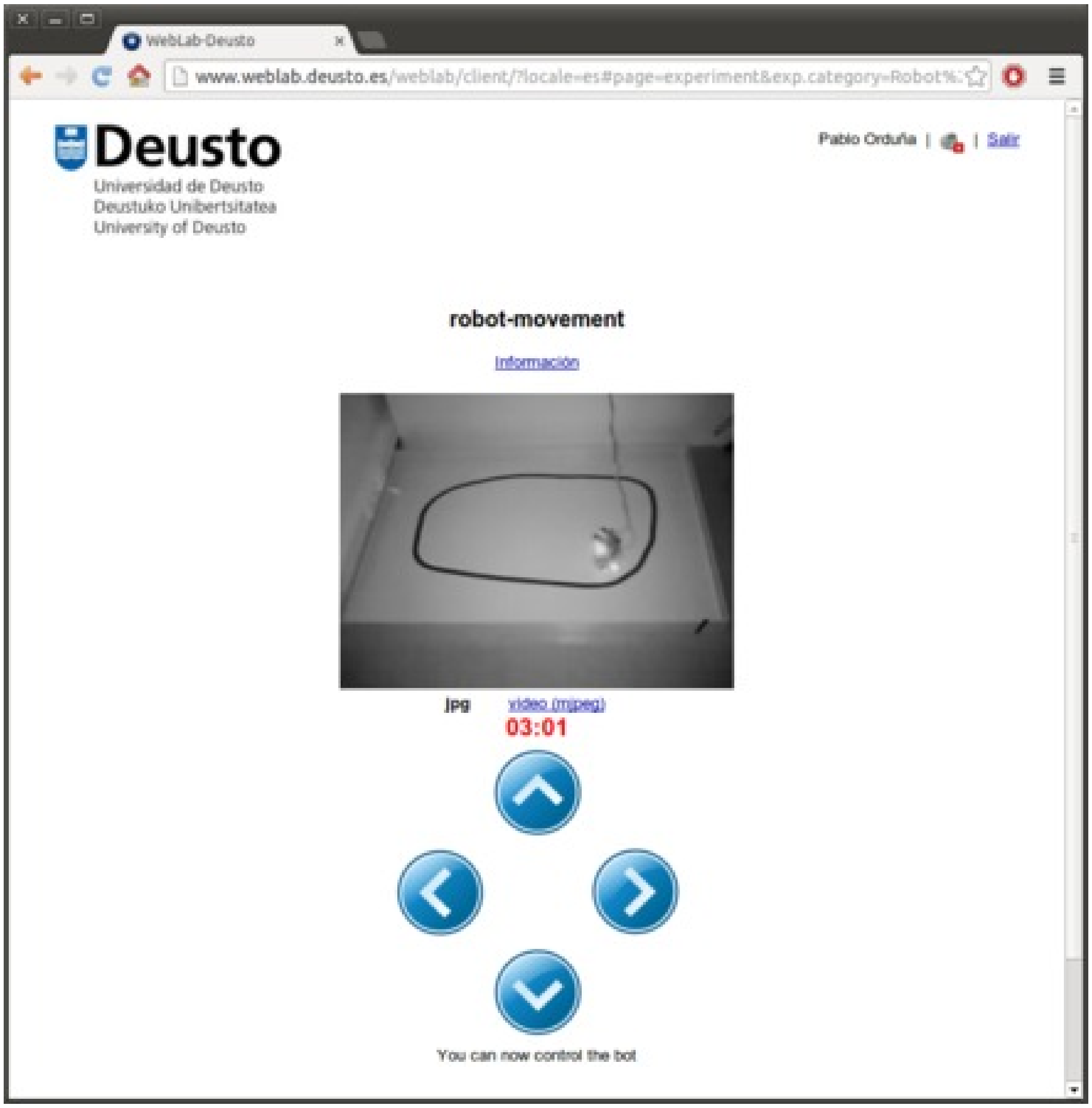
Rotates left and right, non-stop.

robot-movement

robot-movement lets you control a bot remotely. The bot can move forward or backward, and turn to both sides.

To make the bot move, simply click on the appropriate button. Alternatively, you can control the bot by using the arrows on your keyboard. Remember that the bot will move according to its own position, and not to the position of the camera.

The bot will not obey you if it finds a wall in its way, in which case it will try to avoid it.



robot-standard

robot-standard lets you program the bot yourself, with any program you wish.

The bot uses a PIC processor, so the program should be written using the Xilinx PIC compiler. It is noteworthy that the bot has, among other things, infrared sensors, to which the developer has access.

The MPLAB IDE used to build the PIC programs can be downloaded for free from http://www.microchip.com.

Specifics

The microcontroller model of the robot is a PIC 18F4550. It has two different motors for each wheel. The motors can go either forward or backward. It also has two obstacle sensors, which can be used to avoid the walls, and two infrared sensors, which can be used to detect the line.

Obstacle sensors are set to 1 if an obstacle is detected, while infrared sensors are set to 1 if the black line is detected.

Available pins are set up as follows:

```
#define         motorLeftFwd    PORTC,1 ;Forward bit of left Motor
#define         motorLeftBck    PORTC,0 ;Back bit of left Motor
#define         motorRightFwd   PORTD,3 ;Forward bit of right Motor
#define         motorRightBck   PORTC,2 ;Back bit of right Motor
#define         obstacleLeft    PORTA,3 ;Right obstacle sensor
#define         obstacleRight   PORTA,2 ;Left obstacle sensor
#define         infraredRight   PORTA,1 ;Right infrared sensor
#define         infraredLeft    PORTA,0 ;Left infrared sensor
```

It is noteworthy that the bot's firmware relies on a a bootloader, which means that PIC programs must start after a certain number of bytes. This can be seen in the provided example.

Programs should be compiled using absolute addresses (no relocation).

Example

The following program makes the robot run back and forth while trying to avoid the walls:

```
include         "p18F4550.inc"          ; including the header file of PIC 18F4550
radix   hex             ; Unspecified literal hexadecimal-encoded


;********************************Label␣
↪Definition***************************************
#define         motorLeftFwd    PORTC,1 ;Forward bit of left Motor
#define         motorLeftBck    PORTC,0 ;Back bit of left Motor
#define         motorRightFwd   PORTD,3 ;Forward bit of right Motor
#define         motorRightBck   PORTC,2 ;Back bit of right Motor
#define         obstacleLeft            PORTA,3 ;Right obstacle sensor
#define         obstacleRight           PORTA,2 ;Left obstacle sensor

temp1   equ     0x00    ;variable temp1 asociada a registro 0x000 de prop. General
temp2   equ     0x01    ;variable temp2 asociada a registro 0x001 de prop. General
temp3   equ     0x02    ;variable temp3 asociada a registro 0x002 de prop. general
```

(continues on next page)

(continued from previous page)

```
        Org     0x200   ; Program begins at address 0x200
;********************************Configuration␣
↪Section***************************************
                movlw   b'11111000'
                movwf   TRISC                   ;RC0, RC1 y RC2 sets as OUTPUTS
                movlw   b'11110111'
                movwf   TRISD                   ;RD3 set as OUTPUT (Motor ports set␣
↪as outputs)
                setf    TRISA                   ;full PORTA set as INPUT (including␣
↪sensors)
                movlw   0x0f
                movwf   ADCON1                  ;All ports digitals
                movlw   0x07
                movwf   CMCON                   ;Comparators Off

;********************************Program Starts***************************************
goForward       bsf     motorRightFwd
                bsf     motorLeftFwd
                bcf     motorRightBck
                bcf     motorLeftBck
detectRight     btfss   obstacleRight   ; if sensor is "1" skip next instruction (no␣
↪detect)
                bra     turnLeft                ; if previous instruction does not␣
↪jump turn left
                                        ;       to avoid de obstacle detected

detectLeft      btfss   obstacleLeft    ; if sensor is "1" skip next instruction (no␣
↪detect)
                bra     turnRight       ; if previous instruction does not jump turn␣
↪Right
                                        ;       to avoid de obstacle detected
                bra     goForward       ;

turnLeft                Bsf     motorRightFwd
                bcf     motorLeftFwd
                bcf     motorRightBck
                bsf     motorLeftBck
                rcall   halfSec         ;Wait 0,6s
                bra     detectRight

turnRight       Bcf     motorRightFwd
                bsf     motorLeftFwd
                bsf     motorRightBck
                bcf     motorLeftBck
                rcall   halfSec                 ;Wait 0,6s
                bra     detectLeft

halfSec         Movlw   .3
                movwf   temp1
                clrf    temp2
                clrf    temp3                   ; Init vars (temp0=8, temp1=0 y␣
↪temp2=0)
bucle1          decfsz  temp1, F                ; First loop is repeated 8 times.
                bra     bucle2
                return
bucle2          decfsz  temp2, F                ; Second Loop is repeated 256 times␣
↪for each
```

(continues on next page)

(continued from previous page)

```
                bra     bucle3                          ;iteration of the first loop
                bra     bucle1
bucle3          decfsz  temp3, F                ; Third bucle is repeated 256 times␣
↪for each
                bra     bucle3                          ;iteration of the second loop
                bra     bucle2
;considering that each loop takes 3 cycles internal clock
;(1 jump + 1 decrease), the loop takes 3 * 256 * 256 * 3 = 589825
;as 1 cycle is 1 us, rutine takes aprox. 0.6 s

        End
```

Further details:

Full documentation may be downloaded from:

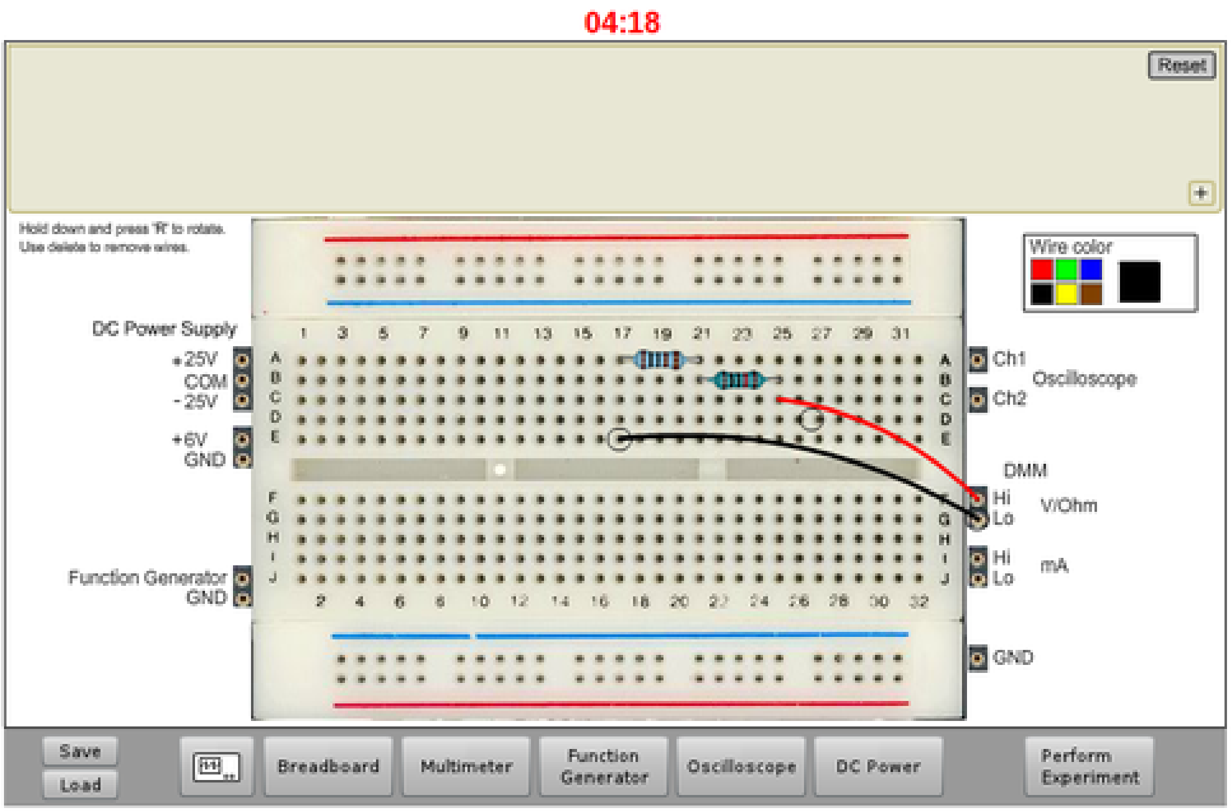
- English: http://www.weblab.deusto.es/pub/docs/robot_module_english.docx
- Spanish: http://www.weblab.deusto.es/pub/docs/robot_module_spanish.docx
- **Target audience:** *engineering students in general, certain secondary schools.*

1.3.5 VISIR

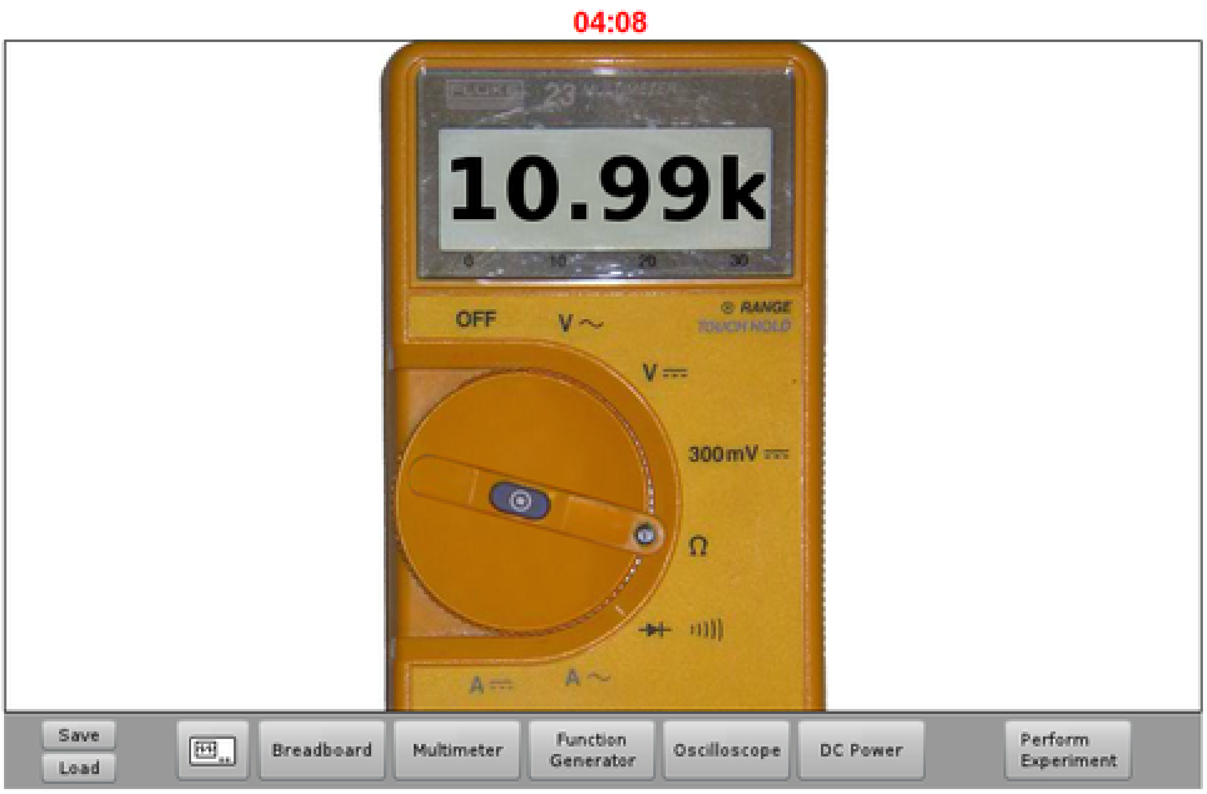
The VISIR experiment lets you access the BTH OpenLabs VISIR through WebLab-Deusto.

BTH OpenLabs VISIR (Virtual Instrument Systems In Reality) is a Remote Laboratory developed in the Blekinge Institute of Technology, which supports remote experimentation with real electronic circuits.

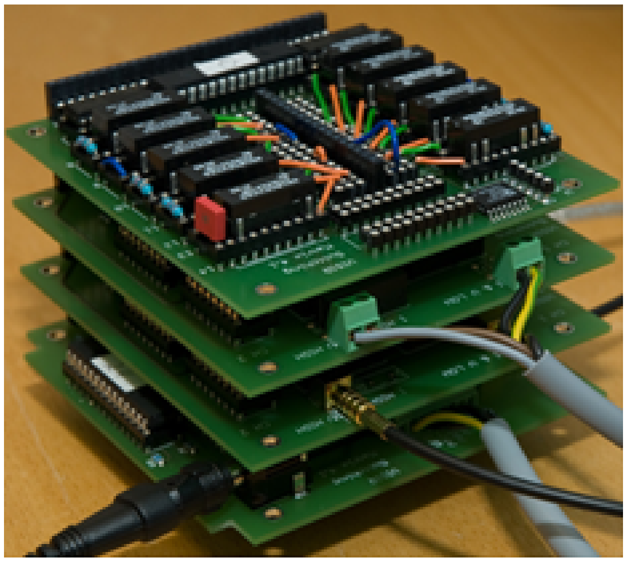
Students create circuits using the web interface, such as the following (where two resistors, of 10k and 1k are placed in serial and connected to the Digital MultiMeter):



And as a result of this, the digital multimeter will show the sum of the two resistors:



This is possible given that VISIR uses a switching matrix, where all the resistors and other components are located, and with a set of relays it creates the circuit requested by the student:



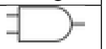
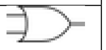
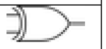
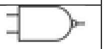
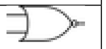
Furthermore, multiple students can access VISIR and take different measurements at the very same time. VISIR will create each circuit and take the measurement each time.

There is more information in the website of the VISIR project or in related papers.

- **Target audience:** It depends on how many principles are taught. It has been used with *secondary school students*, as well as with *electronics engineering courses*.
- **Video:** http://www.youtube.com/watch?v=vI5aM6Yq3S4

1.3.6 ud-logic

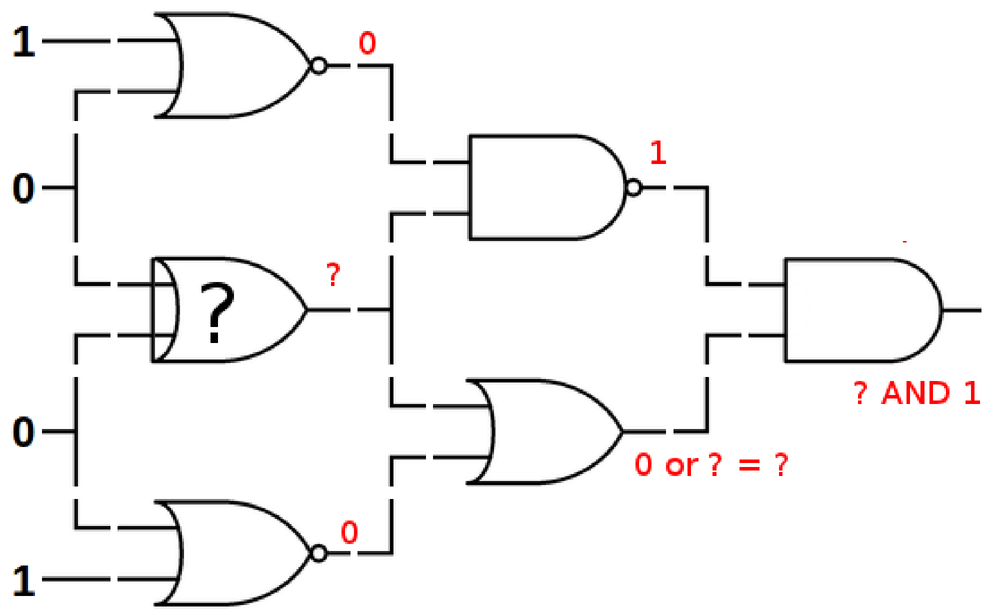
ud-logic is a simple game implemented as an experiment. Players are presented with a circuit diagram made up of 6 connected logic gates. Five of these gates show the type of gate: AND, NAND, OR, NOR or XOR. The symbols, as described in the wikipedia, are the following:

Name	Image
AND	
OR	
XOR	
NAND	
NOR	

Players must choose the type of the sixth gate so the result of the circuit is **1**. Sometimes, several types might yield the desired result, and they will all be considered correct.

When the players succeed, they are awarded one point and a new diagram is generated and they may choose a gate again. The process continues until the time expires or a wrong gate is chosen. When the process finishes, players can see their position in the ranking linked. The more points they get in the provided time, the higher they rank.

This experiment, for demonstration purposes, is usually connected to a hardware board, which can be seen through the provided Webcam stream. Thus, notice that whenever the gate choice is right, a message will appear in the board's screen, and the LEDs of the board will lit.



In the example above, in red it is written what the results will be, regardless the value of the unknown gate. For instance, in the upper level, **1 NOR 0** is **0** (**1 OR 0** is **1**, and **not 1** is **0**). When solving the whole circuit, it is clear that the final output, which must be **1**, is the result of **? AND 1**, being **?** the result of the unknown gate.

Therefore, we need to have **1** as output of the unknown gate. So the question is: which gate has **0** and **0** as inputs and **1** as output? **AND, OR** and **XOR** fail to do this, so the solutions in this case are **NOR** or **NAND**.

- **Target audience:** *secondary school students*, as well as first course of certain engineerings.

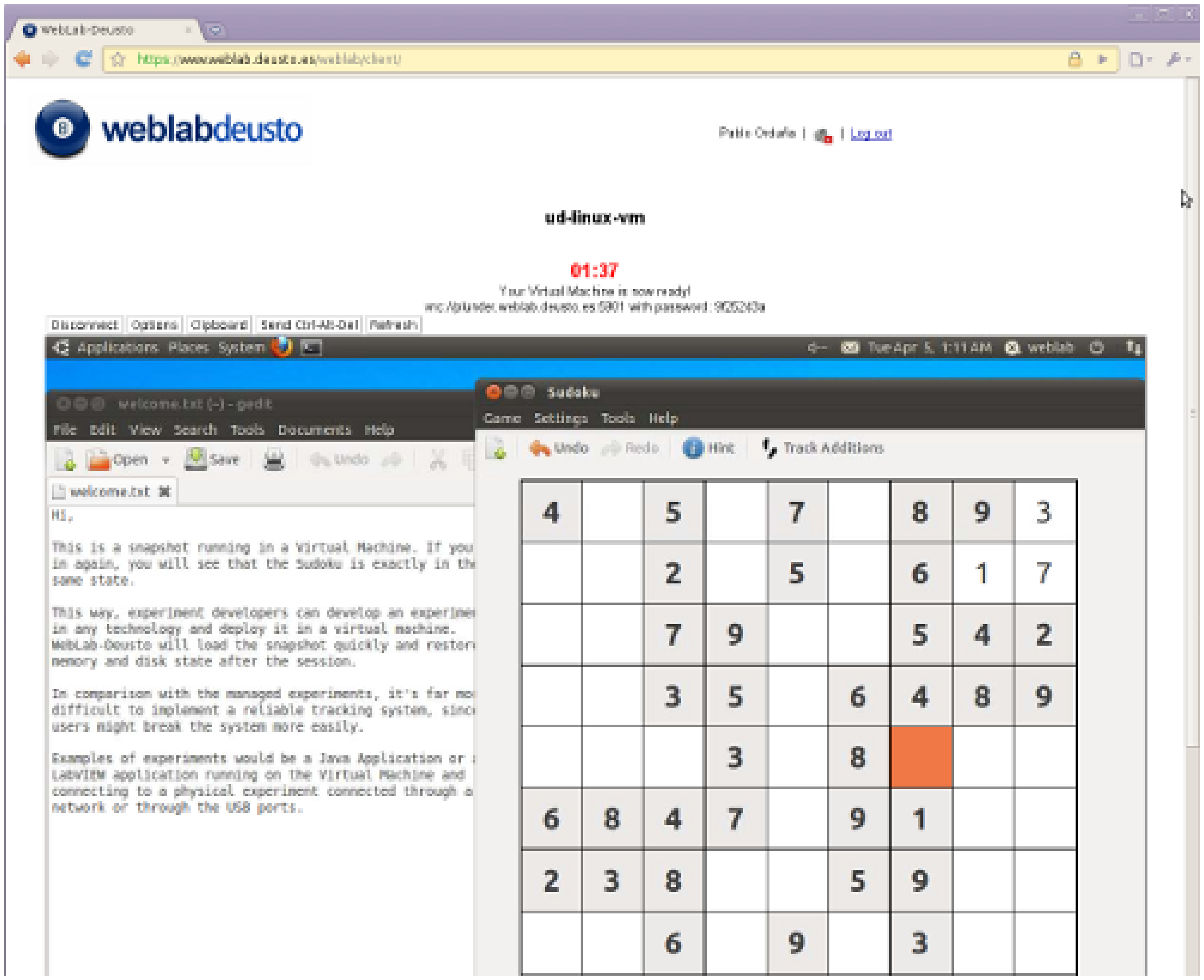
1.3.7 Virtual Machine lab

The linux-vm experiment gives you full access to a virtual machine running the Ubuntu Linux distribution.

The user is presented with a few demo programs, among which is a sample Labview application. The user is free to do whatever he wishes on the machine for the assigned time, and the virtual machine will be reset by Weblab to its original state once the session ends. For instance, you can test that the sudoku game running in the virtual machine is always the same, since the state is always restored.

The purpose of this experiment is mainly to showcase WebLab's ability to host easy-to-develop unmanaged experiments.

More detailed and technical information on VM-based experiments is available *here*.


ud-linux-vm

01:37

Your Virtual Machine is now ready!
vnc://plunder.weblab.deusto.es:5901 with password: 9052d3a

```
Hi,

This is a snapshot running in a Virtual Machine. If you
in again, you will see that the Sudoku is exactly in the
same state.

This way, experiment developers can develop an experimen
in any technology and deploy it in a virtual machine.
WebLab-Deusto will load the snapshot quickly and restore
memory and disk state after the session.

In comparison with the managed experiments, it's far mo
difficult to implement a reliable tracking system, sinc
users might break the system more easily.

Examples of experiments would be a Java Application or
LabVIEW application running on the Virtual Machine and
connecting to a physical experiment connected through a
network or through the USB ports.
```

- **Target audience:** It depends on what equipment is used internally. The one running in the demo is only for demonstration purposes.
- **Video:** http://www.youtube.com/watch?v=b-L2LXRr23A

1.4 Federation

WebLab-Deusto natively supports federating remote laboratories. This means that if two universities install WebLab-Deusto, any of the systems will be able to consume laboratories provided by the other university.

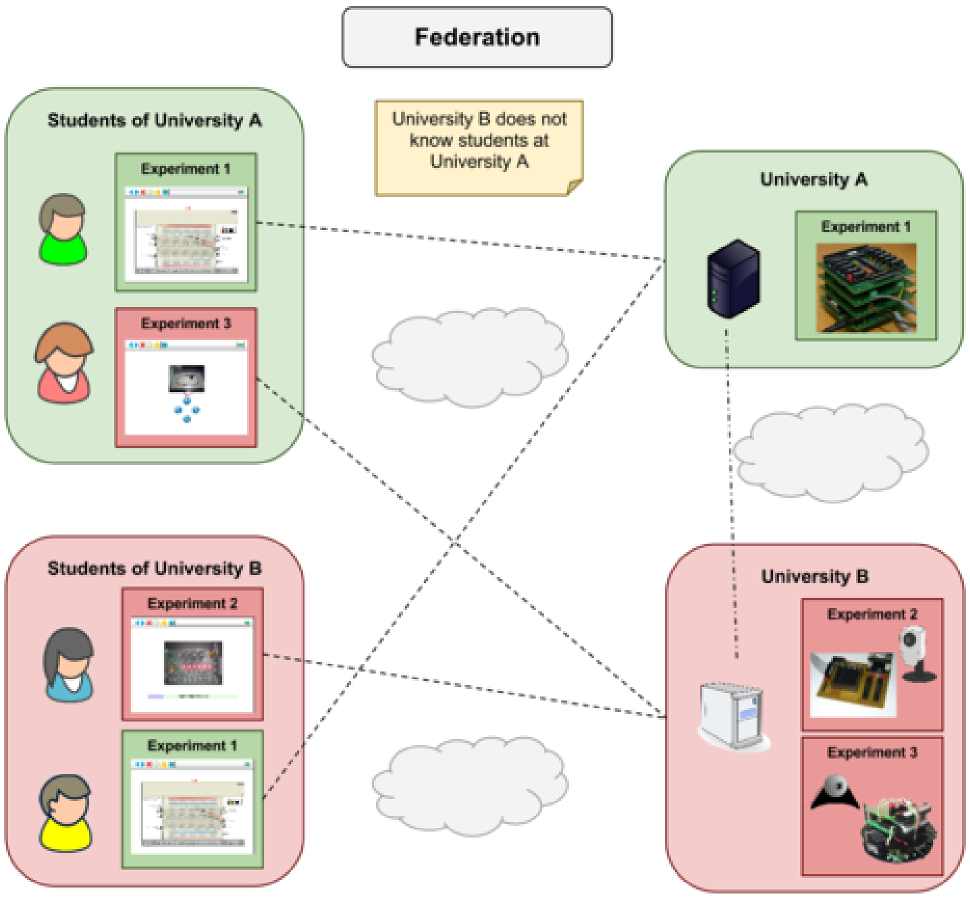


Table of Contents

1.4.1 See it in action

When you run the WebLab-Deusto demo, there is a particular laboratory called *submarine*. If you run it, you'll see that whenever it is reserved, the web page redirects you to other domain (from www.weblab.deusto.es to fish-tank.weblab.deusto.es). Internally, there are two independent WebLab-Deusto deployments there: one is the main system at Deusto, the other is a constrained system running in an ARM device (called IGEPv2). The first one (in this case, the **consumer**) is telling the second one (**provider**), "Hi, I'm 'deusto', and I want to use this laboratory that I'm granted for 250 seconds for a local user here called 'demo'". Later, the consumer will be requesting the provider for the user tracking, so the administrators of WebLab-Deusto will be able to track the 'demo' user.

Other way to test it is by *deploying WebLab-Deusto* (the basic default installation is a straightforward process). By default, the installation is a consumer of a federated system which is the main server of WebLab-Deusto. By adding different users and granting them permissions to the robots lab, and after accessing the lab with this user you'll be able to see in the administrator panel that it has been used.

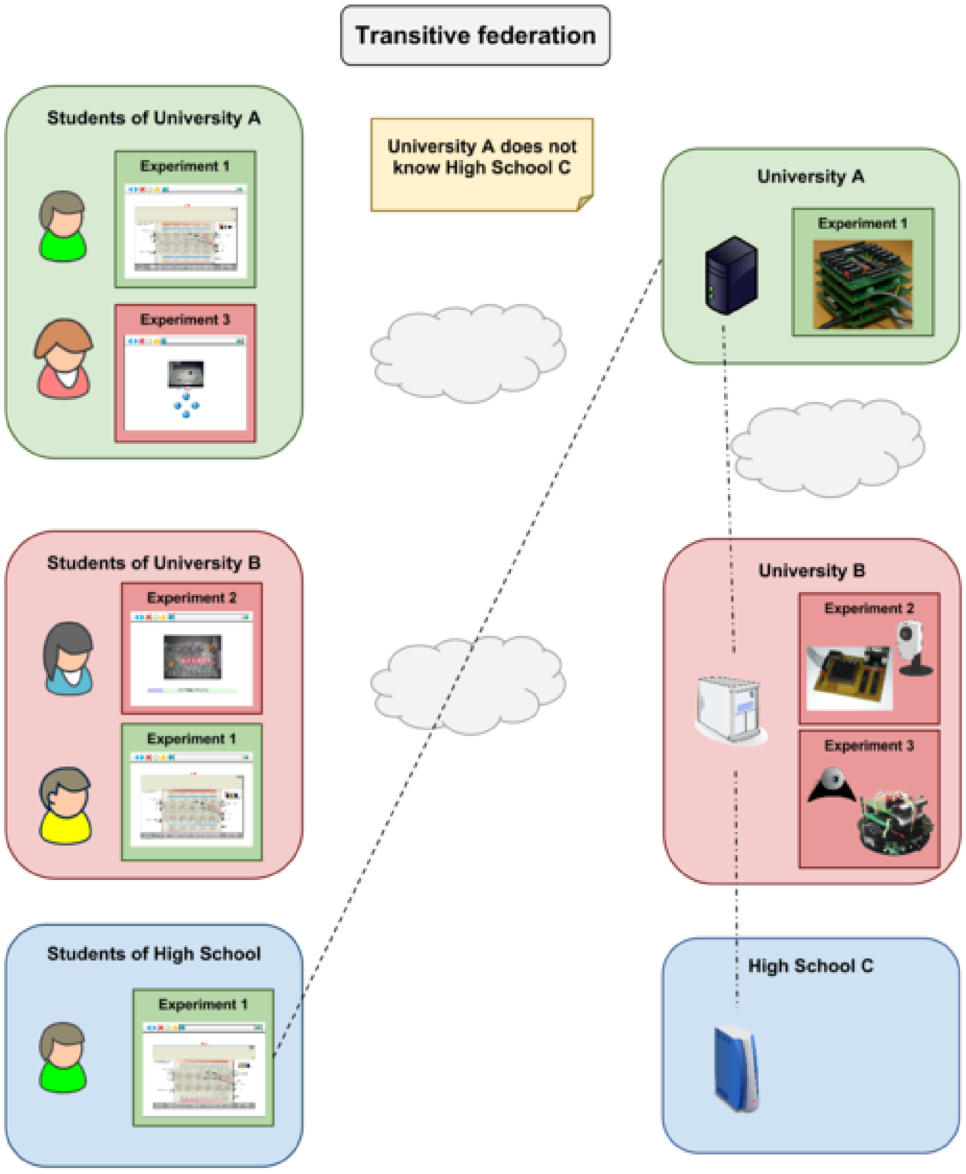
Finally, you can also see the federation video.

1.4.2 Features

Two main features are provided by WebLab-Deusto:

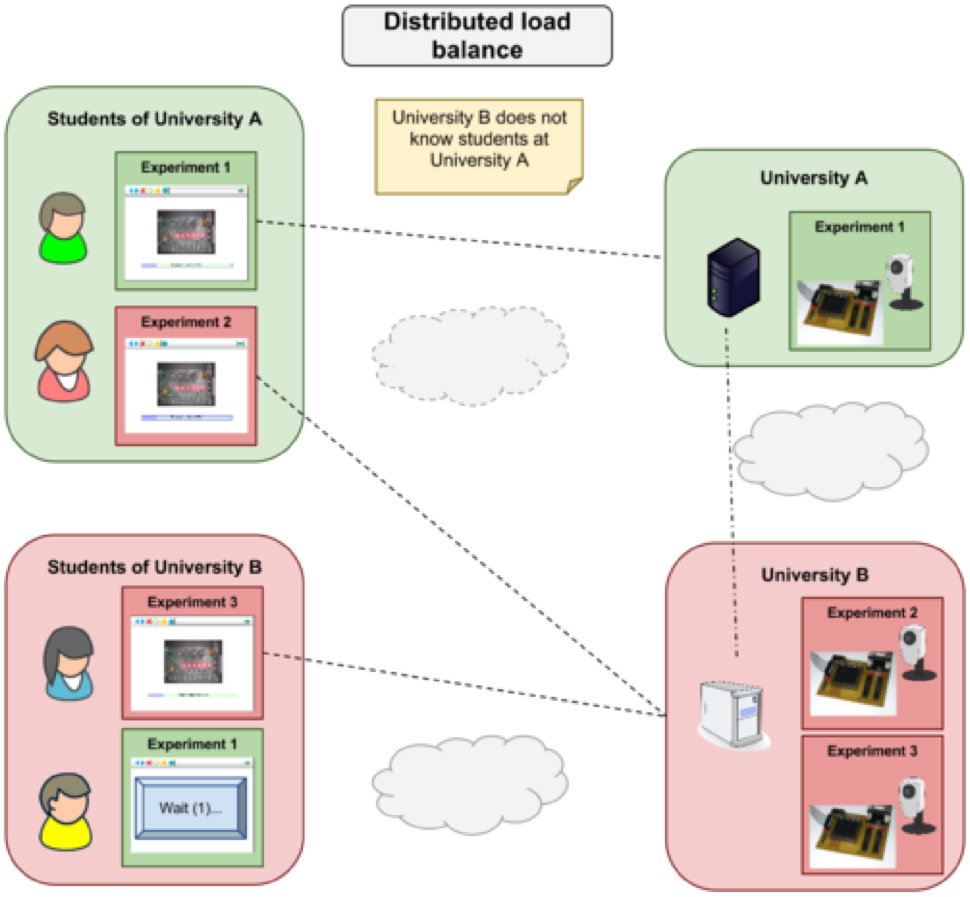
Transitivity

If you're a provider of a laboratory, your consumers may technically re-share this laboratory. Basically, this enables subcontracting laboratories. See the transitive federation video.



Federated load balance

If there are multiple providers of a copy of a laboratory, you can balance the load of users among them automatically.



1.4.3 Examples

Other WebLab-Deusto deployments (in addition to the one in University of Deusto):

- LabsLand: https://weblab.labsland.com
- UNED: https://weblab.ieec.uned.es/
- HBRS: https://fpga-vision-lab.h-brs.de/weblab/
- Slovenská technická univerzita: http://weblab.chtf.stuba.sk/
- UPNA: https://weblab.unavarra.es/
- TU-Dortmund: https://weblab.zhb.tu-dortmund.de
- FH Aachen: https://weblab.fh-aachen.de
- UGA: https://remotelab.engr.uga.edu
- ISEP: https://openlabs.isep.ipp.pt/weblab/
- UNED (Costa Rica): https://labremoto.uned.ac.cr/weblab/
- UNAD: https://lab-remoto-etr.unad.edu.co/weblab/

- UNIFESP: https://weblab.unifesp.br/weblab/
- UFH: https://weblab.ufh.ac.za/weblab/
- University of Michigan: https://weblab.eecs.umich.edu/weblab/
- University of Washington: https://weblab.ece.uw.edu/weblab/
- Université Abdelhamid Ibn Badis Mostaganem: https://weblabdeusto.leog.univ-mosta.dz/weblab/
- PSUT: https://weblab.psut.edu.jo/weblab/

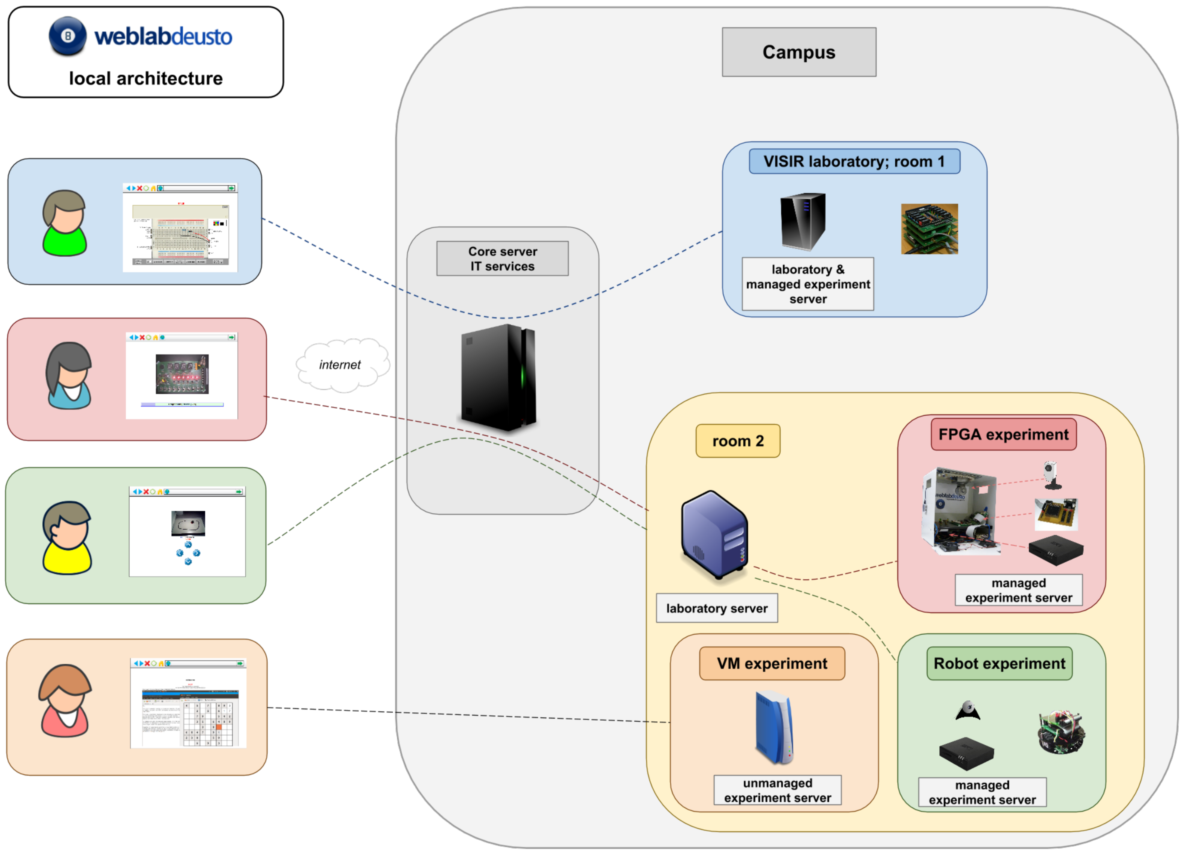
If you wish us to host a new deployment, contact us at weblab@deusto.es.

1.5 Technical description

This section describes the internals of a single WebLab-Deusto deployment. However, the architecture is enriched supporting federation. Go to the *federation* section for further information.

1.5.1 Architecture

Locally, WebLab-Deusto is based on the distributed architecture shown in following diagram:



In this architecture, clients connect to the core servers, using commonly HTTP with JSON. These servers manage the authentication, authorization, user tracking, federation (sharing) and scheduling. From there, the system forwards requests to the laboratory servers, which forward them to the final experiments. One exception are the unmanaged laboratories (such as Remote Panels, Virtual Machines or so), where students directly connect to the final host directly (and therefore user tracking is lost).

As detailed later, the communications however enable that all these servers are spread in different machines in a network, or they can all be running on the same machine or even in the same process. For instance, the login server and the core server are usually always in the same process, while the laboratory server may be in other computer and the experiment server could be in the same process as the laboratory server. It just depends on the deployment desired and the required latency.

1.5.2 Technologies

WebLab-Deusto is developed in Python and using Open Source technologies (MySQL or SQLite, Redis, etc.), but we provide multiple APIs for developing laboratories in different languages. The user interface is developed in HTML, but it supports labs in other legacy technologies.

The server uses an ORM called SQLAlchemy. In theory, WebLab-Deusto should be independent of the database provider, but it has only been tested with MySQL and SQLite. For scheduling, WebLab-Deusto supports two types of back-ends: SQL database (again, MySQL and SQLite) and Redis, which is much faster.

1.5.3 Communications

WebLab-Deusto communications have been built on top of a pluggable system of protocols. When a component tries to connect to other server, it provides the WebLab-Deusto address of this server, and the communications broker will check what possible protocols can be used and it will automatically choose the fastest one (e.g., if both components are in the same process, it calls it directly instead of using any kind of serialization or communication).

1.6 Publications

The members of the WebLab-Deusto project are listed here. Some of them have their publications listed in certain websites (e.g., Javier Garcia-Zubia, Pablo Orduña, Ignacio Angulo, Unai Hernandez or Luis Rodriguez-Gil).

1.7 Contact

For technical support or discussion, please use the weblabdeusto mailing list:

- https://groups.google.com/forum/?fromgroups#!forum/weblabdeusto

You may also contact the WebLab-Deusto developers at weblab@deusto.es

1.8 Professional services

For professional services (e.g., commercial support, consultancy services…), LabsLand is the spin-off of the WebLab-Deusto project:

- http://labsland.com

CHAPTER 2

Users

This section is intended for people who is going to install the WebLab-Deusto system.

2.1 Installation

Installing the core of WebLab-Deusto is pretty straightforward. It does not have many requirements. However, supporting more features and tuning the performance requires installing more software infrastructure. This section covers only the first steps.

Note: If you're familiar with `Python`, `git`, `setuptools` and `virtualenv`, all you need to do (in a `virtualenv`) is:

```
pip install git+https://github.com/weblabdeusto/weblabdeusto.git
```

And go *to the next section*. Please note that, given the size of the repository, it's better if you keep the weblabdeusto git repository downloaded in your computer so you can call `git pull` to upgrade faster.

Otherwise, please read this section.

Through this tutorial, we'll go through the most simple deployment possible. It will not require any web server (such as Apache), neither a database engine (such as MySQL). Instead it will use its internal web server and a simple sqlite database. Then, we will explain how to go deep to more complex deployments.

Note: during the whole documentation, some examples of commands run in a terminal will be presented. Given that terminals are different from system to system, we will show `$` to represent the terminal prompt. For instance, the following example:

```
$ weblab-admin --version
5.0
```

The `$` will represent `C:\something>` in Windows environments and `user@machine:directory$` in certain UNIX (Linux, Mac OS X) environments. You must not write that. Whenever there is no `$` in the beginning of the line (such as `5.0` in the example), is the expected output. Finally, sometimes the output is too long, so `[...]` is used to declare "a long output will be shown".

2.1.1 Obtaining WebLab-Deusto

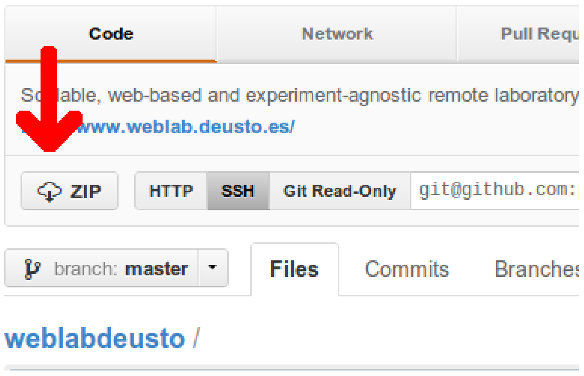
There are two ways to obtain WebLab-Deusto:

1. **Downloading it from github using git. That's the recommended version, since that allows you to upgrade WebLab-Deusto**

    - This process is detailed in *Download using git*.

2. **Downloading it from github using the web browser. That's the simplest version, but we do not recommend it (since you los**

    - **Windows users:** in certain versions of Microsoft Windows, sometimes there are problems with too-long file paths, so if any problem is reported by your uncompressing program, just make sure that the directory where you are uncompressing WebLab is not very long (for instance, uncompressing it in `C:\weblab` or `C:\Users\Tom\weblab` will surely work, whereas downloading it in `C:\Users\My full name\Downloads\Other downloads\Yet other downloads\weblabdeusto-long-name` might fail).



2.1.2 Installing the requirements

1. **Install Python 2.7:**
    - In Linux and Mac OS X, Python is probably installed.
    - In Microsoft Windows, download it from here. Do not download Python 3.x (WebLab-Deusto relies on Python 2.7).
2. **Once installed, put both in the system path:**
    - In Linux and Mac OS X, this is probably done by default.
    - In Microsoft Windows, go to Control Panel -> System -> Advanced -> Environment variables -> (down) PATH -> edit and append: `;C:\Python27\;C:\Python27\Scripts\;`.
3. At this step, you should be able to open a terminal (in Microsoft Windows, click on the Start menu -> run -> type `cmd`) and test that both tools are installed.

Run the following (don't take into account the particular versions, these are just examples):

```
$ python --version

Python 2.7.6
```

Note: If it reports that it is using a higher version (e.g., 3.5.1), then your system is using by default Python 3 instead of Python 2. At the time of this writing, WebLab-Deusto is incompatible with that version. If this is the case, try running `python2.7` to verify that it is installed:

```
$ python2.7 --version
Python 2.7.6
$
```

If it is installed (even if it is not by default), it is fine.

4. Install `setuptools` if you don't have them. In Windows, nowadays the installer of Python comes with `pip`, so you don't need to install anything else. In Linux, you usually can install it from the repositories (e.g., `sudo apt-get install python-pip` in Ubuntu/Debian). If in doubt, follow the instructions.
5. Install `virtualenv` and `virtualenvwrapper`. In Ubuntu/Debian you can use `sudo apt-get install virtualenv virtualenvwrapper` (and in other Linux distributions this is probably available in the repositories). In Windows run the following:

```
C:\Users\Tom> pip install virtualenvwrapper-win
```

In other systems, you may use as an administrator:

```
$ pip install virtualenv virtualenvwrapper
```

6. At this point, you should be able to open a terminal and test that these tools are installed.

Run the following (don't take into account the particular versions):

```
$ pip --version

pip 1.5.4 from /usr/lib/python2.7/dist-packages (python 2.7)

$ virtualenv --version

1.11.4

$ mkvirtualenv --version

1.11.4
```

Troubleshooting

virtualenv and virtualenvwrapper **are not strictly necessary**. If you don't use them, you can always install WebLab-Deusto at system level (using administrator credentials. So if you get problems that you can not solve when installing virtualenv, do not worry and skip that step.

That said, there are some common problems installing virtualenvwrapper, listed here:

- **mkvirtualenv: command not found**: virtualenvwrapper is a bash script, which must be loaded. By default in Ubuntu, it is correctly loaded in all the new terminals, so try closing the current terminal and opening it again.

If the problem persists, you may need to find where is a script called `virtualenvwrapper.sh`, and add to your `~/.bashrc`:

```
source /path/to/virtualenvwrapper.sh
```

- Problems in **Microsoft Windows Windows** with path not found: Check that you have installed virtualenvwrapper-win and not virtualenvwrapper.

If you still have problems with `mkvirtualenv`, try uninstalling it (`pip uninstall virtualenvwrapper`) and installing only the `virtualenv` package. If you do this, you will need to do:

```
$ virtualenv weblab_env
New python executable in weblab_env/bin/python
Installing distribute....................done.
Installing pip...............done.
$
```

Note: Make sure that the virtualenv is in a directory with no spaces. For example, if you have it in a directory such as `/Users/Tom/Google Drive/` or `C:\Users\Tom\Desktop\My folder`, there can be problems with different dependencies of Python. It is safer if you use `/Users/user/projects`.

Note: If by default your system is using Python 3, then make sure you provide the following parameter:

```
$ virtualenv --python=/usr/bin/python2.7 weblab_env
```

And then, each time you want to workin the virtualenv, run:

```
(On UNIX)
$ . ./weblab_env/bin/activate
(weblab_env) user@machine:~$

(On Windows)
C:\> .\weblab_env\Scripts\activate
(weblab_env) C:\>
```

If this also generates problems, you can safely avoid using a virtual environment and install the whole system as administrator:

```
C:\weblab\> python setup.py install
```

2.1.3 Installing WebLab-Deusto

Create a virtualenv. In Linux/Mac OS X systems:

```
user@machine:/opt/weblabdeusto$ cd WHEREVER-IS-WEBLAB

(e.g., /opt/weblabdeusto/  Avoid directories with spaces -e.g., /Users/Tom/Google␣
↪Drive/-)

user@machine:/opt/weblabdeusto$ mkvirtualenv weblab

(weblab) user@machine:/opt/weblabdeusto$
```

Note: If by default your system is using Python 3, then make sure you provide the following parameter:

```
$ mkvirtualenv --python=/usr/bin/python2.7 weblab
```

In Microsoft Windows environments:

```
C:\> cd WHEREVER-IS-WEBLAB

(e.g., C:\weblabdeusto\  Avoid directories with spaces -e.g., C:\Users\Tom\My␣
↪Projects\-)

C:\weblabdeusto> mkvirtualenv weblab

(weblab) C:\weblabdeusto>
```

Then, make sure you're running the latest versions of setuptools and pip:

```
(weblab) $ pip install --upgrade setuptools
(weblab) $ pip install --upgrade pip
```

And then, install WebLab-Deusto:

```
$ python setup.py install
[...]
Finished processing dependencies for weblabdeusto==5.0
```

Once the process is over, you can test the installation by running:

```
$ weblab-admin --version
5.0 - 1ac2e2b03048cf89c8df36c838130212f4ac63d3 (Sunday, October 18, 2015)
```

If it displays 5.0 or higher, then you have successfully installed the system in that virtual environment. Virtual environments in Python are environments where a set of libraries (with particular versions) are installed. For instance, you may have different virtual environments for different applications relying on different versions of libraries. The long code (i.e., 1ac2e2. . .) refers to the currently installed version, and then the date of the latest change in the WebLab-Deusto repository. You should *upgrade the system* from time to time to obtain the latest features.

Whenever you open a new terminal, you'll find that `weblab-admin` is not installed. However, whenever you activate the environment where you installed WebLab-Deusto, it will be installed. For instance, if you open a new terminal, do the following in UNIX (Linux, Mac OS X) systems:

```
user@machine:~$ workon weblab
(weblab) user@machine:~$ weblab-admin --version
5.0 - 1ac2e2b03048cf89c8df36c838130212f4ac63d3 (Sunday, October 18, 2015)
```

Or the following in Microsoft Windows systems:

```
C:\Users\John\Desktop> workon weblab
(weblab) C:\Users\John\Desktop> weblab-admin --version
5.0 - 1ac2e2b03048cf89c8df36c838130212f4ac63d3 (Sunday, October 18, 2015)
```

Now you can continue with the *first steps*.

2.2 First steps

In this section, we will learn to create our first deployment of a WebLab-Deusto instance. This section assumes that you have successfully *installed the system*. It also assumes that you have activated the proper virtual environment in the current terminal, so running weblab-admin works:

```
$ weblab-admin --version
5.0
```

The deployment we are running here is very small and relies of very few technologies. It has successfully been deployed even in Raspberry Pi devices. But it also has several drawbacks: performance, lack of concurrent support for certain operations, etc. We will see how to implement more complex scenarios in *other section*, but for bootstrapping a WebLab-Deusto instance and learning the basic concepts, this is enough.

2.2.1 Creating a WebLab-Deusto instance

A single computer may have multiple instances of WebLab-Deusto. In production, there will be typically a single one, but for testing it may be useful to play with different ones. Each instance will manage its own permissions, its own users, its own queues, etc.

So as to create a new WebLab-Deusto instance, run the following:

```
$ weblab-admin create example --http-server-port=8000
Congratulations!
WebLab-Deusto system created
[...]
Enjoy!

$
```

From this point, in that directory (*example*), a full WebLab-Deusto deployment will be established. If you take a look inside, you will see different directories (for databases -*db*-, web servers -*httpd*-, logs -*logs*, *files_stored*-), and there will be one which contains all the deployment configuration, called *core_machine*. Inside it, you will see a hierarchy of directories with configuration files that apply to each server.

2.2.2 Starting the WebLab-Deusto instance

The WebLab-Deusto instance at this point is configured, but it is not started. So as to start it, we will use once again the *weblab-admin* command. As you'll find out, this is the command that you will use for any management related with the instances. Run the following:

```
$ weblab-admin start example
Press <enter> or send a sigterm or a sigint to finish
```

As you can see, the server is running. By pressing enter, the server will stop:

```
(enter)
Stopping servers...
$
```

So, let's start it again:

```
$ weblab-admin start example
Press <enter> or send a sigterm or a sigint to finish
```

And, while it is started, let's use it for the very first time. Open in your web browser the following address: http://localhost:8000/

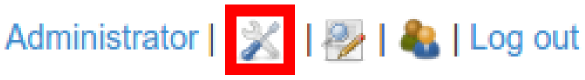
You will find the log in screen of WebLab-Deusto. On it, log in using *admin* as username and *password* as password. You will see that there are some sample laboratories. One of them (*dummy*) is local, and it does not rely on any hardware equipment. The rest are demo laboratories located in the University of Deusto. By default, these laboratories are created and assigned to the administrators group. They use the federation model of WebLab-Deusto to connect to WebLab-Deusto and use real equipment there.

You can safely play with both types of laboratories. With the dummy laboratory, you will see several output lines in the terminal from which you run WebLab-Deusto.

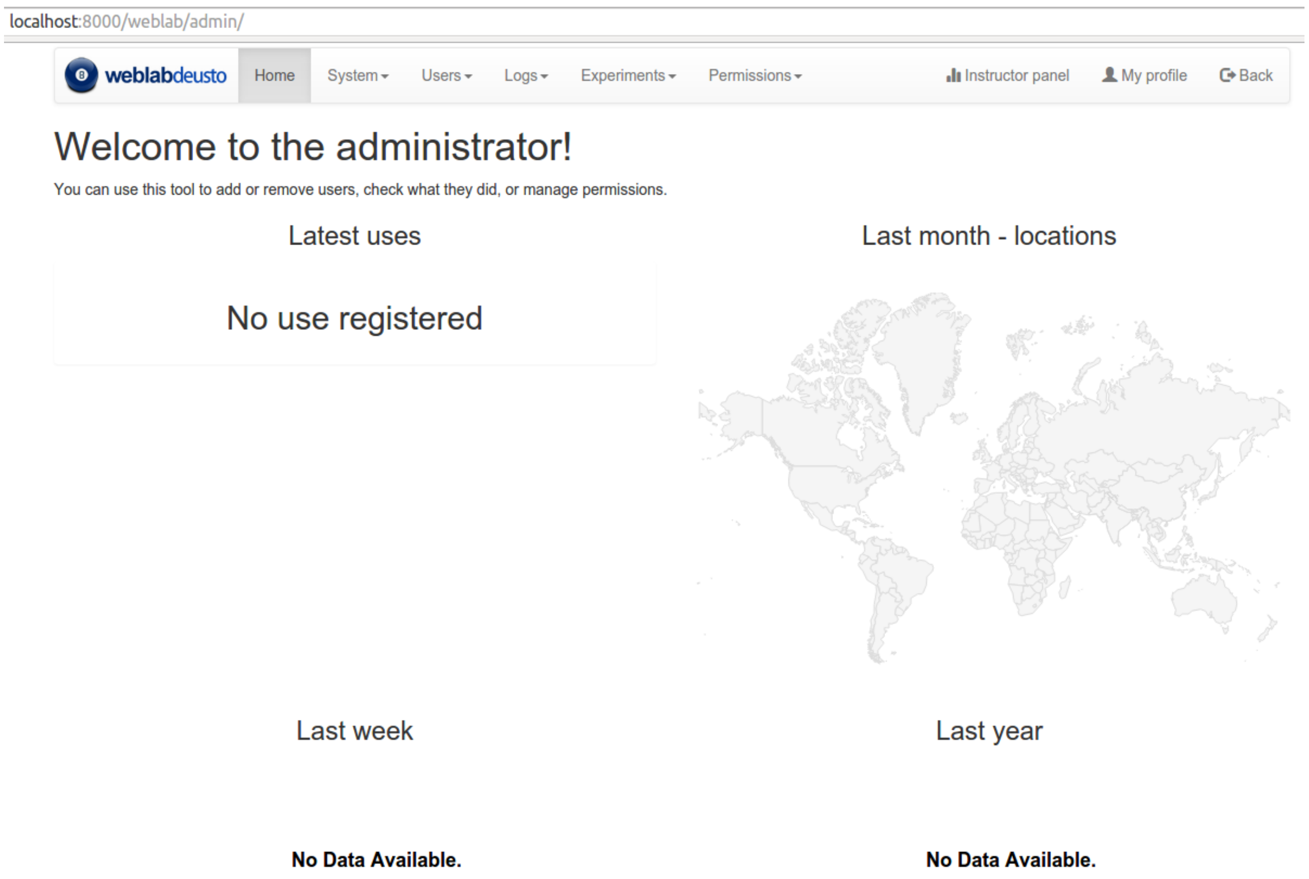
2.2.3 Managing users and permissions

Ok, so everything is working for the *admin* user. What about creating a class of 20 students who can access only the dummy, and other class who can access the federated laboratories?

Using the *admin* user, you'll see the settings button in the top-right corner. Click on it:



And you will see this:

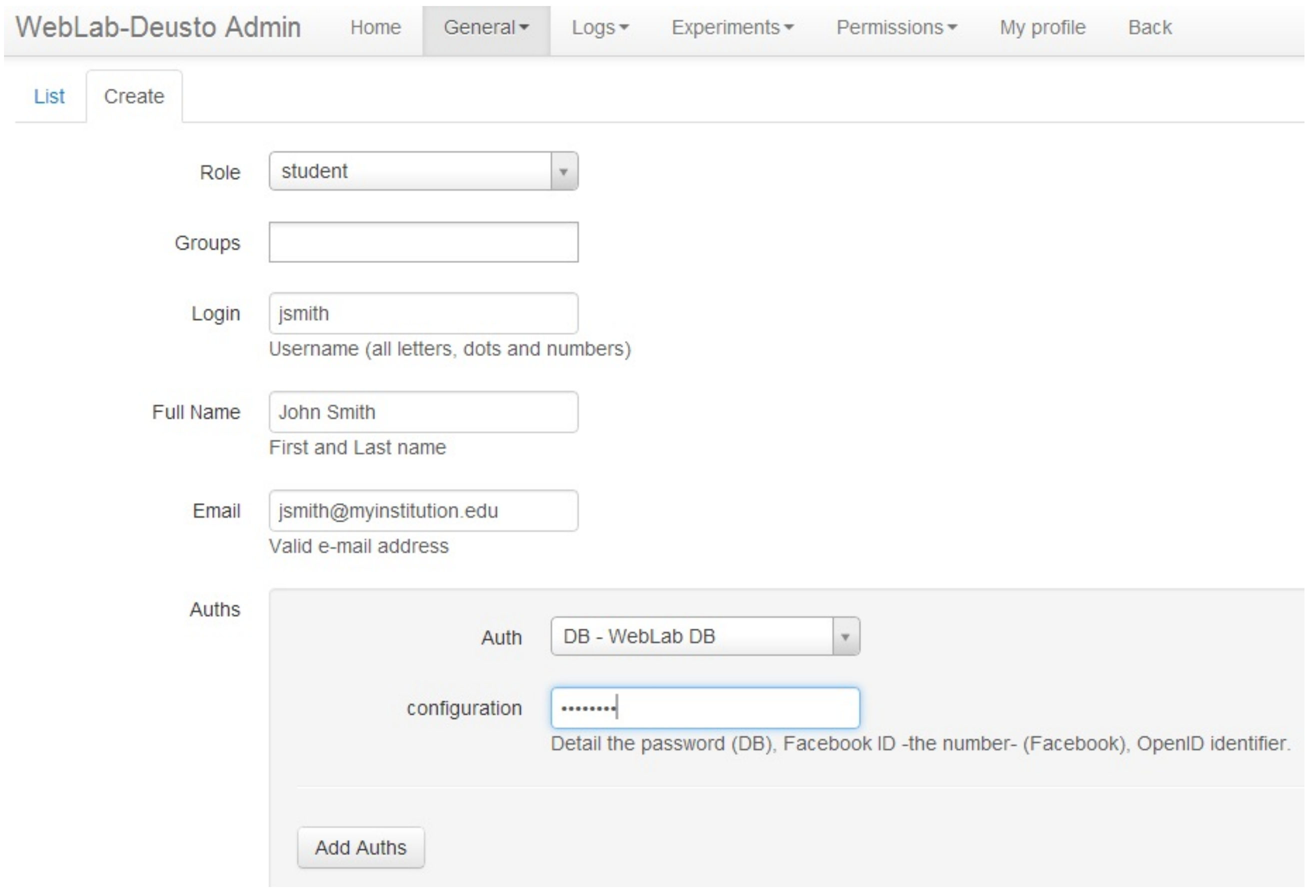


Welcome to the administrator!

You can use this tool to add or remove users, check what they did, or manage permissions.

Once in the administration panel, several operations are available. The number of operations is increasing from month to month, so upgrading the system is highly advisable.

The first thing to do is adding a new user. So as to do this, click on “General” and then on “Users”. There you can see the list of users registered in the system. Then, click on “Create” and fill the following fields:



Role: student

Groups:

Login: jsmith
Username (all letters, dots and numbers)

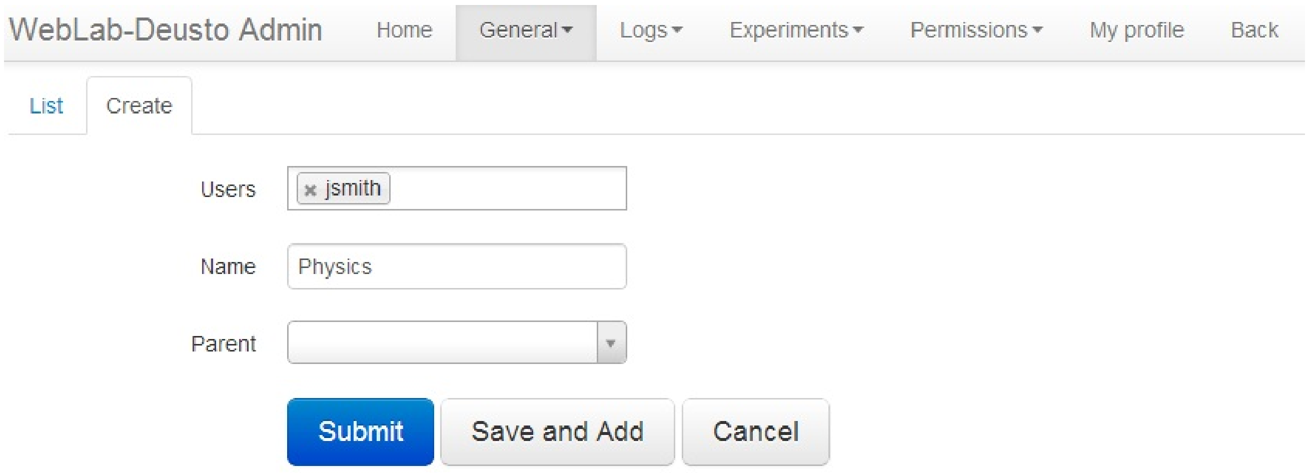
Full Name: John Smith
First and Last name

Email: jsmith@myinstitution.edu
Valid e-mail address

Auths:
Auth: DB - WebLab DB
configuration: ••••••••
Detail the password (DB), Facebook ID -the number- (Facebook), OpenID identifier.

The role “student” is the common one. If you select “administrator”, that user will be able to use the administration panel (and therefore, add or delete other users, experiments, etc.).

Once we have added a user, let’s create a new group called “Physics”. Click on “General” and then on “Groups”. Inside this group, you can click on “Create” and fill the following fields:



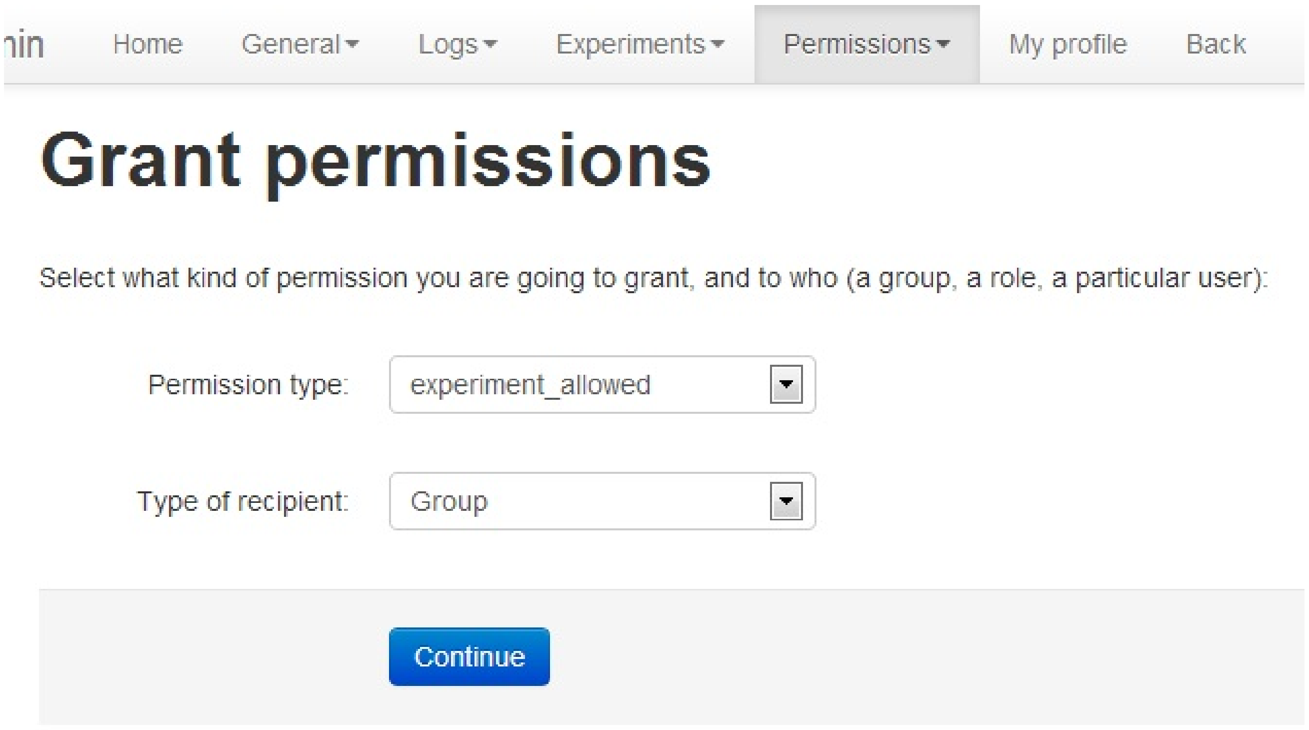
Users: jsmith

Name: Physics

Parent:

The “Users” field contains all the users in the system. So you can add them directly here, or in the “Groups” field when editing a user.

The next step is to grant permission on a laboratory to this user (or this group). To do this, click on "Permissions", and then on "Create". Here you can select what permission to grant ("experiment_allowed" in this case) and to who (a group, a user, or a role).



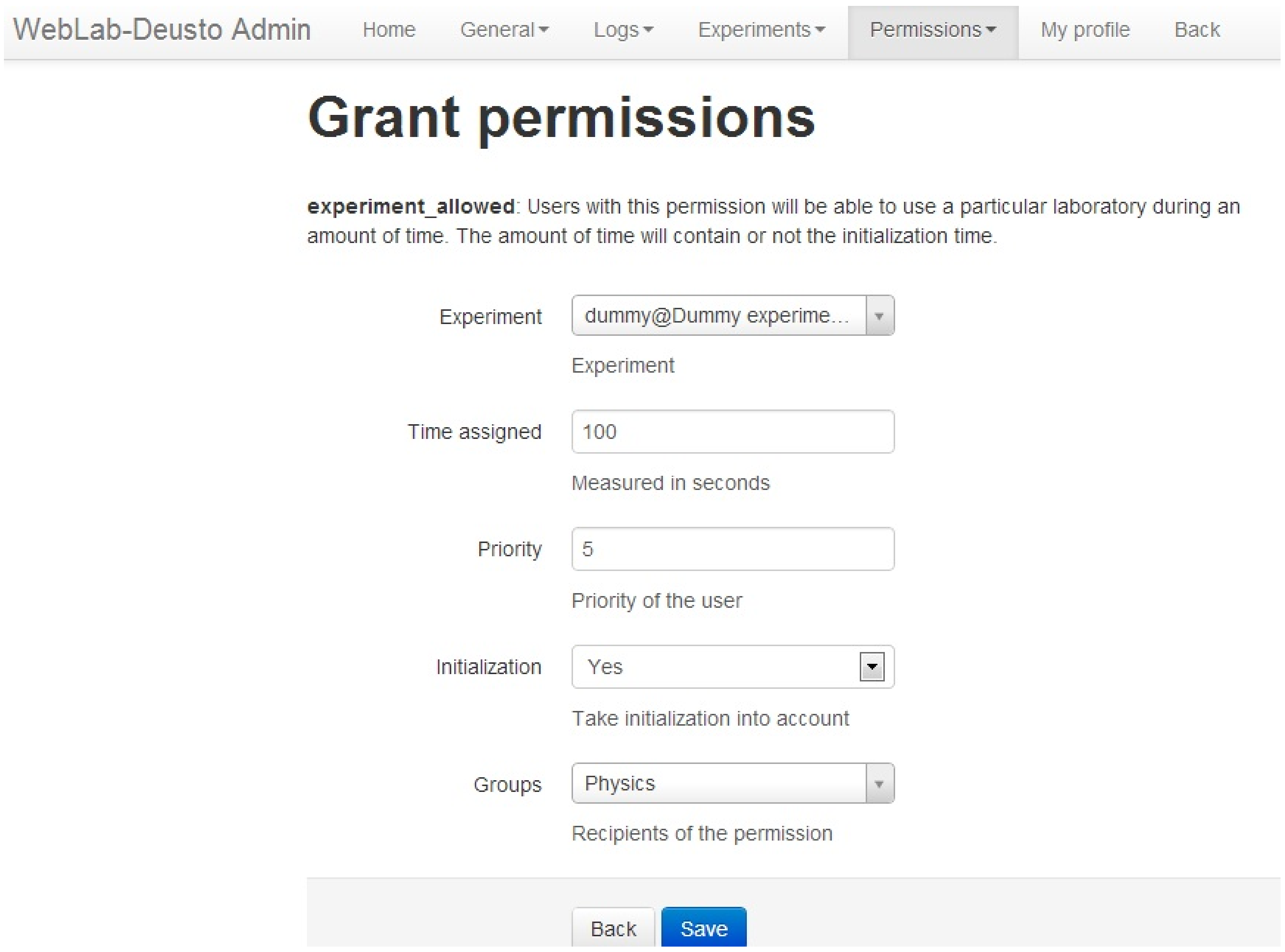
Grant permissions

Select what kind of permission you are going to grant, and to who (a group, a role, a particular user):

Permission type: experiment_allowed

Type of recipient: Group

And then you can select the experiment you want to let the user access, for how long (in seconds), what priority he may have (the lower, the faster they advance in the queue), and to which group you are granting this permission.



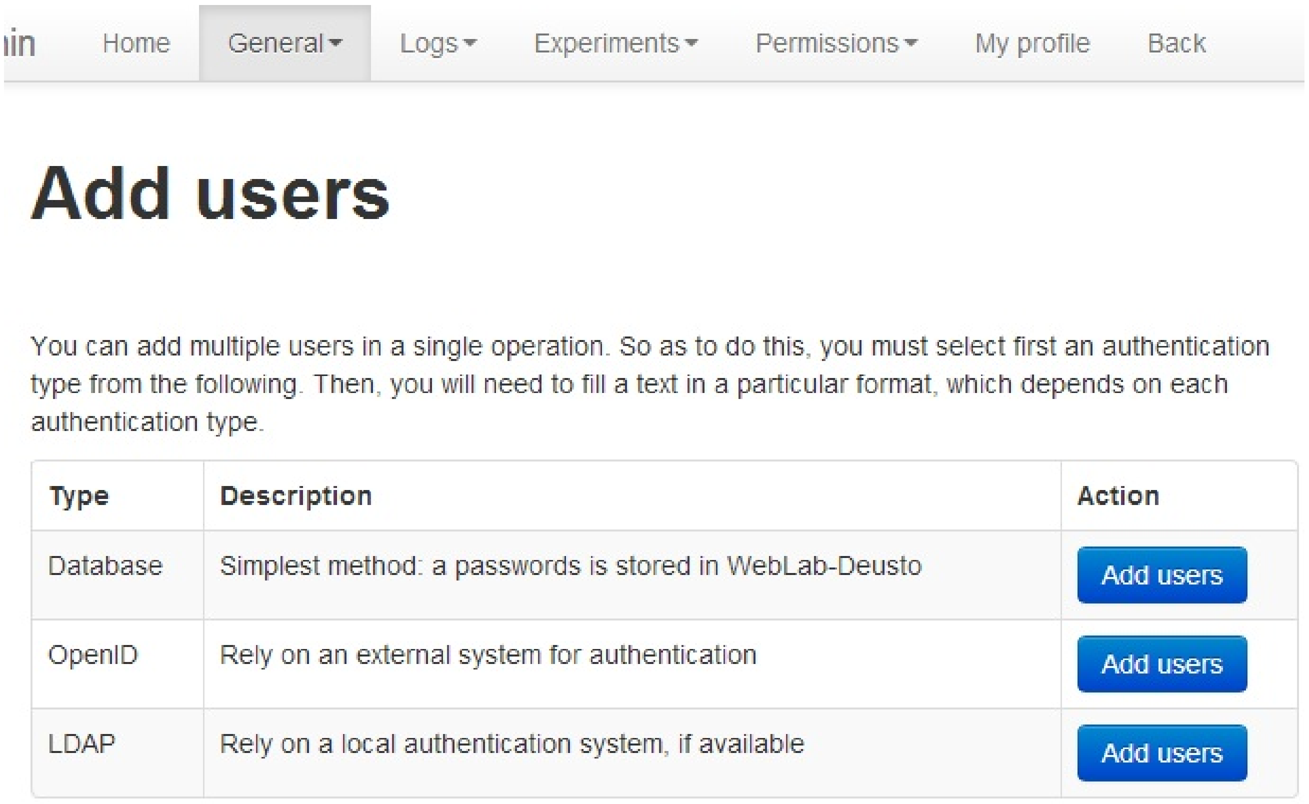
Grant permissions

experiment_allowed: Users with this permission will be able to use a particular laboratory during an amount of time. The amount of time will contain or not the initialization time.

Experiment	dummy@Dummy experime...
	Experiment
Time assigned	100
	Measured in seconds
Priority	5
	Priority of the user
Initialization	Yes
	Take initialization into account
Groups	Physics
	Recipients of the permission

Once this is done, this user (and all the users in that group) can access that laboratory.

Given that adding multiple users one by one might be useful, it is possible to add multiple users at a time. Click on “General”, then on “Add multiple users”.

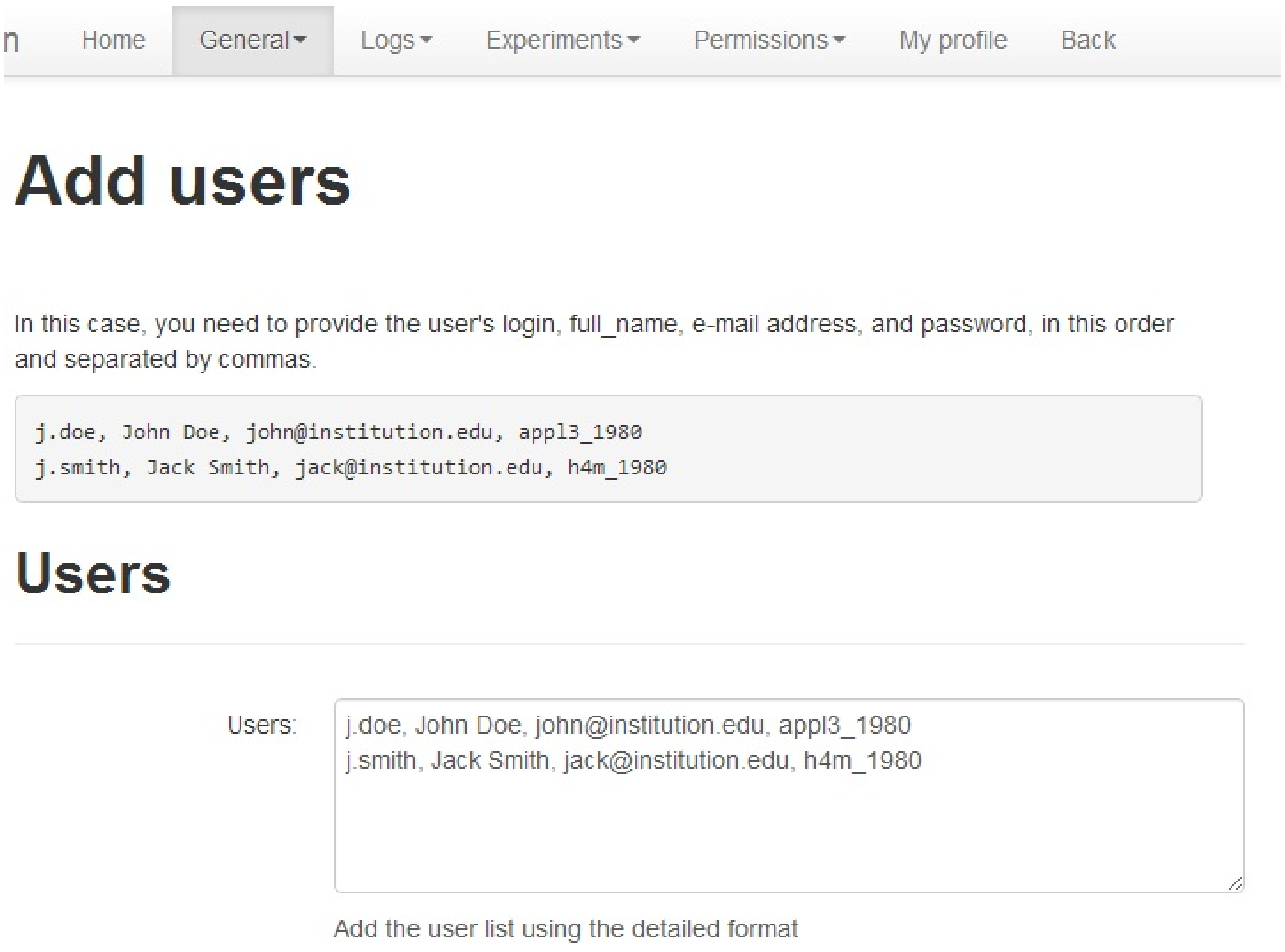


Add users

You can add multiple users in a single operation. So as to do this, you must select first an authentication type from the following. Then, you will need to fill a text in a particular format, which depends on each authentication type.

Type	Description	Action
Database	Simplest method: a passwords is stored in WebLab-Deusto	Add users
OpenID	Rely on an external system for authentication	Add users
LDAP	Rely on a local authentication system, if available	Add users

Click on the "Add users" in the row of "Database". You will be able to add multiple users by writing them in multiple rows separated by commas, using the pattern described. You may even add them to an existing group, or to a new one:



Add users

In this case, you need to provide the user's login, full_name, e-mail address, and password, in this order and separated by commas.

```
j.doe, John Doe, john@institution.edu, appl3_1980
j.smith, Jack Smith, jack@institution.edu, h4m_1980
```

Users

Users:

```
j.doe, John Doe, john@institution.edu, appl3_1980
j.smith, Jack Smith, jack@institution.edu, h4m_1980
```

Add the user list using the detailed format

For instance, if you add them to the Physics groups, they will inherit the permissions granted to this group.

2.2.4 Inviting users to register or to join a group

In some cases, you will want to explicitly manage your users, creating their accounts yourself and placing them into specific groups. However, often you will want to give your users the chance to register themselves, or to join a specific group (to get access to certain labs) on their own.

This is useful, for instance:

- When you don't want to specify the login, name and e-mail for every user.
- When you want a broad group of people to join, without knowing exactly who belong to that group.
- When you want to publicly invite people to join.

To use the invitations system, you can follow the following steps:

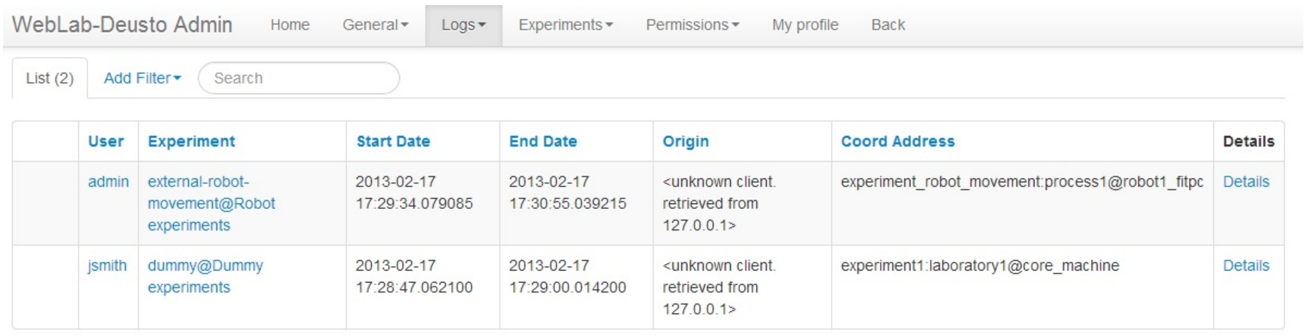
1. First, you will need to create a WebLab group for your invitees to join (unless you have that group already). All invitations are linked to a group, which they automatically join after accepting the invitation. That way, they will automatically get access to the experiments you intend, thanks to that group's privileges.
2. Now, you will need to create the Invitation itself. This can be done through the WebLab administrator. Go to the administrator, then to the "Users" menu, then click on "Invitations". To create the invitation you will need to specify the group to join, the expire date, the max number of people who can accept it, and whether the

invitation can be used to register new users. If that last option is disabled, then only existing WebLab-Deusto users will be able to use the invitation to join a group. Users without an account won't be able to create one.

3. Once you have created the invitation, in the list of invitations you will see the URL of each one. Now all that is left is to share the Invitation URL with your prospective students. Depending on the invitation, they will be able to use it to register a new account altogether, or to join the invitation's group with their existing account.

2.2.5 Tracking users

Now you can start again the WebLab-Deusto instance, and you can use the laboratory with different users. Once you log in the Administration panel, go to "Logs" and you will see who has accessed when:



	User	Experiment	Start Date	End Date	Origin	Coord Address	Details
	admin	external-robot-movement@Robot experiments	2013-02-17 17:29:34.079085	2013-02-17 17:30:55.039215	<unknown client. retrieved from 127.0.0.1>	experiment_robot_movement:process1@robot1_fitpc	Details
	jsmith	dummy@Dummy experiments	2013-02-17 17:28:47.062100	2013-02-17 17:29:00.014200	<unknown client. retrieved from 127.0.0.1>	experiment1:laboratory1@core_machine	Details

By using the "Add filter", you may search by user, date, or similar.

2.2.6 Customizing the deployment

In this section, we have presented a very simple deployment. However, this deployment can be configured. While in the *next section*, we'll learn to configure redis, MySQL or Apache, there are some settings that we can modify at this level.

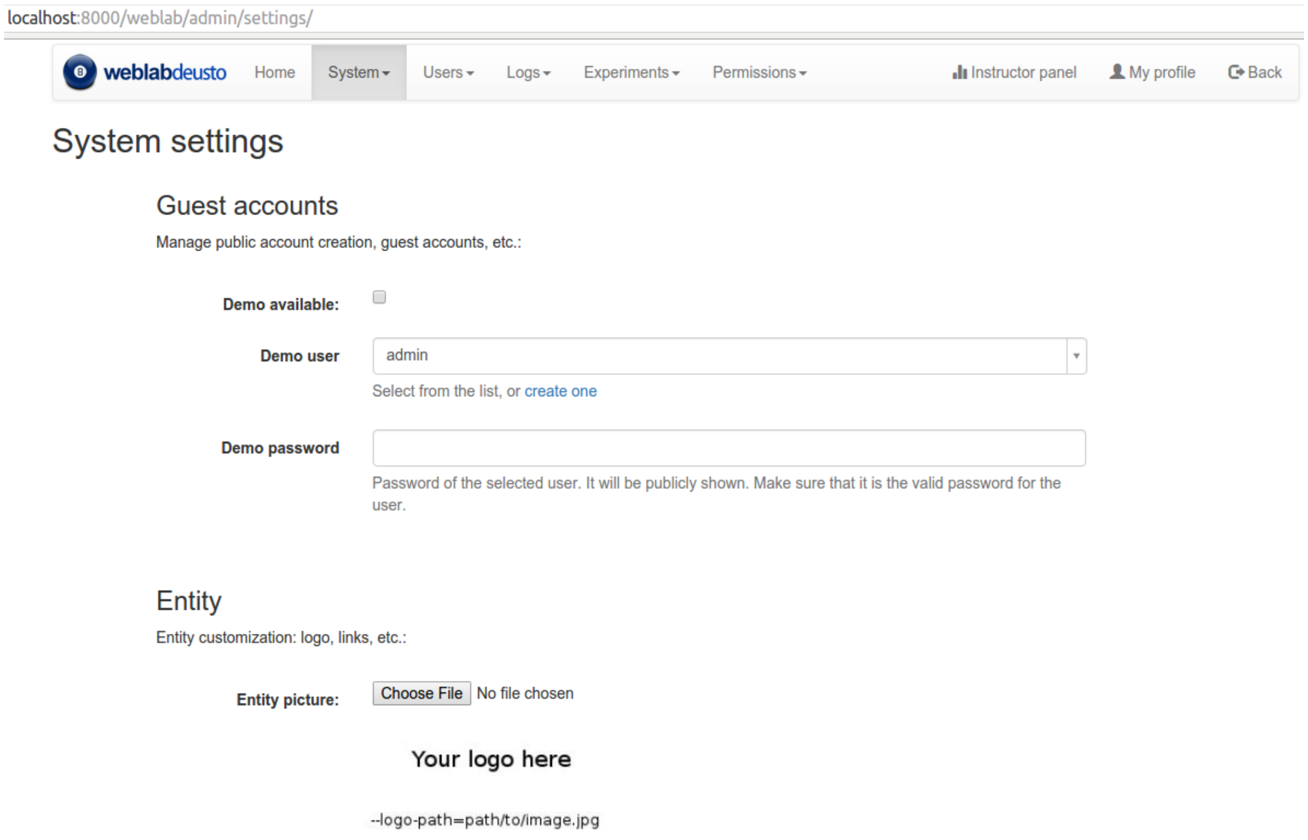
Running:

```
$ weblab-admin create --help
```

Displays the full help regarding the create command. A more advanced example would be:

```
$ weblab-admin create other.example --http-server-port=8001 --start-port=20000 \
--system-identifier='My example' --entity-link='http://www.myuniversity.edu/'  \
--poll-time=300 --admin-user=administrator --admin-name='John Doe'             \
--admin-password=secret --admin-mail='admin@weblab.myuniversity.edu' --logic
```

This example will be run in other port (8001), so you can start it at the same time as the other deployment without problems. Just go to http://localhost:8001/ instead, log in with user *administrator* and password *secret*, and see how there is another laboratory called *logic*. Many of the fields can always be changed with the administration panel. For example, in System and then Settings you can add a demo account, change the URL and logo of the school or provide a Google Analytics code.



System settings

Guest accounts

Manage public account creation, guest accounts, etc.:

Demo available: ☐

Demo user: admin

Select from the list, or create one

Demo password:

Password of the selected user. It will be publicly shown. Make sure that it is the valid password for the user.

Entity

Entity customization: logo, links, etc.:

Entity picture: Choose File No file chosen

Your logo here

--logo-path=path/to/image.jpg

2.2.7 Moving the deployment to a different directory or reinstalling WebLab-Deusto

Say you have installed WebLab-Deusto in a location and you need to move to a different directory. All the web server configuration files will be pointing with absolute paths to the old directory. The easiest way to override the existing HTTPd configuration and make it point to the proper paths is running:

```
$ weblab-admin httpd-config-generate sample
Generating HTTPd configuration files... [done]
$
```

After running this, restarting it and restarting the web server should be enough.

Other examples, such as using Virtual Machines, VISIR, etc., are documented in the *next section*.

2.2.8 Join the community

Once you have installed WebLab-Deusto (or if you have any trouble installing it), please join our small community:

- https://groups.google.com/forum/?fromgroups#!forum/weblabdeusto

So we can all exchange experiences, tips, tricks or concerns on how to create, maintain and share better remote laboratories. If you don't like writing in public, feel free to contact us privately at any point at weblab@deusto.es But exchanging the experience in public can be benefitial for all the members.

2.3 Installation: further steps

Table of Contents

2.3.1 Introduction

As previously detailed, right now you should have the simplest WebLab-Deusto up and running. It uses SQLite as main database (so only one process can be running) and SQLite as scheduling mechanism (which is very slow). Additionally, you have all the servers in a single process, so you can not spread the system in different machines. Finally, you are not using a real HTTP server, but the one built-in, which is very slow and not designed for being used in production. These settings in general are highly not recommended for a production environment.

In this section we will focus on installing and validating the installation of more components, as well as playing with simple deployments which use these installations. With these components, it is possible to enhance the performance in the next section: *Performance*.

2.3.2 Installing external systems

At this point, you have installed the basic requirements. However, now you should install three new external components:

- **Apache HTTP server:** By default, WebLab-Deusto uses a built-in, simple HTTP server written in Python. This web server is not aimed to be used in production, but only for demonstration purposes. For this reason, the Apache web server (or any other supporting proxies) is recommended.
- **MySQL:** By default, WebLab-Deusto uses SQLite. This configuration suits very well low-cost environments (such as Raspberry Pi, where it works). However, in desktop computers or servers, this restricts the number of processes running the Core Server to one, since SQLite can not be accessed concurrently. Even more, it restricts the number of threads to one, so it becomes a bottle neck. For few students, this might be fine, but as the number of students increase, this becomes an important problem. For this reason, it is better to use other database engine. The one used in the University of Deusto for production is MySQL.
- **Redis:** There are two main backends for scheduling: one based on SQL (and therefore, it can use MySQL or SQLite), and other based on Redis (a NoSQL solution that keeps information in memory, becoming very fast). Even in low cost devices, the latter is recommended. However, it is only officially supported for UNIX. Therefore, if you are running Mac OS X or Linux, install Redis and use it as scheduling backend to decrease the time required to process users.

GNU/Linux

All these components are open source and very popular, so they are in most of the package repositories of each distribution. For example, in Ubuntu GNU/Linux, you only need to install the following:

```
sudo apt-get install apache2 mysql-server redis-server
```

If you are not using PHP, it is highly recommended to install the `worker` MPM by running:

```
sudo apt-get install apache2-mpm-worker
```

Note: For apache on Ubuntu (>16.04) `apache2-mpm-worker` is included by default.

This makes that Apache uses threads rather than processes when attending a new request. This way, the amount of memory required with a high number of concurrent students is low. However, it is is usually not recommended when also using PHP, so whenever you install PHP this MPM is usually removed. If you need to run both, you can use the `prefork` MPM, while take into account that it will require more memory. This is explained in detail in the official site.

Regarding `redis`, take into account that redis performs all the operations in memory **but** from time to time it stores everything in disk, adding latency. It is recommended to avoid this. In the `/etc/redis/redis.conf` file, comment the following lines:

```
save 900 1
save 300 10
save 60 10000
```

By adding a # before.

Microsoft Windows

In Microsoft Windows, you can install both the Apache HTTP server and MySQL by using XAMPP. Download it and install it. XAMPP comes with a control panel to start and stop each service. In WebLab-Deusto, we are only interested in Apache and MySQL.

Once installed, it is recommended to have the MySQL client in console, so either do this:

```
set PATH=%PATH%;C:\xampp\mysql\bin
```

Or go to the Microsoft Windows Control Panel -> System -> Advanced -> Environment variables -> (down) PATH -> edit and append: `;C:\xampp\mysql\bin`.

If you have problems with XAMPP, check their FAQ.

Regarding Redis, there is an unofficial version of Redis for Microsoft Windows, with a patch developed by Microsoft. However, while the support is not official or there is an officially supported side project for supporting Microsoft Windows, we are not recommending its use. So if you are running Microsoft Windows, simply skip those sections and use MySQL for scheduling.

Mac OS X

In Mac OS X, Apache is usually installed by default. However, you must install MySQL by using the official page. You can install Redis by downloading it and compiling it directly. If you do not manage to run it, remember that it is an optional requirement and that you can use MySQL as scheduling backend.

2.3.3 Installing native libraries

By default, the installation process installed a set of requirements, which are all pure Python. However, certain native libraries make the system work more efficiently. That said, these libraries require a C compiler to be installed and a set of external C libraries, which might not be available in Microsoft Windows environments. However, in GNU/Linux, they are recommended.

For this reason, in Ubuntu GNU/Linux install the following packages:

```
# Python
$ sudo apt-get install build-essential python-dev
# MySQL client, for an optimized version of the MySQL plug-in
$ sudo apt-get install libmysqlclient-dev
# LDAP
$ sudo apt-get install libldap2-dev
# SASL, SSL for supporting LDAP
$ sudo apt-get install libsasl2-dev libsasl2-dev libssl-dev
# XML libraries for validating the configuration files
$ sudo apt-get install libxml2-dev libxslt1-dev
# Avoid problems with freetype:
$ sudo ln -s /usr/include/freetype2 /usr/include/freetype
```

Once installed, it is now possible to install more optimized Python libraries, by running:

```
$ cd weblab/server/src/
$ pip install -r requirements_suggested.txt
```

From this moment, libraries that improve the performance will be installed.

2.3.4 Scheduling

There are two main database backends for scheduling:

- **SQL based:** using the SQLAlchemy framework. Two database engines are supported:
  - Using `SQLite`, which is fast but it requires a single process to be executed, so multiple users are managed in a single thread and the latency increases.

  - Using `MySQL`, which supports multiple students accessing to different servers, distributed in several processes or even machines.
- **Redis:** which uses redis, and provides faster results but does only work on UNIX environments at this point.

By default in the introduction section, you have used `SQLite`. So as to use `MySQL` as database engine, run the following:

```
$ weblab-admin create sample --coordination-db-engine=mysql
```

Additionally, you may pass other arguments to customize the deployment:

```
$ weblab-admin create sample --coordination-db-engine=mysql \
  --coordination-db-name=WebLabScheduling \
  --coordination-db-user=weblab     --coordination-db-passwd=mypassword \
  --coordination-db-host=localhost  --coordination-db-port=3306
```

However, if you want to use `Redis`, run the following:

```
$ weblab-admin create sample --coordination-engine=redis
```

Additionally, you may pass the other arguments, such as:

```
$ weblab-admin create sample --coordination-engine=redis \
  --coordination-redis-db=4  --coordination-redis-passwd=mypassword \
  --coordination-redis-port=6379
```

So as to change an existing deployment, you may check the variables explained at *Configuration variables*, which are located at a file called `machine_config.py` in the `core_machine` directory.

2.3.5 Database

The WebLab-Deusto database uses SQLAlchemy, which is a ORM for Python which supports several types of database engines. However, in WebLab-Deusto we have only tested two database engines:

- `SQLite`: it is fast and comes by default with Python. It suits very well low cost environments (such as Raspberry Pi).
- `MySQL`: on desktops and servers, it makes more sense to use MySQL and a higher number of processes to distribute the load of users among them.

So as to test this, run the following:

```
$ weblab-admin create sample --db-engine=mysql
```

Additionally, you may customize the deployment with the following arguments:

```
$ weblab-admin create sample --db-engine=mysql  \
  --db-name=MyWebLab     --db-host=localhost    \
  --db-port=3306         --db-user=weblab       \
  --db-passwd=mypassword
```

Note: It may happen that you get an error of authentication when doing this, because in modern Linux servers MySQL does not have by default a username and password for root. If this is the case, run the following:

```
$ sudo mysql -uroot
mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY
↪'password';
mysql> exit
$
```

Then, you will be able to follow the installation if when prompted for a root administrator you provide root and for the password you put whatever you put as 'password'.

You may also change the related variables explained at *Configuration variables*, which are located at a file called `machine_config.py` in the `core_machine` directory.

2.3.6 Secure the deployment

This section covers few minimum steps to secure your WebLab-Deusto deployment.

Secure the communications

WebLab-Deusto supports HTTPS, and it is designed so that it can easily work with it (e.g., in the managed approach, all the connections go through the core server). **We highly recommend you to install SSL certificates** to reduce the risk of potential attacks to your WebLab-Deusto deployment, especially if you or your students submit the credentials through WebLab-Deusto (as it happens when using database passwords or LDAP).

Note: A note about SSL

In case you are unfamiliar with HTTPS (HTTP Secure or HTTP over SSL), all the web uses the HTTP protocol (**http://**). However, this protocol goes unencrypted, so anyone in the middle (people in the same WiFi, ISPs, layers in the middle between the final client and the server. . .) can read the traffic. For this reason, HTTPS (**https://**) was developed, which supports HTTP through an SSL connection, which encrypts the communications. Nowadays there is a big effort to make as much of the web use HTTPS (e.g., not only e-commerce sites but also google.com, Wikipedia, Facebook and even this website where you are reading this. . . all go through HTTPS).

You can generate SSL certificates by yourself (and signed by yourself). However, in general web browsers will not accept them (or they will show a big warning before accessing), because otherwise you could create an SSL certificate for another website that you do not own, and they would not be able to know. This could lead to different types of attacks.

For this reason, web browsers come with a set of CA (Certificate Authorities), and they only trust whatever is signed by them (or signed by whoever they delegate). Additionally, they have other complex mechanisms (such as lists of revoked certificates, etc.).

So, when you install a valid certificate, some CA (or delegated) will verify that you are the valid owner of a server, and it will create and sign a certificate for you. When users access your website using **https://** to your host, when starting the connection they will automatically download the public key (which they will use for encrypting) and the signature of this key provided by a CA. They will validate with the installed CA if this key is valid for this particular domain (e.g., `weblab.yourinstitution.edu`, and if it is, it will proceed to encrypt the connection). Otherwise (e.g., the key is expired, the CA does not recognize the signature, the server name is different -www.weblab.yourinstitution.edu instead of weblab.yourinstitution.edu-, the key is in a revocation list), it will show an error instead.

As a final note, one certificate can server multiple domain names for a particular server. For example, you might have a certificate for `*.weblab.yourinstitution.edu` and you can use it in different servers (e.g., `cams.weblab.yourinstitution.edu`, `www.weblab.yourinstiution.edu`. . .). Those are called *wildcard certificates* (and if you choose to request those, take into account that `*.weblab.yourinstitution.edu` is not valid for `weblab.yourinstitution.edu` so in addition you'll need an alternate name). You may also select different names, listed in what is called the *Alternate names* (manually providing a list, such as `weblab.yourinstitution.edu` and `www.weblab.yourinstitution.edu` and `cams.yourinstitution.edu`, etc.).

So, once you have installed WebLab-Deusto in your **final server** (i.e., with a proper hostname such as `weblab.yourinstitution.edu`), you might want to install the SSL certificates. To do so, there are three approaches:

- `Contact your IT services`: many institutions (e.g., universities, research centers) already have agreements to create free SSL certificates. You should first contact to your IT services to see if they provide you this service.
- `Buy a SSL certificate`: there are many websites where SSL certificates are sold and managed, with different options of security.
- `Get a free SSL certificate by Let's Encrypt`: Let's Encrypt is an open initiative to secure the Internet that provides free SSL certificates in an automatic basis. The certificates only last a couple of months, but you can renew them automatically. All what you need is having your server already configured with the **final** IP address and hostname (so they automatically verify that `weblab.yourinstitution.edu` is indeed your server), and running already a proper web server (e.g., Apache or nginx). For more information on how to do it (it literally takes a couple of minutes), go to the Certbot site created by the EFF (Electronic Frontier Foundation). It tells you what software to install and how. `Let's Encrypt` does not support wildcard certificates, but it supports as many alternate names as you want.

Once you install the certificate in your Apache server (each provider will explain you how), you should go to the `core_host_config.py` file and change the `core_server_url` variable to your final URL (e.g., `https://weblab.yourinstitution.edu/weblab/`).

Additionally, in Apache there is a directive that you might want to use in the `VirtualHost` using the 80 port such as:

```
RedirectMatch ^/weblab/(.*)$ https://weblab.yourinstitution.edu/weblab/$1
```

So that everything that arrives to the 80 port (**http://**) is forwarded to the 443 port (**https://**).

Close access to local services

The internet is a quite dangerous place, where there are robots constantly checking random IPs and searching for open services to attack (such as databases, shared directories, cameras, printers…). In your WebLab-Deusto server, you probably don't want anything open other than the WebLab-Deusto server (and other services that you in purpose want open). There are two ways to do this, and we recommend both:

- First, install a proper firewall. You might use the one provided by your Operating System (such as the Windows Firewall in Microsoft Windows, or iptables in Linux). Make it possible to access only those services that you need open. WebLab-Deusto itself does not require any port open (only those for the web browser, which are 80 and 443).
- Second, review your services. In particular, make sure that both Redis and MySQL are bound to 127.0.0.1 (instead of open to the whole Internet). This is usually established in its configuration files (e.g., search for a parameter called `bind-address` in MySQL or `bind` in redis. It may be called `listen` in other services).

After doing it, or in case of doubt, check from outside (e.g., your home) connecting to those ports:

```
(3306 is the default MySQL port)
$ telnet weblab.myinstitution.edu 3306
Trying 1.2.3.4...
telnet: Unable to connect to remote host: Connection timed out
$
```

(continues on next page)

(continued from previous page)

```
(6379 is the default Redis port)
$ telnet weblab.myinstitution.edu 6379
Trying 1.2.3.4...
telnet: Unable to connect to remote host: Connection timed out
$
```

If the response is something like:

```
telnet: Unable to connect to remote host: Connection refused
```

it's also fine. However, if it ever says:

```
$ telnet weblab.myinstitution.edu
Trying 1.2.3.4...
Connected to weblab.myinstitution.edu.
Escape character is '^]'.
```

It means that those ports are open and can be accessed by attackers. By default, some services (as MySQL) require credentials, but sometimes there is a vulnerability in the software and external attackers can access more than they should. Also, if you are using easy passwords (e.g., the ones in the documentation), the risk of attack increases if the services are open to the Internet.

For those services that you also want to make available but only for you (and not for the general audience), you should also change the default ports. For example, if you use Remote Desktop, VNC or SSH, you can use it in a different port than the default one. For example, SSH is a secure service, but it has had important vulnerability problems in the past. And for those robots that are constantly checking for services open, they might be looking in each IP address for a SSH service running in the 22 port (the default one). If you have it in the 16483 one, it might be more difficult for them to find it and attack it, unless they're indeed targeting your server. As an additional measure, there are approaches such as port-knocking which let you define a set of random ports (e.g., 5356, 15243 and 9513), and when you *knock* them (e.g., trying to connect to them) in that order, suddenly the firewall opens access to these services (e.g., SSH). This way, even if someone checks all the ports open in your server, they will only find the public ones (e.g., Apache), and only if they connect to different ports in an order they will see that service available.

Upgrade your software frequently

All software is inherently subject to have vulnerabilities. Once they are discovered and fixed, when you upgrade them, the vulnerabilities are not there anymore. However, if you upgrade once a month, then you might run into troubles for that month.

This does not mean that you need to use the latest version of the software, just those which are maintained. For example in the case of Ubuntu, you do not need to install the latest Ubuntu distribution. If you are using a Ubuntu Server 12.04 LTS, it will be supported until June 2017. You are of course encouraged to use Ubuntu 16.04 LTS (the latest LTS), but it is not really a priority. What is important is to use an Operating System version that is still supported (and for this reason, in the case of Ubuntu, it is better to install LTS versions -that are supported for longer: e.g., 14.04, 16.04- than not LTS versions -e.g., 16.10-) and upgrade it every day (you can install a script for that). If you are using software not managed by your operating system (e.g., Apache on Windows), you should also upgrade it frequently (and you can join for example their mailing lists to be notified of new versions). This is not required in systems as Linux, where most of the software required by WebLab-Deusto is installed from the repositories. However, you still have to make sure that it is upgraded frequently.

It is also important to *upgrade the WebLab-Deusto* regularly (not so often as every day, but keep it in mind). It's not only about WebLab-Deusto itself, but about the libraries used by WebLab-Deusto (which are automatically upgraded when you upgrade WebLab-Deusto). Usually in the main screen of WebLab-Deusto you have a link to GitHub (where it says `version r<number>`). If you click that link and compare it with this one, you can see if there were new versions since you last upgraded it. You may also use the *WebLab-Deusto mailing list* to receive notifications on potential issues.

2.3.7 Deployment

Note: This section is only for deployments in UNIX environments. In Windows environments you can use services by wrapping WebLab into `.bat` files.

WebLab-Deusto can be run as a script, but you might want to deploy it as a service. However, given that it is very recommendable **not** to install it as root (unless you play with virtuaelnvs to avoid corrupting the system with wrong versions of the libraries), it is better to install it in a system such as supervisor. In supervisor you can add any type of program and they will run as services. You also have a tool to control which services are started, or restart them when required (e.g., when upgrading or modifying the `.py` or `.yml` files).

This section is focused on how to install this tool in a UNIX (e.g., Linux) environment.

Step 1: installation of supervisor

Depending on your Operating System, you might find it in the OS packages itself. For example, in Ubuntu run:

```
$ sudo apt-get install supervisor
```

And you're done. Otherwise go to supervisor docs on installation for futher information.

Once installed, you'll see that you can start supervisor and check the status:

```
$ sudo service supervisor start
$ sudo supervisorctl help

default commands (type help <topic>):
=====================================
add    exit      open  reload  restart   start   tail
avail  fg        pid   remove  shutdown  status  update
clear  maintail  quit  reread  signal    stop    version

$ sudo supervisorctl status
$
```

It is normal that status returns nothing since we have not installed any service yet.

Step 2: prepare WebLab for being used as a service

Let's imagine that you have installed WebLab-Deusto using `virtualenvwrapper` and called it `weblab`. Then, the virtualenv will typically be located in something like:

```
/home/tom/.virtualenvs/weblab/
```

And the activation script will be in:

```
/home/tom/.virtualenvs/weblab/bin/activate
```

And let's imagine that you have created a new WebLab-Deusto instance in your home directory, in a `deployments` directory and called it `example`, such as:

```
$ cd /home/tom/deployments/
$ weblab-admin create example --http-server-port=12345
```

Then, we will create a wrapper file in any folder (e.g., in the `deployments`) directory called for example `weblab-wrapper.sh` which will contain the following three lines:

```
#!/bin/bash
_term() {
   kill -TERM "$child" 2>/dev/null
}

# When SIGTERM is sent, send it to weblab-admin
trap _term SIGTERM

source /home/tom/.virtualenvs/weblab/bin/activate
weblab-admin $@ &

child=$!
wait "$child"
```

And then we will grant execution privileges to that file:

```
$ chmod +x /home/tom/deployments/weblab-wrapper.sh
```

From this point, calling it from anywhere will use the virtualenv will work:

```
$ cd /tmp/
$ /home/tom/deployments/weblab-wrapper.sh
Usage: /home/tom/.virtualenvs/weblab/bin/weblab-admin option DIR [option arguments]

    create                  Create a new weblab instance
    start                   Start an existing weblab instance
    stop                    Stop an existing weblab instance
    monitor                 Monitor the current use of a weblab
    instance
    upgrade                 Upgrade the current setting
    locations               Manage the locations
    database
    httpd-config-generate   Generate the HTTPd
    config files (apache, simple, etc.)

$
```

Step 3: Create the configuration for supervisor

Now what you have to do is to create a file such as `example.conf` (it is important that it ends by `.conf`) for running the example instance as a service. To do so, create a file such as the following:

```
[program:example]
command=/home/tom/deployments/weblab-wrapper.sh start example
directory=/home/tom/deployments/
user=tom
stdout_logfile=/home/tom/deployments/example/logs/stdout.log
stderr_logfile=/home/tom/deployments/example/logs/stderr.log
killasgroup=true
```

There are plenty more of configuration variables in supervisor (such as not exceeding the stdout/stderr logs in more than a number of MB, moving them until you have more than 10 files, etc.): check the documentation at the supervisor [program:x] section documentation.

Step 4: Add the configuration to supervisor

Then, you have to add this file to supervisor. In Ubuntu Linux this is typically done by copying the file to `/etc/supervisor/conf.d/` and then using the `supervisorctl` to add it:

```
$ sudo cp example.conf /etc/supervisor/conf.d/
$ sudo supervisorctl update
example: added process group
$
```

At this point, you might check that your WebLab-Deusto instance is running. By default when you update the supervisorctl, it runs the process. First check in:

```
$ sudo supervisorctl status
example                          RUNNING   pid 12428, uptime 0:00:04
$
```

And then go with your web browser to see if it is running (in the example created, you can go to `http://localhost:12345/`, but you should be using Apache as described above).

Step 5: Try supervisor

Once configured, it becomes easier to start the cycle of the deployment. For example:

```
$ sudo supervisorctl start example
example: started
$ sudo supervisorctl status example
example                          RUNNING   pid 19320, uptime 0:00:18
$ sudo supervisorctl stop example
example: stopped
```

If you have more than WebLab-Deusto deployment, you can always do the following to start them all:

```
$ sudo supervisorctl start all
example1: started
example2: started
$ sudo supervisorctl stop all
example1: stopped
example2: stopped
$
```

If you have to make any change on the `example.conf`, remember to run:

```
$ sudo supervisorctl update
```

So supervisor checks the settings again.

Note: Make sure that supervisor starts itself when you reboot your computer (so try rebooting). In some systems by default it doesn't. In Ubuntu 16.04, for example, you have to run the following command:

```
$ sudo systemctl enable supervisor
```

You might know that supervisor is active because otherwise any command will fail with a message such as:

```
$ sudo supervisorctl status
unix:///var/run/supervisor.sock no such file
$
```

Note: If you want to use this for testing environments, and you don't need them to start every time (e.g., only when you want them to start), you just have to detail that in the `example.conf` file by appending:

```
autostart=false
```

2.3.8 Summary

With these components installed and validated, now it is possible to enhance the performance in the next section: *Performance*.

2.4 Performance

Table of Contents

2.4.1 Introduction

This section focuses on explaining how to increase the performance of the system by customizing it with the proper arguments. You may use the WebLab Bot (*WebLab Bot*) to see what parameters work best in your environment.

2.4.2 Core servers

Python has a Global Interpreter Lock (GIL) that makes the threading model not work as could be expected when coming from other programming languages. Internally in Python, when a thread is being run, it executes a number of instructions which can be configured and then it swaps the context and other thread is executed. Whenever there is an IO (input/output) operation, or some extension developed in C is called, the context will also be swapped. The problem is that during the execution of the pure Python not-IO operations, the GIL is locked, so no other operation can be executed in other processor.

Given the amount of IO in WebLab-Deusto (IO includes not only the requests from the client and the requests to the particular laboratory, but also the database and all the access to the Redis server), certain concurrence will often occur even with a single process. However, in order to take advantage of the nowadays common dual, quad or more core processors, it can be managed using multiple copies of Python processes instead of relying on the Python threading model.

For this reason, WebLab-Deusto has been designed so it can scale and multiple independent processes can be executed not only in different machines, but also in the same machine to mitigate this effect. This way, having 4 processes running WebLab-Deusto in a quad core machine will increase the throughput.

In the following drawing, the WebLab Bot (see *WebLab Bot*), which is a student simulator that tests different loads of highly concurrent users (i.e. users clicking on the *Reserve* button at the very same time), measures times using different numbers of processes (1 and 5) with different database backends (MySQLdb and PyMySQL). This shows the reservation method:

As it is shown, when using a single process (`1 core`), MySQL performs better once you increase the number of concurrent students. There are two Python libraries for using MySQL, one in pure Python (PyMySQL), and a native one, which requires you compiling the code as explained in the section *Installing native libraries*. The native one works faster than the pure Python version.

However, the biggest change is when you increase the number of processes (e.g., `5 core servers`). This is something you can not do if you are using SQLite, but yes with MySQL. For this reason, you will get an error when you run:

```
$ weblab-admin create sample --cores 4
ERROR: sqlite engine selected for coordination, general database is
incompatible with multiple cores
```

The way to create a new WebLab-Deusto deployment with 4 core servers and using MySQL for both database and scheduling is:

```
$ weblab-admin create sample --cores 4 --coordination-db-engine=mysql --db-
↪engine=mysql
```

This generates a WebLab-Deusto system with 4 Core servers and 4 Login servers. Apache will balance the load of users among them, so each of these process will only process a subset of the users.

2.4.3 Scheduling backends

So as to store information (permissions, uses, students, etc.), WebLab-Deusto uses a relational database (MySQL or SQLite). However, for managing the scheduling (who's first), it may use a relational database (again, MySQL or SQLite) or the Redis NoSQL system. This backend also supports load balancing (and therefore, multiple core servers), but since all the information is managed in memory, it is much faster.

Indeed, in the following figure the MySQLdb library with MySQL is compared with Redis, with 1 and 5 core servers. As it can be seen in the drawing, Redis is considerably faster. In older machines, this difference is even multiplied.

For this reason, using Redis is recommended. However, Redis is not officially supported on Microsoft Windows at this moment.

2.4.4 Apache

You should use a robust HTTP server instead of the one that comes by default when you pass the following option:

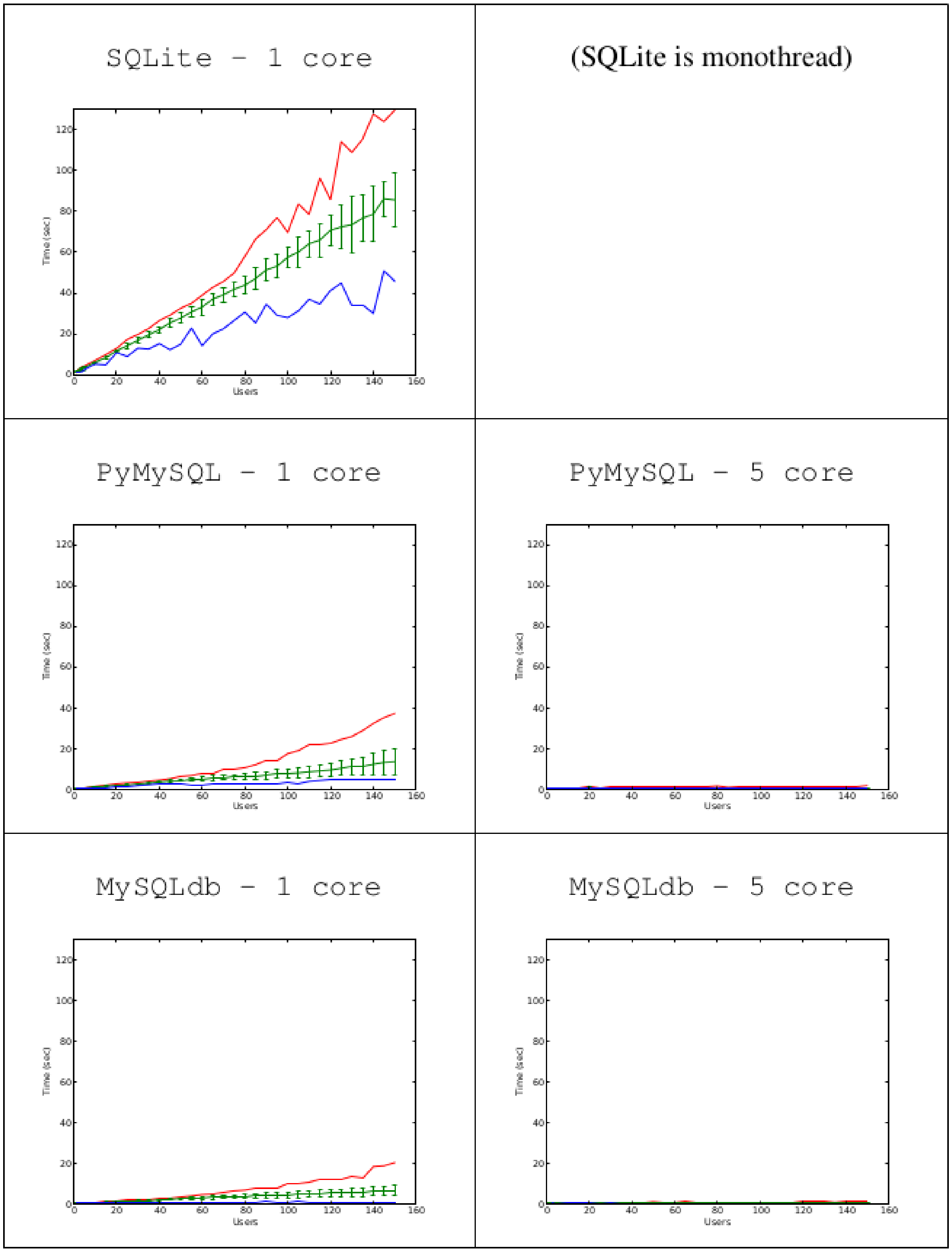


Fig. 1: The red line represents the maximum value, the blue line the minimum value, and the green line the mean and the standard deviation. Each measurement (e.g., 140 students with the MySQL db) have been taken 5 times.

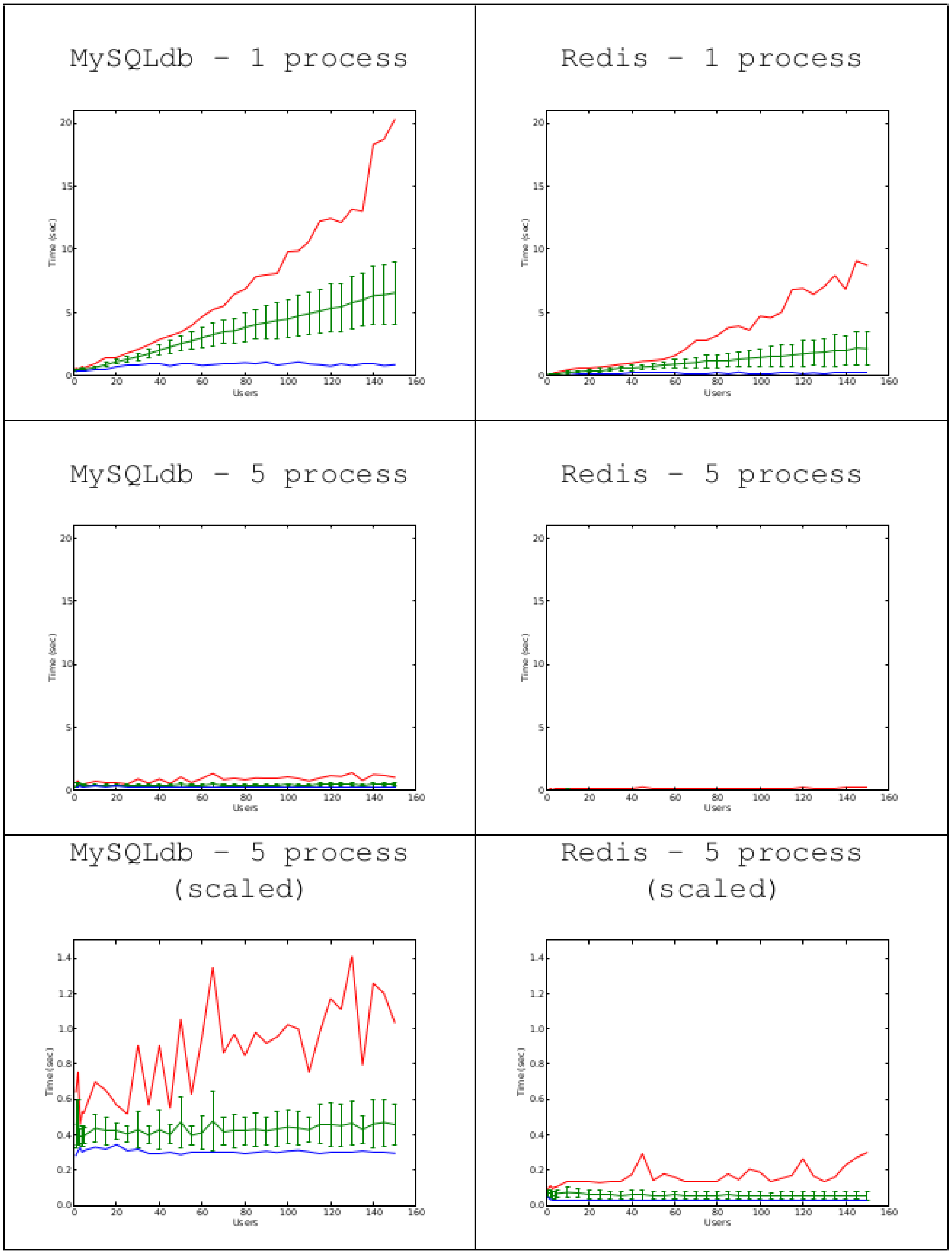


Fig. 2: The red line represents the maximum value, the blue line the minimum value, and the green line the mean and the standard deviation. Each measurement (e.g., 140 students with the MySQL db) have been taken 5 times.

```
$ weblab-admin create sample --http-server-port=12345
```

At this moment, WebLab-Deusto generates the configuration for the Apache HTTP server, so you might use it. Support for autogenerating the configuration of other servers might be added soon. When you create the deployment, the message shown explains what you need to add in which files. For example, in GNU/Linux, at the time of this writing, it details the following:

```
$ weblab-admin create sample

Congratulations!
WebLab-Deusto system created

Append the following to a new file that you must create called /etc/apache/conf.d/
↪weblab

    Include "/tmp/sample/httpd/apache_weblab_generic.conf"

And enable the modules proxy proxy_balancer proxy_http headers.
For instance, in Ubuntu you can run:

    $ sudo a2enmod proxy proxy_balancer proxy_http headers

Then restart apache. If you don't have apache don't worry, delete sample and
run the creation script again but passing --http-server-port=8000 (or any free port).
```

This message is different in each operating system, and it takes into account what files it finds.

Additionally, as previously explained, Apache has different MPMs. In GNU/Linux, when PHP is installed, Apache typically uses the `prefork` MPM. The `worker` MPM consumes much less memory, so it is recommended. However, if you need to support PHP or you are working in other operating systems, you may use the existing MPM, although you should measure how much memory is Apache consuming.

2.4.5 Raspberry Pi and low cost devices

WebLab-Deusto is a very light system, which does not require much memory. Indeed, we have successfully deployed the whole system even in Raspberry Pi devices, and measured the results. As you can see in the following drawing, this ARM device, with only 256 MB RAM, could manage different amounts of users, while the amount of time increased fastly. It was using SQLite as database, everything (Experiment Server, Core Server, Laboratory Server and Login Server) in a single process, and `Redis` (left) and `SQLite` (right) for scheduling.

As it can be seen there, even in a Raspberry Pi device, `Redis` is more suitable. However, in such a cheap device (around $ 35) the system becomes substantially slower. The typical deployment is having a set of regular servers for the main services (Core Server and Login Server), and multiple raspberries for the different experiments.

2.4.6 Summary

In this section, more complex deployments have been addressed. It uses extensively the `weblab-admin` script, and therefore, it does not explain how this is managed internally. So as to understand the files generated by this script, continue with the next section, *Directory hierarchy*.

2.5 Directory hierarchy

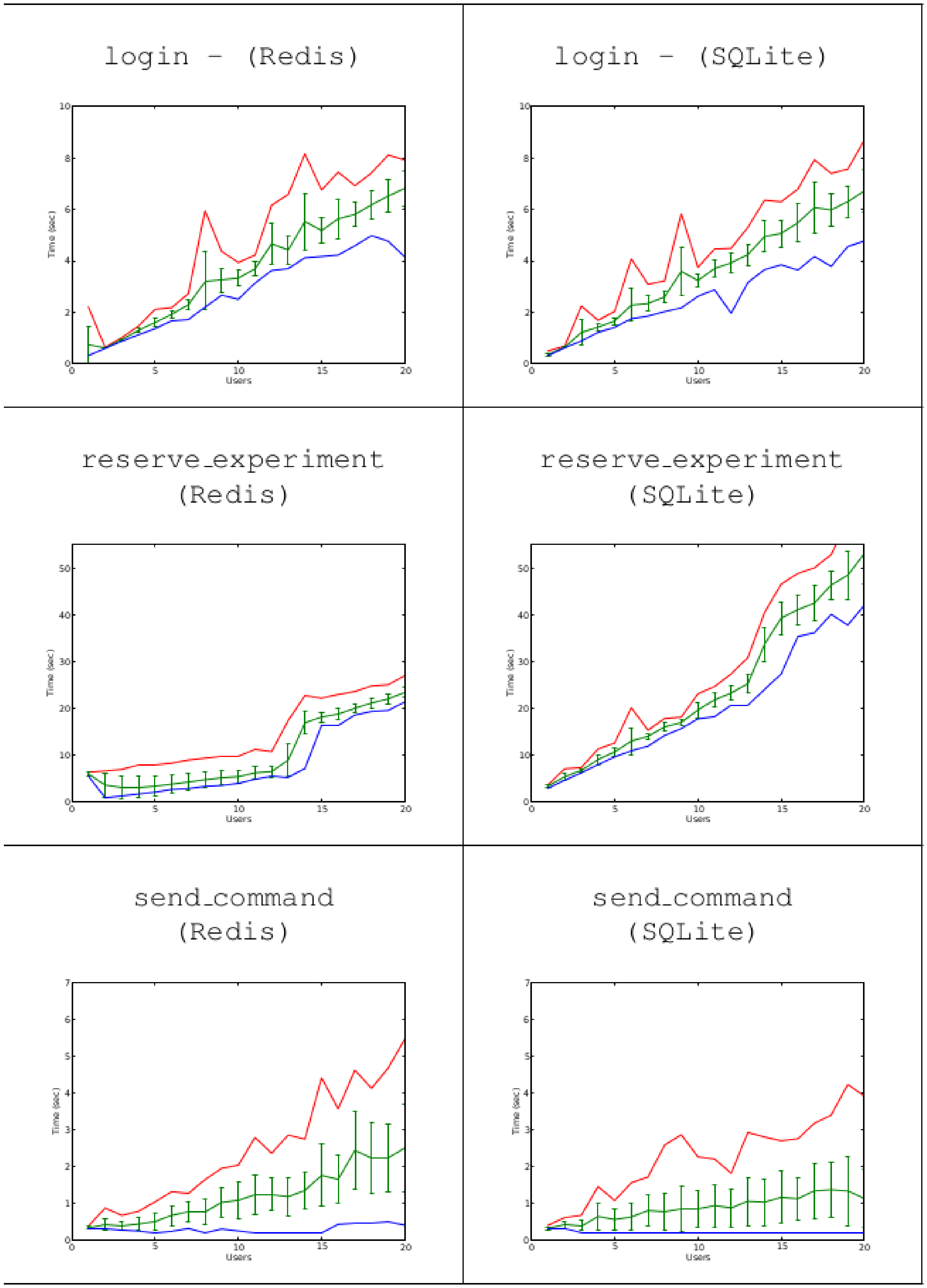


Fig. 3: The red line represents the maximum value, the blue line the minimum value, and the green line the mean and the standard deviation. Each measurement (e.g., 140 students with the MySQL db) have been taken 5 times. Note that each row has a different scale.

Table of Contents

2.5.1 Introduction

WebLab-Deusto uses a directory hierarchy which is also used for managing configuration. Basically, if you create a simple WebLab-Deusto instance:

```
$ weblab-admin create sample
```

You will see that it generates a set of files and directories:

```
+ sample
  + client
    + images
      - logo.jpg
      - logo-mobile.jpg
  - configuration.yml
  - core_host_config.py
  + db
  - debugging.py
  + files_stored
  + httpd
    - apache_weblab_generic.conf
    - simple_server_config.conf
  - lab1_config.py
  + logs
    + config
      - (...)
    - (...)
  + pub
      (empty)
  - run.py
```

2.5.2 Basic files and directories

The following files and directories are simple:

- `client`: contains the logo images. You can replace the images there directly.
- `db`: contains the databases, if stored in disk. When using SQLite, this will be the case and several `*.db` files will be stored. When using MySQL, this directory will be empty.
- `debugging.py`: contains information about which ports are mapped to which functionalities. It is used by the command `weblab-admin monitor sample` command, as well as by the Bot so as to know to what it must be connected.
- `files_stored`: if storing files in a laboratory (such as the FPGA, CPLD or PIC laboratories), by default files will be located in this directory. Please note that so as to store files, you have to configure the `core_store_students_programs` to `True` in the `core_host_config.py` file.
- `httpd`: contains configuration files for HTTP servers. By default, WebLab-Deusto comes with a built-in low performance HTTP server for testing. However, it is recommended to use the Apache HTTP server. This directory contains the configuration for both. In the future, we might generate configuration for other HTTP servers, such as nginx.
- `logs`: contains the log files generated by the application. It also contains the `config` directory, which contains the configuration on how much WebLab-Deusto should store.
- `pub`: contains public files or directories. They are available in http://localhost:8000/weblab/web/pub/ . You can always change them here, or in the Administration panel (in System: Public directory). If you are developing a new laboratory in JavaScript for example, you can put it there.
- `run.py`: the script that will launch this WebLab-Deusto instance.

Note that we have skipped a set of files (`configuration.yml`, `core_host_config.py`, `lab1_config.py`) in purpose. They are explained in the whole following section.

2.5.3 Configuration hierarchy

WebLab-Deusto uses a configuration hierarchy. This hiearchy is based on three major concepts:

- **Host**: refers to a physical computer.
- **Process**: refers to a process running in a host*.
- **Component**: refers to a functionality running in an process*.

Note: If you come from a previous WebLab-Deusto installation, you might be wondering that the terms do not fit. What it used to be **machine** is now called **host**, what it used to be **instance** is now called **process**, and what it used to be called **server** is now called **component**. The previous names were not explaining much what they were doing.

All the components described in *Technical description* are *components* using these categories. Each experiment server (e.g., a Robotics experiment) is a *component*.

Now, *components* can be grouped in a single *process* (at operating system level, this is indeed a single process) in a single *host*. However, they may also be distributed among different *hosts* (computers), each one containing multiple *processes*. For this reason, WebLab-Deusto provides a middleware that manages the communications, providing an addressing and registry system. For example, core servers are not implemented knowing where are the laboratory server. They ask the registry for **a** laboratory server, and they get the closest one, wherever it is and whatever the communication protocol is used.

This enables flexibility supporting multiple types of deployments. For instance, in a standalone system in a Single Board Computer (such as a Raspberry Pi), it is possible to deploy the whole thing in a single process. The communications among all the different components will not use HTTP or so, but simply a function call in Python. This optimization is provided by this middleware: if a Core server and a Laboratory server are in the same process, the communication will always be direct: when the Core server calls a method of the Laboratory server, internally it will be simply calling that method in the Laboratory server. However, if they are separated in a different network, it will use a network based protocol.

Basic structure example

Let's see a couple of example prior to proceeding. By running (as before):

```
$ weblab-admin create sample
```

We can see how this is generated (skipping the basic files explained above):

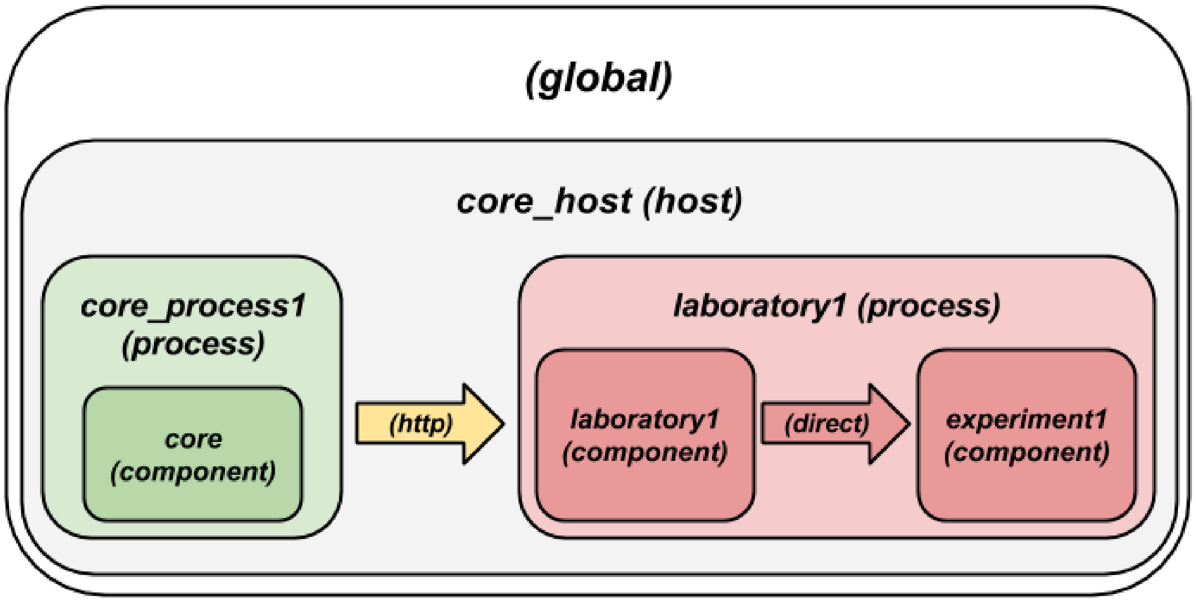
```
(...)
- configuration.yml
- core_host_config.py
- lab1_config.py
 (...)
```

If we open the `configuration.yml` file, we find the following:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
```

If you are not familiar with YAML, it is a very simple format where you can describe information quite condensed. In this case, you can see that there is a list of hosts, which is `core_host`, which has some properties (such as runner: `run.py` or config_file: `core_host_config.py`). It also has `processes`, and in this case, the list of processes contains two: `core_process1` and `laboratory1`. The first one has a single component called `core` (which is a Core Server according to its `type`) and the second one contains two components `experiment1` and `laboratory1` (which is a Laboratory Server, according to its `type`).

In this case, the core server will communicate with the Laboratory Server through a network in localhost, using an HTTP message in this case, as shown in the following diagram, while the Laboratory server will contact the Experiment Server using a simple Python call (it is in the same *process*).



Single process example

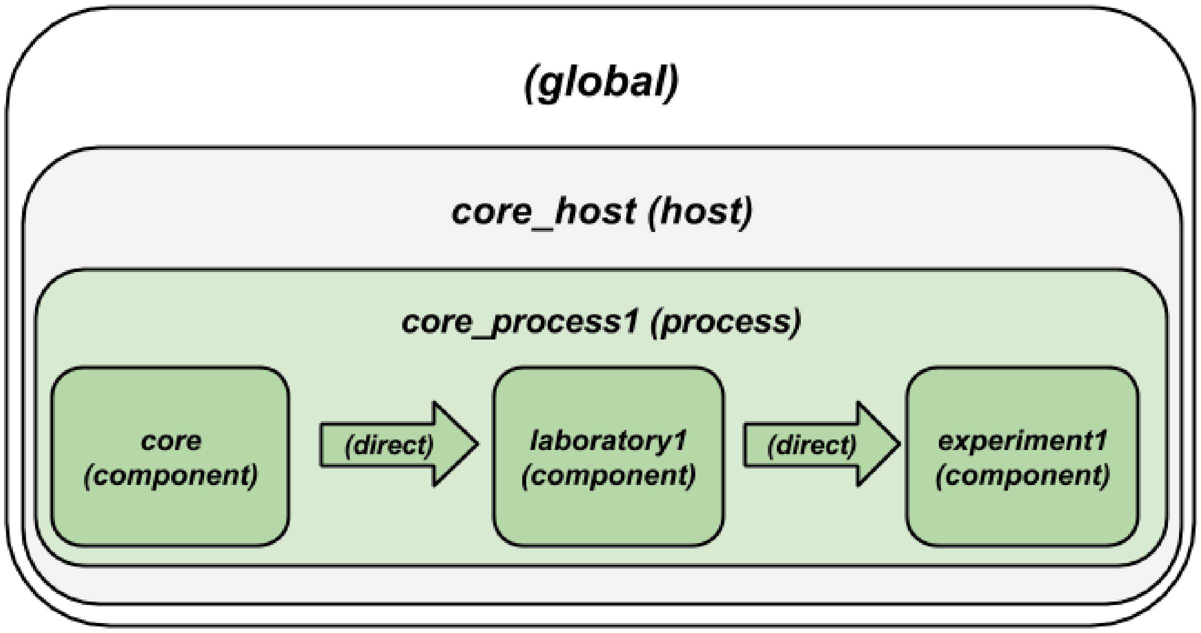
So as to illustrate a more compact example, where all the servers are running in a single process, run the following:

```
$ weblab-admin create sample2 --inline-lab-server
```

While the files are kind of the same, you can notice that the configuration.yml changes considerably:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            type: laboratory
```

As explained above, this hierarchy represents a single *host* (`core_ohst1`) running a single *process* (`core_process1`), running three *components* (`experiment1`, `laboratory1` and `core`). Since they are all in the same process, all the communication between the different servers will use the so-called `Direct` protocol (calling directly the function without using any network), regardless the configured protocols. Therefore, the generated structure is as follows:



Propagating configuration

During the example above, we've seen that it was possible to add configuration files such as:

```
config_file: lab1_config.py
```

or configuration variables directly such as:

```
config:
  core_facade_port: 10000
  core_facade_server_route: route1
```

There is also third approach which is:

```
config_files: [ lab1_config.py, lab2_config.py ]
```

or, alternatively:

```
config_files:
   lab1_config.py
   lab2_config.py
```

Furthermore, the mechanisms can be combined, so the following is valid:

```
config_file: general_config.py
config:
   port: 12345
```

However, each mechanism can not be repeated (so you can't have two `config` or two `config_file` for the *same level*). This is not a problem, though (in a single `config` you can put as many values as you want, and if you need more than one `config_file`, then you need a `config_files`).

These parameters can be put in any level of the hierarchy (global, *host*, *process* or *component*). When a component is running, it will have access to all those values which are accessible in its direct path to the root. In case of conflict (a variable defined in two levels), the one defined at a lowest level shadows the other for that component (e.g., if something is declared at a process level and at host level, the component under that process will obtain the value defined at process level).

So as to show this more clear, if we have this scenario:

```
(...)
config:
  var1: global
  var2: global
hosts:
  core_host:
    config:
      var2: host
      var3: host
    processes:
      core_process1:
        config:
          var3: process
          var4: process
        components:
          core:
            type: core
            config:
              var4: component
```

The `core` component will see that `var1` is "global", `var2` is "host", `var3` is "process" and `var4` is "component".

The full list of configuration variables are listed in *Configuration variables*.

Multiple core servers

Let's take a more complex example, involving more laboratories and more core servers. Here we assume that you have installed MySQL and the PyMySQL driver as suggested in *Installation: further steps*, and therefore we can run more than one core server. Running:

```
$ weblab-admin create sample3 --lab-copies=2 --dummy-copies=5 --cores=3 --db-
↪engine=mysql --coordination-db-engine=mysql
```

With this command, we are creating a new deployment where there will be 5 copies of an experiment, 3 core and login servers and 2 laboratory servers. The use of MySQL both for database backend and for coordination is required, since otherwise it will be using SQLite, which does not support concurrent access by multiple processes.

The generated hierarchy is the following:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
```

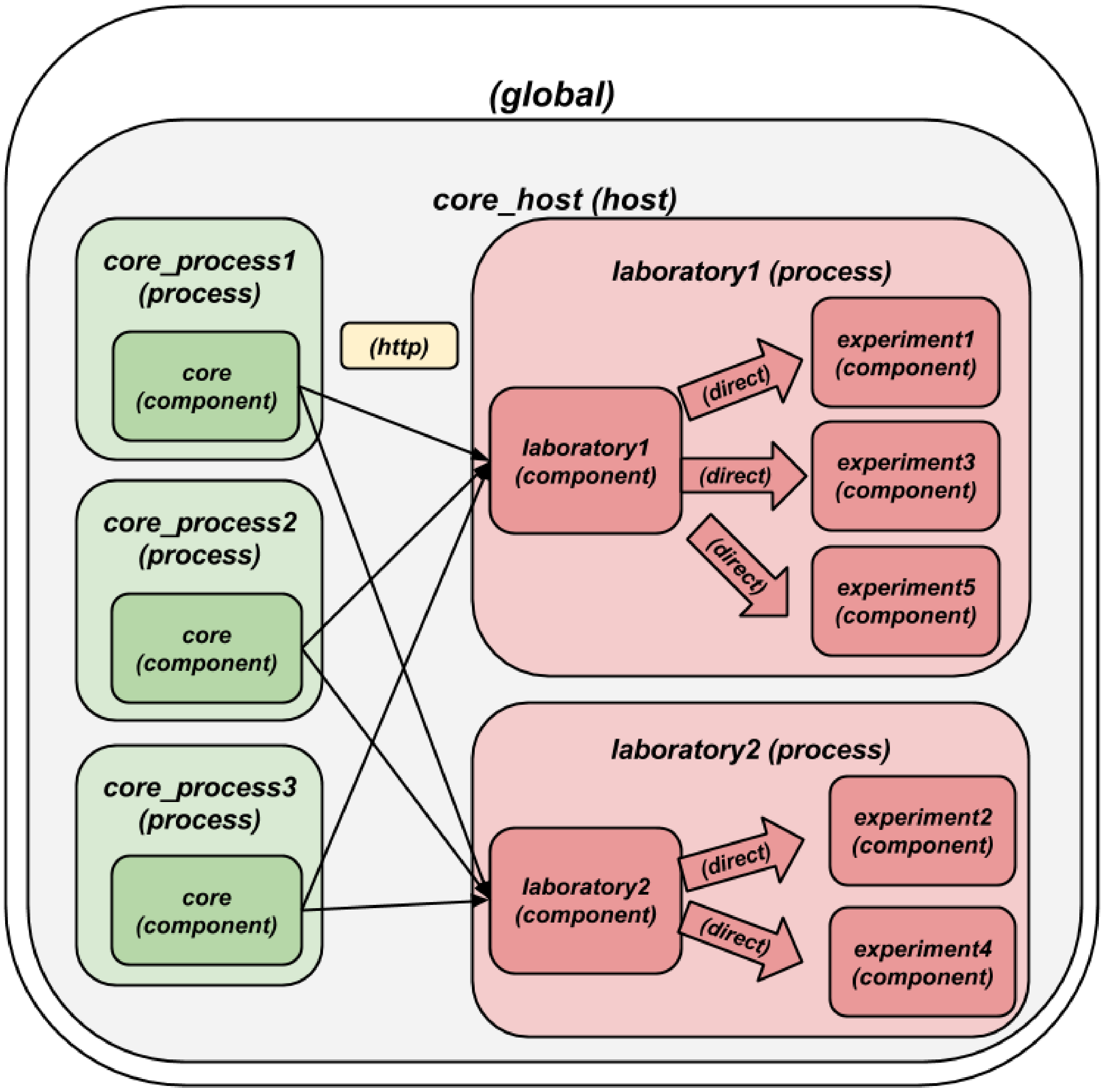
(continues on next page)

(continued from previous page)

```
              core_facade_server_route: route1
            type: core
      core_process2:
        components:
          core:
            config:
              core_facade_port: 10001
              core_facade_server_route: route2
            type: core
      core_process3:
        components:
          core:
            config:
              core_facade_port: 10002
              core_facade_server_route: route3
            type: core
      laboratory1:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          experiment3:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          experiment5:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10003
            type: laboratory
      laboratory2:
        components:
          experiment2:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          experiment4:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory2:
            config_file: lab2_config.py
            protocols:
              port: 10004
            type: laboratory
```

As requested, 3 Core servers have been created. Each pair has been created in a single *process*, so there are `core_process1`, `core_process2` and `core_process3`. Each of them will have a `core` component. On the other hand, it was requested to create 5 copies of an experiment (and therefore, 5 Experiment servers) and only 2 Laboratory servers. Since an Experiment server can only be associated to a single Laboratory server, the number of Experiment servers have been divided among the available Laboratory servers. The communication between each Laboratory server and each Experiment server will be `Direct`, since they will be in the same *process*. However, the communication among the Core servers and the Laboratory servers will use the most suitable network protocol, which by default it will be a HTTP format.

This configuration is represented with the following figure:

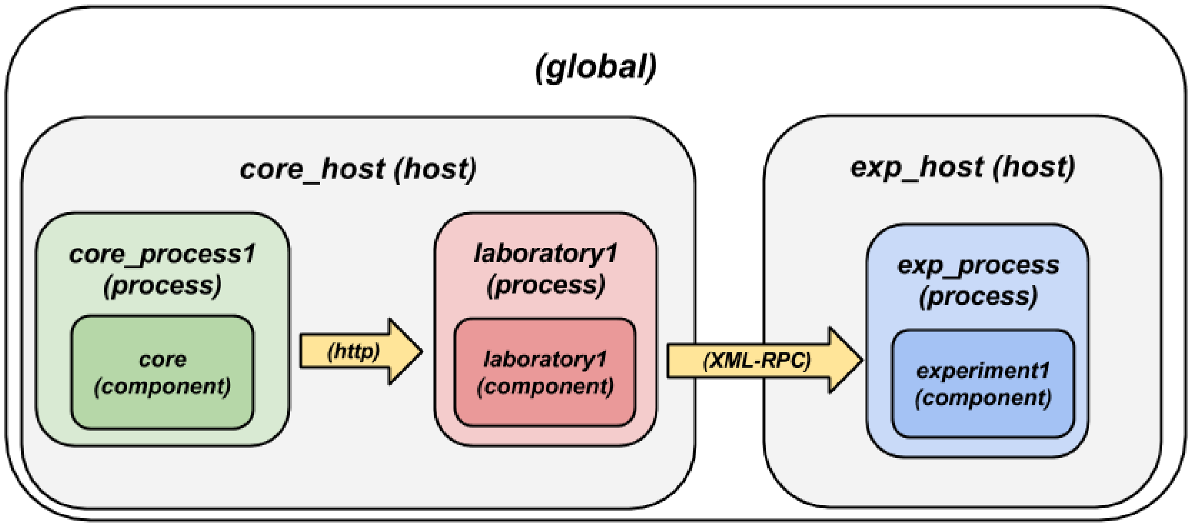


Multiple machines

So as to generate more than one machine with the `weblab-admin` script, run the following:

```
$ weblab-admin create sample4 --xmlrpc-experiment
```

This command is intended for deploying laboratories that use XML-RPC (such as those laboratories developed in programming languages other than Python). This command generates the deployment detailed in the following figure:



If we look at the `configuration.yml` file, we can appreciate the following:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
  exp_host:
    runner: run-xmlrpc.py
    host: 127.0.0.1
    processes:
      exp_process:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            protocols:
              port: 10002
              supports: xmlrpc
            type: experiment
```

There are two hosts: `exp_host` and `core_host`. The `core_host` contains the Laboratory server (in the `laboratory1` *process*) and the Core server (in the `core_process1` *process*). The `exp_host` has a single *process* which has a single *component* which is the `experiment1`. Since `experiment1` states that it only supports: xmlrpc, then the Laboratory Server will use XML-RPC to contact it.

2.5.4 Notes on addressing

In the addressing system used, one *component* called experiment1 at the *process* laboratory1 at the *host* core_machine will be refered as:

```
experiment1:laboratory1@core_host
```

For this reason, in some parts of the configuration files you will notice that the core server defines:

```
core_coordinator_laboratory_servers = {
    'laboratory1:core_process1@core_host' : {
        'exp1|dummy|Dummy experiments'         : 'dummy1@dummy',
    },
}
```

Where it defines "there is a Laboratory server which is identified by laboratory1 in the core_process1 process, which is in the core_host host. Similarly, you will see that the Laboratory is configured as:

```
laboratory_assigned_experiments = {
        'exp1:dummy@Dummy experiments' : {
                'coord_address' : 'experiment1:core_process1@core_host',
                'checkers' : ()
            },
    }
```

Here, the configuration establishes that a particular experiment (at database level) is located in a particular address. You will notice that this address is using the format explained.

2.5.5 Security

If you are going to deploy one of the servers in a different network and you want some basic security (e.g., the core server in a location and a laboratory server in a different location), you're encouraged to use one of the two following options (or both):

- Configure your firewall so only your core server can access (via IP)
- Use the auth parameter in the protocols section

For example, if you put the following in your configuration.yml:

```
exp_host:
 runner: run-xmlrpc.py
 host: 127.0.0.1
 processes:
   exp_process:
     components:
       experiment1:
         class: experiments.dummy.DummyExperiment
         protocols:
           port: 10002
           auth: RANDOM-SHARED-KEY
         type: experiment
```

Then the clients will always provide that shared key (RANDOM-SHARED-KEY in this case, put something random in your case) when contacting that server, and the server will require that shared key so as to process requests.

2.5.6 Summary

The focus of this section is showing the basics of the configuration subsystem of WebLab-Deusto. You may use the type of setting that suits better your system, even modifying it by yourself instead of using the `weblab-admin` script (or modifying the results of this script). With this section, you should be able to customize these aspects of the deployments.

2.6 Authentication

Table of Contents

WebLab-Deusto provides an extensible authentication mechanism. This way, all users are stored in the database, but different UserAuth mechanisms can be used for each user. The system will check for each user what mechanisms are available, and will check the credentials with each system. If any of the mechanisms say that the user is valid, the authentication mechanism will understand that it's a valid user.

For instance, if a password is provided by 'student1', who has two UserAuth, one providing a password hash stored in the database, and another one detailing a certain LDAP server that is valid for this user, then the system will check one system and then the other. If any of them says that it is correct, it does not check more systems. The order of these systems is detailed in the database, so it will first check local passwords and then it will check LDAP servers, for instance.

2.6.1 OAuth 2.0

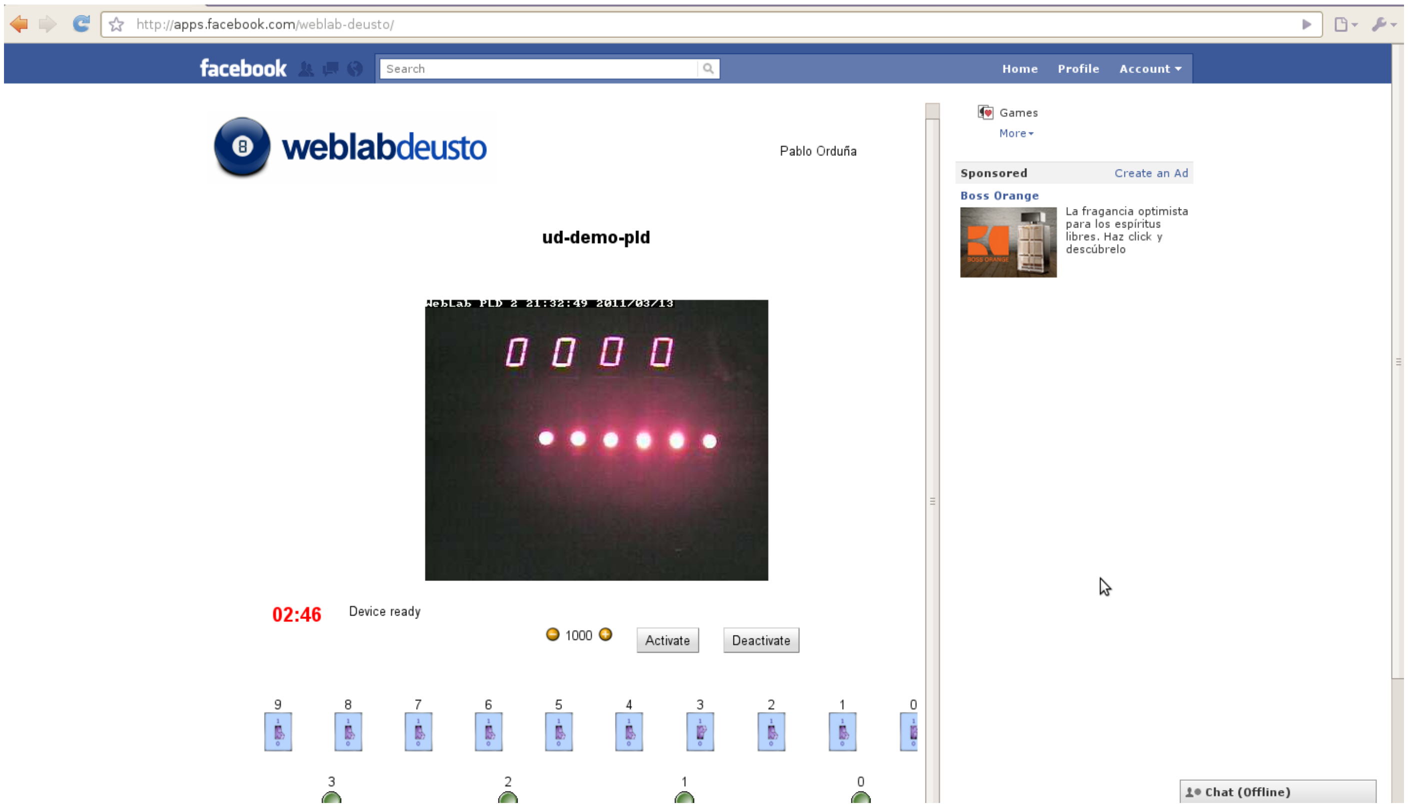
WebLab-Deusto can be easily integrated in Facebook through OAuth 2.0, as seen in http://apps.facebook.com/weblab-deusto/

Supporting other OAuth 2.0 systems for authentication should be simple, although some work would be required.

When first logging in the application, a website will offer two choices:

- Providing the WebLab-Deusto credentials, so the facebook account and the WebLab-Deusto accounts will be linked.
- Create a new WebLab-Deusto user, linked to this facebook account. When creating this user, the system will grant the same permissions the demo user has.

From this point, users will automatically see the experiments they have permissions to use, just as if they were logging in the WebLab-Deusto client, with the only difference that experiments are adapted to the Facebook constraints. For instance, Flash and Java experiments are resized to fit in the constraints imposed by Facebook (with a small fixed width).



Registering the application

In http://www.facebook.com/developers/ anyone can create applications for free. The application must use the iframe mode, pointing to the `/weblab/login/facebook/`.

The Login Server must be configured with the following Facebook parameters:

```
# The server where the WebLab-Deusto is deployed
# The link to the Facebook application, as you registered it.
login_facebook_url             = ""
login_facebook_client_address  = ""
login_facebook_auth_url        = "http://www.facebook.com/dialog/oauth?client_id=%s&
↪redirect_uri=%s&scope=email"
# The Facebook Application identifier, available in http://www.facebook.com/
↪developers/
login_facebook_app_id          = ""
login_facebook_canvas_url      = ""
```

Additionally, the Login Server must be configured to establish which permissions will have new users created through Facebook (if this is enabled) with the following configuration values in the Login Server configuration file:

```
login_not_linkable_users = ['demo']
login_default_groups_for_external_users = ['Demos']
```

(continues on next page)

(continued from previous page)

```
login_creating_external_users = True
login_linking_external_users = True
```

The `login_not_linkable_users` points to which users you don't want anyone to link. For instance, in the University of Deusto we use a user called "demo" with a public password ("demo") for demos. Since we don't want anyone to acquire this username, we set this property.

The `login_default_groups_for_external_users` property refers to the groups that will be used for new users through Facebook. Later administrators can select what permissions do these groups have.

Finally, the boolean properties `login_creating_external_users` and `login_linking_external_users` can be established if these features are not desired.

2.6.2 OpenID

OpenID is an open standard that enables the decentralized authentication. The authentication process, which consists on a user demonstrating the system that he really is who claims to be, can be handled by remote servers in a transparent way.

Use case

This way, if a university (University A) wants to share their experiments with other university (University B) in a very simple way, students of University B can be registered as OpenID users. As long as the consumer university (University B) counts with an OpenID server (e.g., in Spain the RedIRIS SIR provides http://yo.rediris.es/soy/username@universitydomain as OpenID for those universities enroled), they can provide a list such as:

```
student1, Student One, student1@universityb.edu, http://oid.universityb.edu/student1
student2, Student Two, student2@universityb.edu, http://oid.universityb.edu/student2
...
```

The provider university (University A) can then use the Database Manager to add these users as OpenID users. From this point, these students can log in:

```
/weblab/login/web/openid/verify?user_id=http://sso.universityb.edu/openid/student1
```

At this point, WebLab-Deusto will redirect student1 to the OpenID handler at universityb.edu. Student1 will provide the credentials to his university (University B), and University B will then redirect again to WebLab-Deusto, with certain tokens known by WebLab-Deusto, and therefore logging in automatically.

It's important to note that even if the authentication phase is performed in other university, these users are still local users for University A.

This type of deployment is especially useful when dealing with Learning Management Systems that use some kind of Single Sign-On system. If student1 enters in http://moodle.universityb.edu/, which first requires authentication redirecting http://sso.universityb.edu/, then when http://moodle.universityb.edu/ shows an iframe pointing to http://weblab.universitya.edu/weblab/login/web/openid/verify?user_id=http://sso.universityb.edu/openid/student1, the system will automatically show WebLab-Deusto logged in.

Settings

The following configuration values can be defined in the Login Server configuration (default values are provided):

```
# Only used when connecting to /weblab/login/web/openid/, shows a form where the %s␣
↪will be user ID
login_openid_domains = {
    'UNED'        : 'http://yo.rediris.es/soy/%s@uned.es',
    'UNED-INNOVA' : 'http://yo.rediris.es/soy/%s@innova.uned.es',
    'DEUSTO'      : 'http://yo.rediris.es/soy/%s@deusto.es'
}
login_openid_host         = 'https://www.weblab.deusto.es'
login_openid_client_url   = '/weblab/client/'
login_openid_base_openid = '/weblab/login/web/openid/'
```

2.6.3 Based on IP

Under certain and limited circumstances, administrators may want to be able to authenticate as a given local user without providing a password. For instance, University A could have two students of University B (called student1 and student2). University A could define "I will let http://moodle.universityb.edu/ to log in as student1 and student2 without asking for a password".

In order to do so, WebLab-Deusto supports the "TrustedIP" system. In order to do so, a new row is inserted in the Auth table, referencing to TRUSTED-IP-ADDRESSES in AuthType. The configuration defines the supported IP addresses, separated by commas if multiple are required (such as 127.0.0.1, 130.206.138.16). Then, new rows are required in UserAuth, one per each User, pointing to the new Auth. No configuration is required in the UserAuth. From this point, those users can be logged in through /weblab/login/web/login/?username=student1 without providing a password from the defined IP addresses.

2.6.4 LDAP

LDAP is an application protocol for reading and writing directories. Through these protocols it's possible to gather information of students from a LDAP infrastructure of the University, and it is possible to use LDAP to authenticate users.

WebLab-Deusto uses LDAP to register users and to check that the password provided by the user is the password used in the system. Therefore, for a certain amount of time, the university credentials are handled by WebLab-Deusto. It does not store it in any format, but if the WebLab-Deusto server is hacked, the credentials of those users using the system during that time are in danger. In the University of Deusto this is the system used with our students. However, if you still don't trust it and prefer other solutions, check other systems.

Note: How to use LDAP has not been yet documented.

2.6.5 Extending the system

The authentication system is based on plug-ins. It can be extended by implementing a proper plug-in in Python. This section covers how to implement one system.

WebLab-Deusto differentiates among two different types of authentication systems:

- *Simple:* those systems which receive the username and password, and check if the user is who claims to be. Examples of these systems are LDAP, password stored in the database, or checking that it comes from a particular IP address.

- *Web protocol systems:* those systems which do not receive simply a username and password, but which require an external web protocol. For example, using OAuth 2.0, the user will be forwarded to a particular page that must exist. Or in OpenID, the foreign system will redirect users to a particular page that also must exist.

So basically: if the system you are trying to design requires that WebLab-Deusto provides a new web service or anything to a third system, you should use the second approach. However, if you receive a certain username and password, you may use the first approach.

Simple

All the protocols implemented using the *Simple* approach are located in the weblab.login.simple package. On it, you will see different modules, one per each system. The most simple plug-in would be the following:

```python
from weblab.core.login.simple import SimpleAuthnUserAuth

class MyPluginUserAuth(SimpleAuthnUserAuth):

    NAME = 'MY-PLUGIN'

    def __init__(self, auth_configuration, user_auth_configuration):
        """ auth_configuration is how the particular system is configured in an
        instance. For instance, 30 students may use a LDAP repository, while other
        30 students are using other LDAP repository. Therefore a plug-in for LDAP
        is implemented, and later with the administration panel you may establish
        that the first 30 students use an instance of the LDAP plug-in, and other
        30 students other instance. The details of the repository would come in the
        auth_configuration (common for many users).

        However, in the case of the hashed passwords in the database, the
        auth_configuration is empty, and user_auth_configuration contains the
        particular hashed pasword.

        Both arguments are strings.
        """
        pass

    def authenticate(self, login, password):
        # Do something with auth_configuration, user_auth_configuration and
        # then return True or False if the login and password match proper
        # credentials.
        return True
```

Once this class is created and is located in the proper module, the last lines of the weblab/login/simple/__init__.py to register the plug-in. In this example:

```python
from weblab.core.login.simple.db_auth import WebLabDbUserAuth
from weblab.core.login.simple.ldap_auth import LdapUserAuth
from weblab.core.login.simple.ip_auth import TrustedIpAddressesUserAuth
# Just added
from weblab.core.login.simple.my_plugin import MyPluginUserAuth

SIMPLE_PLUGINS = {
    WebLabDbUserAuth.NAME              : WebLabDbUserAuth,
    LdapUserAuth.NAME                  : LdapUserAuth,
    TrustedIpAddressesUserAuth.NAME : TrustedIpAddressesUserAuth,
    # Put your plug-in here.
```

(continues on next page)

(continued from previous page)

```
    MyPluginUserAuth.NAME            : MyPluginUserAuth
}
```

From this point, those user with this authentication mechanism would be validated by it.

Web protocol systems

So as to support those systems using a login subsystem that requires an external protocol, a slightly more complicated process is required. You may find examples in the weblab.login.web package. As you will notice, two classes are required, so the most simple system that you can implement is the following:

```
from weblab.core.login.web import ExternalSystemManager, weblab_api
import weblab.core.server as core_api

from weblab.data.dto.users import User
from weblab.data.dto.users import StudentRole

class MyManager(ExternalSystemManager):

    NAME = 'MYPLUGIN'

    @logged(log.level.Warning)
    def get_user(self, credentials):
        """Use credentials to validate in the remote system."""

        # credentials might be a token to retrieve information
        # such as the full name, the email or the login.

        login     = "user2132@myplugin"
        full_name = "John Doe"
        email     = "john.doe@deusto.es"

        user = User(login, full_name, email, StudentRole())
        return user

    def get_user_id(self, credentials):
        login = self.get_user(credentials).login
        # login is "13122142321@myplugin"
        return login.split('@')[0]

@weblab_api.route_login_web('/my/')
def my_web():
    """ This is a complete Flask-compliant system, although
    some methods are inherited from WebPlugin that make it
    easier to work with. """

    # Here you can contact other URLs or provide multiple
    # different methods.

    # Once you have something to check credentials with
    # such as tokens or whatever, you may call the following
    # method:
    session_id = core_api.extensible_login(MyManager.NAME, whatever_token)

    # And you may pass it however you want to the final user:
```

(continues on next page)

(continued from previous page)

```
    return ("<html><body><b>This HTML content will be "
        "displayed %s</b></html>" % session_id.id)
```

Once you write the WSGI-compliant web application, you can register it in the last lines of the weblab/core/login/web/__init__.py as follows:

```
from weblab.core.login.web.login      import LoginPlugin
from weblab.core.login.web.facebook   import FacebookManager
from weblab.core.login.web.openid_web import OpenIDManager
from weblab.core.login.web.myplugin   import MyManager

EXTERNAL_MANAGERS = {
    FacebookManager.NAME : FacebookManager(),
    OpenIdManager.NAME   : OpenIDManager(),
    # Your plug-in here
    MyManager.NAME       : MyManager(),
}
```

2.7 Tools

Table of Contents

2.7.1 WebLab Admin

WebLab-Deusto provides a command called `weblab-admin` for interacting with installations of WebLab-Deusto. You'll find the latest documentation by running:

```
$ weblab-admin --help
$ weblab-admin <command> --help
```

The following is the output of these commands as of June 2016.

Instance creation

Running `weblab-admin create --help` returns:

```
Usage: weblab-admin create DIR [options]

Options:
  -h, --help            show this help message and exit
  -f, --force           Overwrite the contents even if the directory already
                        existed.
  -q, --quiet           Do not display any output.
  -v, --verbose         Show more information about the process.
  --not-interactive     Run the script in not interactive mode. Recommended
                        for scripts only.
  --socket-wait=PORT    Wait for a socket connection rather than sigterm/input
  --add-test-data       Populate the database with sample data
  --cores=CORES         Number of core servers.
  --start-port=START_PORTS
                        From which port start counting.
  -i SYSTEM_IDENTIFIER, --system-identifier=SYSTEM_IDENTIFIER
                        A human readable identifier for this system.
  --enable-https        Tell external federated servers that they must use
                        https when connecting here
  --base-url=BASE_URL   Base location, before /weblab/. Example: /deusto.
  --http-server-port=HTTP_SERVER_PORT
                        Enable the builtin HTTP server (so as to not require
                        apache while testing) and listen in that port.
  --entity-link=ENTITY_LINK
                        Link of the host entity (e.g. http://www.deusto.es ).
  --logo-path=IMG_FILE_PATH
                        Path of the entity logo.
  --server-host=SERVER_HOST
                        Host address of this machine. Example: weblab.domain.
  --poll-time=POLL_TIME
                        Time in seconds that will wait before expiring a user
                        session.
  --no-lab              Do not create any laboratory server or experiment
                        server.
  --inline-lab-server   Laboratory server included in the same process as the
                        core server. Only available if a single core is used.
  --lab-copies=LAB_COPIES
                        Each experiment can be managed by a single laboratory
                        server. However, if the number of experiments managed
                        by a single laboratory server is high, it can become a
                        bottleneck. This bottleneck effect can be reduced by
                        balancing the amount of experiments among different
                        copies of the laboratories. By establishing a higher
                        number of laboratories, the generated deployment will
                        have the experiments balanced among them.
  --ignore-locations    Ignore locations. Otherwise, it will tell you to
                        download two files for GeoLocation

  Administrator data:
    Administrator basic data: username, password, etc.

    --admin-user=ADMIN_USER
                        Username for the WebLab-Deusto administrator
    --admin-name=ADMIN_NAME
```

(continues on next page)

(continued from previous page)

```
                        Full name of the administrator
    --admin-password=ADMIN_PASSWORD
                        Administrator password ('password' is the default)
    --admin-mail=ADMIN_MAIL
                        E-mail address of the system administrator.

  Experiments options:
    While most laboratories are specific to a particular equipment, some
    of them are useful anywhere (such as the VM experiment, as long as you
    have a VirtualBox virtual machine that you'd like to deploy, or the
    logic game, which does not require any equipment). Other experiments,
    such as VISIR, have been deployed in many universities. Finally, for
    development purposes, the XML-RPC experiment is particularly useful.

    --xmlrpc-experiment
                        By default, the Experiment Server is located in the
                        same process as the  Laboratory server. However, it is
                        possible to force that the laboratory  uses XML-RPC to
                        contact the Experiment Server. If you want to test a
                        Java, C++, .NET, etc. Experiment Server, you can
                        enable this option, and the system will try to find
                        the Experiment Server in other port
    --xmlrpc-experiment-port=XMLRPC_EXPERIMENT_PORT
                        What port should the Experiment Server use. Useful for
                        development.
    --dummy-experiment-name=DUMMY_NAME
                        There is a testing experiment called 'dummy'. You may
                        change this name (e.g. to dummy1 or whatever) by
                        changing this option.
    --dummy-category-name=DUMMY_CATEGORY_NAME
                        You can change the category name of the dummy
                        experiments. (by default, Dummy experiments).
    --dummy-copies=DUMMY_COPIES
                        You may want to test the load balance among different
                        copies of dummy.
    --dummy-silent      Not show the commands sent to the dummy experiment.
    --visir, --visir-server
                        Add a VISIR server to the deployed system.
    --visir-slots=SLOTS
                        Number of concurrent users of VISIR.
    --visir-experiment-name=EXPERIMENT_NAME
                        Name of the VISIR experiment.
    --visir-base-url=VISIR_BASE_URL
                        URL of the VISIR system (e.g. http://weblab-
                        visir.deusto.es/electronics/ ). It should contain
                        login.php, for instance.
    --visir-measurement-server=MEASUREMENT_SERVER
                        Measurement server. E.g. weblab-visir.deusto.es:8080
    --visir-use-php     VISIR can manage the authentication through a PHP
                        code. This option is slower, but required if that
                        scheme is used.
    --visir-login=USERNAME
                        If the PHP version is used, define which username
                        should be used. Default: guest.
    --visir-password=PASSWORD
                        If the PHP version is used, define which password
                        should be used. Default: guest.
```

(continues on next page)

(continued from previous page)

```
  --logic, --logic-server
                        Add a logic server to the deployed system.
  --vm, --virtual-machine, --vm-server
                        Add a VM server to the deployed system.
  --vm-experiment-name=EXPERIMENT_NAME
                        Name of the VM experiment.
  --vm-storage-dir=STORAGE_DIR
                        Directory where the VirtualBox machines are located.
                        For example: c:\users\lrg\.VirtualBox\Machines
  --vbox-vm-name=VBOX_VM_NAME
                        Name of the Virtual Box machine which this experiment
                        uses. Is often different from the Hard Disk name.
  --vbox-base-snapshot=VBOX_BASE_SNAPSHOT
                        Name of the VirtualBox snapshot to which the system
                        will be reset after every usage. It should be an
                        snapshot of an started machine. Otherwise, it will
                        take too long to start.
  --vm-url=URL          URL which will be provided to users so that they can
                        access the VM through VNC. For instance:
                        vnc://192.168.51.82:5901
  --http-query-user-manager-url=URL
                        URL through which the user manager (which runs on the
                        VM and resets it when needed) can be reached. For
                        instance: http://192.168.51.82:18080
  --vm-estimated-load-time=LOAD_TIME
                        Estimated time which is required for restarting the
                        VM. Does not need to be accurate. It is displayed to
                        the user and is essentially for cosmetic purposes.

Federation options:
  WebLab-Deusto at the University of Deusto federates a set of
  laboratories. You may put them by default in your WebLab-Deusto
  instance.

  --add-fed-submarine
                        Add the submarine laboratory.
  --add-fed-logic       Add the logic laboratory.
  --add-fed-visir       Add the VISIR laboratory.

Session options:
  WebLab-Deusto may store sessions in a database, in memory or in
  redis.Choose one system and configure it.

  --session-storage=SESSION_STORAGE
                        Session storage used. Values: sql, redis, memory.
  --session-db-engine=SESSION_DB_ENGINE
                        Select the engine of the sessions database.
  --session-db-host=SESSION_DB_HOST
                        Select the host of the session server, if any.
  --session-db-port=SESSION_DB_PORT
                        Select the port of the session server, if any.
  --session-db-name=SESSION_DB_NAME
                        Select the name of the sessions database.
  --session-db-user=SESSION_DB_USER
                        Select the username to access the sessions database.
  --session-db-passwd=SESSION_DB_PASSWD
                        Select the password to access the sessions database.
```

(continues on next page)

(continued from previous page)

```
  --session-redis-db=SESSION_REDIS_DB
                        Select the redis db on which store the sessions.
  --session-redis-host=SESSION_REDIS_HOST
                        Select the redis server host on which store the
                        sessions.
  --session-redis-port=SESSION_REDIS_PORT
                        Select the redis server port on which store the
                        sessions.

Database options:
  WebLab-Deusto uses a relational database for storing users,
  permissions, etc.The database must be configured: which engine,
  database name, user and password.

  --db-engine=DB_ENGINE
                        Core database engine to use. Values: mysql, sqlite.
  --db-name=DB_NAME   Core database name.
  --db-host=DB_HOST   Core database host.
  --db-port=DB_PORT   Core database port.
  --db-user=DB_USER   Core database username.
  --db-passwd=DB_PASSWD
                        Core database password.

Scheduling options:
  These options are related to the scheduling system.  You must select
  if you want to use a database or redis, and configure it.

  --coordination-engine=COORD_ENGINE
                        Coordination engine used. Values: sql, redis.
  --coordination-db-engine=COORD_DB_ENGINE
                        Coordination database engine used, if the coordination
                        is based on a database. Values: mysql, sqlite.
  --coordination-db-name=COORD_DB_NAME
                        Coordination database name used, if the coordination
                        is based on a database.
  --coordination-db-user=COORD_DB_USER
                        Coordination database userused, if the coordination is
                        based on a database.
  --coordination-db-passwd=COORD_DB_PASSWD
                        Coordination database password used, if the
                        coordination is based on a database.
  --coordination-db-host=COORD_DB_HOST
                        Coordination database host used, if the coordination
                        is based on a database.
  --coordination-db-port=COORD_DB_PORT
                        Coordination database port used, if the coordination
                        is based on a database.
  --coordination-redis-db=COORD_REDIS_DB
                        Coordination redis DB used, if the coordination is
                        based on redis.
  --coordination-redis-passwd=COORD_REDIS_PASSWD
                        Coordination redis password used, if the coordination
                        is based on redis.
  --coordination-redis-host=COORD_REDIS_HOST
                        Coordination redis host used, if the coordination is
                        based on redis.
  --coordination-redis-port=COORD_REDIS_PORT
```

(continues on next page)

(continued from previous page)

```
                        Coordination redis port used, if the coordination is
                        based on redis.
```

Starting an instance

Running `weblab-admin start --help` returns:

```
Usage: weblab-admin start DIR [options]

Options:
  -h, --help            show this help message and exit
  -m HOST, --host=HOST, --machine=HOST
                        If there is more than one host in the configuration,
                        which one should be started.
  -l, --list-hosts, --list-machines
                        List hosts.
  -s SCRIPT, --script=SCRIPT
                        If the runner option is not available, which script
                        should be used.
```

Stopping an instance

The command `weblab-admin stop <instance_directory>` does not have any option. It stops all the processes of the instance.

Upgrading an instance

The command `weblab-admin upgrade <instance_directory> --help` returns:

```
usage: weblab-admin [-h] [-y]

WebLab upgrade tool.

optional arguments:
  -h, --help  show this help message and exit
  -y, --yes   Say yes to everything
```

Upgrading locations of an instance

The command `weblab-admin locations <instance_directory> --help` returns:

```
usage: weblab-admin locations DIR [options]

optional arguments:
  -h, --help        show this help message and exit
  --redownload      Force redownload of databases
  --reset-database  Reset the database, forcing the server to download all the
                    data again
  --reset-cache     Reset the database, forcing the server to download all the
                    data again
```

Upgrading the web server configurations of an instance

The command `weblab-admin httpd-config-generate <instance_directory` does not have any option. It just re-generates the web server configuration.

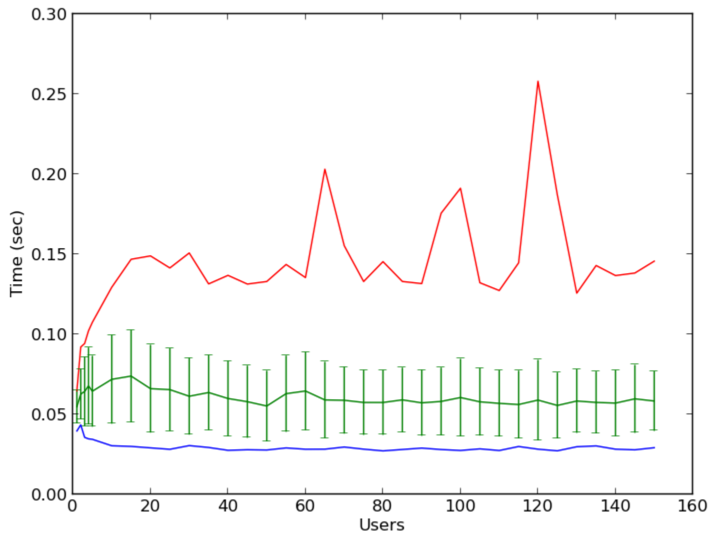
2.7.2 WebLab Bot

A Remote Laboratory is a software system that requires a complex workflow and that will require to face big load of users in certain moments. There are different constraints that have an impact on the latency and performance of WebLab-Deusto:

- Deployment configuration: only one server, multiple servers, storing sessions in database or in memory…
- Deployed system: what machine, operating system, Python or MySQL versions…
- Tens or hundreds of students being queued
- Tens or hundreds of students using experiments

In order to test these variables easily, a students simulator has been implemented, and it is called WebLab Bot. The WebLab Bot tool is used for three purposes:

- Measure the time with each configuration
- Perform stress tests of the system, finding the errors created when developing new features
- Test the system in new operating systems or software versions



So as to run it, you need a configuration file, such as the one available in tools/Bot/configuration.py.dist. Copy it to `configuration.py` and change the required variables (e.g., change the credentials, URLs, etc.). The `consumer/run.py` referes to the `run.py` file generated whenever you created an environment, such as:

```
$ weblab-admin create consumer
```

The number of iterations define how many times the same scenario will be repeated. The number of concurrent users is defined in the generate_scenarios method, in the different two `for` loops. You may add other loops or change these, but the idea is that in this example, it will be tested with 1 student, 2, 3, 4, 5, 10, 15, 20, 25 …, 140, 145 and 150:

```
for protocol in cfg_util.get_supported_protocols():
    for number in range(1, 5):
        scenarios.append(
                Scenario(
                    cfg_util.new_bot_users(number, new_standard_bot_user, 0, STEP_
↪DELAY, protocol),
                    protocol, number
                )
            )

    for number in range(5, 151, 5):
        scenarios.append(
                Scenario(
                    cfg_util.new_bot_users(number, new_standard_bot_user, STEP_DELAY␣
↪* (5 -1), STEP_DELAY, protocol),
                    protocol, number
                )
            )
```

Additionally, you need to install matplotlib:

```
# (in Ubuntu, the following requires some packages, such as build-essential,␣
↪libfreetype6-dev or libpng-dev)
pip install matplotlib
```

Then, simply call:

```
weblab-bot.py
```

This will start the WebLab-Deusto instance, run the proposed scenario, and then stop it, for each iteration and scenario defined. Running it will generate the following output:

```
********************
CONFIGURATION consumer/run.py
Unique id: D_2013_03_31_T_11_38_17_
********************

New trial. 1 iterations
 iteration 0 .  {'route1': 1} [ 0 exceptions ]
Cleaning results... Sun Mar 31 11:38:28 2013
Storing results... Sun Mar 31 11:38:28 2013
Results stored Sun Mar 31 11:38:28 2013
   -> Scenario: <Scenario category="JSON" identifier="1" />
   -> Results stored in logs/botclient_D_2013_03_31_T_11_38_17__SCEN_0_CONFIG_0.pickle
   -> Serializing results...
   -> Done

[...]

New trial. 1 iterations
 iteration 0 ....  {'route1': 4} [ 0 exceptions ]
Cleaning results... Sun Mar 31 11:39:19 2013
Storing results... Sun Mar 31 11:39:19 2013
Results stored Sun Mar 31 11:39:19 2013
   -> Scenario: <Scenario category="JSON" identifier="4" />
   -> Results stored in logs/botclient_D_2013_03_31_T_11_38_17__SCEN_3_CONFIG_0.pickle
   -> Serializing results...
```

(continues on next page)

(continued from previous page)

```
   -> Done
Writing results to file raw_information_D_2013_03_31_T_11_38_17_.dump... 2013-03-31␣
↪11:39:19.866922
Generating graphics...
Executing figures/generate_figures_D_2013_03_31_T_11_38_17_.py... [done]
HTML file available in botclient_D_2013_03_31_T_11_38_17_.html
Finished plotting; Sun Mar 31 11:39:31 2013, 251 millis
Done 2013-03-31 11:39:31.251789
```

The HTML file that it points out contains all the graphics for each method.

If you don't want to start the process each time (e.g., you want to test it with an existing WebLab-Deusto instance that you don't want to stop), then, pass the following argument:

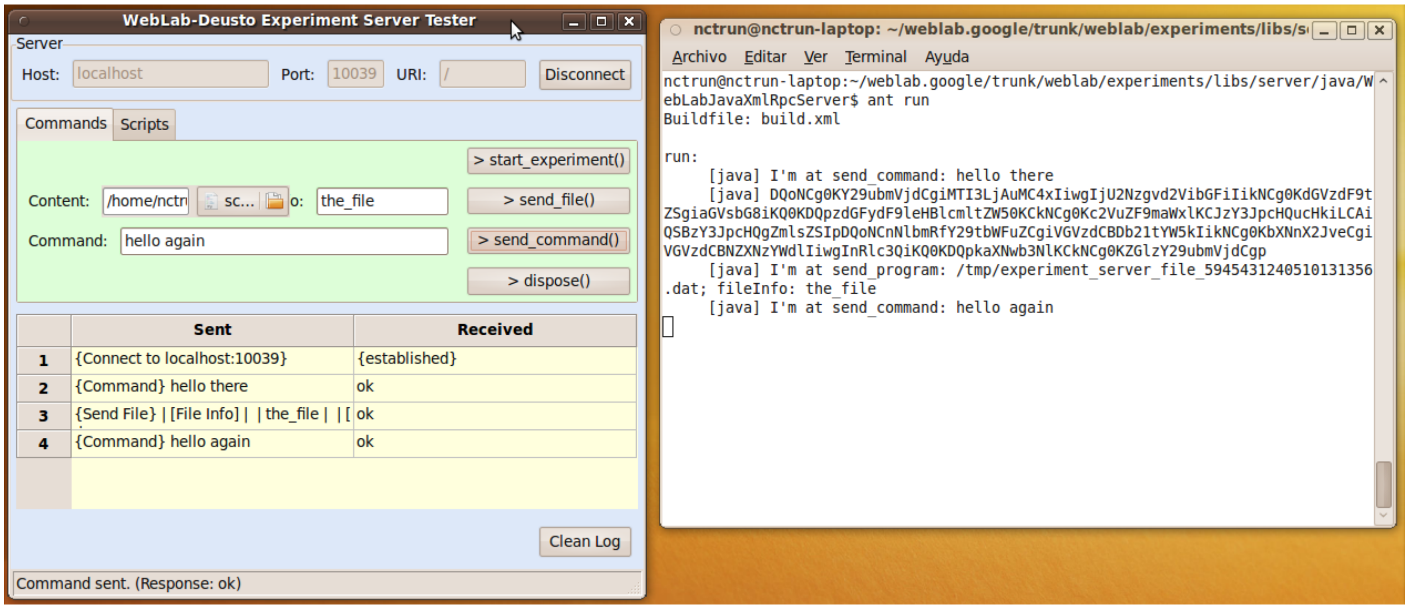
```
weblab-bot.py --dont-start-processes
```

As in the case of `weblab-admin`, in UNIX systems you may also use `weblab-bot` (instead of `weblab-bot.py`).

2.7.3 Experiment Server Tester

Warning: THIS TOOL NEEDS TO BE UPGRADED TO SUPPORT THE NEW APIs

In order to make it easy to test the experiment server under development, WebLab-Deusto provides a tool called ExperimentServerTester (available in tools/ExperimentServerTester). This is a Python application (requires both Python 2.6 and wxPython, both available for GNU/Linux, Microsoft Windows and Mac OS X) that makes it easy to interact with the server as WebLab-Deusto would do it. You can use the provided assistant (pressing on "Send command" will send the command you have written):

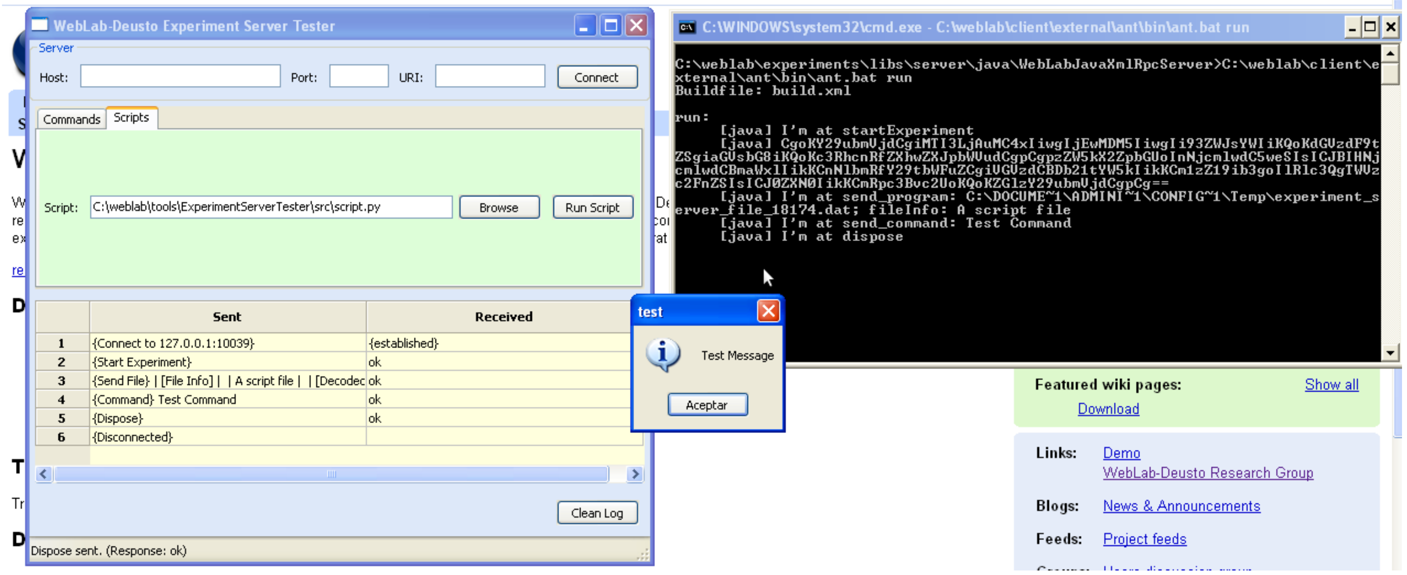


	Sent	Received
1	{Connect to localhost:10039}	{established}
2	{Command} hello there	ok
3	{Send File} | [File Info] | | the_file | | [	ok
4	{Command} hello again	ok

```
nctrun@nctrun-laptop:~/weblab.google/trunk/weblab/experiments/libs/server/java/W
ebLabJavaXmlRpcServer$ ant run
Buildfile: build.xml

run:
     [java] I'm at send_command: hello there
     [java] DQoNCg0KY29ubmVjdCgiMTI3LjAuMC4xIiwgIjU2Nzgvd2VibGFiIikNCg0KdGVzdF9t
ZSgiaGVsbG8iKQ0KDQpzdGFydF9leHBlcmltZW50KCkNCg0Kc2VuZF9maWxlKCJzY3JpcHQucHkiLCAi
QSBzY3JpcHQgZmlsZSIpDQoNCnNlbmRfY29tbWFuZCgiVGVzdCBDb21tYW5kIikNCg0KbXNnX2JveCgi
VGVzdCBNZXNzYWdlIiwgInRlc3QiKQ0KDQpkaXNwb3NlKCkNCg0KZGlzY29ubmVjdCgp
     [java] I'm at send_program: /tmp/experiment_server_file_5945431240510131356
.dat; fileInfo: the_file
     [java] I'm at send_command: hello again
```

Or you can make a script. This could be a full example of the provided API (in addition to all the Python API):

```
connect("127.0.0.1", "10039", "/weblab")
test_me("hello")

start_experiment()
send_file("script.py", "A script file")
send_command("Test Command")
msg_box("Test Message", "test")
dispose()

disconnect()
```



	Sent	Received
1	{Connect to 127.0.0.1:10039}	{established}
2	{Start Experiment}	ok
3	{Send File} | [File Info] | | A script file | | [Decodec	ok
4	{Command} Test Command	ok
5	{Dispose}	ok
6	{Disconnected}	

Dispose sent. (Response: ok)

```
C:\WINDOWS\system32\cmd.exe - C:\weblab\client\external\ant\bin\ant.bat run

C:\weblab\experiments\libs\server\java\WebLabJavaXmlRpcServer>C:\weblab\client\e
xternal\ant\bin\ant.bat run
Buildfile: build.xml

run:
     [java] I'm at startExperiment
     [java] CgoKY29ubmVjdCgiMTI3LjAuMC4xIiwgIjEwMDM5IiwgIi93ZWJsYWIiKQoKdGVzdF9t
ZSgiaGVsbG8iKQoKc3RhcnRfZXhwZXJpbWVudCgpCgpzZW5kX2ZpbGUoInNjcmlwdC5weSIsICJBIHNj
cmlwdCBmaWxlIikKCnNlbmRfY29tbWFuZCgiVGVzdCBDb21tYW5kIikKCm1zZ19ib3goIlRlc3QgTWVz
c2FnZSIsICJ0ZXN0IikKCmRpc3Bvc2UoKQoKZGlzY29ubmVjdCgpCg==
     [java] I'm at send_program: C:\DOCUME~1\ADMINI~1\CONFIG~1\Temp\experiment_s
erver_file_18174.dat; fileInfo: A script file
     [java] I'm at send_command: Test Command
     [java] I'm at dispose
```

While this tool is still in an experimental status, it can already help the development of experiments.

2.7.4 VISIR Battle Tester

The VISIR Battle Tester (available in tools/VisirBattleTester is an automated tool to evaluate the performance of WebLab-Deusto with VISIR. It simulates multiple concurrent students interacting with a VISIR in a WebLab-Deusto system, testing different measurements and validating that the results are the expected, in certain range.

For example, it may send a command which is a request that it knows that it should return 900, and checks that there is up to a 20% of error margin:

```
before = time.time()
response = weblab.send_command(reservation_id, Command(visir_commands.visir_request_
↪900 % visir_sessionid))
after = time.time()
result = visir_commands.parse_command_response(response)
ar3 = AssertionResult(900.0, 900.0 * 0.2, result)
if DEBUG and ar3.failed:
    print "[Failed at 3rd]" + str(ar3)
if not IGNORE_ASSERTIONS:
    assertions.append(ar3)
times.append(after - before)
```

So as to run it, change the credentials and URL in the `run.py` file and run it.

2.8 Configuration variables

This section covers the available configuration variables, organized by servers. Take a look at *Technical description* to identify the different servers described here.

Note: At the time of this writing, not all the variables have been documented. We're working on this (June 2016). Take into account that these variablse are the type of variables you'll find in the .py configuration files. They are not variables for commands.

Table of Contents

2.8.1 Laboratory Server

The laboratory server is closer to the experiment server and checks if it is alive, maintains the sessions and acts as a bridge between the pool of core servers and the experiments.

General

Property	*Type*	*Default value*	*Description*
laboratory_session_type	basestring	Memory	What type of session manager the Core Server will use: Memory or MySQL.
laboratory_session_pool_id	basestring	LaboratoryServer	See “core_session_pool_id” in the core server.
laboratory_assigned_experiments	list		List of strings representing which experiments are available through this particular laboratory server. Each string contains something like ‘exp1|ud-fpga|FPGA experiments;fpga:inst@mach’, where exp1|ud-fpga|FPGA experiments is the identifier of the experiment, and “fpga:inst@mach” is the WebLab Address of the experiment server.
laboratory_exclude_checking	list	[]	List of ids of experiments upon which checks will not be run

2.8.2 Experiments

This section includes the configuration of existing laboratories.

HTTP

The HTTP experiment is a type of unmanaged laboratory which enables you to develop your own laboratory. WebLab-Deusto will call certain methods in that laboratory, and your laboratory will act taking that into account.

Property	*Type*	*Default value*	*Description*
http_experiment_url	basestring		The base URL for the experiment server. Example: ‘http://address/mylab/’ will perform requests to ‘http://address/mylab/weblab/
http_experiment_username	basestring	None	The username used for performing that request. If not present, it will not use any credentials (and it will assume that the server is filtering the address by IP address or so).
http_experiment_password	basestring	None	The password used for performing that request. If not present, it will not use any credentials.
http_experiment_batch	bool	False	Does the system manage its own scheduling mechanism? If so, users requesting access will automatically be forwarded, and it is the experiment server the one who has to manage what to do with them.
http_experiment_api	basestring	None	The API is calculated automatically. However, you may force a particular API (such as 0, which is the oldest one).
http_experiment_extension	basestring	None	Is it using the standard routing system provided? Or is it using something like ‘.php’ in all the files?
http_experiment_request_format	basestring	json	What format should be used for performing the request? JSON directly? Or standard http form?

2.8.3 Common configuration

These variables affect all the servers. For instance, certain servers use a session manager (e.g. the Core server for users, but also the Laboratory server).

General

These variables are simple variables which are general to the whole project.

Property	*Type*	*Default value*	*Description*
debug_mode	bool	False	If True, errors and exceptions are shown instead of generic feedback (like WebLabInternalServerError)
server_admin	basestring	None	WebLab-Deusto administrator's email address for notifications. See Admin Notifier settings below.
server_hostaddress	basestring		Host address of this WebLab-Deusto deployment
propagate_stack_traces_to_client	bool	False	If True, stacktraces are propagated to the client (useful for debugging).
facade_timeout	float	0.5	Seconds that the facade will wait accepting a connection before checking again for shutdown requests.

Admin Notifier

The Admin notifier is mainly used by the core server for notifying administrators of certain activity such as broken laboratories.

Property	*Type*	*Default value*	*Description*
mail_notification_enabled	bool		Enables or Disables mail notifications
mail_server_host	basestring		Host to use for sending mail
mail_server_helo	basestring		Address to be used on the mail's HELO
mail_server_use_tls	basestring	no	Use TLS or not. Values: 'yes' or 'no'
mail_notification_sender	basestring		Address of the mail's sender
mail_notification_subject	basestring	[WebLab] CRITICAL ERROR!	(Optional) Subject of the notification mail

Sessions

The session configuration is mainly used by the Core Server, but also by the Laboratory Server and by certain Experiment Servers.

Property	*Type*	*Default value*	*Description*
session_sqlalchemy_engine	basestring	mysql	Database engine used for sessions the database. Example: mysql
session_sqlalchemy_host	basestring	localhost	Location of the sessions database server
session_sqlalchemy_port	int	None	Location of the sessions database server
session_sqlalchemy_db_name	basestring	WebLabSessions	Database name of the sessions database
session_sqlalchemy_username	basestring		Username for connecting to the sessions database
session_sqlalchemy_password	basestring		Password for connecting to the sessions database
session_lock_sqlalchemy_engine	basestring	mysql	Database engine used for locking the database. Example: mysql
session_lock_sqlalchemy_host	basestring	localhost	Location of the locking database server
session_lock_sqlalchemy_port	int	None	Location of the locking database server
session_lock_sqlalchemy_db_name	basestring	WebLabSessions	Database name of the locking database
session_lock_sqlalchemy_username	basestring		Username for connecting to the locking database
session_lock_sqlalchemy_password	basestring		Password for connecting to the locking database
session_manager_default_timeout	int	7200	Maximum time that a session will be stored in a Session Manager. In seconds.
session_memory_gateway_serialize	bool	False	Sessions can be stored in a database or in memory. If they are stored in memory, they can be serialized in memory or not, to check the behaviour

2.8.4 Core Server

This configuration is used only by the Core servers. The Core server manages the scheduling, life cycle of the users, the sessions, and the incoming web services calls. Note that there is other common configuration which affects the Core server, so also take a look at the Common Configuration in this document.

General

General variables for the Core server: what type of session, should we store students programs, etc.

Property	*Type*	*Default value*	*Description*
core_server_url	basestring		The base URL for this server. For instance, http://your-uni.edu/weblab/
core_universal_identifier	basestring	00000000	Unique global ID for this WebLab-Deusto deployment. Used in federated environments, where multiple nodes register each other and do not want to enter in a loop. You should generate one (search for online GUID or UUID generators or use the uuid module in Python).
core_universal_identifier_human	basestring	WARNING; MISCONFIGURED SERVER. ADD A UNIVERSAL IDENTIFIER	Message such as 'University A', which identifies which system is using performing the reservation. The unique identifier above must be unique, but this one only helps debugging.
core_session_type	basestring	Memory	What type of session manager the Core Server will use: Memory or MySQL.
core_session_pool_id	basestring	UserProcessingServer	A unique identifier of the type of sessions, in order to manage them. For instance, if there are four servers (A, B, C and D), the load of users can be splitted in two groups: those being sent to A and B, and those being sent to C and D. A and B can share those sessions to provide fault tolerance (if A falls down, B can keep working from the same point A was) using a MySQL session manager, and the same may apply to C and D. The problem is that if A and B want to delete all the sessions -at the beginning, for example-, but they don't want to delete sessions of C and D, then they need a unique identifier shared for A and B, and another for C and D. In this case, "UserProcessing_A_B" and "UserProcessing_C_D" would be enough.
core_store_students_programs	bool	False	Whether files submitted by users should be stored or not.
core_store_students_programs_path	basestring	None	If files are stored, in which local directory should be stored.
geoip2_city_filepath	basestring	GeoLite2-City.mmdb	If the maxminds city database is downloaded, use it
geoip2_country_filepath	basestring	GeoLite2-Country.mmdb	If the maxminds country database is downloaded, use it
local_city	basestring	None	Local city (e.g., if deployed in Bilbao, should be Bilbao). This is used so WebLab-Deusto uses it for resolving local IP addresses
local_country	basestring	None	Local country, in ISO 3166 format (e.g., if deployed in Spain, should be ES). This is used so WebLab-Deusto uses it for resolving local IP addresses
ignore_locations	bool	False	Ignore the locations system (and therefore do not print any error if the files are not found)
logo_path	basestring	client/images/logo.jpg	File path of the logo.
logo_small_path	basestring	client/images/logo-mobile.jpg	File path of the small version of the logo.

Scheduling

This is the configuration variables used by the scheduling backend (called Coordinator). Basically, you can choose among redis or a SQL based one, and customize the one selected.

Property	*Type*	*Default value*	*Description*
core_coordination_impl	basestring	sqlalchemy	Which scheduling backend will be used. Current implementations: 'redis', 'sqlalchemy'.
core_coordinator_db_host	basestring	localhost	Host of the database server.
core_coordinator_db_port	int	None	Port of the database server.
core_coordinator_db_name	basestring	WebLabCoordination	Name of the coordination database.
core_coordinator_db_username	basestring		Username to access the coordination database.
core_coordinator_db_password	basestring		Password to access the coordination database.
core_coordinator_db_engine	basestring	mysql	Driver used for the coordination database. We currently have only tested MySQL, although it should be possible to use other engines.
core_coordinator_laboratory_servers	list		Available laboratory servers. It's a list of strings, having each string this format: "lab1:inst@mach;exp1|ud-fpga|FPGA experiments", for the "lab1" in the instance "inst" at the machine "mach", which will handle the experiment instance "exp1" of the experiment type "ud-fpga" of the category "FPGA experiments". A laboratory can handle many experiments, and each experiment type may have many experiment instances with unique identifiers (such as "exp1" of "ud-fpga|FPGA experiments").

Facade

Here you can customize the general web services consumed by the clients. Stuff like which ports will be used, etc.

Property	*Type*	*Default value*	*Description*
core_facade_server_route	basestring	default-route-to-server	Identifier of the server or groups of servers that will receive requests, for load balancing purposes.
core_facade_bind	basestring		Binding address for the main facade at Core server
core_facade_port	int		Binding address for the main facade at Core Server

Database

The database configuration stores the users, groups, uses, etc.

Property	*Type*	*Default value*	*Description*
db_host	basestring	localhost	Location of the database server
db_port	int	None	Port where the database is listening, if any
db_database	basestring	WebLab	Name of the main database
db_engine	basestring	mysql	Engine used. Example: mysql, sqlite
db_echo	bool	False	Display in stdout all the SQL sentences
db_pool_size	int	5	Maximum number of spare connections to the database.
db_max_overflow	int	35	Maximum number of connections to the database.
weblab_db_username	basestring	weblab	WebLab database username
weblab_db_password	basestring		WebLab database user password
weblab_db_force_engine_creation	bool	False	Force the creation of an engine each time

2.9 Upgrading

You have installed WebLab-Deusto. However, you notice that there is a super-cool feature in a newer version. And you want to upgrade your current setup to this version.

There are several things that may change from one version to other:

1. The client code
2. The server code
3. Some new or old parameters changed
4. The database schema

The first two points only require you to download the changes and re-deploy it. The other two will require you to also modify your WebLab instance. We provide tools for all this.

Table of Contents

2.9.1 Upgrading the base system

So as to download the latest version, download the latest changes from the git repository. Basically, go to the directory where WebLab-Deusto is and do the following:

```
# Go wherever you downloaded it
$ cd /opt/weblabdeusto/
$ git pull
```

Then, the code changes will be there, but they will still not be deployed. Now you need to deploy both the code, by running:

```
# Go wherever you downloaded it
$ cd /opt/weblabdeusto/
$ python setup.py install
```

Warning: Before running the `setup.py install` process, you may need to delete the directory called `build` in the `server/src` directory. The reason is that sometimes, some old files are left there. Most of the times this step is not mandatory, but from time to time, it is required.

This will install all the new requirements, will copy everything to the deployment directory. From this point, you may create new WebLab-Deusto instances using the new deployment, by running:

```
$ weblab-admin create sample
```

As already explained in *First steps*.

2.9.2 Upgrading an existing instance

If you already have a running WebLab-Deusto instance, then you may want to upgrade it. This means changing the structure of the database, configuration variables and so on.

WebLab-Deusto, through its `weblab-admin` command, manages this, modifying the database and converting the old variables. However, this command may fail (there are too many combinations), and if it fails, your system might end up in an unrecoverable state. For this reason, you are encouraged to make a backup of both the WebLab-Deusto instance directory and the database (if it is SQLite, it's inside the directory, but if it is MySQL, it is outside it, and you might need a command like mysqldump).

Warning: You are encouraged to make backups of your data before proceeding, and even to run the following command in a copy of the directory using a different database.

So as to use the automatic upgrader, first stop your current instance, and then run the following:

```
$ weblab-admin upgrade sample
```

If you have made further changes (such as the location of the virtualenv, or the directory where the deployment is), you need to reconfigure the paths, by running the following and restart the web server (e.g., Apache):

```
$ weblab-admin httpd-config-generate sample
```

Once finished, you will be able to start again your system:

```
$ weblab-admin start sample
```

If there is any error, please *report it*.

CHAPTER 3

Remote laboratory development and management

This section is intended for people who is going to create a new laboratory using the WebLab-Deusto tools.

3.1 Remote laboratory development

Table of Contents

3.1.1 Introduction

This section covers the development of new remote laboratories using WebLab-Deusto. As detailed in *Technical description*, WebLab-Deusto provides a set of libraries so experiment developers can create their own remote laboratories.

There are two major approaches for using WebLab-Deusto:

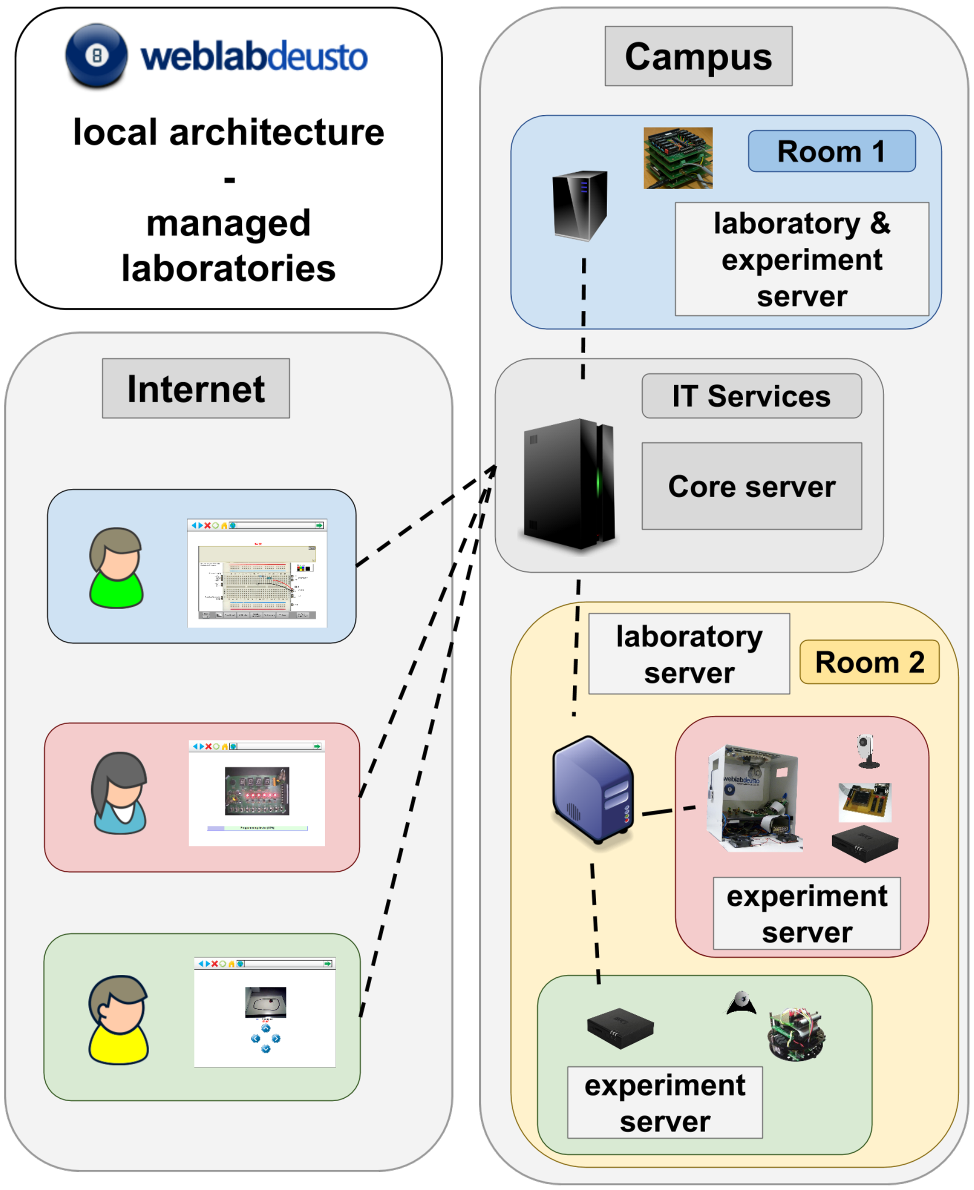
1. Managed laboratories
2. Unmanaged laboratories

Which are described below.

Note: This section explains the provided APIs and tools for development. However, you need to read *the following section* to register the new laboratory and use it. So you probably need to go from one document to the other during the development cycle.

Note: If you are familiar with Python, you could go to http://developers.labsland.com/weblablib/ which is a Python library suitable for web developers using Python and Flask.

Managed laboratories



Managed laboratories are those laboratories developed with the API of WebLab-Deusto. They basically have two parts:

- A **client**, developed using one of the libraries provided by WebLab-Deusto (see *below*).
- A **server**, developed using one of the provided server libraries or using XML-RPC directly (see *below*).

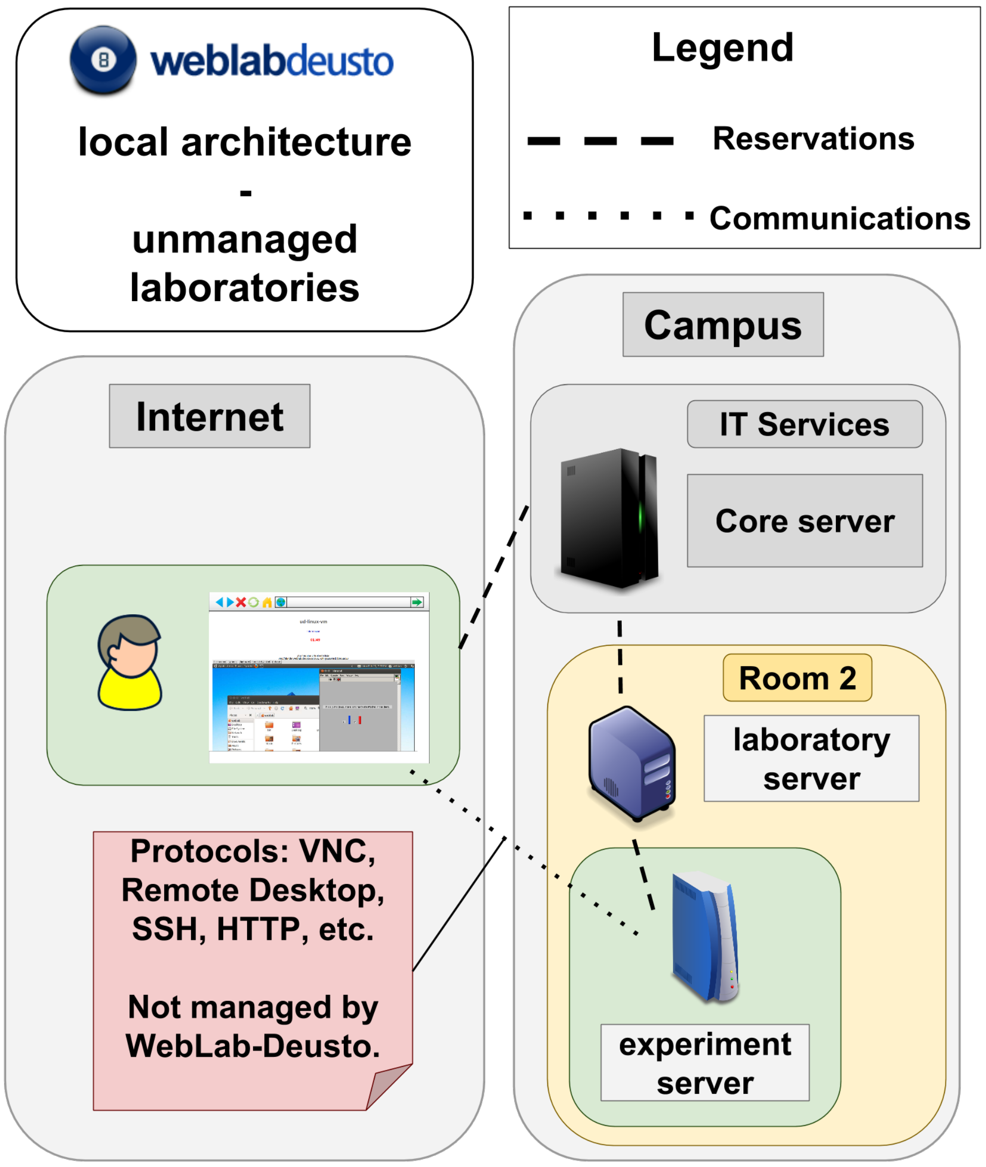
This way, the client will run on the web browser and will basically display the user interface. Whenever the user interface requires accessing the equipment, it will use the provided API to submit a command and retrieve a response. For example, a typical application might perform an action when the user presses a button. This button will submit a message (command) using the API, and WebLab-Deusto will call a particular method in the server side with that particular message.

Therefore, managed laboratories count with the following advantages:

- Experiment developer does not manage any type of **communications**. The client API has a method for submitting a command, which the WebLab-Deusto client will propagate as securely as the system has been configured (e.g., supporting HTTPS) to the server, which once in the campus side, the server will submit the command to the particular equipment (regardless where it is deployed in the internal topology of the campus side network). All commands submitted through WebLab-Deusto will go through pure HTTP, crossing firewalls and proxies.
- All the **information is stored** in the database by default, so it is possible to perform learning analytics. By default, administrators and instructors can track what exact commands were submitted by the student. This process however does not add a relevant latency, since instead of storing each command whenever is sent, it adds it to a memory queue (which is a fast operation), and other thread is continuosly retrieving information from the queue and storing it in the database in a batch basis.

WebLab-Deusto supports and provides libraries for multiple programming languages.

Unmanaged laboratories



However, not everybody in the remote laboratory community is comfortable with developing a remote laboratory from scratch by programming. For this reason, WebLab-Deusto also supports unmanaged laboratories, which are those where the communication is not sent through WebLab-Deusto, but directly to the final server.

A typical unmanaged environment works as follows: #. The user selects a laboratory in WebLab-Deusto #. When the user attempts to use a laboratory, WebLab-Deusto contacts the laboratory. Some secret is exchanged between both WebLab-Deusto and the laboratory, and WebLab-Deusto provides the user with a URL which contains a secret so the laboratory can identify the user. #. From that point, the user is redirected to that URL and he interacts directly with the laboratory.

This way, WebLab-Deusto still manages the reservation process, authentication (i.e., who is the user), authorization (i.e., in what groups are the user), scheduling (i.e., the queue of users) or user tracking (but only when did the user enter, not what was submitted by the user). However, the final communications are not managed by WebLab-Deusto.

The main **advantages** of unmanaged laboratories are:

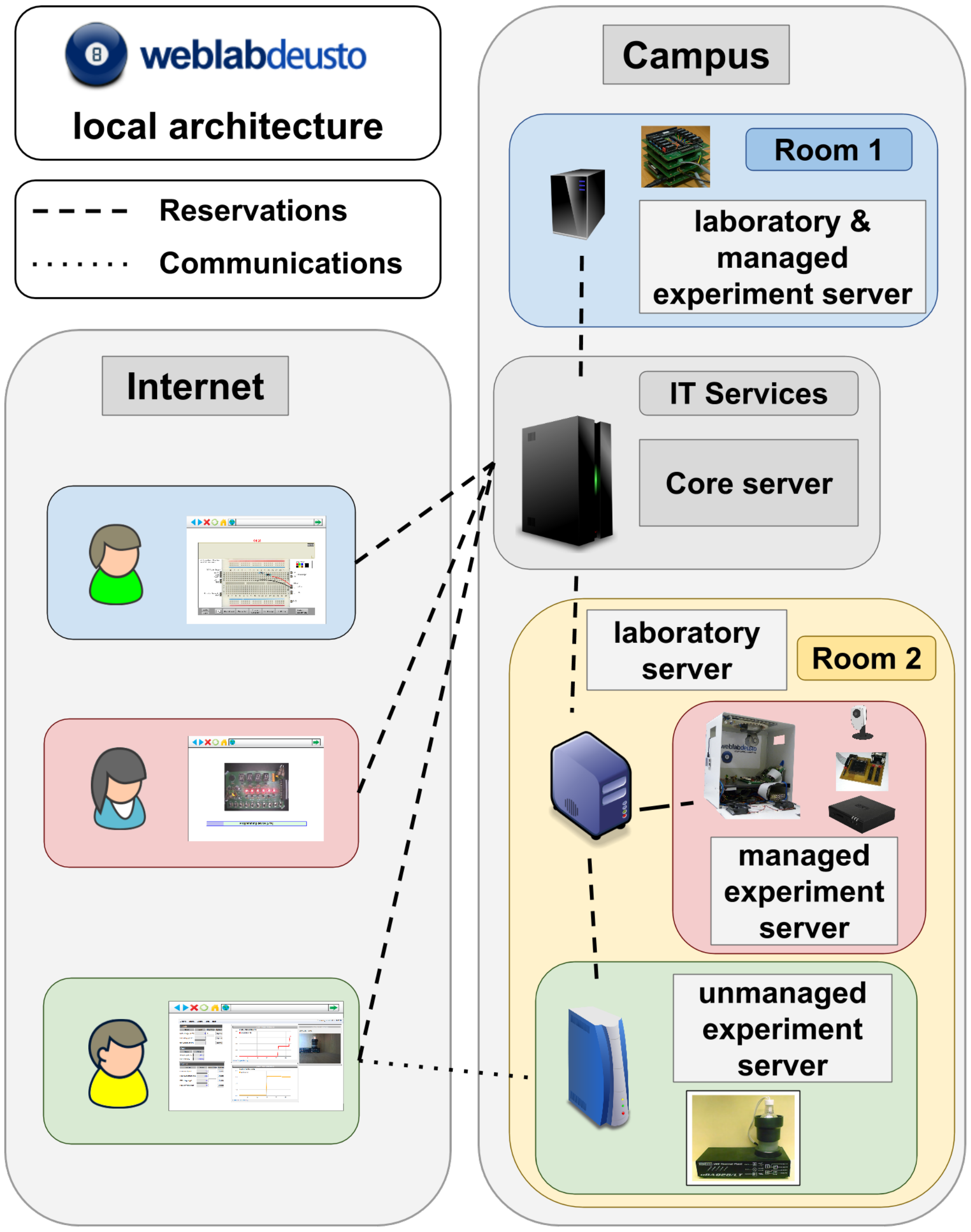
- You can use any web framework in any web framework you already know. There is no restriction on how the communications have to be managed.
- It supports further protocols not supported by WebLab-Deusto. For example, you might use WebSockets, which is not natively supported by WebLab-Deusto. But in an unmanaged laboratory, you can use them. Or if you use Virtual Machines, you can use SSH/VNC/Remote Desktop or whatever protocol you consider necessary for your laboratory.
- You can use libraries such as http://developers.labsland.com/weblablib/

Which one should I use?

It depends on your background. WebLab-Deusto supports both approaches because none of them is suitable for all publics:

1. If you are familiar with developing code in Java, .NET, Python or so on, but have little experience with web development, it might be easier for you to develop a managed laboratory. If you're not familiar with HTML + JavaScript, you might find tutorials on the Internet. Also, if you already know JavaScript and want to have the laboratory as something isolated in your network (so you do not need to deal with unauthenticated users or requests, with efficient approaches to know if the user has finished the session or not, etc.), then the managed approach is better for you. In this case, jump to *Managed laboratories*.
2. If you are familiar with developing complete web applications (e.g., in web frameworks such as Flask, Django or other technologies such as Node.js, PHP, ASP.NET or so), or you want to use special advances features (WebSockets, etc.), you might prefer to deploy the remote laboratory using one of these technologies and be in charge of the complete stack (e.g., managing who has access, checking when the user disconnected, etc.), so using the unmanaged approach might be more suitable for you. In this case, jump to *Unmanaged laboratories*.

In any case, both approaches are compatible in the same WebLab-Deusto server, so you might manage laboratories developed in each technology.



So the next step is to start with any of the two approaches:

- *Managed laboratories*
- *Unmanaged laboratories*

3.1.2 Managed laboratories

This section describes how to develop experiments using the managed model.

Introduction

As previously defined, in the managed laboratories, all the communications are managed by WebLab-Deusto. This basically means that, as seen on the following figure, the client code will call a set of methods such as:

```
// In the client side (JavaScript in this case)
weblab.sendCommand("press button1")
    .done(function (response) {
        console.log(response);
    })
    .fail(function (error) {
        console.log(error);
    });
```

And WebLab-Deusto guarantees that this string will be forwarded to the proper experiment server. In the experiment server, there will be a method such as:

```
//  (example in Java)
public String sendCommand(String command) throws WebLabException {
    // Manage the command and return the results
    if (command.startsWith("press ")) {
        String what = command.substring("press ".length);
        pressButton(what);
        return getStatus();
    } else {
        return "unknown";
    }
}
```

So as to do this, WebLab-Deusto provides *APIs for the client*, which wrap the communications submitting the commands to the server side using HTTP (and HTTPS if available), adding the required metadata (such as the session identifier). This is *step 1* in the following figure. Once in the Core server (check *the technical description if lost*), it checks if that the session is still available and with an experiment assigned. If so, it submits the command to the Laboratory server in charge of the assigned experiment (there might be different laboratory servers) and stores the command in the database. This process is faster than it may sound, since it uses memory structures and internal queues so there is only a single thread using the database for adding the commands submitted. This is *step 2* in the figure. Once in the Laboratory server, it checks to which Experiment server the command should be submitted, and submits it (this is *step 3*). If the Experiment Server was developed with one of the *libraries for servers*, then this gets the message in the programming language used and passes it to the Experiment server code.

This way, it is entirely up to the experiment developer to choose the proper programming environment for its experiments. Furthermore, developers will select the format of the contents submitted as commands. WebLab-Deusto does not impose any restriction on this side, so developers may send a simple string such as `press button1` that will later parse, or they may use an XML or JSON format.

For this reason, in the case of the managed model, developers do not need to handle:

- Scheduling (the core server manages it)
- Communications (the libraries manage it)
- User tracking (every command exchanged is already stored in the database)
- Complex deployments (e.g., load balancing: it is configured at WebLab-Deusto level)

So the next step is to develop the client and the server components in the technologies you select. Feel free to jump to:

- *Server side*
- *Client side*

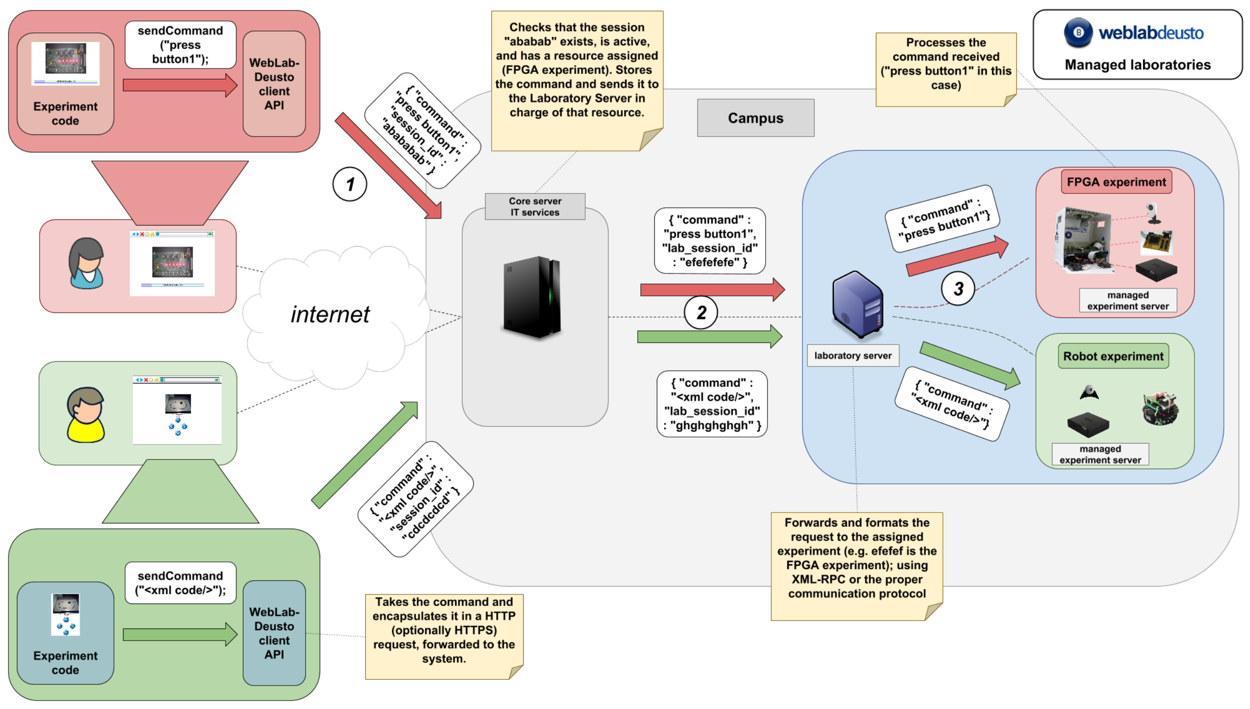

sendCommand ("press button1");
WebLab-Deusto client API
Experiment code
{ "command": "press button1", "session_id": "abababab" }
1
Checks that the session "ababab" exists, is active, and has a resource assigned (FPGA experiment). Stores the command and sends it to the Laboratory Server in charge of that resource.
Campus
Processes the command received ("press button1" in this case)
weblabdeusto
Managed laboratories
internet
Core server IT services
{ "command" : "press button1", "lab_session_id" : "efefefefe" }
2
{ "command" : "<xml code/>", "lab_session_id" : "ghghghghgh" }
laboratory server
{ "command" : "press button1"}
3
{ "command" : "<xml code/>"}
FPGA experiment
managed experiment server
Robot experiment
managed experiment server
{ "command": "<xml code/>", "session_id": "cdcdcdcd" }
Forwards and formats the request to the assigned experiment (e.g. efefef is the FPGA experiment); using XML-RPC or the proper communication protocol
sendCommand ("<xml code/>");
WebLab-Deusto client API
Experiment code
Takes the command and encapsulates it in a HTTP (optionally HTTPS) request, forwarded to the system.

Fig. 1: Command sent through the managed model. See the diagram in full size here.

Server side

There are two ways to develop a remote laboratory using the WebLab-Deusto API in the managed model:

- Using Python (which is the programming language used by the rest of the WebLab-Deusto system) as a native laboratory (therefore managing even the configuration through WebLab-Deusto).
- Running an external process which acts as a XML-RPC server. We provide libraries for doing this automatically, described below.

In this case, there is no prefered way to develop the laboratories, whatever is easier for the laboratory developer.

All the libraries can be found in the repository, in the experiments/managed/libs/server directory.

Before starting, there is a concept of API version or level for the Experiment server API. Basically, we started with a very simple API which contained the following methods:

```
void startExperiment();
void dispose();
String sendCommand(String);
String sendFile(String content, String fileInfo);
```

Changing this API breaks compatibility with existing laboratories. For this reason, we implemented a method called `get_api`, which returns the current API. And at the moment of this writing, there are 3 APIs:

- `1`, which is the one presented above.
- `2`, which is the one used in the majority of our laboratories, but not in all the libraries at this moment.
- `2_concurrent`, which right now is only provided in Python, while it should be easy to change the underlying XML-RPC services in each library to support it.

In the API v2, the methods we support are the following. We use Java syntax so it is clearer for any reader.

```
/**
* Receives two JSON documents. The first one (clientInitialData) is provided
* by the experiment client. The second one is provided by the core server,
* so it includes secure data that can be trusted. This second one might
* receive:
*
* - request.locale: language used by the client
* - request.username: login of the student
* - request.full_name: full name of the student (at this point, it's still
*      the username)
* - request.experiment_id.category_name: category of the experiment
* - request.experiment_id.experiment_name: experiment name
* - priority.queue.slot.length: time in seconds for the particular user
* - priority.queue.slot.start: since when counting this time
* - priority.queue.slot.initialization_in_accounting: whether the
*      initialization is counted or not in that time
*
* More parameters will be added in future versions.
*
* The startExperiment returns a JSON document too. It can be simply "{}".
* But it may contain the following information:
*
* - initial_configuration: a String that will be sent to the client in its
*      initialization. It may contain for example webcam URLs or similar of
*      the particular server.
* - batch: a boolean value stating if the current experiment is batch (and
*      therefore the experiment will be stopped just after calling this
*      method).
*
* So, examples of return values are:
*
*  - {}   (normal return)
*  - { "initial_configuration": "{\"webcam_url\": \"http://.../\"}" }
*  - { "initial_configuration": "(result)", "batch": true }
*
*/
String startExperiment(String clientInitialData, String serverInitialData);

/**
* returning "1", "2", "2_concurrent" and more in the future.
*/
String getApi();

/**
* Report WebLab-Deusto if the current user should be kicked out or not.
* In some laboratories, if certain circumstance happens, the user should not
* be using the laboratory more time. This method provides a mechanism for
* developers to activate this. To this end, this method will be called
* periodically. If such feature is not required, the laboratory should just
* return 0 (stating "don't call me again"). If the lab should be contacted
* often (e.g., every 30 seconds), the laboratory should return that time in
* seconds. So if it returns 5, it will be contacted in 5 seconds
* approximately, and if then it returns 10, it will be contacted in 10
* seconds approximately. Finally, if the user should be kicked out, -1
* should be returned.
*/
int shouldFinish();
```

(continues on next page)

(continued from previous page)

```
/**
* Send a command to the laboratory. You can encode whatever message here:
* either a JSON/XML or a simple string. WebLab-Deusto will not process this,
* it will just store it. Whatever is returned will be also sent to the
* client.
*/
String sendCommand(String message);

/**
* Send a file to the laboratory.
*/
String sendFile(File f, String info);

/**
* Tell the laboratory to clean the resources. Whenever the laboratory
* returns from this method, the laboratory will be assigned to someone else.
* If the laboratory might take long in cleaning resources, it should return
* a JSON stating that. It may also provide some information for the client
* to be displayed after finishing. To do this, it should return a JSON with
* some contents:
*
* - finished: true/false (if finished)
* - data: data to be returned to the client
* - ask_again: true/false (if not finished and want to be called again until
*       cleaning resources is finished.
*
* By default, just return "{}" (an empty JSON message).
*/
String dispose();
```

The Laboratory server can define which is the API of a laboratory. If it is not stated by the Laboratory server, the system attempts to request the API version. If it fails to provide it, it will assume that it is version `1` (where there was no such concept, and therefore, no explicit method detailing it). From that point, it will know which version the Experiment server is running and it will call the methods in one way or other (e.g., providing arguments to the startExperiment or not, using more methods, etc.).

In the concurrent version of the API, the same exact API is provided but it receives an additional argument at the beginning identifying the user session. This way, you can make a laboratory that supports 30 concurrent students accessing, and you can still identify who is who by the session identifier provided. So the methods become the following (the internals are the same as in the previous code):

```
/**
* Same as before, but it is told to receive a new user with sessionId.
*/
String startExperiment(String sessionId, String clientInitialData, String␣
↪serverInitialData);

/**
* returning "2_concurrent" and more in the future.
*/
String getApi();

/**
* Same as before but referring to a particular user.
*/
```

(continues on next page)

(continued from previous page)

```
int shouldFinish(String sessionId);

/**
* Same as before but referring to a particular user
*/
String sendCommand(String sessionId, String message);

/**
* Same as before but referring to a particular user
*/
String sendFile(String sessionId, File f, String info);

/**
* Same as before but referring to a particular user
*/
String dispose(String sessionId);
```

So as to make the development process easier, we provide libraries for different languages. However, some of them are in different versions (e.g., version 1, or version 2 or version 2_concurrent). If you want to support a different version, feel free to develop the library yourself and contribute it to our github (*Contribute*), or simply contact us to develop it (*Contact*). All the libraries can be found in the repository, in the experiments/managed/libs/server directory.

The following are available:

- *WebLab-Deusto server (Python)*
- *Java*
- *.NET*
- *C*
- *C++*
- *Node.js*
- *LabVIEW*
- *Python*

WebLab-Deusto server (Python)

There are two ways to develop the laboratory in Python. One is using all the WebLab-Deusto toolkit, and another using a simple script. The second one would be recommended for constrained devices (e.g., Raspberry Pi), while the first one could be more convenient for regular deployments. This section covers the first one. If you're interested on the second one, jump to *Python*.

In the case of Python, no external library is required, other than WebLab-Deusto itself. A dummy example would be the following:

```
import json

from weblab.experiment.experiment import Experiment
import weblab.experiment.level as ExperimentApiLevel

class DummyExperiment(Experiment):

    def __init__(self, coord_address, locator, cfg_manager, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super(DummyExperiment,self).__init__(*args, **kwargs)

        # Keep an instance of the configuration manager
        self._cfg_manager = cfg_manager

        # Retrieve a configuration variable from the configuration file:
        self._cfg_manager.get_value("property_name", "default_value")

    def do_start_experiment(self, client_initial_data, server_initial_data):
        """A new student is granted access to the laboratory (scheduled,
        authenticated, etc.)"""

        # Data provided by the client
        print "Client initial data:", json.loads(client_initial_data)
        # Data provided by the server (username, time slot...)
        print "Server initial data:", json.loads(server_initial_data)

        # Default response
        return "{}"

        # If you want to provide some initial data (URLs to cameras or so)
        # return json.dumps({ "initial_configuration" : "this will be batch", "batch"␣
↪: False })

        # In case of batch laboratories, use the following:
        # return json.dumps({ "initial_configuration" : "this will be batch", "batch"␣
↪: True })

    def do_get_api(self):
        # The current Laboratory API is the version 2. Whenever we add new
        # methods or change the API, it will not affect you if you are
        # stating that the API that the rest of the system must use with
        # this experiment is v2.
        return ExperimentApiLevel.level_2

    def do_dispose(self):
        """ The user exited (or the time slot finished). Clean resources. """

        print "User left"

        return "{}"

    def do_send_file_to_device(self, file_content, file_info):
        """ A file, encoded in BASE64, has been sent. Do something with it """

        return "A response that the client will receive"

    def do_send_command_to_device(self, command):
        """ A command has been submitted. Do something with it and reply. """

        print "Command received:", command

        return "Got your command"

    def do_should_finish(self):
        """
        Should the experiment finish? If the experiment server should be able to
```

(continues on next page)

(continued from previous page)

```
        say "I've finished", it will be asked every few time; if the experiment
        is completely interactive (so it's up to the user and the permissions of
        the user to say when the session should finish), it will never be asked.

        Therefore, this method will return a numeric result, being:
          - result > 0: it hasn't finished but ask within result seconds.
          - result == 0: completely interactive, don't ask again
          - result < 0: it has finished.
        """
        return 0
```

From this point, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

However, it is worth mentioning that there is other API called the concurrent API, which enables that the Experiment server manages multiple concurrent users at the same time. For example, imagine that you want that a fixed number (e.g., 10) students talk each other while using the laboratory. You can change this in the deployment, as it is later explained in *Concurrency*. But then, you would not be able to differentiate who is accessing, or send different messages to each student. For this reason, the concurrent API provides a unique identifier (which is a random number, and is not maintained across sessions) called `session_id`. This `session_id` is passed through all the methods, as seen below:

```
from weblab.experiment.concurrent_experiment import ConcurrentExperiment
import weblab.experiment.level as ExperimentApiLevel

class DummyConcurrentExperiment(ConcurrentExperiment):

    def __init__(self, coord_address, locator, cfg_manager, *args, **kwargs):
        super(DummyConcurrentExperiment,self).__init__(*args, **kwargs)

        # Keep an instance of the configuration manager
        self._cfg_manager = cfg_manager

        # Retrieve a configuration variable from the configuration file:
        self._cfg_manager.get_value("property_name", "default_value")

    def do_start_experiment(self, session_id, client_initial_data, server_initial_
↪data):
        # Store in a local dictionary that there is a new user defined as
        # session_id

        return "{}"

    def do_get_api(self):
        return ExperimentApiLevel.level_2_concurrent

    def do_dispose(self, session_id):
        # Remove that particular user from the active users
        return "{}"

    def do_send_file_to_device(self, session_id, file_content, file_info):
        # That user (identified by session_id) is sending a file

        return "A response that the client will receive"

    def do_send_command_to_device(self, session_id, command):
```

(continues on next page)

(continued from previous page)

```
        # That user (identified by session_id) is sending a command

        print "Command received:", command

        return "Got your command"

    def do_should_finish(self, session_id):
        # Should that user be kicked out?
        return 0
```

Now yes, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

Java

The Java library can be found in the experiments/managed/libs/server/java library. It is an Eclipse project, so you should be able to import it if you are using this IDE. Otherwise, you can use ant to compile it, by running:

```
$ ant build
$ ant run
```

The structure of the source code is the following:

```
+ src
  + es/deusto/weblab/experimentservers
    + exceptions
      - (defined exceptions)
    - ExperimentServer.java
    - Launcher.java
    - (Other auxiliar classes)
  + com/example/weblab
    - DummyExperimentServerMain.java
    - DummyExperimentServer.java
```

There, the important classes are those available in the package `es.deusto`. The ones in the `com.example` can be removed and replaced by the proper package of your application. They are there as a working example of what the interface is.

The two important classes are `ExperimentServer` and `Launcher`. The former is a class which defines all the optional methods which can be implemented by the experiment developer (e.g., a method for receiving commands). The latter is a class that will start a XML-RPC server taking an instance of the class generated by the experiment developer.

The first thing you must implement is a class which inherits from `ExperimentServer`. An example of this is the `DummyExperimentServer` class, which supports multiple methods such as:

```
// A new user comes in
public String startExperiment(String clientInitialData, String serverInitialData)␣
↪throws WebLabException {
    System.out.println("I'm at startExperiment");
    System.out.println("The client provided me this data: " + clientInitialData);
    System.out.println("The server provided me this data: " + serverInitialData);
    return "{}";
}
```

(continues on next page)

(continued from previous page)

```
// Typical server initial data:
// [java] The server provided me this data:
//        {
//          "request.locale": "es",
//          "request.experiment_id.experiment_name": "dummy",
//          "request.experiment_id.category_name": "Dummy experiments",
//          "priority.queue.slot.initialization_in_accounting": true,
//          "priority.queue.slot.start": "2013-03-27 00:36:08.397675",
//          "priority.queue.slot.length": "200",
//          "request.username": "admin"
//        }

// A user leaves (or is kicked out)
public String dispose() {
    System.out.println("I'm at dispose");
    return "ok";
}

public String sendFile(File file, String fileInfo)  throws WebLabException {
    System.out.println("I'm at send_program: " + file.getAbsolutePath() + ";␣
↪fileInfo: " + fileInfo);
    return "ok";
}

public String sendCommand(String command)  throws WebLabException {
    System.out.println("I'm at send_command: " + command);
    return "ok";
}
```

Those methods should parse the command send by the client and do the required actions (such as interact with certain equipment and return some response).

Once you have implemented this class, you can use the `Launcher` as:

```
public class DummyExperimentServerMain {
    public static void main(String [] args) throws Exception{
        int port = 10039;
        IExperimentServer experimentServer = new DummyExperimentServer();
        Launcher launcher = new Launcher(port, experimentServer);
        launcher.start();
    }
}
```

This way, you willhave the experiment running on port `10039` in this case. Once you have the server running, you will need to register it in WebLab-Deusto.

From this point, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

.NET

The .NET library is available here (so you have it in your WebLab-Deusto installation in `experiments/managed/libs/server/dotnet`):

https://github.com/weblabdeusto/weblabdeusto/tree/master/experiments/managed/libs/server/dotnet

At the time of this writing, it supports API v1. You can fill the `DummyExperimentServer.cs` example that uses the library:

```
using System;
using System.IO;

class SampleExperimentServer : WebLabDeusto.ExperimentServer {
        public string SendFile(byte [] file, string fileInfo){
            int length = file.Length;
            Console.WriteLine("File received: {0}", length);
            return "File received " + length;
        }

        public string SendCommand(string command){
            Console.WriteLine("Command received: {0}", command);
            return "Command received: " + command;
        }

        public void StartExperiment(){
            Console.WriteLine("Experiment started");
        }

        public void Dispose(){
            Console.WriteLine("Experiment disposed");
        }
}

class Tester{
    public static void Main(){
        WebLabDeusto.Runner runner = new WebLabDeusto.Runner(
                                            new SampleExperimentServer(),
                                            5678,
                                            "weblab"
                                    );
        runner.Start();
        Console.WriteLine("Press to shutdown");
        Console.ReadLine();

    }
}
```

From this point, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

C

The C library is available here (so you have it in your WebLab-Deusto installation in `experiments/managed/libs/server/c`):

https://github.com/weblabdeusto/weblabdeusto/tree/master/experiments/managed/libs/server/c

At the time of this writing, it supports API v2. You can fill the `dummy_experiment_server.c` example that uses the library:

```
#include "weblabdeusto_experiment_server.h"

char * start_experiment(){
```

(continues on next page)

(continued from previous page)

```
    return "{'initial_configuration' : {}, 'batch' : false}";
}

char * send_file(char * encoded_file, char * fileinfo){
    return "ok";
}

char * send_command(char * command){
    return "ok";
}

char * dispose(){
    return "ok";
}

int main(int const argc, const char ** const argv) {

    struct ExperimentServer handlers;
    handlers.start_experiment  = start_experiment;
    handlers.send_command      = send_command;
    handlers.send_file         = send_file;
    handlers.dispose           = dispose;

    /* For optional methods, you can use the default
       implementation by pointing to default_<handler-name> */
    handlers.is_up_and_running = default_is_up_and_running;
    handlers.should_finish = default_should_finish;

    launch(12345, handlers);

    return 0;
}
```

From this point, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

C++

The C++ library is available here (so you have it in your WebLab-Deusto installation in `experiments/managed/libs/server/cpp`):

> https://github.com/weblabdeusto/weblabdeusto/tree/master/experiments/managed/libs/server/cpp

At the time of this writing, it supports API v1. You can fill the `dummy_experiment_server.cpp` example that uses the library:

```
#include "weblabdeusto_experiment_server.hpp"

#include <iostream>


class DummyExperimentServer : public ExperimentServer
{
public:

    virtual std::string onStartExperiment()
```

(continues on next page)

(continued from previous page)

```
    {
        return "{'initial_configuration' : {}, 'batch' : false}";
    }

    virtual std::string onSendFile(std::string const & encoded_file, std::string␣
↪const & fileinfo)
    {
        return "ok";
    }

    virtual std::string onSendCommand(std::string const & command)
    {
        return "ok";
    }

    virtual std::string onDispose()
    {
        return "ok";
    }
};



int main(int argc, char const * argv[])
{
    DummyExperimentServer testServer;
    testServer.launch(12345, "rpc_log.txt");
}
```

From this point, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

Node.js

The Node.js library is available here (so you have it in your WebLab-Deusto installation in `experiments/managed/libs/server/nodejs`):

https://github.com/weblabdeusto/weblabdeusto/tree/master/experiments/managed/libs/server/nodejs

At the time of this writing, it supports API v2. You can fill the `dummyexperimentserver.js` example that uses the library:

```
experimentserver = require("./node.weblab.experimentserver");

DummyExperimentServer = new function() {

    this.test_me = function(message) {
        console.log("On test_me");
        return message;
    }

    // Is the experiment up and running?
    // The scheduling system will ensure that the experiment will not be
    // assigned to other student while this method is called. The result
    // is an array of integer + String, where the first argument is:
    //   - result >= 0: "the experiment is OK; please check again
```

(continues on next page)

(continued from previous page)

```
    //                  within $result seconds"
    //    - result == 0: the experiment is OK and I can't perform a proper
    //                  estimation
    //    - result == -1: "the experiment is broken"
    // And the second (String) argument is the message detailing while
    // it failed.
    this.is_up_and_running = function() {
        console.log("On is_up_and_running");
        return [600, ""];
    }

    this.start_experiment = function(client_initial_data, server_initial_data) {
        // Start experiment can return a JSON string specifying the initial␣
↪configuration.
        // The "config" object can contain anything. It will be delivered as-is to␣
↪the client.
        var config = {};
        var initial_config = { "initial_configuration" : config, "batch" : false };
        return JSON.stringify(initial_config);
    }

    this.send_file = function (content, file_info) {
        console.log("On send_file");
        return "ok";
    }

    this.send_command = function (command_string) {
        console.log("On send_command");
        return "ok";
    }

    // Returns a numeric result, defined as follows:
    // result > 0: it hasn't finished but ask within result seconds.
    // result == 0: completely interactive, don't ask again
    // result < 0: it has finished.
    this.should_finish = function() {
        return 0;
    }

    // May optionally return data as a string, which will often be json-encoded.
    this.dispose = function () {
        console.log("On dispose");
        return "ok";
    }
}


experimentserver.launch(12345, DummyExperimentServer);
```

From this point, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

LabVIEW

Note: This is the LabVIEW *managed* library. It does not support using Remote panels or so on. It only supports that you serialize the messages and write your own client in JavaScript using it.

The LabVIEW library is available here (so you have it in your WebLab-Deusto installation in `experiments/managed/libs/server/labview`):

> https://github.com/weblabdeusto/weblabdeusto/tree/master/experiments/managed/libs/server/labview

At the time of this writing, it supports API v1. You can fill the `DummyExperimentServer.vi` example that uses the library.

From this point, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

Python

There are two ways to develop the laboratory in Python. One is using all the WebLab-Deusto toolkit, and another using a simple script. The second one would be recommended for constrained devices (e.g., Raspberry Pi), while the first one could be more convenient for regular deployments. This section covers the second one. If you're interested on the first one, jump to *WebLab-Deusto server (Python)*.

The Python library can be found in the experiments/managed/libs/server/python. It does not rely on any external library, since Python comes with an XML-RPC server included. You will find two modules:

- `weblab_server.py`, which includes the `ExperimentServer` and the `Launcher`.
- `sample.py`, which includes an example of how to use the `ExperimentServer` code, and how to run it with the `Launcher`.

Basically, you have to create a class which inherits from `ExperimentServer` and implements a subset of the following methods (none of these are required since they are already implemented in the parent class):

```python
from weblab_server import ExperimentServer, Launcher

class DummyExperimentServer(ExperimentServer):

    def start_experiment(self, client_initial_data, server_initial_data):
        print "start_experiment", client_initial_data, server_initial_data
        return "{}"

    def get_api(self):
        return "2"

    def send_file(self, content, file_info):
        print "send_file", file_info
        return "ok"

    def send_command(self, command_string):
        print "send_command", command_string
        return "ok"

    def dispose(self):
        print "dispose"
        return "{}"

    def should_finish(self):
        return 0
```

Then, you only need to create a Launcher with a port, and start it:

```
launcher = Launcher(12345, DummyExperimentServer())
launcher.start()
```

From this point, you can now deploy the experiment, as explained in the *following section* steps 1 to 4, or to jump to the client development (*Client side*).

Client side

The client code is focused on two tasks:

- Providing the user interface
- Submitting commands to the Experiment server and managing the responses

While WebLab-Deusto supports some web libraries, it is highly recommended to use the JavaScript library (as opposed to Flash or Java applets). Those laboratories developed on top of it will be available for mobile devices, and the number of conflicts in different platforms will be highly decreased, since they will not need any plug-in installed.

In the following sections describe how to use each of the provided APIs. Jump to the technology you are more comfortable with:

- *JavaScript*
- *Flash applets*
- *Java applets*
- *Google Web Toolkit*

JavaScript

The recommended programming language for managed laboratories is JavaScript:

- It is easy. You simply develop an HTML file without any restriction. You include a JavaScript that WebLab provides to interface with the server.
- Does not have any dependency, other than a JavaScript script file and jQuery.
- Can easily make use of any kind of JavaScript library or framework.
- Possible to develop and test the experiments offline, without deploying a WebLab first. You can just open the HTML file in a browser.

What to develop

In order to create a new experiment, essentially you need:

- An experiment server
- **An experiment client**

This section is dedicated to the latter (an experiment client). An experiment client provides the user interface and client-side logic that your particular experiment requires. It communicates with WebLab and the experiment server through a very simple API.

When you create a WebLab-Deusto environment, it creates a `pub` directory. Whatever you put on this directory is available in http://localhost/weblab/web/pub/ . You can put HTML/JS/CSS files there. The most basic version of your first JavaScript lab will look like this:

```
<html>
   <head>
     <script src="https://code.jquery.com/jquery-1.11.3.min.js"></script>
     <script src="../static/weblabjs/weblab.v1.js"></script>
   </head>
   <body>
     <p>Hello world</p>
   </body>
</html>
```

Make sure that the weblab.v1.js file is properly configured. On a typical environment, it is available in http://localhost/weblab/web/static/weblabjs/weblab.v1.js, so one file called http://localhost/weblab/web/pub/mylab.html will refer to it as `../static/weblabjs/weblab.v1.js`, but if you create the file in a different directory (e.g., in a directory `mylab` in the pub directory), then you need more `../`.

This HTML that you have just created is meant to be your experiment's interface. It will appear within WebLab-Deusto as an iframe. If we continue with the aforementioned example, you might want to add, for instance, a webcam feed to your HTML, and maybe some JavaScript button. Because it is just standard HTML, you can use any library or framework you wish to make your work easier.

Once you draw buttons or things, you only need to interact with the expeirment server, by sending and receiving commands. This is done through the JavaScript API, which will be explained next.

JavaScript API

The WebLab API is relatively simple. The basic API provides these base functions, which is all you really need:

- Sending a command.
- Sending a file.
- Receiving an experiment-starts notification.
- Receiving an experiment-ends notification.
- Forcing the experiment to end early.

For JavaScript, this API can be found in the following place /weblab/web/static/weblabjs/weblab.v1.js

The API follows:

```
    //! Sends a command to the experiment server.
//!
//! @param text Text of the command.
//! @returns a jQuery.Deferred object. You might use it to register callbacks, if␣
↪desired.
//! Takes a single string as argument.
//!
weblab.sendCommand(text)
    .done(function(message) {
        // ...
    })
    .fail(function(error) {
        // ...
    })

    //! Sends a file to the experiment server.
//!
```

(continues on next page)

(continued from previous page)

```
//! @param file An <input type="file"> element (the result of a document.
↪getElementById or $("#fileinput"))
//! @param fileInfo A string describing the file (e.g., a file name or whatever).
//! @returns a jQuery.Deferred object. You might use it to register callbacks, if␣
↪desired.
//! Takes a single string as argument.
//!
weblab.sendFile(file, fileInfo)
    .done(function(message) {
        // ...
    })
    .fail(function(error) {
        // ...
    })


//! Sets the callback that will be invoked when the experiment finishes. Generally,
//! an experiment finishes when it runs out of allocated time, but it may also
//! be finished explicitly by the user or the experiment code, or by errors and
//! and disconnections.
//!
weblab.onFinish(function() {
    // Do something when the user finishes
});

//! Sets the startInteractionCallback. This is the callback that will be invoked
//! after the Weblab experiment is successfully reserved, and the user can start
//! interacting with the experiment.
weblab.onStart(function (time, initialConfig) {
    // Work with the initialConfig (provided by your experiment server) and the
    // remaining time (you're responsible of keep track of it once received)
});
```

Using the API is easy. Once the script has been included, you can simply call:

```
weblab.sendCommand( "LIGHTBULB ON")
        .done(function(response) {
                console.log("Light turned on successfully");
        })
.fail(function(response) {
                console.error("Light failed to turn on");
        });
```

Note that as you can see above, there are some functions that start with "dbg". Those are for development purposes. Sometimes, for instance, it is convenient to be able to run your HTML interface stand-alone. In order for the experiment to behave in a way that more closely resembles in its intended way, you can use these to simulate command responses from the server and the like.

Note: If you want to start the experiment from your JavaScript code, you may call `weblab.startReservation();`

After reading this, keep either reading in this document about tools (*Tools*) or summary (*Summary*), or jump to the deployment section (*Remote laboratory deployment*; in particular *JavaScript* if you've done the previous steps in that document) or to the sharing section (*Remote laboratory sharing*).

Java applets

You can develop the client side using Java Applets. They will not work in tablets (iPad, Android) neither in mobile phones, and many web browsers nowadays do not support Java applets. For this reason, it is recommended to avoid it and rely on JavaScript or Google Web Toolkit (see below).

However, if you are going to reuse code or you need to use Java Applets for other reasons, WebLab-Deusto supports it. If you go to experiments/managed/libs/client, you will find the Java applets source code. It is an Eclipse project, so you should be able to import it if you are using this IDE. Otherwise, you can use ant to compile it, by running:

```
$ ant package
```

The package hierarchy is the following:

```
+ es.deusto.weblab.client.experiment.plugins
  + es.deusto.weblab.javadummy
    + commands
      - PulseCommand
    - JavaDummyApplet
  + java
    - WebLabApplet
    - ICommandCallback
    - ResponseCommand
    - ConfigurationManager
    - BoardController
    - Command
```

The `java` package is the library itself, used by WebLab-Deusto. The `es.deusto.weblab.javadummy` package is just an example of a user interface built using this library. You may remove it and use your own package, even outside (e.g., `edu.myuniversity.mylab`). However, you must include the `java` package.

The first step is to make a class which inherits from `WebLabApplet` (view code). In the example, this class is `JavaDummyApplet` (view applet code). This new class is the one which will be instanciated by WebLab-Deusto. It will be instanciated whenever the user selects the laboratory, before reserving it. Then, there are three methods that should be implemented by this class:

```
public void startInteraction() {
    // When this method is called, student has access to
    // the remote equipment (he has been assigned). You
    // can show a cool user interface for your remote
    // laboratory and call the sendCommand methods (later
    // explained).
}

public void setTime(int time) {
    // This method is called to inform you how many seconds
    // the user will be using this laboratory. You should
    // print it somewhere and maintain a custom counter.
}

public void end() {
    // When this method is called, the user has stated that
    // he is not using the laboratory anymore, or the system
    // has kicked him out (e.g., because his slot finished).
}
```

From this point, the client knows when the user interfaces should be loaded. So as to interact with the Experiment server, the `WebLabApplet` provides a method which gives access to the `BoardController`. The

BoardController provides a set of methods for submitting commands.

```
// From the class which inherits from JavaDummyApplet:
MyCommandCallback callback = new MyCommandCallback();

// Send a message to the Experiment server, and provide a callback
// which will be called when the method comes back.
this.getBoardController().sendCommand("turn switch on", callback);
```

The callback itself can be defined as follows:

```
// Somewhere else
public class MyCommandCallback inherits ICommandCallback {

    public void onSuccess(ResponseCommand response) {
        String responseText = response.getCommandString();
        // Do something with the message returned from the
        // Experiment server.
    }

    public void onFailure(String message) {
        // Something failed in the server side or in the
        // communications. Do something with the error.
    }
}
```

Additionally, the WebLabApplet class provides other methods, such as:

```
// Call this if you want to terminate the current session
this.getBoardController().onClean();

// Retrieve String properties from the configuration.js file
this.getConfigurationManager().getProperty("my.property", "default value");

// Retrieve int properties from the configuration.js file
this.getConfigurationManager().getIntProperty("my.property");
```

After reading this, keep either reading in this document about tools (*Tools*) or summary (*Summary*), or jump to the deployment section (*Remote laboratory deployment*; in particular *Java applets* if you've done the previous steps in that document) or to the sharing section (*Remote laboratory sharing*).

Flash applets

We provide a .fla project in experiments/managed/libs/client/flash to see a simple sample of !WebLab accessed from Adobe Flash, as well as a .as file with all the glue code that the experiment developer might use. You can see it in github.

In order to create a new experiment using WeblabFlash, weblab.util.WeblabFlash must be imported.

Done this, it is possible to access the singleton instance of WeblabFlash through its static method getInstance().

```
//! Retrieves a reference to the only instance of WeblabFlash.
//!
public static function getInstance() : WeblabFlash {
    // ...
}
```

After getting a reference to this instance the programmer should register the weblab callbacks. This is done through the registerCallbacks method. The callback functions to call whenever events take place are passed to registerCallbacks as parameters. If being notified of a certain event is not required, it is possible to pass null for it instead. Events are, in order: `onSetTime`, `onStartInteraction`, `onEnd`, `onSecondEllapsed`. This last one is null by default and hence may not be passed at all.

```
//! Registers the JS callbacks setTime, startInteraction and end, so that
//! the appropiate user-specified delegate is automatically called when appropiate.
//! Optionally registers a seconds timer.
//!
//! @param setTime Function called to set the time.
//! @param startInteraction Function called to start interacting with the user.
//! @param end Function called when the experiment ends.
//! @param onSecondEllapsed Function called every second.
public function registerCallbacks(setTime : Function, startInteraction : Function,␣
↪end : Function, onSecondEllapsed : Function = null) : void{
   //...
}
```

Anyway, the developer can ask for the state at any moment. An experiment may be found in one of three different states:

- `WeblabFlash.STATE_WAITING`: When weblab is yet to call startInteraction.
- `WeblabFlash.STATE_INTERACTING`: When startInteraction has been called and therefore the experiment has started, and is not done yet.
- `WeblabFlash.STATE_FINISHED`: When weblab has called `onEnd()` or when `onClean()` has been called locally.

Current state may be obtained through the getcurrentState() method.

The method sendCommand is used to send commands to the server. It takes a string with the command as first parameter plus two callback functions. First of those will be called in case the command succeeds and the other one in case it fails. Both callbacks are passed a message string when called.

```
//! Sends a command to the server. Its response will be received asynchroneously␣
↪through
//! two alternative callbacks.
//!
//! @param command_str The command string.
//! @param onSuccess Function to call if the command succeeds. Should take the␣
↪response string
//! as a parameter.
//! @param onError Function to call if the command fails. Should take the response␣
↪string
//! as a paramter.
public function sendCommand(command_str : String, onSuccess : Function, onError :␣
↪Function) {
   //...
}
```

The time of an experiment is limited. `WeblabFlash` internally keeps a timer. This timer is initialized to the value passed by weblab through its `setTime()` call and starts decreasing once interaction starts. When zero is reached, `onClean()` is automatically called and the experiment is considered to be finished. `onClean()` may also be called explicitly before the timer reaches zero. Moreover, weblab may call `onEnd()` at any time to finish the experiment. Whenever this happens, the programmer is responsible to clean all resources in use (such as timers).

In order to retrieve the global configuration values, stored in the configuration.xml file in the client side, the developer may call these methods:

```
//! Retrieves a property as a string.
//!
//! @param prop The name of the property.
//! @param def The default value to return if the property is not found.
public function getPropertyDef( prop : String, def : String):String {
    // ...
}

//! Retrieves a property as a string.
//!
//! @param prop The name of the property.
public function getProperty( prop : String ):String {
    // ...
}

//! Retrieves an integer property.
//!
//! @param prop The name of the property.
//! @param def The default value to return if the property is not found.
public function getIntPropertyDef( prop : String, def : int ):int {
    // ...
}

//! Retrieves an integer property.
//!
//! @param prop The name of the property.
public function getIntProperty( prop : String ):int {
    // ...
}
```

After reading this, keep either reading in this document about tools (*Tools*) or summary (*Summary*), or jump to the deployment section (*Remote laboratory deployment*; in particular *Flash* if you've done the previous steps in that document) or to the sharing section (*Remote laboratory sharing*).

Google Web Toolkit

Google Web Toolkit (GWT) used to be the technology used by WebLab-Deusto in all the remote laboratories. Nowadays, it is still used in certain legacy laboratories and systems.

How to develop the lab

The source code is in `client/src`. The main package is `es.deusto.weblab.client`, and inside this, all the laboatories are in the `experiments` package (see here). You can check any, but the binary one is one of the simplest ones. As you can see there, you have to create a class which inherits from `IExperimentCreatorFactory`. We'll go to this one later. Then you see a package called `ui`. It contains both the class inheriting from `ExperimentBase` and the different user interfaces.

First, let's see the class inheriting from `ExperimentBase`. In this case, it is called `BinaryExperiment`. The ExperimentBase class is the one from which all the remote laboratories inherit. It contains many methods, most of them optinally implemented by the experiment class. The methods are the following:

```
/**
 * User selected this experiment. It can start showing the UI. It can
 * load the VM used (Adobe Flash, Java VM, Silverlight/Moonlight, etc.),
 * or define requirements of the (i.e. require 2 files, etc.). It should
 * also show options to gather information that will be sent to the
```

(continues on next page)

(continued from previous page)

```
 * initialization method of the experiment server, that later will be
 * retrieved through the {@link #getInitialData()} method.
 */
public void initialize(){}

/**
 * A user, who performed the reservation outside the regular client (in
 * a LMS or a federated environment) is going to start using this
 * experiment. Basically it is like the {@link #initialize()} method,
 * except for that it should be very fast, and take into account that no
 * configuration can be provided (since the reservation has already been
 * done).
 */
public void initializeReserved(){
    initialize();
}

/**
 * Retrieves information sent to the experiment when reserving the
 * experiment. It might have been collected in the UI of the
 * {@link #initialize()} method.
 */
public JSONValue getInitialData(){
    return null;
}

/**
 * User is in a queue. Thetype filter text typical behavior will be to hide the UI␣
↪shown
 * in the {@link #initialize()} method.
 */
public void queued(){}

/**
 * User grabs the control of the experiment (in the server side, the
 * experiment is already reserved for the user).
 *
 * @param time Seconds remaining. This time is the maximum permission time.
 * @param initialConfiguration Data sent by the experiment server in the
 * initialization method.
 */
public void start(int time, String initialConfiguration){}

/**
 * User experiment session finished. The experiment should clean
 * its resources, or notify the user that it has finished. It may still
 * wait for the {@link #postEnd(String)} method to be called so as to
 * receive the information sent by the experiment when disposing
 * resources.
 */
public void end(){}
```

Therefore, it is up to the Experiment client to do something when these methods are called. Most typically, developers will implement the `start` method and the `end` method. Additionally, all the classes inheriting from `ExperimentBase` also inherit the following attributes:

```
protected final IBoardBaseController boardController;
protected final IConfigurationRetriever configurationRetriever;
protected static final IWebLabI18N i18n = GWT.create(IWebLabI18N.class);
```

The first one allows developers to interact with the Experiment Server, as documented here, but the most relevant are the following:

```
////////////////////////////////////
//
// General information
//

/**
 * Is the user accessing through facebook?
 */
public boolean isFacebook();

/**
 * What is the session id of the user? It is useful when using other type of␣
↪communications, such
 * as iframes in the LabVIEW experiment.
 */
public SessionID getSessionId();

////////////////////////////////////
//
// Sending commands
//

/**
 * Send a string command, don't care about the result
 */
public void sendCommand(String command);

/**
 * Send a string command, notify me with the result
 */
public void sendCommand(String command, IResponseCommandCallback callback);


////////////////////////////////////
//
// Cleaning
//

/**
 * Clean the experiment widgets and move to the list of experiments
 */
public void clean();

/**
 * Finish the experiment.
 */
public void finish();
```

So basically, what you do when implementing an experiment is to inherit from `ExperimentBase`, override the `start` method to be notified when the laboratory has been assigned, and then start showing the user interface and interacting with the server through commands. In the `end` method, you usually clean up the remaining resources (e.g., stop camera streams, cancel timers, etc.).

However, there are some common operations, such as putting a panel in the screen, or cleaning up the resources. For this reason, there is a utility class which inherits from `ExperimentBase` called UIExperimentBase, which provides the following extra operations:

```
protected void putWidget(Widget widget);

protected void addDisposableWidgets(IWlDisposableWidget widget);
```

So you don't need to implement the `end` method, and simply use these methods to add the resources to be cleaned. In the binary experiment, you can see that `BinaryExperiment` (see code) inherits from this class. In that class, you may see that it simply calls to the `putWidget` method, providing the panels implemented in that laboratory. For instance, you can see the user interface in XML, and the attached Java code which manages the event handlers and calls the `sendCommand` method in the `processSwitch` method to interact with the laboratory.

Finally, going back to the class which inherits from `IExperimentCreatorFactory`, it will have two methods, and you can copy and paste code as most of them are very similar:

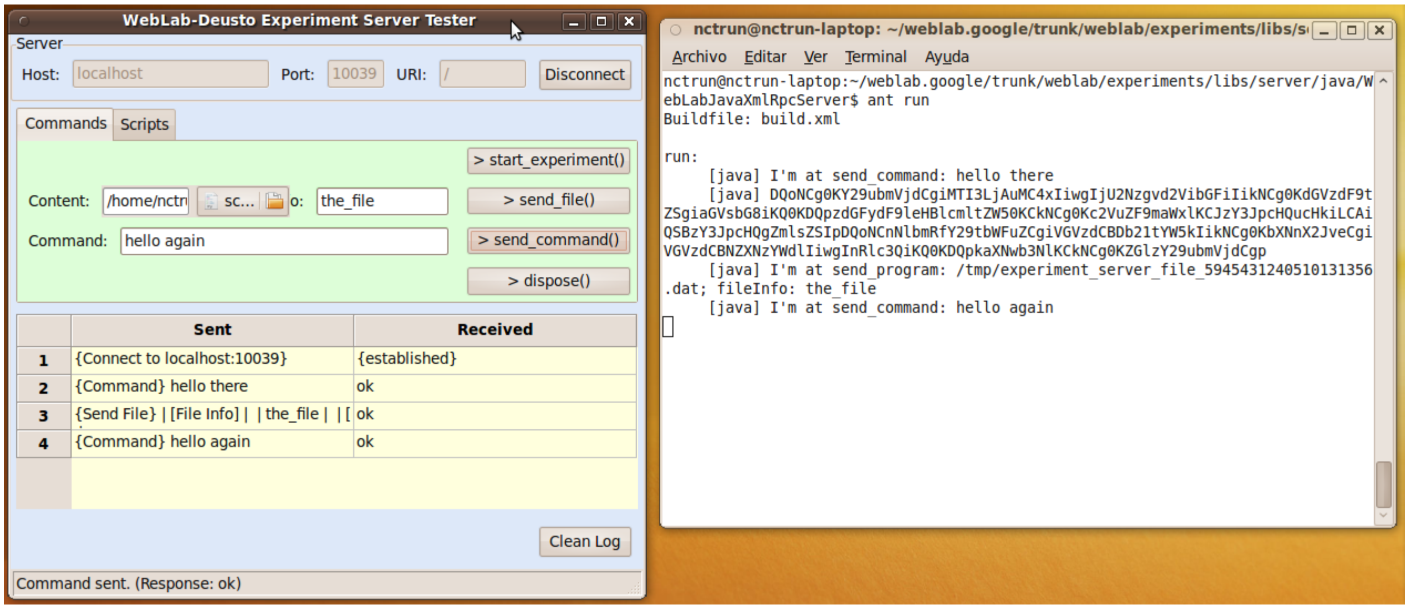
1. `getCodeName`, which returns a unique code for that laboratory, and will later be used in the deployment.
2. `createExperimentCreator`, which creates an `ExperimentCreator` class where you define the support in mobile phones (so later in mobile phones the user interface will show a different color), and where internally it will create the user interface. Note that it uses `GWT.runAsync` to define that this code will not be compiled in the same JavaScript and it will be loaded only once the user has clicked on this laboratory. The `ExperimentCreator` has other method called `createMobile`, used if you want to pass an alternative user interface for mobile devices, as done in the logic laboratory.

```
@Override
public void createWeb(final IBoardBaseController boardController, final␣
↪IExperimentLoadedCallback callback) {
    GWT.runAsync(new RunAsyncCallback() {
        @Override
        public void onSuccess() {
            callback.onExperimentLoaded(new BinaryExperiment(
                    configurationRetriever,
                    boardController
                ));
        }

        @Override
        public void onFailure(Throwable e){
            callback.onFailure(e);
        }
    });
}
```

After reading this, keep either reading in this document about tools (*Tools*) or summary (*Summary*), or jump to the deployment section (*Remote laboratory deployment*; in particular *Configuring the client in a managed laboratory*) or to the sharing section (*Remote laboratory sharing*).

Tools

The *Experiment Server Tester* helps you in the process of writing the server code, by being able to test it easily with a graphic tool, and even create your own scripts to verify that it works as expected. Refer to that documentation for using it.



	Sent	Received
1	{Connect to localhost:10039}	{established}
2	{Command} hello there	ok
3	{Send File} | [File Info] | | the_file | | [	ok
4	{Command} hello again	ok

Command sent. (Response: ok)

```
nctrun@nctrun-laptop:~/weblab.google/trunk/weblab/experiments/libs/server/java/W
ebLabJavaXmlRpcServer$ ant run
Buildfile: build.xml

run:
     [java] I'm at send_command: hello there
     [java] DQoNCg0KY29ubmVjdCgiMTI3LjAuMC4xIiwgIjU2Nzgvd2VibGFiIikNCg0KdGVzdF9t
ZSgiaGVsbG8iKQ0KDQpzdGFydF9leHBlcmltZW50KCkNCg0Kc2VuZF9maWxlKCJzY3JpcHQucHkiLCAi
QSBzY3JpcHQgZmlsZSIpDQoNCnNlbmRfY29tbWFuZCgiVGVzdCBDb21tYW5kIikNCg0KbXNnX2JveCgi
VGVzdCBNZXNzYWdlIiwgInRlc3QiKQ0KDQpkaXNwb3NlKCkNCg0KZGlzY29ubmVjdCgp
     [java] I'm at send_program: /tmp/experiment_server_file_5945431240510131356
.dat; fileInfo: the_file
     [java] I'm at send_command: hello again
```

3.1.3 Unmanaged laboratories

The unmanaged approach is just a different model for developing remote laboratories. At the time of this writing, there are mainly three types of unmanaged laboratories:

- *Using weblablib* (recommended)
- *HTTP unmanaged laboratories* (recommended)
- *LabVIEW Remote Panels* (experimental)
- *Virtual machines* (limited)

Using weblablib

weblablib is a Python library which enables you to develop unmanaged laboratories easily if you have certain Python knowledge.

So as to put a simple example:

```
# Install using "pip install weblablib"
weblab = WebLab(app)

@weblab.initial_url
def initial_url():
    return url_for("index")

@weblab.on_dispose
def stop():
    print("Cleaning code...")

@app.route("/")
@requires_active
def index():
    return "Hello {}".format(weblab_user.username)
```

HTTP unmanaged laboratories

The HTTP unmanaged laboratories target that you can develop laboratories in your preferred web technology. It is by far the most flexible approach, and the most powerful, but also the one that requires developers to be in charge of more tasks.

The basis is that developers implement a interface (detaled in *Interface specification*) that WebLab-Deusto will use as a client to contact your server for five tasks:

1. Notifying that a new user comes. Your server does not need to control the queue of users or user authentication: WebLab-Deusto is still charge of that, so it will notify you only whenever a valid user has a valid reservation and must be able to access. WebLab-Deusto will tell you for how long, what's the username, and some more data. You will have to generate a URL that should include a private session or token that you generate so that anyone going to that website will be able to use the laboratory.
2. Requesting if the user is still there. A user might be assigned 10 minutes, but he might leave after 30 seconds. If this happens, the laboratory might be assigned to that user still for 9 minutes more, potentially with people in the queue. You are responsible of checking whether the user has left or not.
3. Notifying that a user must finish. When WebLab-Deusto establishes it (because the time is over, an administrator kicked the user or similar), the laboratory must be able to make sure that the user is not valid anymore. You are responsible of making sure that the user can't do anything else after this happens, and that he is redirected to a URL provided by WebLab-Deusto in the beginning.
4. Providing the API level. The HTTP unmanaged laboratories API has different versions. Knowing what version is running on your server lets WebLab-Deusto contact using more or less parameters. So, as long as you report what API version you're running, you will be fine (and WebLab-Deusto will use that API version to contact you, instead of a newer version with more arguments).
5. Providing a test service. The HTTP unmanaged laboratories rely on a set of credentials to verify that it is WebLab-Deusto the one contacting your server to the main methods described above. So as to be able to automatically check problems in advance, you must provide a test method that will help us tell the administrator what is going wrong.

So, your server will be serving two different web applications: one for WebLab-Deusto (which you can even limit by IP address or listen on a different port if you prefer), and one for the final users.

Interface specification

This section explains in detail each of the five functions explained above. You might see also examples in the section examples.

All the functions called from WebLab-Deusto provide a shared secret, which is essentially a username and password in HTTP Basic format. As explained in *Unmanaged server*, there are two configuration variables (`http_experiment_username` and `http_experiment_password`) that must be configured by the administrator. These two variables should never be sent to the user. But all the methods described below include the regular HTTP header such as:

```
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
```

For "weblab" and password "password" (which is "weblab:password" in base64). You are responsible of checking this in all the methods to ensure that nobody else from the Internet (if this API is publicly exposed) can access this information.

Additionally, there are two ways to call the functions: as a REST service, or as a set of files with extensions. The default version uses a regular REST service where the URLs will be something like the following:

```
GET /weblab/sessions/api
GET /weblab/sessions/test
POST /weblab/sessions/  (with contents in JSON)
GET /weblab/sessions/ace76a23-5ccc-45eb-a03c-54dd67b016a5/status
POST /weblab/sessions/ace76a23-5ccc-45eb-a03c-54dd67b016a5 (with contents in JSON)
```

If you are using a routing engine like you do in most modern web frameworks, this should be easy to manage. However, it might be easier for you to work with some `.php` or `.jsp` or `.asp` files if you prefer. In that case, we provide another version of the API which looks like:

```
GET /weblab/sessions/api.php
GET /weblab/sessions/test.php
POST /weblab/sessions/new.php
GET /weblab/sessions/status.php?session_id=ace76a23-5ccc-45eb-a03c-54dd67b016a5
POST /weblab/sessions/action.php?session_id=ace76a23-5ccc-45eb-a03c-54dd67b016a5
```

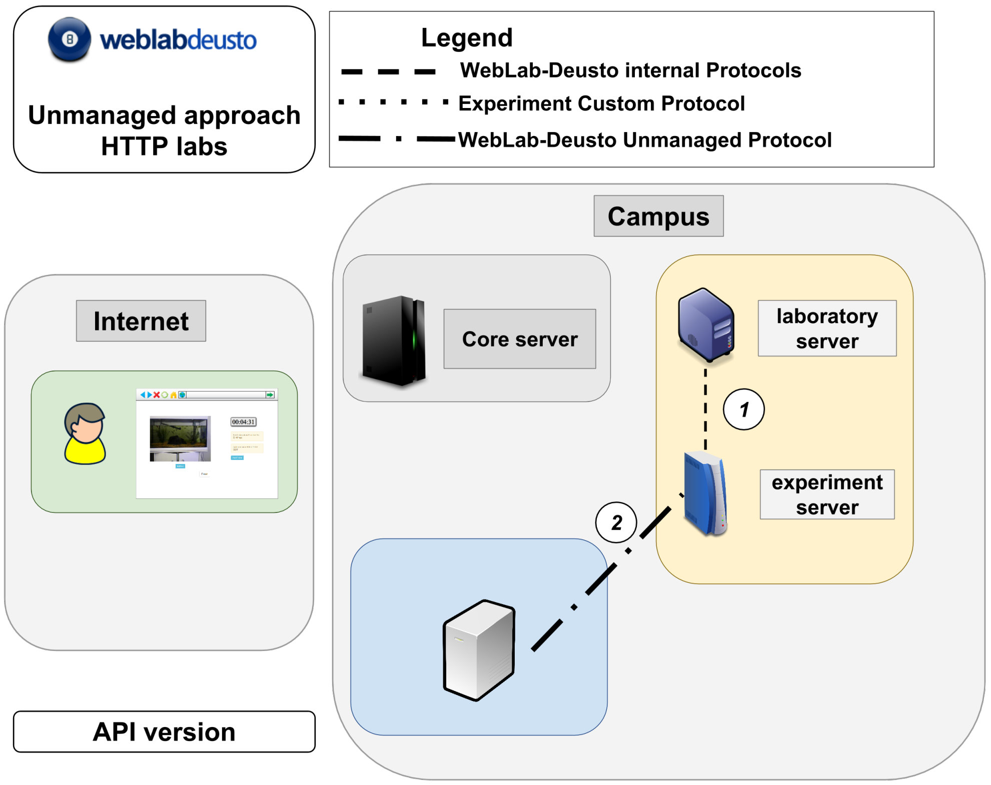
So as to activate this feature, you have to provide a `http_experiment_extension` configuration variable to `.php` or `.asp` or `.jsp` or something else. Whatever you put will be appended to `/weblab/sessions/api` or to `/weblab/sessions/new`.

Finally, we rely in JSON for the communications, so basically when we call `/weblab/sessions/` with a `POST` to perform a new request, we submit a JSON message by default. However, sometimes this is difficult for some developers in some web frameworks. If this is your case, and you prefer that we submit you the data like in a regular form, you may change the `http_experiment_request_format` configuration variable so it is set to `form`. If you prefer a different format (such as XML or whatever), feel free to contact us (*Contact*) and we can add it.

Function 1: Get API version

This method lets WebLab-Deusto to know what is the set of methods that is available and in what format. Basically in the future we might add new features to this API, which would make this API incompatible. So as to keep backwards compatibility, we provide this method. As long as you provide your version here, we will be able to guarantee you that you will be safe. The API version of the current documentation is `1`.

In this particular function, it will be the laboratory server the one contacting your server in a single step 1:



This is the only method where Authorization is not required:

```
GET /weblab/sessions/api HTTP/1.0
[...]
```

The expected response is:

```
HTTP/1.0 200 OK
Content-type: application/json
[...]

{
    "api_version": "1",
}
```

If the `http_experiment_extension` is set, for example, to `.php`, the request will be:

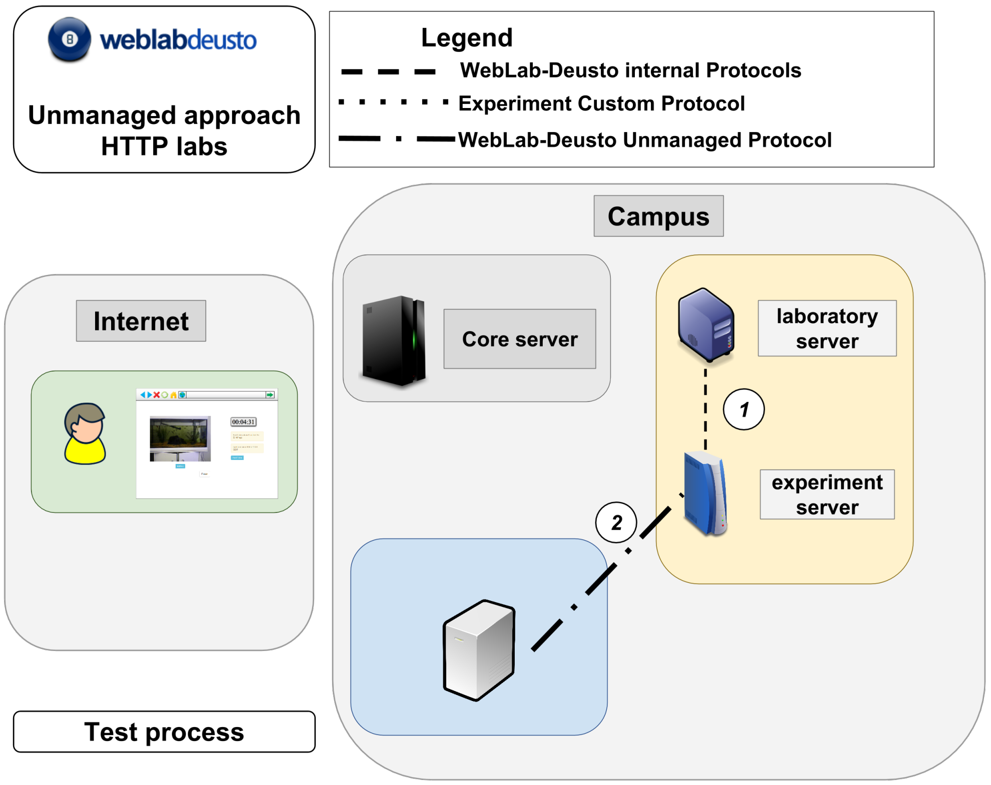
```
GET /weblab/sessions/api HTTP/1.0
[...]
```

And the response will be the same.

Note: Take into account that the version is a string (and not an int), as it happens in the managed API.

Function 2: Test connection

This method checks whether the connection is valid or not, according to the provided parameters (e.g., if the username and password are correct). As in the previous case, it is started directly by the Laboratory server:



In this case, the `Authorization` header will be provided, and you are responsible of checking it:

```
GET /weblab/sessions/test HTTP/1.0
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
[...]
```

The expected response is the following if correct:

```
HTTP/1.0 200 OK
Content-type: application/json
[...]

{
    "valid": true,
}
```

Or the following if not correct:

```
HTTP/1.0 200 OK
Content-type: application/json
[...]

{
    "valid": false,
    "error_messages": ["Invalid credentials"]
}
```

If the `http_experiment_extension` is set, for example, to `.php`, the request will be:

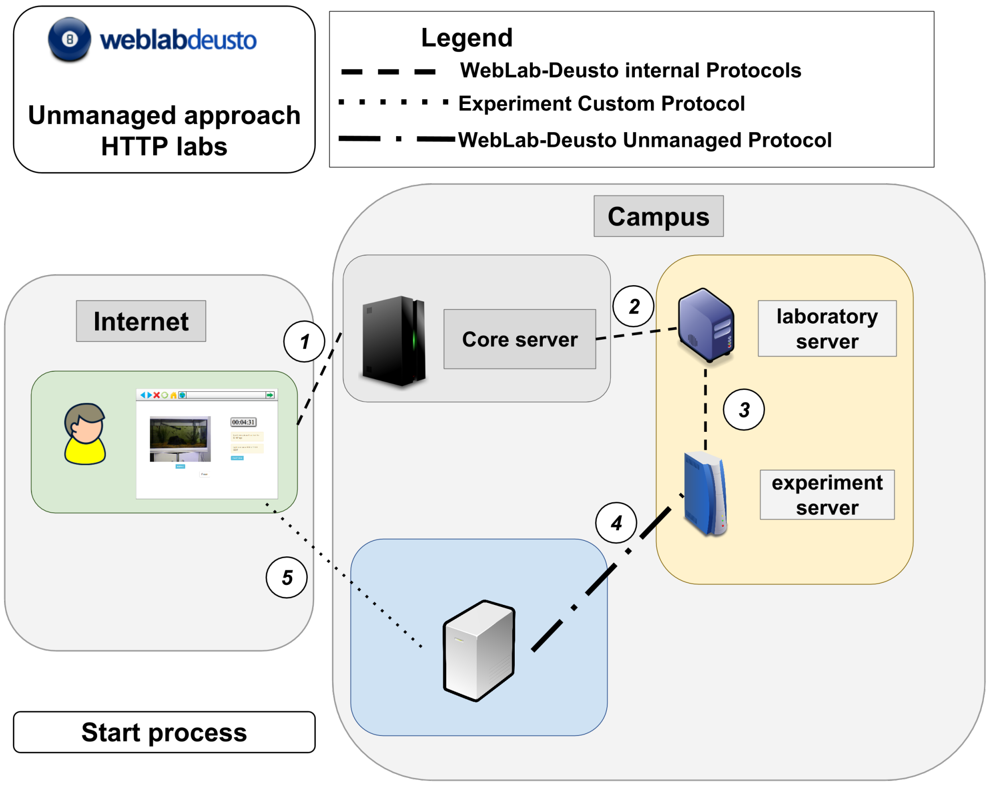
```
GET /weblab/sessions/test.php HTTP/1.0
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
[...]
```

And the response will be the same.

Note: The `error_messages` is a list, just in case there are multiple reasons why the connection is failing. And it is a list of human-readable messages to show the administrator when something goes wrong.

Function 3: Start

As mentioned, this method notifies the server to let a new user access the laboratory. In the following diagram:



The steps described in the diagram are the following:

1. The user will contact WebLab-Deusto requesting a reservation. If there was somebody already using the system, the WebLab-Deusto client will be contacting the server and showing that the user is in a queue.
2. Whenever the user can access the laboratory, WebLab-Deusto will initialize the session contacting the Laboratory server.
3. The Laboratory server will then contact the Experiment Server, which is provided by WebLab-Deusto in this case. It is a wrapper that wraps the requests to WebLab-Deusto to your server using the HTTP interface.
4. The Experiment server will contact your server calling the start function, as defined below. You are expected to provide a URL and let that student access with that URL, as well as a session identifier so the Experiment Server can contact your server for that session.

5. All the layers will return that URL to the user, so the user will automatically be redirected to that URL. In this step, the user will go to that URL directly.

So, in this method, an HTTP request is done to your server (step 4). The request is the following:

```
POST /weblab/sessions/ HTTP/1.0
Content-Type: application/json
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
[...]

{
    "back": "http://.../",
    "client_initial_data": {
    },
    "server_initial_data": {
        "request.locale": "es",
        "request.username": "porduna",
        "request.full_name": "porduna",
        "request.experiment_id.category_name": "Aquatic experiments",
        "request.experiment_id.experiment_name": "aquariumg",
        "priority.queue.slot.length": 148
    }
}
```

The parameters are: * `back`: indicating the URL to which the user is expected to be redirected

after. So, whenever the user session is finished, you should redirect the user to that URL.

- `client_initial_data`: a JSON-serialized document with the information sent by the user interface.
- `server_initial_data`: a JSON-serialized document with the information sent by the WebLab-Deusto server. It includes:
  - `request.locale`: language used by the client
  - `request.username`: login of the student
  - **`request.full_name`: full name of the student (at this point, it's still** the username)
  - `request.experiment_id.category_name`: category of the experiment
  - `request.experiment_id.experiment_name`: experiment name
  - `priority.queue.slot.length`: time in seconds for the particular user
  - `priority.queue.slot.start`: since when counting this time
  - **`priority.queue.slot.initialization_in_accounting`: whether the** initialization is counted or not in that time

If the `http_experiment_extension` is set, for example, to `.php`, and the `http_experiment_request_format` is set to `form` the request will be:

```
POST /weblab/sessions/new.php HTTP/1.0
Content-Type: multipart/form-data
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
[...]

client_initial_data=%7B%7D&server_initial_data=%7B%22request.experiment_id.experiment_
↪name%22%3A+%22aquariumg%22%2C+%22request.experiment_id.category_name%22%3A+
↪%22Aquatic+experiments%22%2C+%22priority.queue.slot.length%22%3A+148%2C+%22request.
↪username%22%3A+%22porduna%22%2C+%22request.full_name%22%3A+%22porduna%22%2C+
↪%22request.locale%22%3A+%22es%22%7D&back=http%3A%2F%2F...%2F
```

Which essentially is the quoted version of:

- `back`: the URL in string format
- `client_initial_data`: the dictionary above in JSON format (encoded). So in PHP for example you may access by using `json_decode($_POST['client_initial_data'], true)`
- `server_initial_data`: the dictionary above in JSON format (encoded). So in PHP for example you may access by using `json_decode($_POST['server_initial_data'], true)`

The expected response is the following:

```
HTTP/1.0 200 OK
Content-Type: application/json
[...]

{
    "session_id": "ace76a23-5ccc-45eb-a03c-54dd67b016a5",
    "url": "http://myserver.com/lab/?token=ace76a23-5ccc-45eb-a03c-54dd67b016a5
}
```

The returned `url` is where the user will be redirected to. The `session_id` will be used by the rest of the methods to identify this user. For example, for notifying you that this user should be kicked out, WebLab-Deusto will use that `session_id`.

Note: When creating such URL, you can use something like:

http://myserver/mylab?token=0ff5345e-c2d7-4e1e-84c1-54df43de60f5

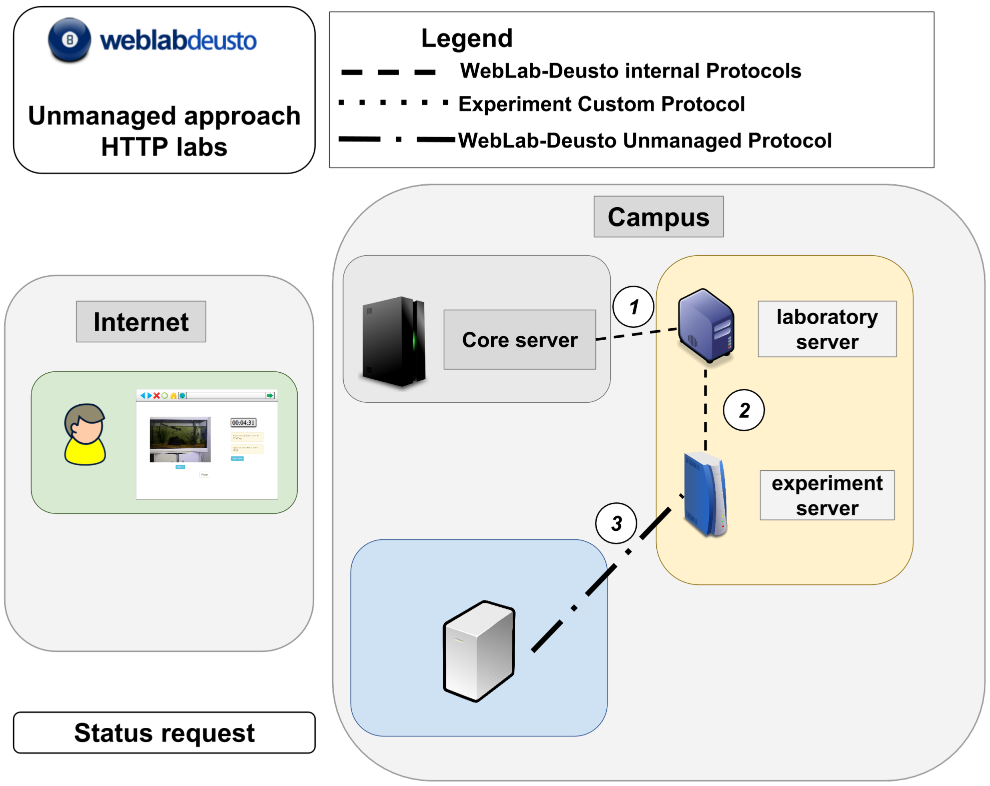
However, ideally you should pass it with the # so as to avoid the token to be logged in all the proxies and similar, and ideally it should be removed just after. For example, if you provide this link:

http://myserver/mylab#token=0ff5345e-c2d7-4e1e-84c1-54df43de60f5

And internally when accessing, the client in JavaScript takes the `location.hash`, uses that token and changes the `location.hash`, it would remove certain security problems. Ideally, you should also use HTTPS instead of HTTP.

Function 4: Status

So as to know that if the user is still using the laboratory or not, WebLab-Deusto will periodically call this function. As described in the diagram:



1. The WebLab-Deusto core server will call the Laboratory Server to see if the laboratory is still in use or not.
2. The Laboratory server will ask the Experiment server.
3. The Experiment server will ask your server to verify this.

Therefore, the user is not involved at any point. It is your responsability to use a proper mechanism to know if you user is still using the laboratory. You can simply write a JavaScript code that calls a dummy service every 20 seconds and if it has not been called in 40 seconds, then you report that he's not using the laboratory anymore.

The HTTP method in particular is:

```
GET /weblab/sessions/ace76a23-5ccc-45eb-a03c-54dd67b016a5/status HTTP/1.0
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
[...]
```

Where `ace76a23-5ccc-45eb-a03c-54dd67b016a5` is the `session_id` provided in the start method.

If the `http_experiment_extension` is set, for example, to `.php`, the request will be:

```
GET /weblab/sessions/status.php?session_id=ace76a23-5ccc-45eb-a03c-54dd67b016a5 HTTP/
↪1.0
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
[...]
```

The expected response is:

```
HTTP/1.0 200 OK
Content-type: application/json
[...]

{
```

(continues on next page)

(continued from previous page)

```
    "should_finish": 10,
}
```

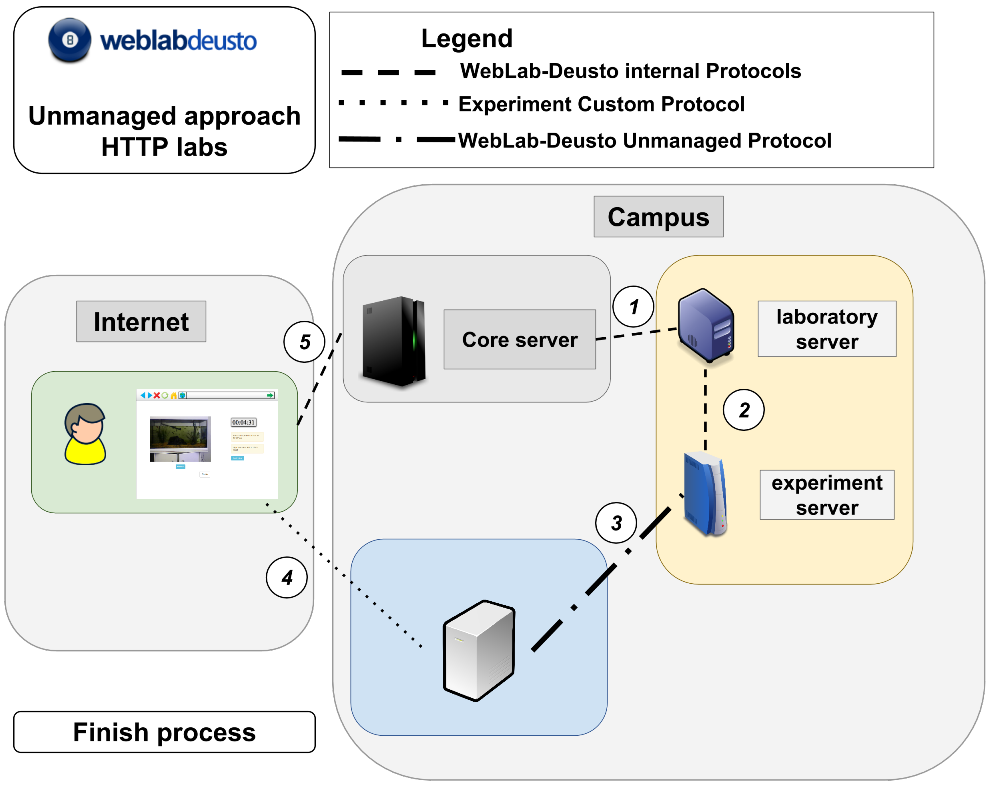
The value of `should_finish` is an integer. It represents the following:

- If it is -1, it means that the user must be kicked out.
- If it is 0, it means that WebLab-Deusto should not contact the server again for this session and wait until the time expires.
- If it is over 0, it means that WebLab-Deusto should contact again after that number of seconds. For example, it may return 10 so it calls again in 10 seconds. If the second time it returns 30, then the third call will call it 30 seconds later.

Note: You may use JavaScript to be notified that the user has closed the window. This is a good approach so you know as soon as possible that the user has left. However, don't rely uniquely on this approach, since if the user's computer shuts down, suspends, gets disconnected, that event will not be sent. So relying on two mechanisms (e.g., storing what was the last action while sending periodically an event + JavaScript) makes the overall system more efficient.

Function 5: Stop

Finally, WebLab-Deusto will call the stop function whenever the user should be kicked out. As seen on the diagram:



This is usually triggered by the Core Server. The steps are the following:

1. The WebLab-Deusto Core server notifies the Laboratory server that it should finish.
2. The Laboratory Server notifies this to the Experiment server.

3. The Experiment Server notifies this to your server.
4. Whenever the user performs a new request to your server, you must notify him that the session is over. He should be redirected whenever you consider to the `back` URL provided in the start function.

The HTTP request is the following:

```
POST /weblab/sessions/ace76a23-5ccc-45eb-a03c-54dd67b016a5 HTTP/1.0
Content-Type: multipart/form-data
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
[...]

{
    "action": "delete",
}
```

If the `http_experiment_extension` is set, for example, to `.php`, and the `http_experiment_request_format` is set to `form` the request will be:

```
POST /weblab/sessions/action.php?session_id=ace76a23-5ccc-45eb-a03c-54dd67b016a5 HTTP/
↪1.0
Content-Type: application/json
Authorization: Basic d2VibGFiOnBhc3N3b3Jk
[...]

action=delete
```

The expected HTTP response is the following. The simplest example would be:

```
HTTP/1.0 200 OK
Content-Type: application/json
[...]

{}
```

Another example would be:

```
HTTP/1.0 200 OK
Content-Type: application/json
[...]

{
    "finished": false,
    "ask_again": 10.0,
}
```

And, 10 seconds later:

```
HTTP/1.0 200 OK
Content-Type: application/json
[...]

{
    "finished": true,
    "data": "Result=10"
}
```

It may contain the following values:

- `finished`: in case it has not finished. By default, `true` is assumed. But if the resource disposal takes time, return `false` and the method will be called again, and return `true` whenever it is successfully cleaned.
- `data`: in case some data should be returned to the experiment client or logged.
- `ask_again`: if `finished` is `false`, you can provide a float that it's a number of seconds to be waited to be called again. If you return `"ask_again": 30.5`, it will call again in approximately 30.5 seconds.

Examples

We provide the following examples:

- *Flask (with a library)* (which uses weblablib, so it is more reliable)
- *Flask (simple)* (which shows how to implement a simplified version)
- *PHP (multiple files)* (which uses multiple files and regular forms)
- *PHP (single file)* (which uses a single file and the standard form)

Flask (with a library)

Go to http://developers.labsland.com/weblablib/ to see how to use and install weblablib. Then, check *Deployment*.

Flask (simple)

In the following URL you have a simplified version on how an unmanaged remote lab could be implemented from scratch. The target of this code is only to be shown as an example (so you could implement something similar with your framework). If you want to use Flask, we encourage you to use the library explained above. The code itself is available at:

- https://github.com/weblabdeusto/weblabdeusto/blob/master/experiments/unmanaged/http/python/flask/sample.py

Here we will only cover some parts of it. In the following snippet, we keep in two dictionaries in memory what are the current active sessions and expired sessions. You shouldn't do this, but rely on a session mechanism, Redis, SQL or similar. While using memory, make sure that you don't run the server in multiple processes (otherwise it will fail):

```
###############################
#
# Store in DATA dictionaries
# representing users.
#
DATA = {
}

##################################
#
# Store in EXPIRED_DATA, expired
# addresses pointing to their
# previous URLs
#
EXPIRED_DATA = {
}
```

In the laboratory code, we require the user to provide us a `session_id`. With it, we check in the dictionaries above and see whether the user exists (if it is in `DATA`), existed (if it is in `EXPIRED_DATA`, and will be redirected to a different URL), or never existed.

```
#####################################
#
# Main method. Authorized users
# come here directly, with a secret
# which is their identifier. This
# should be stored in a Redis or
# SQL database.
#
@app.route('/lab/<session_id>/')
def index(session_id):
    data = DATA.get(session_id, None)
    if data is None:
        back_url = EXPIRED_DATA.get(session_id, None)
        if back_url is None:
            return "Session identifier not found"
        else:
            return redirect(back_url)


    data['last_poll'] = datetime.datetime.now()
    return """<html>
    <head>
        <meta http-equiv="refresh" content="10">
    </head>
    <body>
        Hi %(username)s. You still have %(seconds)s seconds
    </body>
    </head>
    """ % dict(username=data['username'], seconds=(data['max_date'] - datetime.
↪datetime.now()).seconds)
```

We check using HTTP Basic to see if the user is valid or not by providing this auxiliar function:

```
def check_http_credentials(testing=False):
    auth = request.authorization
    if auth:
        username = auth.username
        password = auth.password
    else:
        username = password = "No credentials"

    weblab_username = app.config['WEBLAB_USERNAME']
    weblab_password = app.config['WEBLAB_PASSWORD']
    if username != weblab_username or password != weblab_password:
        if testing:
            return Response(json.dumps(dict(valid=False, error_messages=["Invalid␣
↪credentials"])), status=401, headers = {'WWW-Authenticate':'Basic realm="Login␣
↪Required"', 'Content-Type': 'application/json'})

        print("In theory this is weblab. However, it provided as credentials: {} : {}
↪".format(username, password))
        return Response(response=("You don't seem to be a WebLab-Instance"),␣
↪status=401, headers = {'WWW-Authenticate':'Basic realm="Login Required"'})
```

(continues on next page)

(continued from previous page)

```
    return None
```

Then, we provide the basic `api` and `test` functions:

```
@app.route("/foo/weblab/sessions/api")
def api_version():
    return jsonify(api_version="1")

@app.route("/foo/weblab/sessions/test")
def test():
    response = check_http_credentials(testing=True)
    if response is not None:
        return response
    return jsonify(valid=True)
```

And the start function, that stores data in `DATA` as new users come, and provide the link to the laboratory code:

```
@app.route("/foo/weblab/sessions/", methods=['POST'])
def start_experiment():
    response = check_http_credentials()
    if response is not None:
        return response

    # Parse it: it is a JSON file containing two fields:
    request_data = request.get_json(force=True)

    client_initial_data = request_data['client_initial_data']
    server_initial_data = request_data['server_initial_data']

    print server_initial_data

    # Parse the initial date + assigned time to know the maximum time
    start_date_str = server_initial_data['priority.queue.slot.start']
    start_date_str, microseconds = start_date_str.split('.')
    start_date = datetime.datetime.strptime(start_date_str, "%Y-%m-%d %H:%M:%S") +␣
↪datetime.timedelta(microseconds = int(microseconds))
    max_date = start_date + datetime.timedelta(seconds = float(server_initial_data[
↪'priority.queue.slot.length']))

    # Create a global session
    session_id = str(random.randint(0, 10e8)) # Not especially secure 0:-)
    DATA[session_id] = {
        'username'  : server_initial_data['request.username'],
        'max_date'  : max_date,
        'last_poll' : datetime.datetime.now(),
        'back'      : request_data['back']
    }

    link = url_for('index', session_id=session_id, _external = True)
    print "Assigned session_id: %s" % session_id
    print "See:",link
    return jsonify(url=link, session_id=session_id)
```

In the `status` function we tell WebLab what is the status taking into account the information from the sessions:

```
@app.route('/foo/weblab/sessions/<session_id>/status')
def status(session_id):
```

(continues on next page)

(continued from previous page)

```
    response = check_http_credentials()
    if response is not None:
        return response

    data = DATA.get(session_id, None)
    if data is not None:
        print "Did not poll in", datetime.datetime.now() - data['last_poll'], "seconds
↪"
        print "User %s still has %s seconds" % (data['username'], (data['max_date'] -␣
↪datetime.datetime.now()).seconds)
        if (datetime.datetime.now() - data['last_poll']).seconds > 30:
            print "Kick out the user, please"
            return jsonify(should_finish=-1)

    print "Ask in 10 seconds..."
    #
    # If the user is considered expired here, we can return -1 instead of 10.
    # The WebLab-Deusto scheduler will mark it as finished and will reassign
    # other user.
    #
    return jsonify(should_finish=10)
```

And in the last one we end the session and move it to `EXPIRED_DATA`:

```
@app.route('/foo/weblab/sessions/<session_id>', methods=['POST'])
def dispose_experiment(session_id):
    response = check_http_credentials()
    if response is not None:
        return response

    request_data = request.get_json(force=True)
    if 'action' in request_data and request_data['action'] == 'delete':
        if session_id in DATA:
            data = DATA.pop(session_id, None)
            if data is not None:
                EXPIRED_DATA[session_id] = data['back']
            return jsonify(message='Deleted')
        return jsonify(message='Not found')
    return jsonify(message='Unknown action')
```

Note: `EXPIRED_DATA` will never be cleaned until you restart that server.

So as to test it, you might start deploying it and checking how it works. Jump to *Deployment*.

PHP (multiple files)

In:

- https://github.com/weblabdeusto/weblabdeusto/tree/master/experiments/unmanaged/http/php

You have the PHP code sample for the unmanaged laboratories. You should copy all that code to `/var/www/html/phplabs`. Then, you should create the database, as:

```
$ mysql -uroot -p
mysql> CREATE DATABASE phplab DEFAULT CHARSET 'utf8';
Query OK, 1 row affected (0.00 sec)
mysql> GRANT ALL ON phplab.* TO 'phplab'@'localhost' IDENTIFIED BY 'phplab';
Query OK, 0 rows affected (0.00 sec)
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)

$ mysql -uphplab -p phplab < initial.sql
```

Once created, you can use the following configuration (more info on *Remote laboratory deployment*):

```
http_experiment_url: http://localhost/phplab/multifile
http_experiment_username: admin
http_experiment_password: password
http_experiment_request_format: form
http_experiment_extension: .php
```

So it will not submit JSON messages but regular form messages, to files called `new.php`, `action.php`, etc. as shown in the code.

PHP (single file)

In:

- https://github.com/weblabdeusto/weblabdeusto/tree/master/experiments/unmanaged/http/php

You have the PHP code sample for the unmanaged laboratories. You should copy all that code to `/var/www/html/phplabs`. Then, you should create the database, as:

```
$ mysql -uroot -p
mysql> CREATE DATABASE phplab DEFAULT CHARSET 'utf8';
Query OK, 1 row affected (0.00 sec)
mysql> GRANT ALL ON phplab.* TO 'phplab'@'localhost' IDENTIFIED BY 'phplab';
Query OK, 0 rows affected (0.00 sec)
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)

$ mysql -uphplab -p phplab < initial.sql
```

Once created, you can use the following configuration (more info on *Remote laboratory deployment*):

```
http_experiment_url: http://localhost/phplab/weblab.php
http_experiment_username: admin
http_experiment_password: password
```

Internally, you can see how `weblab.php` implements the defined calls, and how the user will be redirected to `index.php`.

Deployment

So as to see how to deploy this type of lab, go to *Remote laboratory deployment*.

LabVIEW Remote Panels

Note: This approach is experimental. Don't hesitate to *Contact* for further and updated information.

The disadvantage is that it has all the disadvantages of the LabVIEW remote panels: it does not work in most web browsers, requires several ports to be open, etc. However, it is the easiest approach to implement a new remote laboratory for LabVIEW developers nowadays.

If you want to try this approach, take the `.vi` files from:

> https://github.com/weblabdeusto/weblabdeusto/tree/master/experiments/unmanaged/labview

And in the the deployment section (*Remote laboratory deployment*; in particular *Unmanaged server*), use the same steps, except for using `experiments.labview_remote_panels.LabviewRemotePanels` instead of `experiments.http_experiment.HttpExperiment`. The following options should be also added in that segment:

```
electronics:
  class: experiments.labview_remote_panels.LabviewRemotePanels
  type: experiment
  config:
     labview_host: 127.0.0.1
     labview_port: 20000
     labview_url: http://mylabviewserver.myinstitution.edu:8080/index.html
     labview_shared_secret: 31e6b0b6-757d-4331-b731-27aeb8f8f04d
```

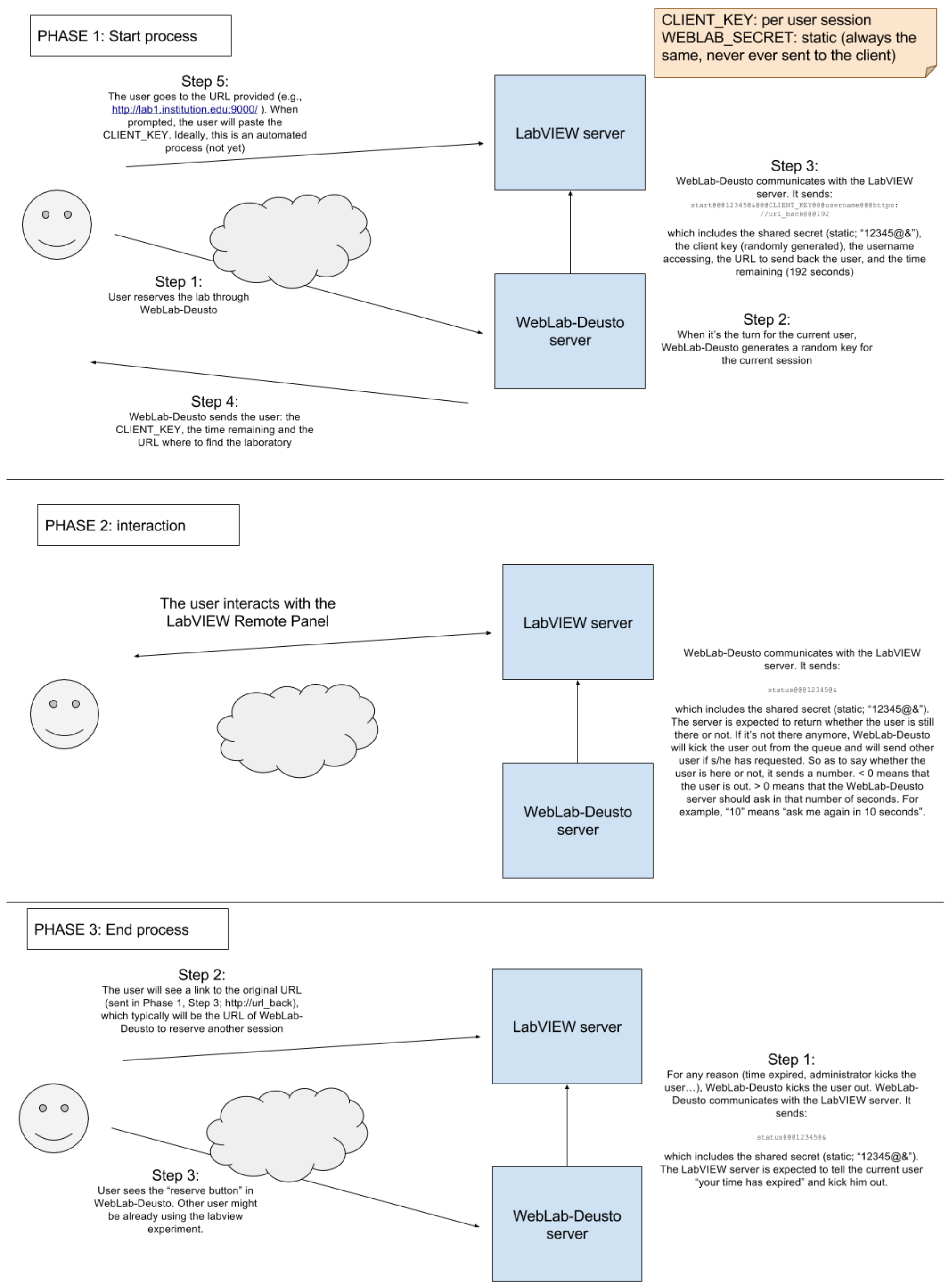
Additionally, you can set debugging information by adding:

```
electronics:
  class: experiments.labview_remote_panels.LabviewRemotePanels
  type: experiment
  config:
     labview_host: 127.0.0.1
     labview_port: 20000
     labview_url: http://mylabviewserver.myinstitution.edu:8080/index.html
     labview_shared_secret: 31e6b0b6-757d-4331-b731-27aeb8f8f04d
     labview_debug_message: true
     labview_debug_command: true
```

The meaning of the values is:

- `labview_host`: the IP address of the LabVIEW server. It can be in a different machine, so you must put what's the IP address.
- `labview_port`: the port of the WebLab-Deusto LabVIEW server. In the `.vi` you will see that you can configure it. Note that it's not the port of the Remote Panel (i.e., the public port that users will use), but only the one that accepts connections from WebLab-Deusto.
- `labview_shared_secret`: a secret that you will also configure in the `.vi` file. The key is that it will tell LabVIEW that the message comes from WebLab-Deusto (and not from somewhere else).
- `labview_url`: the URL of the remote panel. The LabVIEW remote panel must be publicly available.

Whenever a user comes to WebLab-Deusto, a session identifier will randomly be generated and sent to the LabVIEW server. Then, it will be shown to the user in the client, with a button to go to the `labview_url`. Once there, the LabVIEW code (your code) will have make sure that the user is valid if he provides such code. From that moment, the user will use the Remote Panel as usual, and different triggers in the `.vi` code will tell you that you should tell the user to finish (e.g., when a different user comes).



CLIENT_KEY: per user session
WEBLAB_SECRET: static (always the same, never ever sent to the client)

Step 5:
The user goes to the URL provided (e.g., http://lab1.institution.edu:9000/). When prompted, the user will paste the CLIENT_KEY. Ideally, this is an automated process (not yet)

Step 3:
WebLab-Deusto communicates with the LabVIEW server. It sends:

```
start@@@12345@&@@@CLIENT_KEY@@@username@@@https:
//url_back@@@192
```

which includes the shared secret (static; "12345@&"), the client key (randomly generated), the username accessing, the URL to send back the user, and the time remaining (192 seconds)

Step 1:
User reserves the lab through WebLab-Deusto

Step 2:
When it's the turn for the current user, WebLab-Deusto generates a random key for the current session

Step 4:
WebLab-Deusto sends the user: the CLIENT_KEY, the time remaining and the URL where to find the laboratory

The user interacts with the LabVIEW Remote Panel

WebLab-Deusto communicates with the LabVIEW server. It sends:

```
status@@@12345@&
```

which includes the shared secret (static; "12345@&"). The server is expected to return whether the user is still there or not. If it's not there anymore, WebLab-Deusto will kick the user out from the queue and will send other user if s/he has requested. So as to say whether the user is here or not, it sends a number. < 0 means that the user is out. > 0 means that the WebLab-Deusto server should ask in that number of seconds. For example, "10" means "ask me again in 10 seconds".

Step 2:
The user will see a link to the original URL (sent in Phase 1, Step 3; http://url_back), which typically will be the URL of WebLab-Deusto to reserve another session

Step 1:
For any reason (time expired, administrator kicks the user…), WebLab-Deusto kicks the user out. WebLab-Deusto communicates with the LabVIEW server. It sends:

```
status@@@12345@&
```

which includes the shared secret (static; "12345@&"). The LabVIEW server is expected to tell the current user "your time has expired" and kick him out.

Step 3:
User sees the "reserve button" in WebLab-Deusto. Other user might be already using the labview experiment.

Virtual machines

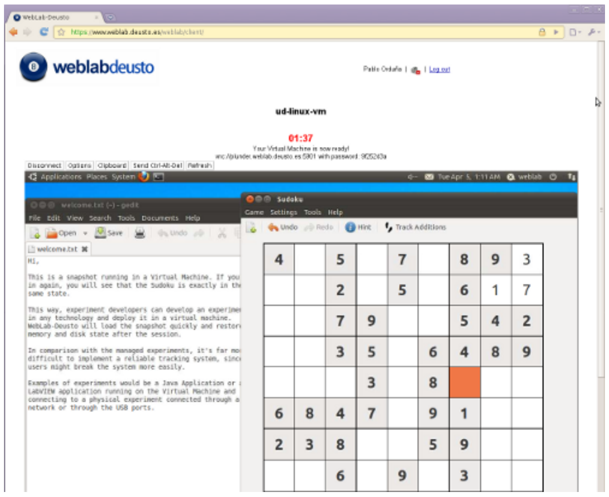
Note: The support for remote laboratories based on virtual machines is limited. While it works, the flexibility provided by virtual machines is in general not enough for most remote laboratories.

Virtual Machines based remote labs

Introduction

WebLab-Deusto supports two kinds of experiments: **Managed** and **Unmanaged** experiments.

Managed experiments are fully integrated with WebLab-Deusto and, generally, developed specifically for it through the use of its provided API. Data during the experiment tends to be transmitted through WebLab, which means logging and accountability is both easy and accurate. They are the most frequent kind of experiment, and, when possible, it is the most advisable type of experiment to use. However, it is not always possible or convenient to have such integration. Sometimes, an experiment developer might be unable or unwilling to adequate his experiment, due to the nature of the experiment itself or the time it would require to do so. For these cases, WebLab-Deusto provides Virtual Machine based Unmanaged experiments.



ud-linux-vm

01:37

Your Virtual Machine is now ready!
vnc://plunder.weblab.deusto.es:5901 with password: 9f2524a

```
Hi,

This is a snapshot running in a Virtual Machine. If you
in again, you will see that the Sudoku is exactly in the
same state.

This way, experiment developers can develop an experimen
in any technology and deploy it in a virtual machine.
WebLab-Deusto will load the snapshot quickly and restor
memory and disk state after the session.

In comparison with the managed experiments, it's far mo
difficult to implement a reliable tracking system, sinc
users might break the system more easily.

Examples of experiments would be a Java Application or
LabVIEW application running on the Virtual Machine and
connecting to a physical experiment connected through a
network or through the USB ports.
```

4		5		7		8	9	3
		2		5		6	1	7
		7	9			5	4	2
		3	5		6	4	8	9
			3		8	🟧		
6	8	4	7		9	1		
2	3	8			5	9		
		6		9		3		

Virtual Machine Experiment

Virtual Machine experiments work differently than traditional Managed experiments. A Virtual Machine experiment does not include any WebLab-specific code. Instead, the experiment is developed anyhow and deployed on a Virtual Machine image. WebLab-Deusto then manages that Virtual Machine. WebLab-Deusto users will, upon experiment startup through the standard WebLab-Deusto web, receive a password. Then, they may use that password to connect to the Virtual Machine through either Remote Desktop or VNC. From then on, the user can freely use the Virtual Machine and the experiment deployed on it. Once the time has run out, WebLab-Deusto will close the Virtual Machine and restore it to a pre-defined snapshot before the next use.

Supported Virtual Machine, OSes, and Protocols

Currently WebLab-Deusto VM system has been tested on Linux, under which the VNC protocol through the TightVNC server (and compatible clients) is supported. It has also been tested on Windows, under which both UltraVNC and Remote Desktop are supported.

The Virtual Machine software currently supported is Virtual Box.

It is noteworthy that this list is likely to be extended in the future, and that the system is easily extensible, so it would not be particularly hard for a developer to add support to new VM software or new Protocols.

WebLab-Deusto VM software

For the password of a Virtual Machine remote managing protocol to be changed upon WebLab-Deusto's request, it is necessary to install certain software on the Virtual Machine snapshot. Currently, WebLab-Deusto provides a Vino password changer for Linux and a Remote Desktop and UltraVNC password changer for Windows.

Safety

The power this system gives to the user is significant, as it grants potentially full access to a Virtual Machine. However, WebLab-Deusto uses the snapshot system provided by most Virtual Machine software to reset the machine to a predefined state before every use. Thus, though during the available time the user is free to use the hard disk or even break the system, no changes will be permanent. Hence, from that viewpoint, these experiments are completely safe.

Nonetheless, as usual, the experiment developer will need to be aware not to add experiment-specific risks and vulnerabilities.

It is also noteworthy that in the case of an unmanaged experiment, this is slightly more important. Though the network traffic of a managed experiment goes through the WebLab server and may hence be easily logged, users connect to the Virtual Machine directly. Consequently, though there is still a certain degree of logging and accountability through the standard WebLab login, reservation and experiment start-up mechaniams, obtaining information about the exact usage of the experiment is not possible anymore.

Creating a new VM based experiment

In this section we will create and deploy a new VM based experiment.

Though the process is rather long and it will be described in detail, it is advisable to read everything carefully, and to carry out and check each step before following to the next.

Our goal here will be to deploy a new VM experiment, with the following characteristics:

- Will use VirtualBox as a VM engine.
- The VM will run a Windows OS.
- Users will access the VM through RDP (Remote Desktop Protocol).

Note: WebLab-Deusto supports several VM OSs (Windows, Linux-likes, etc.) and a few access protocols (VNC, RDP). Setting up a VM with those other OSes and protocols is beyond the scope of this guide. However, the process would be very similar and only a few changes would be needed.

Warning: To make sure you are always on the right track, we are providing a **checklist** at the start and/or end of some sections. Upon encountering such a checklist, please **check carefully every point** before going on.

Prerrequisites

WebLab-Deusto instance

Before being able to create a new VM based experiment, and before being able to start following this guide, you will need to have a *working instance* of WebLab-Deusto.

If you do not have a working instance yet, you can find out how to create it in first steps.

Warning: CHECKLIST *(Ensure the following before skipping to the next section)*

1. I have a WebLab-Deusto instance.
2. My WebLab-Deusto instance is successfully deployed.
3. I have tested at least one experiment in my WebLab-Deusto instance, and it is working fine.

VirtualBox

Oracle VM VirtualBox is a virtualization engine. It will be the engine under which the machine with our experiment will be run.

You may download the VirtualBox software package from the virtualbox downloads website, and install it normally.

Once installed, some further actions are required.

In order for WebLab-Deusto to be able to properly interact with VirtualBox, certain utilities that come with VirtualBox need to be accessible from the command line. To do this:

1. Locate the VirtualBox installation folder. Often, this will be *c:\Program Files\Oracle\VirtualBox* or similar. Go to that location through the windows file explorer, and make sure the VirtualBox files are there. Copy that exact path to that folder to your clipboard (through ctrl+c).
2. We will now need to add that folder to our windows PATH environment variable. To do this under Windows 7, open the *system properties* dialog. Go to *advanced settings* and then to *environment variables*. Among *system variables* you will find a variable named **PATH**. Modify it, and append the VirtualBox path. Make sure a semicolon separates it from the last path in the variable.

Warning: CHECKLIST *(Ensure the following before skipping to the next section)*

1. I have successfully installed *Oracle VirtualBox*
2. VBoxManage is accessible. To check this: Open a Windows console terminal. Type *vboxmanage -v* and hit enter. If it **is** accessible, a version number should appear (such as *4.1.12*). If an error occurs, then it **is not** accessible, and the previous steps should be redone.

Virtualized Windows machine

We now have *Oracle VirtualBox* installed. However, we do not really have a *Virtual Machine* yet. We will create one now. In order to do this, we will require a copy of any version of Windows with RDP support. Windows XP is recommended, though later versions of Windows should also work.

Note: Make sure that the Windows version you want to install supports the Remote Desktop server. Users will connect to the Virtual Machine through Remote Desktop, so this is particularly important. *Professional* and higher versions support the Remote Desktop server, but certain lower versions as well. If in doubt, check your specific version. You can check through the official Microsoft website, or by checking whether the *enable remote access* option exists in your *system properties*.

We can have our copy either in CD/DVD form, or in .ISO image form (other image formats supported by VirtualBox are also fine).

Once we have it, create the VM by following these broad steps:

1. Start *Oracle VirtualBox*
2. Hit the *New* button on the toolbar. A wizard dialog should pop up.
3. Go on in the dialog. Eventually, you will be asked to write a name for your VM. Give whichever name you want. This name will identify the VM, and we will later refer to it. In this guide, we will refer to is as the *VM name*. Choose also the right settings for the *Operative System* and *Version* fields. The exact values will depend upon the version of Windows you wish to install.
4. Go on. The next screens should be rather straightforward. Make sure to give enough RAM to your Virtual Machine (at least 512 MB is probably advisable, though it depends on the version you are installing, on the experiment you want to place on it, and on the real machine you will be running the Virtual Machine from). Make sure to give it enough Hard Disk space as well. Depending again on the circumstances, a good minimum would probably be 10gb-20gb.
5. Eventually the wizard will let you select the installation media. Depending on whether you want to install Windows from your CD/DVD drive, or from a .ISO image, you will need to configure it appropriately.
6. After setting the right installation media and proceeding, the Virtual Machine should start. If it doesn't, start it manually (Virtual Machines appear in VirtualBox on the left. Yours should appear there, with the *VM name* you chose).
7. If the VM starts, and after a while the Windows Setup appears, then congratulations, you are on the right track. If nothing happens, or if the VM starts but no installation media is found, then check the previous steps (particularly, make sure you configured the installation media right, and that your CD or ISO image is right).
8. Install Windows normally.
9. Apart from whichever administrator account you create, create a second admin account called *weblab*. Naming it *weblab* is important.
10. Once Windows is installed, make sure the Internet can be accessed from the Virtual Machine.

Note: The *weblab* account we created in previous steps could actually be named differently. But then, additional configuration changes would be required in the In-VM Manager (which we will install in later sections of this guide), and for simplicity, these won't be covered here.

Congratulations. If everything went ok, you now have a virtual windows machine on your VirtualBox.

Warning: CHECKLIST *(Ensure the following before skipping to the next section)*

1. My Windows VM appears in VirtualBox. (Generally, on the left).
2. My Windows VM can be started through VirtualBox, and the virtualized Windows seems to work fine.
3. I can access the Internet from the virtualized Windows.
4. My virtualized Windows supports the Remote Desktop server. You can check whether the *enable remote access* option exists in your *system properties*. (If you check this way, enable remote access now, and you will save a step for later).

Installing the WebLab In-VM Manager

Warning: CHECKLIST *(Before proceeding to this section, please check the following. Feel free to skip those checks you have done already.)*

1. I have a working, Windows VM which uses VirtualBox as its engine.
2. My Windows VM supports Remote Desktop server.
3. I can access the Internet from my Windows VM.
4. My virtualized Windows supports the Remote Desktop server. You can check whether the *enable remote access* option exists in your *system properties*. (If you check this way, enable remote access now, and you will save a step for later).
5. The terminal command VBoxManage is accessible. To check this: Open a Windows console terminal. Type *vboxmanage -v* and hit enter. If it **is** accessible, a version number should appear (such as *4.1.12*). If an error occurs, then it **is not** accessible, and the previous steps should be redone.

What is the Manager?

Users will access the virtualized Windows machine through the RDP protocol (that is, Windows' Remote Desktop). So that only one user (the one who has a reservation) can access the machine at a given time, a different, unique, random password will be provided for each session.

This means that somehow, something will need to change the password of the virtualized Windows each session.

That is the mission of the WebLab In-VM Manager.

The WebLab In-VM Manager is a service which will run within the virtualized Windows, and its main purpose will be to receive password change requests from WebLab.

Because as of now, the VM you have created does not yet have such a service, we will need to install it.

Manager Prerrequisites

.NET Framework 3.5

The In-VM Manager requires the Microsoft .NET Framework version 3.5. The In-VM Manager is meant to run within the Windows VM, so it is that machine, and not your physical, host machine, which needs to have it installed.

You may download Microsoft .NET Framework 3.5 from the official Microsoft website. It is advisable that you download it from the Windows VM itself. Once downloaded, install it.

Some versions of Windows may come with .NET Framework 3.5 pre-installed. That is, however, likely not the case.

Making the VM accessible

Configuring the network

The VM needs to be accessible from the host machine through an IP address, so the VM network settings will need to be configured properly.

Especifically, the host machine will need to connect to two ports on the windows Virtual Machine:

- Remote Desktop port (3389). The port end users will connect to. VM will need to accept connections to it from the Internet.

- In-VM Manager port (6789). The WebLab-Deusto server will connect to it and command a password change when needed.

Warning: RDP port needs to be accessible from the Internet. Otherwise, end-users will not be able to connect to the machine. The VM Manager port, however, **should only** be accessible from the host machine. Otherwise, an attacker could change the password of the VM at will. Note that, however, the security risk isn't high. An attacker could gain temporary control over the VM (which will last until the next experiment session begins, and the VM is reset). However, the host system itself would not be compromised.

To open the *network settings* dialog:

1. Go to VirtualBox administrator dialog (the one with the VM list on the left)
2. Right click on the windows VM
3. Go to *settings*
4. When the *settings* dialog appears, go to *network*

There are essentially two ways to configure the network:

1. **NAT**: The VM will connect to the Internet through the host machine's connection. In order for it to work, you would need to forward port 3389 and 6791 properly. That is not particularly hard, but isn't trivial either, so NAT **is not recommended**.
2. **Bridged Adapter**: The VM will connect to the Internet directly. This **is the recommended** way. The Windows VM will be given its own IP on your local network. If your local network doesn't support DHCP, further configuration may be needed. Note, however, that choosing this configuration means that the Guest and Host machines will communicate through your local network directly. If your local network is somehow restricted or filtered by a firewall, this may lead to issues (See the third note).

It is hence suggested that you choose *Bridged Adapter*.

Note: You might need to restart the VM before network configuration changes take effect.

Note: From this point, this guide will assume that you are indeed using a *Bridged Adapter* network.

Note: Choosing the Bridged Adapter configuration means that the communication between the Host and the Guest machine will be carried out through the local network. If your local network is restricted or filtered by a firewall, problems may arise. If you do indeed have a firewall, you will need to make sure that port 3389 (RDP) and port 6789 (communication between Guest and Host) are open. Port 3389 is easy to test, as you can assume that if RDP works, the port is open. Port 6789 is harder to test, and if it is being blocked, you will only notice later on this guide, when you carry out the suggested tests. If the firewall can't be turned off or configured, then you could also use **NAT** instead of **Bridged Adapter**. Though the concept is similar, using **NAT** is not fully covered in this guide. The only difference, however, should be that you would need to configure *Port Forwarding* within the Virtual Box configuration, so that you can access the required ports from the Host machine.

Checking the network config

If the network was properly configured, the virtualized Windows:

- Will still have Internet access
- Will have been assigned an IP in the local network

We will now find out which IP has been assigned to the VM.

There are several ways to do this. The easiest is (everything is done on the virtualized Windows):

- Open a terminal (a command line)
- Type *ipconfig*

You will see a list of every network adapter in your machine, along with its IP addresses. The adapter we seek is our standard *Local Network Ethernet Connection* (or a similar name). The IP we seek is the *IPv4 address*. Write out that IP address. From now on, we will refer to that IP as the *VM IP*.

Note: An example of a valid IP would be *192.168.1.105*, or any LAN IP. An example of an *invalid* IP would be *localhost* or *127.0.0.1*. Often, but not always, an IP that starts with *10* won't be valid either. If any of this happens, and further checks are unable to access the VM, then re-check your network settings.

We should now be able to access our VM through the VM IP.

Our first check will be the following:

1. Start a command line.
2. Type *ping <VM IP>* on it. Replace <VM IP> with your actual VM IP. For instance: *ping 192.168.1.105*. Hit enter.

If timeout errors appear, then the test failed. Your VM, for some reason, is not reachable through that IP. Check the previous steps. If, however, ping does send several packets, and certain times appear on the screen, then congratulations, your machine, for now, seems to be reachable.

We will now carry out yet another check. In your host machine (not your VM one) open the Windows Remote Desktop client. Try to connect to the VM IP. It should work. If it doesn't:

1. Check that the version of Windows that the VM is running supports the Remote Desktop server.
2. Check (in the VM) that remote access is enabled.
3. Check this section again and ensure that the network is configured properly.

Warning: CHECKLIST *(Ensure the following before skipping to the next section)*

1. My guest Windows (virtualized Windows) supports Microsoft .NET Framework 3.5
2. My guest Windows can be accessed through Remote Desktop from my host Windows.
3. The firewall on my local network should not prevent access to port 6789.

Installing the In-VM Manager itself

Deploying the binaries

Locate the In-VM Manager binaries. All WebLab distributions should include them. If %WEBLAB% is the WebLab folder, then the binaries we seek should be in: *%WEBLAB%\experiments\unmanaged\vm_services\WindowsVM\WindowsVMService\bin\Release*

Place those binaries into your guest Windows. For instance, you may place them into the *c:\vmservice* folder (create it, it won't exist).

Installing as a service

The Manager will run as a Windows service. To install it, you can't execute WindowsVMService.exe straightaway. Instead, you should execute the *sc_install_service.bat* script. If your VM is running a relatively modern version of Windows you should right click on the script and **run as administrator**.

Warning: If you do not run the script with administrator priviledges, it will be unable to install it properly.

If for any reason *sc_install_service.bat* failed, you may try with *install_service.bat*, though this is not recommended.

When those succeed, your service will be installed as a standard Windows service, and can be started and stopped as one. Alternatively, *sc_start_service* and *sc_stop_service* scripts are provided, but they probably will only work if the service was installed through *sc_install_service*.

Before going on, check whether your service is indeed installed.

Open the windows *Service Manager* by running *services.msc* (hit [WINKEY]+R to be able to run programs).

Among the services there, a new one, *WeblabVMService*, should appear. If it doesn't, do not go on, as something went wrong.

Starting the service

Locate the service in the Windows *Service Manager*. If the service is not started already, then click on it and start it.

Note: Alternatively, if you installed the service through the *sc_install_service* script, you may use the *sc_start_service* and the *sc_stop_service* to start and stop it.

If for any reason it fails to start, then something went wrong. Do not go on. Verify that you have .NET 3.0, and that the service is installed properly.

Testing the service

Warning: **CHECKLIST** *(Ensure the following before starting this section. All of them apply to the **guest** Windows (virtualized one))*

1. WeblabVMService appears in my list of processes (which can be checked through the Windows' *services.msc* utility).
2. When I start WeblabVMService, no errors occur. The status of the service changes to *started* and stays so.

We will now carry out a few tests to check whether WeblabVMService is working as expected with our current settings.

Test 1 This test should be done within the guest OS. That is, within the virtualized Windows. *(This test is meant to check that the In-VM Manager is working properly and can change the local password).*

1. If the WeblabVMService is not running already, start it.
2. Open a browser window.
3. Do the following query: *http://localhost:6789/?sessionid=testone*
4. It should take a while and then take you to a blank page with only the word *Done* written in black. If the page cannot be loaded or if an error occurs, then the service is either not running or failing. If that is the case, do not proceed. It is suggested that you contact the developers for support. From this point on, we will assume *Done* was printed.
5. The password of your Windows *weblab* account has now been changed to *testone*. That is essentially what the previous query did. You should now verify that this is indeed the case. Logout and try to login into your *weblab* account, using *testone* as password.

6. If you manage to login using that password, then congratulations, the first test was successful, you may go on. If you can't login using that password, then something failed. If *done* was printed to the screen, this is fortunately unlikely. Make sure you followed every step right. If it still doesn't work, please contact the developers for support.

Test 2 This test assumes that the first test was successful. We will try the following, *(This test is meant to check that the In-VM Manager can indeed be accessed and used from the Host machine).*

1. If the WeblabVMService is not running already, start it.
2. Find out the IP that has been assigned to your Virtual Machine in your local network. This is the IP we used in previous sections to connect to the machine through RDP. It will most likely be something such as *192.168.100.5*, but it may start with *172* instead, or with other digits.
3. Open a browser in your **host** machine (that is, **not** your guest machine).
4. Do the following query: *http://192.168.100.5:6789/?sessionid=testtwo*. Replace 192.168.100.5 with your actual VM IP.
5. It should take you to the same page as in the first test. A blank page with *Done* in black. If it worked, congratulations. The second test was successful. You may try to login with the password *testtwo* into the *weblab* account of your Virtual Machine if you wish to be sure. If it didn't, see the following note.

Note: The previous test should have loaded a blank page with *Done* written in black. If it did, you may skip this note. If it didn't, something went wrong. Most often, this means that the guest OS is not accessible from the host OS. Try to login into your host OS through RDP, in the same way you did before when you configured the *network settings* of the VM (in previous sections of this guide). If you still can connect to the machine through RDP, then you should repeat the first test, to make sure the service is still working. If RDP is working, and the first test is working, but the second test is still failing, please make sure that you have no firewall which may be blocking port 6789 on your local network (see the previous *network settings* section if you do). You may also try to repeat the second test with a different browser. If it still does not work, please *contact* the developers for support. (In this guide, from this point, we will assume that the second test did work. If it didn't, you may not want to proceed until the issue is solved).

Congratulations, if you are here, both tests should have passed. This means that WeblabVMService is properly installed and working.

Warning: CHECKLIST *(Ensure the following before going on to the next section.)*

1. Test 1 was completed and works as expected.
2. Test 2 was completed and works as expected.

Preparing the Virtual Machine: Base Snapshot

What is a snapshot?

Most VM systems (such as VirtualBox) support snapshots. Snapshots describe the exact state of the Virtual Machine at a given point of time. Once you have taken a shapshot, you can at any time restore your VM to it. This is how WebLab-Deusto VM experiment ensures that any change a user makes to the VM, is restored before the next session.

Base Snapshot

We will take an snapshot, which will be our *base snapshot*, the one every user will get to use. After the user is done, WebLab-Deusto will restore the system to that same *base snapshot* again.

To prepare a first base snapshot, you can follow these steps:

1. Start your guest Windows.
2. Login into your *weblab* account.
3. Start the In-VM Manager if it is not running already.
4. Install any software you wish the users to have.
5. Open any program that you want the users to see.
6. Prepare everything for the user. Arrange every open window.
7. Your machine should now be ready. Without closing it, it's time to take a snapshot. In VirtualBox menu, go to Machine->Take a snapshot. It will let you choose a name. Type *base*. You can actually choose a different one and configure it later, but we will use *base* for simplicity.

Note: Probably, your actual experiment is not ready yet. When it is, you will probably have to modify the base snapshot to include it. Fortunately, that is easy. Though the previous steps are somewhat linear, really the only important things are:

1. Your guest windows needs to be logged in the *weblab* account.
2. The In-VM manager needs to be started.
3. The machine must be turned on when you take the snapshot. If it isn't, it will have to be boot-up everytime, and this takes too long of a time.

These are essentially the three points that you have to take into account when creating your own base snapshots.

Testing the Base Snapshot

We will make sure we are on the right track. Do the following:

1. Start your guest Windows.
2. In your guest Windows, create some new file, and add it to the desktop. It can be any file, and have any name. For instance, you may create a *TESTING.TXT* text file.
3. From this point on, we will use the command line, to ensure that it is working as expected too.
4. Open a command line in your **host** Windows. (That is, not on your virtualized Windows). We will use it to manage virtualbox.
5. Recall your *VM name*. As we established in previous sections of this guide, that is the name that appears in VirtualBox's list, and which you can right click to start the machine, etc.
6. Type the following command in the command line: *vboxmanage controlvm "Windows VM" poweroff*. You should replace *Windows VM* with your actual *VM name*. The following is what should happen:

```
C:\Users\lrg>vboxmanage controlvm "Windows VM" poweroff
Oracle VM VirtualBox Command Line Management Interface Version 3.2.10
(C) 2005-2010 Oracle Corporation
All rights reserved.
```

(continues on next page)

(continued from previous page)

```
0%...10%...20%...30%...40%...50%...60%...70%...80%...90%...100%
```

Your machine should turn off. If it doesn't, make sure you installed VirtualBox properly, as described in previous sections, and that you specified the right *VM name* in your command.

7. We will now restore the *base* snapshot using the command line. Type the following: *vboxmanage snapshot "Windows VM" restore "base"*. Replace *Windows VM* with the actual name of your Virtual Machine, and replace *base* with the actual name of your snapshot (which is most likely *base* too, if you followed the previous sections accurately). The following is what should happen:

```
C:\Users\lrg>vboxmanage snapshot "Windows VM" restore "base"
    Oracle VM VirtualBox Command Line Management Interface Version 3.2.10
    (C) 2005-2010 Oracle Corporation
    All rights reserved.
```

8. Finally, we will start the VM through the command line. Type the following: *vboxmanage startvm "Windows VM"*. Again, replace *Windows VM* with the actual name of your Virtual Machine. The Virtual Machine should appear, loading your virtualized Windows, and the following should appear in your console:

```
C:\Users\lrg>vboxmanage startvm "Windows VM"
Oracle VM VirtualBox Command Line Management Interface Version 3.2.10
(C) 2005-2010 Oracle Corporation
All rights reserved.
```

9. If an error occurs, something is wrong. Check the previous steps. Note that your Windows snapshot should have loaded. What you see is exactly what your experiment users will see. If something is amiss, for instance, if Windows had to boot (if it wasn't started already) or if the programs you left open when you created your *base* snapshot are not open anymore, then you probably did not create the snapshot properly or you did not restore it. You might want to check the previous sections if that is the case.

Warning: CHECKLIST *(Please ensure the following before going on to the next section)*

1. My VM was loaded properly. Windows did not need to boot.
2. The programs I left open when I created my *base* snapshot were there still.
3. I was able to accomplish all of the above through the command line.

If nothing went wrong, congratulations, your snapshot is ready.

Configuring the WebLab instance

If you have followed the guide up to here, every prerrequisite is now ready. In this last section, we will configure and test the WebLab experiment itself. That is, the experiment server which will actually control the VM we have created, through the means we have provided.

Recalling important variables

Before going on we will need to remember some variables which we established during the previous sections. We need the following:

1. **VM name**: The name you gave to your VM. The one you used with the *vboxmanage* commands.

2. **VM ip**: The local IP of your VM. That is the IP that you used to connect to it through RDP.

Creating the instance through Weblab-admin

As we mentioned in the first sections of this guide, you need to have the weblab-admin script properly installed. The next steps assume you do.

We will next use weblab-admin to create a new WebLab instance with our VM experiment. We will run the following command:

```
weblab-admin create WLTest --vm --vbox-base-snapshot "base" --vbox-vm-name "Windows VM
↪" --vm-estimated-load-time 30
--http-query-user-manager-url "http://192.168.64.143:6789" --vm-url 192.168.64.143 --
↪http-server-port 8000
```

However, you will need to make a few changes to the command:

1. Change the IP, 192.168.64.143 for your **VM ip**. You need to change it for the http-query-user-manager-url, which is, essentially, the address to which the password changing queries that we have explained in previous sections are sent. You also need to change it for the vm-url variable. This isn't that important, because it is simply the URL that will be displayed to experiment users, so that they can connect through RDP.
2. Change the Virtual Machine name, "Windows VM", for your own **VM name**.
3. Note that VMDeploy is, in this case, the name we have given to our new instance. You may change it, if you want to.

This is what should happen:

```
(weblab.dev) C:\shared\weblab_github\weblabdeusto_lrg\server\src>weblab-admin
create WLTest --force --vm --vbox-base-snapshot base --vbox-vm-name "Windows VM"
--vm-estimated-load-time 30 --http-query-user-manager-url "http://192.168.64.143:6789"
--vm-url 192.168.64.143 --http-server-port 8000

patchZsiPyExpat skipped; ZSI not installed
patchZsiFaultFromException skipped; ZSI not installed

Congratulations!
WebLab-Deusto system created

Run:

         weblab-admin start VMTestTwoS

to start the WebLab-Deusto system. From that point, you'll be able to access:

   http://localhost:8000/

Or in production if you later want to deploy it in Apache:

         http://localhost/weblab/

And log in as 'admin' using 'password' as password.

You should also configure the images directory with two images called:

         sample.png and sample-mobile.png

```

(continues on next page)

(continued from previous page)

```
You can also add users, permissions, etc. from the admin CLI by typing:

        weblab-admin admin VMTestTwoS

Enjoy!
```

If an error occurs here, it is likely that it is not related to the VM experiment. Please, make sure that you have installed weblab properly and that you can deploy instances (without vm experiments) through *weblab-admin*. Unfortunately, doing that is beyond the scope of this guide, but you can check the weblab installation documentation. From this point, we will assume your instance was deployed properly.

Testing our new instance

To start our new instance, type the following:

```
weblab-admin start WLTest
```

Replace *WLTest* with the name you gave to your weblab instance (which is most likely *WLTest* either way.

Your WebLab instance should now start.

Open a browser in your computer, and connect to it through http://localhost:8000, which is the port we specified.

You can log into it with the account name *admin* and the password *password*. There should be a few experiments, among them, your new VM experiment.

Check, reserve, and check whether it works as expected.

If the experiment is reserved properly, and you can connect to your Virtual Machine through RDP using the provided address, and if you can see the snapshot we set in previous sections. **Congratulations, you have successfully deployed an VM experiment!**

If no error occurred, this guide is **over**.

If something went wrong, take a look at the next section.

Something failed

If you are in this section, some problem occurred and your VM deployment is not working. We will here describe some likely errors.

My experiment seems to work properly, but I can't connect through RDP to the provided address.

Make sure that the IP that you are being provided with is the same as the IP you tested with, on the *network configuration* section of this guide. If it is and the *network configuration* section's test still succeeds, please *contact* the Weblab developers for support.

The VM experiment appears, but an error occurs before the reservation succeeds.

Make sure that you installed Weblab properly. That is, make sure that experiments other than the VM one work. If other experiments don't work, then the problem is most likely not related to VMs. Check the weblab installation guide. If it's only the VM experiment which does not work, then please *contact* the Weblab developers for support.

The VM experiment appears, I can reserve, but when the experiment loads, the progress bar never finishes.

Make sure that you have installed the In-VM Manager properly by carrying out every suggested test and checklist. Particularly, make sure that you can change your Guest's password from your Host machine through http://{guest-ip}/?sessionid=newpassword. Check the console in case there is an error. If there is, please *contact* the Weblab developers for support.

3.1.4 Summary

With this section, you have learnt to develop a new remote laboratory using WebLab-Deusto. Now it's time to deploy it, going to *the following section*.

3.2 Remote laboratory deployment

Table of Contents

3.2.1 Introduction

In the *previous section* we have covered how to create new remote laboratories using the WebLab-Deusto APIs. After it, you have a working (yet draft or very initial) code that you want to use. However, we have not covered how to use them in an existing deployment of WebLab-Deusto. This section covers this task. This way, here we will see how to register the already developed clients and servers.

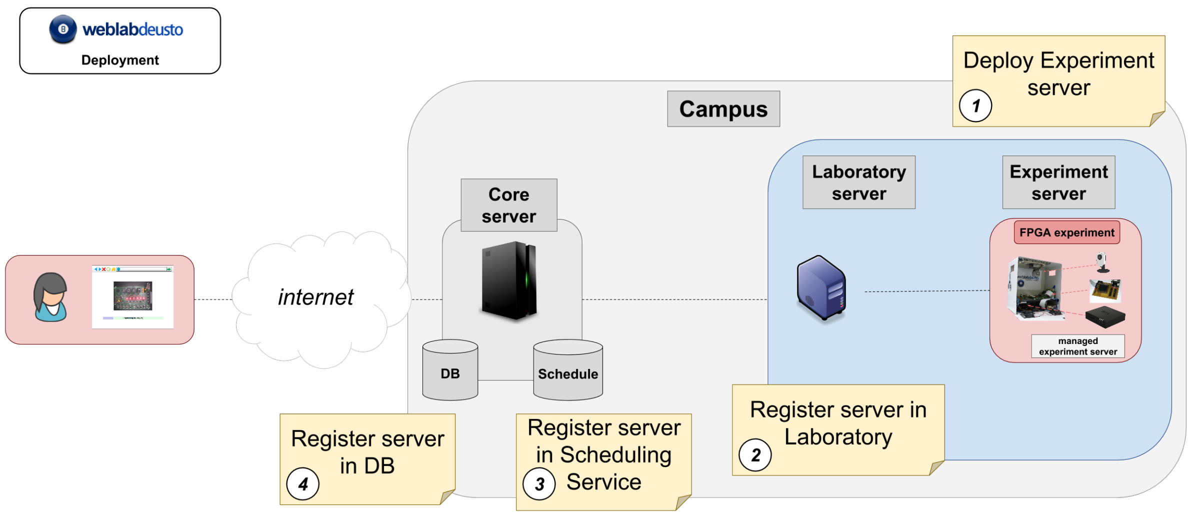


Fig. 2: Steps to deploy a remote laboratory in WebLab-Deusto.

This process is compounded of the following steps:

1. *Step 1: Deploying the Experiment server*
2. *Step 2: Registering the experiment server in a Laboratory server*
3. *Step 3: Registering a scheduling system for the experiment*
4. *Step 4: Add the experiment server to the database and grant permissions*

After these steps, your laboratory should be working.

3.2.2 Step 1: Deploying the Experiment server

As *previously explained*, there are two major ways to develop a WebLab-Deusto Experiment server:

1. *Managed*, which includes Experiment servers developed in Python, as well as experiments developed in other platforms (e.g., Java, .NET, LabVIEW, C, C++. . .)
   1. If the Experiment server was developed in Python, then it might use any of the protocols of WebLab-Deusto. This part is explained below in *WebLab-Deusto Python server*.
   2. However, if other platform was used (e.g., Java, .NET, C, C++), then the XML-RPC approach must be taken. This is explained below in *Non-Python managed servers (XML-RPC based)*.
2. *Unmanaged*, such as external HTTP applications.

This section assumes that you have previously read the following two sections:

- *Directory hierarchy*
- *Technical description*

Managed server

This section describes how to deploy a laboratory using the managed approach.

1. If the Experiment server was developed in Python, then it might use any of the protocols of WebLab-Deusto. This part is explained below in *WebLab-Deusto Python server*.

2. However, if other platform was used (e.g., Java, .NET, C, C++), then the XML-RPC approach must be taken. This is explained below in *Non-Python managed servers (XML-RPC based)*.

WebLab-Deusto Python server

As explained in *Directory hierarchy*, WebLab-Deusto uses a directory hierarchy for configuring how the communications among different nodes is managed. In the case of WebLab-Deusto Python servers, you may run them inside the same process as the Laboratory server, being able to use the configuration subsystem and being easier to manage.

So as to do this, let us assume that there is a simple system as the one created by:

```
$ weblab-admin create sample --http-server-port=12345
```

And that you have developed an experiment using the Python API as explained in *WebLab-Deusto server (Python)*. Your experiment can be something as simple as:

```
import json

from weblab.experiment.experiment import Experiment
import weblab.experiment.level as ExperimentApiLevel

class ElectronicsLab(Experiment):
    def __init__(self, coord_address, locator, config, *args, **kwargs):
        super(ElectronicsLab,self).__init__(*args, **kwargs)
        self.config = config

    def do_start_experiment(self, client_initial_data, server_initial_data):
        print("Start experiment")
        print("Client initial data:", json.loads(client_initial_data))
        print("Server initial data:", json.loads(server_initial_data))
        print("Camera:", self.config.get('my_camera_url'))
        return json.dumps({ "initial_configuration" : "cam='cam1'"})

    def do_get_api(self):
        return ExperimentApiLevel.level_2

    def do_dispose(self):
        print("User left")
        return "{}"

    def do_send_command_to_device(self, command):
        print("Command: ", command)
        return "Got your command"

    def do_should_finish(self):
        print("Checking if the user should exit. If returned 0, will not ask again.␣
↪If return 1, WebLab will kick him out")
        return 5
```

Let us also assume that this code is in a file called `myexperiments.py`, and that is in a directory called `/home/tom/experiments`.

Then, first, we will need to make sure that WebLab-Deusto can access that file. To this end, we would add that directory to the `PYTHONPATH`.

In Windows we can run the following each time before running weblab-admin start:

```
(weblab) C:\Users\Tom> set PYTHONPATH=C:\Users\Tom\experiments
```

In Linux / Mac OS X we can run the following:

```
(weblab) tom@tom-laptop:~$ export PYTHONPATH=/home/tom/experiments:$PYTHONPATH
```

To verify that this is correct, you should be able to do the following:

```
$ python
[...]
>>> import myexperiments
>>>
```

If no ImportError occurs, it means that everything required (e.g., your code and WebLab-Deusto code) is available.

Warning: The `PYTHONPATH` path must be absolute (e.g., `/home/tom/experiments`) and not relative (e.g., `../experiments`). When running `weblab-admin start`, the current working directory is changed and could lead to wrong results.

In this case, the Python class identifier of your Python laboratory would be `myexperiments.ElectronicsLab` (since it's the class `ElectronicsLab` of the module `myexperiments.py`). If you had a more complex hierarchy (for example: a Python package called `myinstitution` and inside several modules like `myexperiments.py`), then the Python class identifier would be `myinstitution.myexperiments.ElectronicsLab`.

The next step is to modify the `configuration.yml` file generated by `weblab-admin create sample`. Originally, it looks like the following:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
```

Which looks like the following:

But we want to add a new laboratory called `electronics` so it becomes the following:

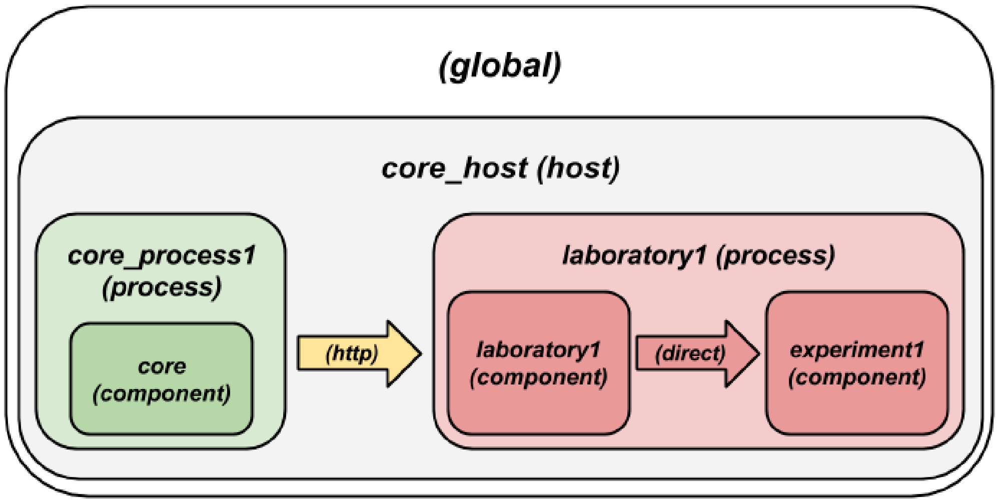


Fig. 3: `sample` as created by default

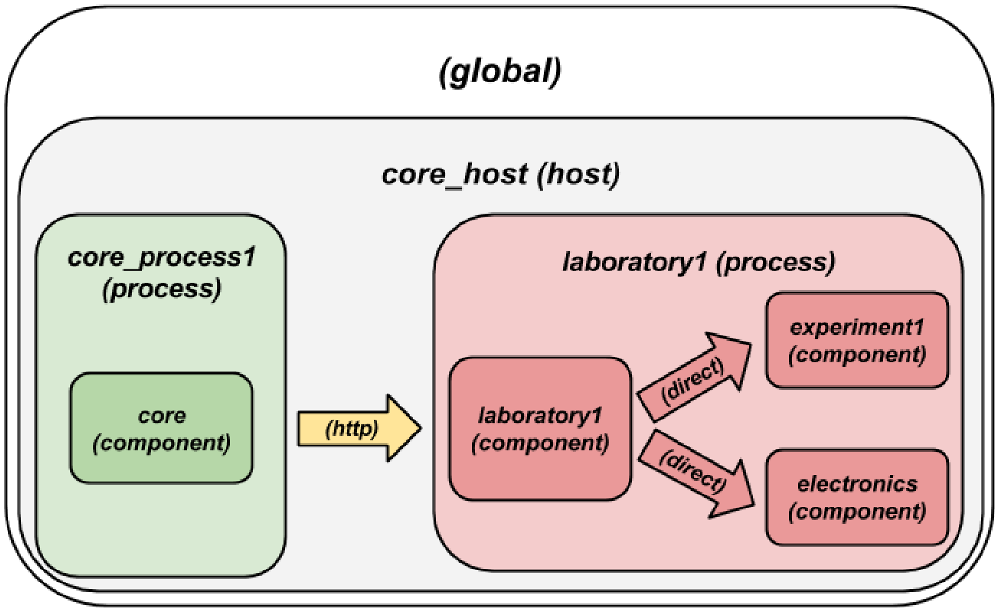


Fig. 4: `sample` after the modification

So as to have this new component which is an experiment running your code, you have to add it inside the `components` of `laboratory1`, as follows:

```yaml
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          electronics:
            class: myexperiments.ElectronicsLab
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
```

If you want to add configuration variables, then you can either add them to the component or to any of the upper layers (to the host, process or globally), and either add them in a configuration file or inline as follows:

```yaml
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          electronics:
            class: myexperiments.ElectronicsLab
            config:
              my_camera_url: http://cams.weblab.deusto.es/webcam/electronics.jpg
            type: experiment
```

(continues on next page)

(continued from previous page)

```
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
```

From the Python code, you may access that variable.

From this point, the internal WebLab-Deusto address of your Experiment server is `electronics:laboratory1@core_host`. You might see it later when seeing which device was accessed by students, or in logs.

However, refer to *Directory hierarchy* for further details for more complex deployments.

Warning: Avoid naming conflicts with your laboratory name. For instance, `myexperiments.ElectronicsLab` relies on the fact that there is no other `myexperiments` directory in the `PYTHONPATH`. If you use other names, such as `experiments.ElectronicsLab`, `voodoo.ElectronicsLab` or `weblab.ElectronicsLab`, you will enter in naming conflicts with existing modules of WebLab-Deusto or of libraries used by WebLab-Deusto.

To verify that the configuration is fine, start the server:

```
$ weblab-admin start sample
 * Running on http://0.0.0.0:12345/ (Press CTRL+C to quit)
Press <enter> or send a sigterm or a sigint to finish
```

If no error is reported in a few seconds, you can press enter to stop it and continue. If the following error appears:

```
$ weblab-admin start sample
 * Running on http://0.0.0.0:12345/ (Press CTRL+C to quit)
Press <enter> or send a sigterm or a sigint to finish
[...]
voodoo.gen.exc.LoadingError: Error loading component: 'myexperiments.ElectronicsLab'␣
↪for server electronics:laboratory1@core_host: No module named myexperiments
```

It means that the myexperiments.py file does not seem to be available. Verify that running in the same terminal reports no error:

```
$ python
[...]
>>> import myexperiments
>>> print(myexperiments.ElectronicsLab)
<class 'myexperiments.ElectronicsLab'>
>>>
```

If it reports an ImportError, verify that you configured the `PYTHONPATH` according to what it was defined earlier in this subsection.

After you start WebLab-Deusto with no error, you can now jump to the *Step 2: Registering the experiment server in a Laboratory server*.

Non-Python managed servers (XML-RPC based)

As explained in *Directory hierarchy*, WebLab-Deusto uses a directory hierarchy for configuring how the communications among different nodes is managed. In the case of experiments using XML-RPC, it is required to *lie the system*, by stating that there is an experiment server listening through XML-RPC in a particular port, with a particular configuration that will never be run.

The easiest way to see an example of this configuration is running the following:

```
$ weblab-admin create sample --xmlrpc-experiment --xmlrpc-experiment-port=10039 --
↪http-server-port=12345
```

This will generate a particular configuration, with two *hosts* at WebLab-Deusto level: one called `core_host`, and the other `exp_host`.

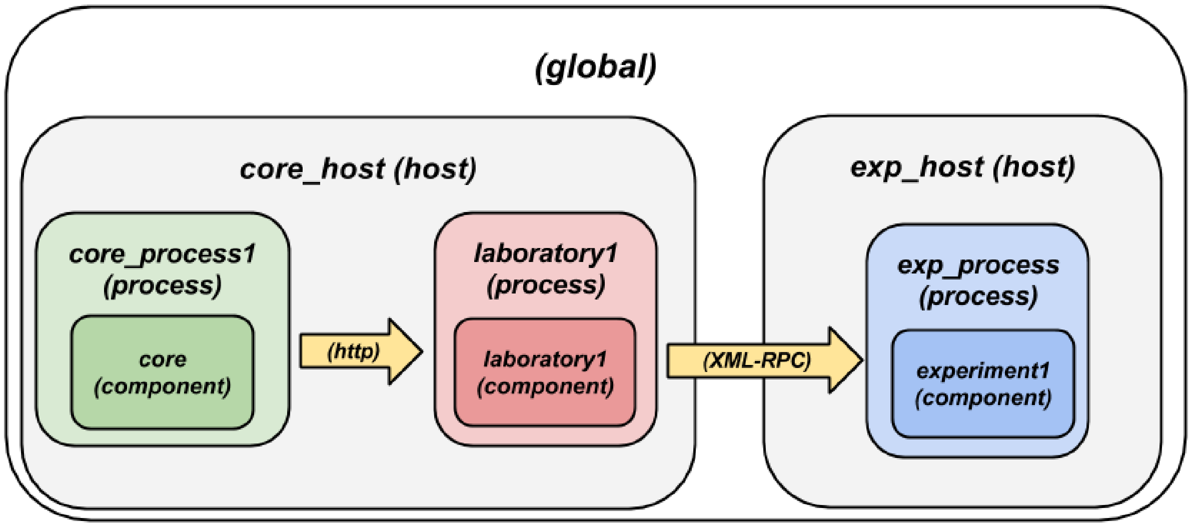


Fig. 5: Default settings when creating an XMLRPC lab.

The generated configuration is the following:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
```

(continues on next page)

(continued from previous page)

```
  exp_host:
    runner: run-xmlrpc.py
    host: 127.0.0.1
    processes:
      exp_process:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            protocols:
              port: 10039
              supports: xmlrpc
            type: experiment
```

So as to run the first one, you should run:

```
$ weblab-admin start sample --host core_host
```

You may also run:

```
$ weblab-admin start sample --host exp_host
```

In other console at the same time. That way, there would be a Python Experiment server listening on port `10039`. However, this is not what we want here. What we want here is to be able to run other laboratories, such as a Java or .NET Experiment server. So if we don't execute this last command, and instead we run our Java (or .NET, C++, C. . .) application listening in that port, everything will work.

For this reason, using the `weblab-admin` command with those arguments is the simplest way to get a laboratory running. If you only want to test the system with your new developed remote laboratory, you can simply use the `--xmlrpc-experiment` flags, in the `configuration.yml` change `experiment1` for `electronics` and jump to the *Step 2: Registering the experiment server in a Laboratory server*.

However, the typical action is to use the *Directory hierarchy* documentation to establish at WebLab-Deusto level that there will be an Experiment server listening in a particular port.

So, let's start from scratch. Let's imagine that we create other example, such as:

```
$ weblab-admin create sample --http-server-port=12345
```

This will generate the following schema:

And the following configuration:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          experiment1:
```

(continues on next page)

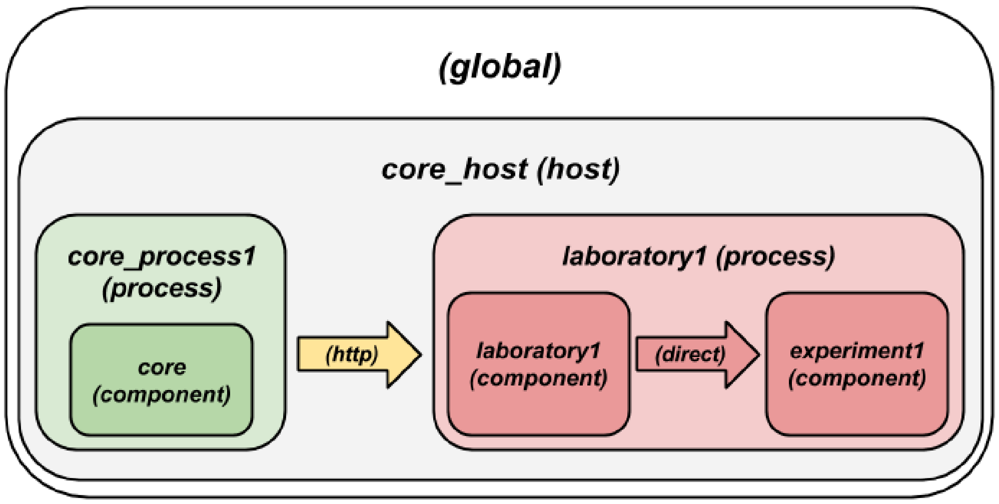


Fig. 6: `sample` as created by default

(continued from previous page)

```
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
```

We want to add an external Experiment server in a different host. So as to do this, we will append at the end the following:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            protocols:
```

(continues on next page)

(continued from previous page)

```
              port: 10001
            type: laboratory
  exp_host:
    runner: run-xmlrpc.py
    host: 127.0.0.1
    processes:
      exp_process:
        components:
          electronics:
            class: experiments.dummy.DummyExperiment
            protocols:
              port: 10039
              supports: xmlrpc
            type: experiment
```

Note: `exp_host` is another host, so must have the same indentation (number of spaces before) as `core_host:`

Actually, the values of `runner` and `class` in this case are not relevant, since they will not be used. With these changes, the structure will be the following:

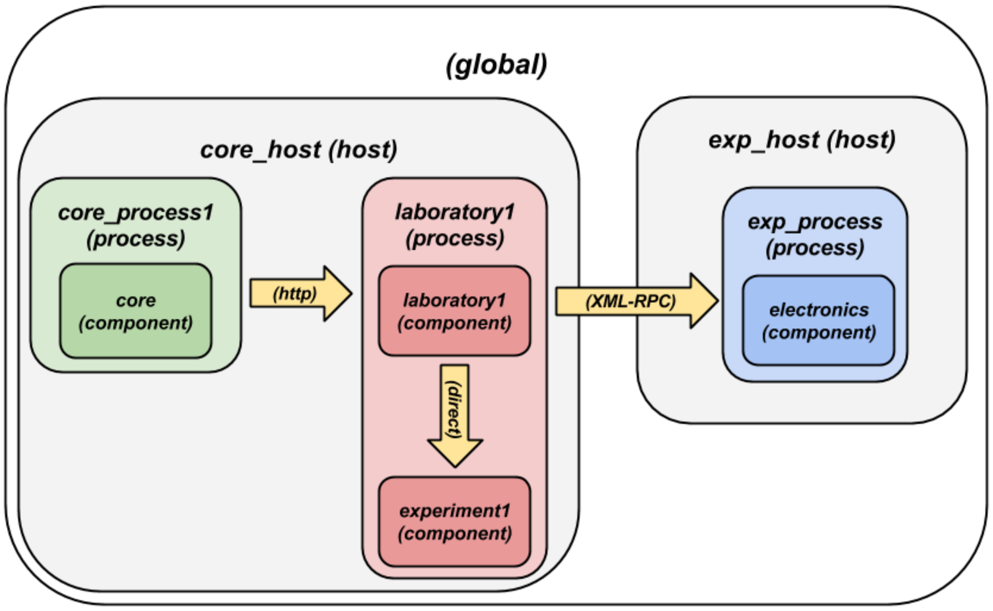


Fig. 7: `sample` modified to support a new `electronics` laboratory.

Doing this, the Experiment server will have been registered. You can test that running the following will start without errors the core host:

```
$ weblab-admin start sample --host core_host
```

However, you must make sure that you start the Experiment server (developed in other technology: .NET, C++…) every time you start the WebLab-Deusto servers (preferably, just before than just after).

Warning: By default, WebLab-Deusto will attempt to perform XML-RPC requests to `http://127.0.0.1:10039/`.

However, certain libraries (such as the one of .NET) does not support this scheme, and it requires that WebLab-Deusto calls `http://127.0.0.1:10039/weblab`. For this reason, in .NET and LabVIEW, you need to configure the system adding `path` to the component configuration:

```
protocols:
  port: 10039
  supports: xmlrpc
  path: /weblab
```

In the following sections, you will address the Experiment server as `electronics:exp_process@exp_host`.

You can now jump to the *Step 2: Registering the experiment server in a Laboratory server*.

Unmanaged server

The unmanaged laboratories internally are a managed laboratory where the managed server maps the calls to the unmanaged server. Therefore, the steps are just a subset of the steps of the previous section. For the sake of simplicity, we repeat here that subset, focused on an unmanaged server.

The first step is to modify the `configuration.yml` file generated by `weblab-admin create sample`. Originally, it looks like the following:

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
```

Which looks like the following:

But we want to add a new laboratory called `electronics` so it becomes the following:

So as to have this new component which is an experiment running your code, you have to add it inside the `components` of `laboratory1`, as follows:

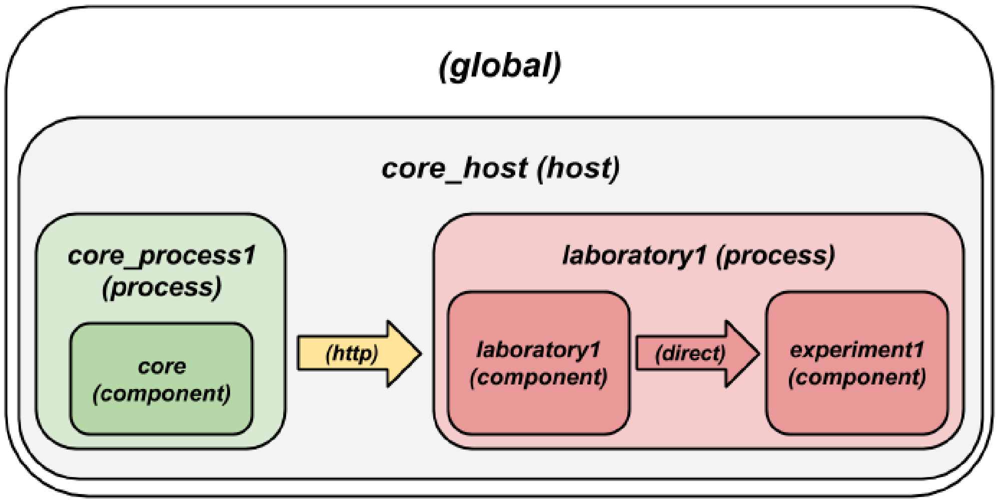


Fig. 8: `sample` as created by default

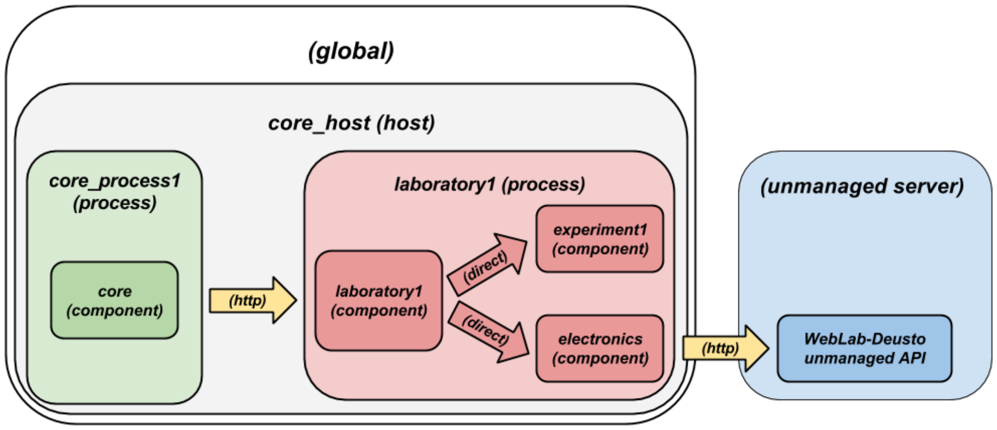


Fig. 9: `sample` after the modification

```
hosts:
  core_host:
    runner: run.py
    config_file: core_host_config.py
    processes:
      core_process1:
        components:
          core:
            config:
              core_facade_port: 10000
              core_facade_server_route: route1
            type: core
      laboratory1:
        components:
          experiment1:
            class: experiments.dummy.DummyExperiment
            config:
              dummy_verbose: true
            type: experiment
          laboratory1:
            config_file: lab1_config.py
            protocols:
              port: 10001
            type: laboratory
          electronics:
            class: experiments.http_experiment.HttpExperiment
            config:
              http_experiment_url: http://server.myinstitution.edu/experiment1/
              http_experiment_username: weblab
              http_experiment_password: bdca31bd-b5d4-4f2f-995a-e6cd9d0a1b2d
            type: experiment
```

Note: Please take into account that `electronics` must be indented as `laboratory1` (same number of spaces before), as shown in the example.

The name of `experiments.http_experiment.HttpExperiment` is fixed. The configuration variables are the ones you provide so as to let the unmanaged laboratory know that it's WebLab-Deusto accessing. Make sure that in the code you develop you take this into account and check these credentials.

You can now jump to the *Step 2: Registering the experiment server in a Laboratory server*.

3.2.3 Step 2: Registering the experiment server in a Laboratory server

In the following figure, we have already finished step 1, which is the most complex. The rest of the steps are independent of the technology used, and they are only focusing on registering the laboratory in the different layers. In this subsection, we're in the step 2: registering the server in the Laboratory server.

Each Experiment Server must be registered in a single Laboratory server. One Laboratory Server can manage multiple Experiment servers. So as to register a Experiment server, we have to go to the Laboratory server configuration file. In the near future, this configuration will disappear and everything will be configured in the database. When you create a WebLab-Deusto instance doing:

```
$ weblab-admin create sample
```

This file by default is called `lab1_config.py`, and by default it contains the following:

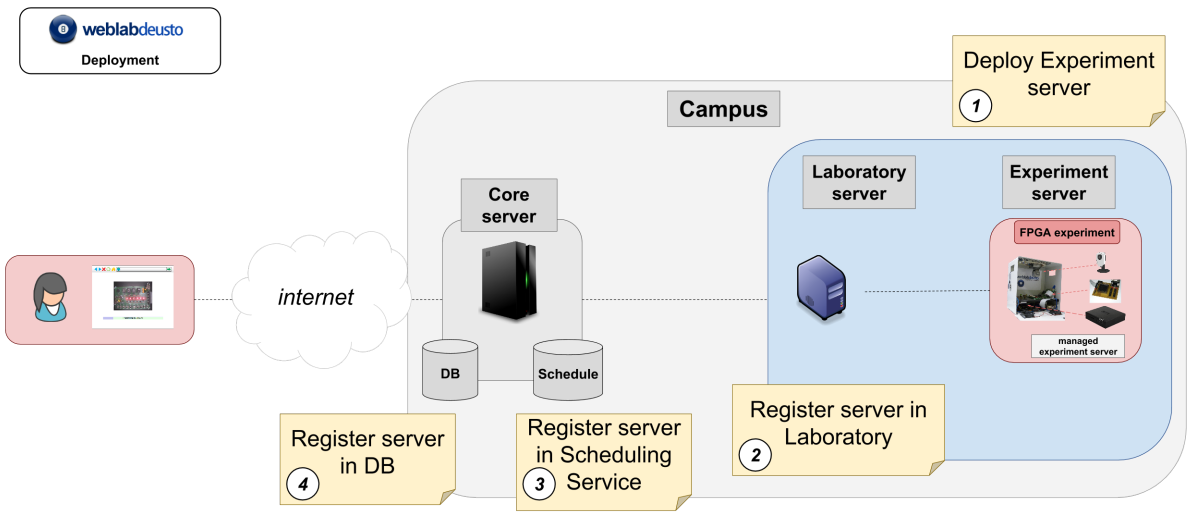


Fig. 10: We're in step 3.

```
laboratory_assigned_experiments = {
        'exp1:dummy@Dummy experiments' : {
                'coord_address' : 'experiment1:laboratory1@core_host',
                'checkers' : ()
            },
    }
```

This means that the current Laboratory Server has one Experiment Server assigned.

- `exp1:dummy@Dummy experiments` is the identifier for this resource at the Laboratory Server. Typically `dummy` is the name as it will be in the database and `Dummy experiments` is the category name as it will be in the database. `exp1` is not published anywhere, but will be used by the Core server in the following step.
- `experiment1:laboratory1@core_host` is the identifier at WebLab-Deusto level of the experiment. It establishes that it is the component `experiment1` of the process `laboratory1` of the host `core_host`.

You can find in *Multiple core servers* more elaborated examples.

So as to add the new experiment, you must add a new entry in that Python dictionary. For example, if you have added an electronics laboratory, and in the previous step you have located them in the `laboratory1` instance in the `core_host`, you should edit this file to add the following:

```
laboratory_assigned_experiments = {
        'exp1:dummy@Dummy experiments' : {
                'coord_address' : 'experiment1:laboratory1@core_host',
                'checkers' : ()
            },
        'exp1:electronics@Electronics experiments' : {
                'coord_address' : 'electronics:laboratory1@core_host',
                'checkers' : (),
                'api'      : '2',
            },
    }
```

If your laboratory is an unmanaged laboratory, then the client `redirect` could cause problems since when the user is redirected, WebLab-Deusto might assume that the user is not anymore logged in and the laboratory should be re-scheduled to other user. So as to avoid this, if your experiment is unmanaged, add also the `manages_polling` variable:

```
laboratory_assigned_experiments = {
        'exp1:dummy@Dummy experiments' : {
                'coord_address' : 'experiment1:laboratory1@core_host',
                'checkers' : (),
                'manages_polling': True,
            },
    }
```

If you have used XML-RPC, the experiment server is somewhere else outside the `core_host`, but you only need to put in `coord_address` the identifier. For example, if you created a new laboratory using Java, you will need to add something like:

```
laboratory_assigned_experiments = {
        'exp1:dummy@Dummy experiments' : {
                'coord_address' : 'experiment1:laboratory1@core_host',
                'checkers' : ()
            },
        'exp1:electronics@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2'
            },
    }
```

The `api` variable indicates that the API version is `2`. If in the future we change the Experiment server API, the system will still call your Experiment server using the API available at this time. If you are using an old library, you might state `api` to `1` and it will work.

One of the duties of the Laboratory server is to check frequently whether the Experiment server is alive or not. This may happen due to a set of reasons, such as:

- The laboratory uses a camera which is broken
- The connection failed
- The Experiment server was not started or failed

By default, every few seconds the system checks if the communication with the Experiment server works. If it is broken, it will notify the administrator (if the mailing variables are configured) and will remove it from the queue. If it comes back, it marks it as fixed again.

However, you may customize the `checkers` that are applied. The default checkers are defined in `weblab.lab.status_handler` (code). At the time of this writing, there are two:

- `HostIsUpAndRunningHandler`, which opens a TCP/IP connection to a particular host and port. If the connection fails, it marks the experiment as broken.
- `WebcamIsUpAndRunningHandler`, which downloads an image from a URL and checks that the image is a JPEG or PNG.

So as to use them, you have to add them to the `checkers` variable in the Laboratory server configuration. For example, if you have a FPGA laboratory with a camera and a microcontroller that does something, you may have the following:

```
'exp1:ud-fpga@FPGA experiments' : {
    'coord_address' : 'fpga:process1@box_fpga1',
    'checkers' : (
                    ('WebcamIsUpAndRunningHandler', ("https://www.weblab.deusto.es/
↪webcam/proxied.py/fpga1",)),
```

(continues on next page)

(continued from previous page)

```
                    ('HostIsUpAndRunningHandler', ("192.168.0.70", 10532)),
                ),
    'api'       : '2',
},
```

In this case, the system will check from time to time that URL to find out an image, and will connect to that port in that IP address, as well as the default checking (calling a method in the Experiment server to see that it is running).

You can develop your own checkers in Python, inheriting the `AbstractLightweightIsUpAndRunningHandler` class and adding the class to the global `HANDLERS` variable of that module.

Additionally, if you have laboratories that you don't want to check, you may use the following optional variable in the Laboratory server. It will simply skip this process.

```
laboratory_exclude_checking = [
    'exp1:electronics@Electronics experiments',
    'exp1:physics@Physics experiments',
]
```

After this, you can jump to *Step 3: Registering a scheduling system for the experiment*.

3.2.4 Step 3: Registering a scheduling system for the experiment

Now we move to the Core server. The Core server manages, among other features, the scheduling of the experiments. At the moment of this writing, there are different scheduling options (federation, iLabs compatibility, and priority queues). We do not support booking using a calendar at this moment.

All the configuration of the Core server related to scheduling is by default in the `core_host_config.py` file. It is placed there so if you have multiple Core servers in different instances (*which is highly recommended*), you have the configuration in a single location. In this file, you will find information about the database, the scheduling backend, etc.

The most important information for registering a remote laboratory is the following:

```
core_scheduling_systems = {
        'dummy'             : ('PRIORITY_QUEUE', {}),
        'robot_external'    : weblabdeusto_federation_demo,
}
```

Here, it is defined the different schedulers available for each remote laboratory *type*. WebLab-Deusto supports load balancing, so it assumes that you may have multiple copies of a remote laboratory. In that sense, we will say that one *experiment type* might have multiple *experiment instances*. This variable (`core_scheduling_systems`) defines which scheduling system applies to a particular *experiment type*. Say that you have one of five copies of a experiment identified by `electronics` (of category `Electronics experiments`). Then you will add a single *experiment type* to this variable. If you only have one, it's the same procedure (adding a single *experiment type*). The name used is only used inside this file, and it has no relation with previous names.

```
core_scheduling_systems = {
        'dummy_queue'       : ('PRIORITY_QUEUE', {}),
        'robot_external'    : weblabdeusto_federation_demo,
        'electronics_queue' : ('PRIORITY_QUEUE', {}),
}
```

However, we still have to map the experiment instances to this experiment type. So as to do this, you will see that there is another variable in the Core server which by default it has:

```
core_coordinator_laboratory_servers = {
    'laboratory1:laboratory1@core_host' : {
            'exp1|dummy|Dummy experiments' : 'dummy1@dummy_queue',
        },
}
```

This variable defines which Laboratory servers are associated, which *experiment instances* are associated to each of them, and how they are related to the scheduling system. For instance, with this default value, it is stating that there is a Laboratory server located at `core_host`, then in `laboratory1` and then in `laboratory1`. This Laboratory server manages a single experiment server, identified by `exp1` of the experiment type `dummy` of category `Dummy experiments`. This *experiment instance* represents a slot called `dummy1` of the scheduler identified by `dummy_queue`.

So, when a user attempts to use an experiment of `dummy` (category `Dummy experiments`), the system is going to look for how many are available. It will see that there is only one slot (`dummy1`) in the queue (`dummy_queue`) that is of that type. So if it is available, it will call that Laboratory server asking for `exp1` of that *experiment type*. But if there was no slot available (e.g., some other student is using it), it will simply wait for that slot to be available.

Therefore, if you have added a single Experiment server of electronics to the existing Laboratory server, you can safely add:

```
core_coordinator_laboratory_servers = {
    'laboratory1:laboratory1@core_host' : {
            'exp1|dummy|Dummy experiments'       : 'dummy1@dummy_queue',
            'exp1|electronics|Electronics experiments' : 'electronics1@electronics_
↪queue',
        },
}
```

In the near future, all this will be in the database and therefore it will not be dealt with file-based configurations. However, in the meanwhile it's very important to understand what names are mapped among the different files.

The name `exp1|electronics|Electronics experiments` is mapped to the name `exp1:electronics@Electronics experiments` that we used in the previous section in the Laboratory Server. However, the separators are changed from `:` or `@` to `|`. The name `exp1` is only used in those two files. However, the other two components are the experiment name (`electronics`) and category name (`Electronics experiments`) in the database.

The name `electronics1` is not used anywhere else, so feel free to use any other name (e.g., `slot1`, etc.).

With this information, you are ready to jump to *Step 4: Add the experiment server to the database and grant permissions*. However, here we document other special scenarios, such as balancing the load of users among different copies of the laboratories, or supporting more than one user in a single laboratory at the same time.

Load balancing

If you have two copies of the same type of laboratory, you can add:

```
core_coordinator_laboratory_servers = {
    'laboratory1:laboratory1@core_host' : {
            'exp1|dummy|Dummy experiments'             : 'dummy1@dummy_queue',
            'exp1|electronics|Electronics experiments' : 'electronics1@electronics_
↪queue',
            'exp2|electronics|Electronics experiments' : 'electronics2@electronics_
↪queue',
        },
}
```

This means that if two students come it asking for an `electronics` laboratory, one will go to one of the copies and the other to the other. The process is random. A third user would wait for one of these two students to leave.

If you have two different experiments (one of electronics and one of physics), then you should add:

```
core_coordinator_laboratory_servers = {
    'laboratory1:laboratory1@core_host' : {
            'exp1|dummy|Dummy experiments'             : 'dummy1@dummy',
            'exp1|electronics|Electronics experiments' : 'electronics1@electronics_
↪queue',
            'exp1|physics|Physics experiments'         : 'physics1@physics_queue',
        },
}
```

Sharing resources among laboratories

This system is designed to be flexible. For instance, it supports to have more than one Experiment server associated to the same physical equipment. For example, in WebLab-Deusto we have the CPLDs and the FPGAs, with one Experiment server that allows users to submit their own programs. However, we also have other Experiment servers called `demo`, which are publicly available and anyone can use them. These Experiment servers do not allow users to submit their own program, though: they use their own default program for demonstration purposes. Additionally, we have two CPLDs, so the load of users is balanced between these two copies, and a single FPGA. The configuration is the following:

```
core_coordinator_laboratory_servers = {
    'laboratory1:laboratory1@core_host' : {

            # Normal experiments:
            'exp1|ud-pld|PLD experiments'    : 'pld1@pld_queue',
            'exp2|ud-pld|PLD experiments'    : 'pld2@pld_queue',
            'exp1|ud-fpga|FPGA experiments'  : 'fpga1@fpga_queue',

            # Demo experiments: note that the scheduling side is the same
            # so they are using the same physical equipment.
            'exp1|ud-demo-pld|PLD experiments' : 'pld1@pld_queue',
            'exp2|ud-demo-pld|PLD experiments' : 'pld2@pld_queue',
            'exp1|ud-demo-fpga|FPGA experiments' : 'fpga1@fpga_queue',
        },
}
```

In this case, if three students reserve `ud-pld@PLD experiments`, two of them will go to the two copies, but the third one will be in the queue. If somebody reserves a `ud-demo-pld@PLD experiments`, he will also be in the queue, even if the laboratory and the code that he will execute is different. The reason is that it is using the same exact device, so it makes sense decoupling the scheduling subsystem of the experiment servers and clients.

Concurrency

Finally, one feature of this system is that it enables that you provide more than one time slot to a single resource. For example, you may establish at Core server that there are 10 different `copies` of the laboratory, even if there is a single one:

```
core_coordinator_laboratory_servers = {
    'laboratory1:laboratory1@core_host' : {
            'exp1|dummy|Dummy experiments'             : 'dummy1@dummy_queue',
            'exp1|electronics|Electronics experiments' : 'electronics1@electronics_
↪queue',
```

(continues on next page)

(continued from previous page)

```
            'exp2|electronics|Electronics experiments' : 'electronics2@electronics_
↪queue',
            'exp3|electronics|Electronics experiments' : 'electronics3@electronics_
↪queue',
            'exp4|electronics|Electronics experiments' : 'electronics4@electronics_
↪queue',
            'exp5|electronics|Electronics experiments' : 'electronics5@electronics_
↪queue',
        },
}
```

Then, in the Laboratory server you must create those registries, but they can point to the same laboratory:

```
laboratory_assigned_experiments = {
        'exp1:dummy@Dummy experiments' : {
                'coord_address' : 'experiment1:laboratory1@core_host',
                'checkers' : ()
            },
        'exp1:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2'
            },
        'exp2:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2'
            },
        'exp3:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2'
            },
        'exp4:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2'
            },
        'exp5:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2'
            },
    }
```

This way, five students will be able to enter to the laboratory at the same time, and they will be able to interact each other. The main problem is that by default, the server API does not support knowing which student is submitting each request, since the methods are essentially something like:

```
String sendCommand(String command);
```

However, there is other API, called the Concurrent API (see *WebLab-Deusto server (Python)*), not supported at the moment by most of the libraries but yes by the Python experiments, which supports this. It which basically adds a `lab_session_id` string to the beginning of each parameter. That way, the method for sending commands, for instance, is as follows:

```
String sendCommand(String labSessionId, String command);
```

Using this, the Experiment developer can identify who is accessing in the laboratory and reply different messages to each user. So as to configure this, the Laboratory server must use the following `api`:

```
laboratory_assigned_experiments = {
        'exp1:dummy@Dummy experiments' : {
                'coord_address' : 'experiment1:laboratory1@core_host',
                'checkers' : ()
            },
        'exp1:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2_concurrent'
            },
        'exp2:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2_concurrent'
            },
        'exp3:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2_concurrent'
            },
        'exp4:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2_concurrent'
            },
        'exp5:electronics-lesson-1@Electronics experiments' : {
                'coord_address' : 'electronics:exp_process@exp_host',
                'checkers' : (),
                'api'      : '2_concurrent'
            },
    }
```

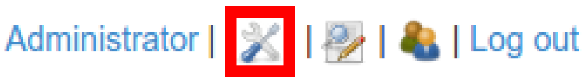
3.2.5 Step 4: Add the experiment server to the database and grant permissions

At this point, we have the Experiment server running, the Laboratory has registered the Experiment server and the Core server has registered that this experiment has an associated scheduling scheme (queue) and knows in which Laboratory server it is located.

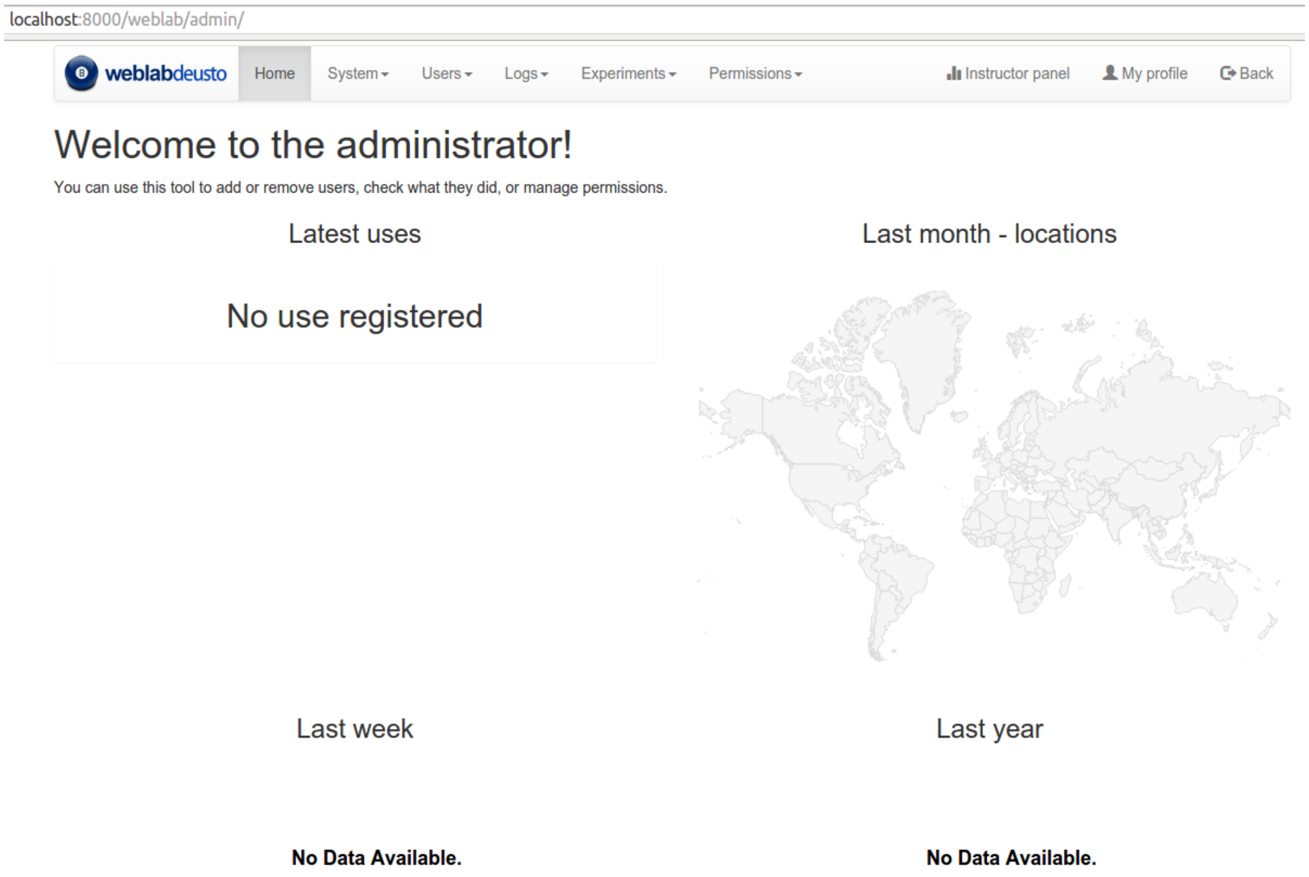
Now we need to make it accessible for the users. The first thing is to register the remote laboratory in the database. So, start the WebLab-Deusto instance:

```
$ weblab-admin start sample
```

Go to the administrator panel by clicking on the top right corner the following icon:



You will see this:



Welcome to the administrator!

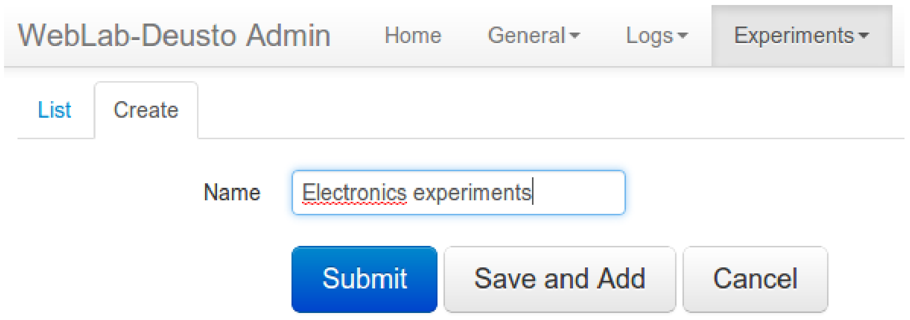
You can use this tool to add or remove users, check what they did, or manage permissions.

No use registered

No Data Available.

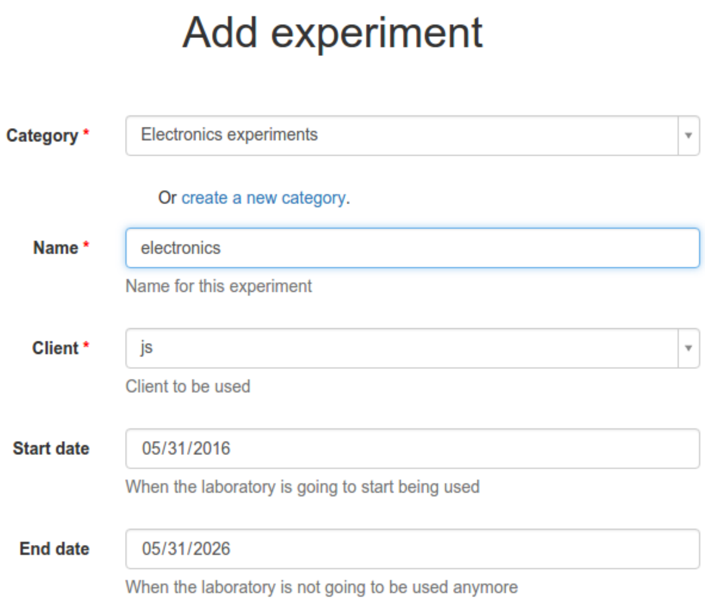
No Data Available.

On it, go to `Experiments`, then on `Categories`, and then on `Create`. You will be able to add a new category (if it did not exist), such as `Electronics experiments`, and click on Submit:



Name: Electronics experiments

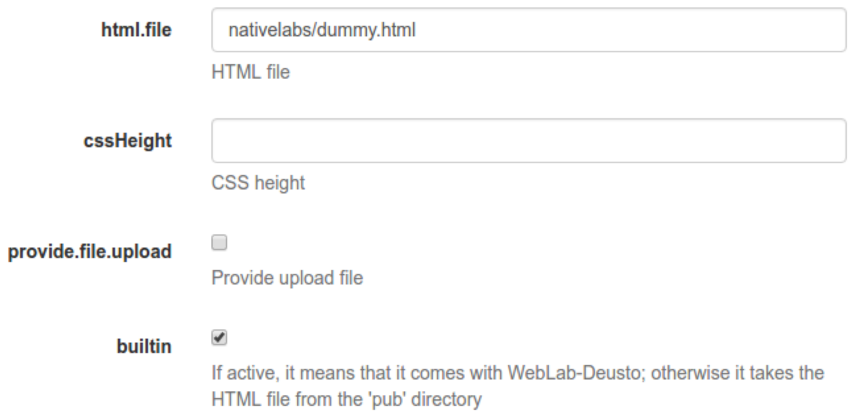
Then, go back to `Experiments`, then `Experiments`, and then on `Create`. You will be able to add a new experiment, such as `electronics`, using the category just created. The Start and End dates refer to the usage data. At this moment, no more action is taken on these data, but you should define since when the experiment is available and until when. For now, make sure that the `client` is `js`:



Add experiment

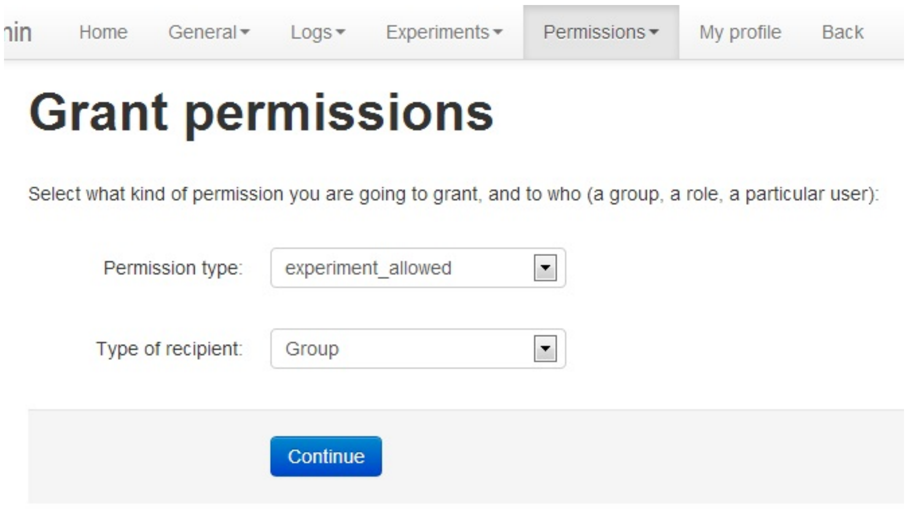
Category *	Electronics experiments
	Or create a new category.
Name *	electronics
	Name for this experiment
Client *	js
	Client to be used
Start date	05/31/2016
	When the laboratory is going to start being used
End date	05/31/2026
	When the laboratory is not going to be used anymore

And also make sure that later you select `builtin` and in `html.file` you type `nativelabs/dummy.html`:



html.file	nativelabs/dummy.html
	HTML file
cssHeight	
	CSS height
provide.file.upload	☐
	Provide upload file
builtin	✓
	If active, it means that it comes with WebLab-Deusto; otherwise it takes the HTML file from the 'pub' directory

Then click on `Save`. At this moment, the laboratory has been added to the database. Now you can guarantee the permissions on users. So as to do this, click on `Permissions`, `Create`. Select that you want to grant permission to a Group, of permission type `experiment_allowed`.

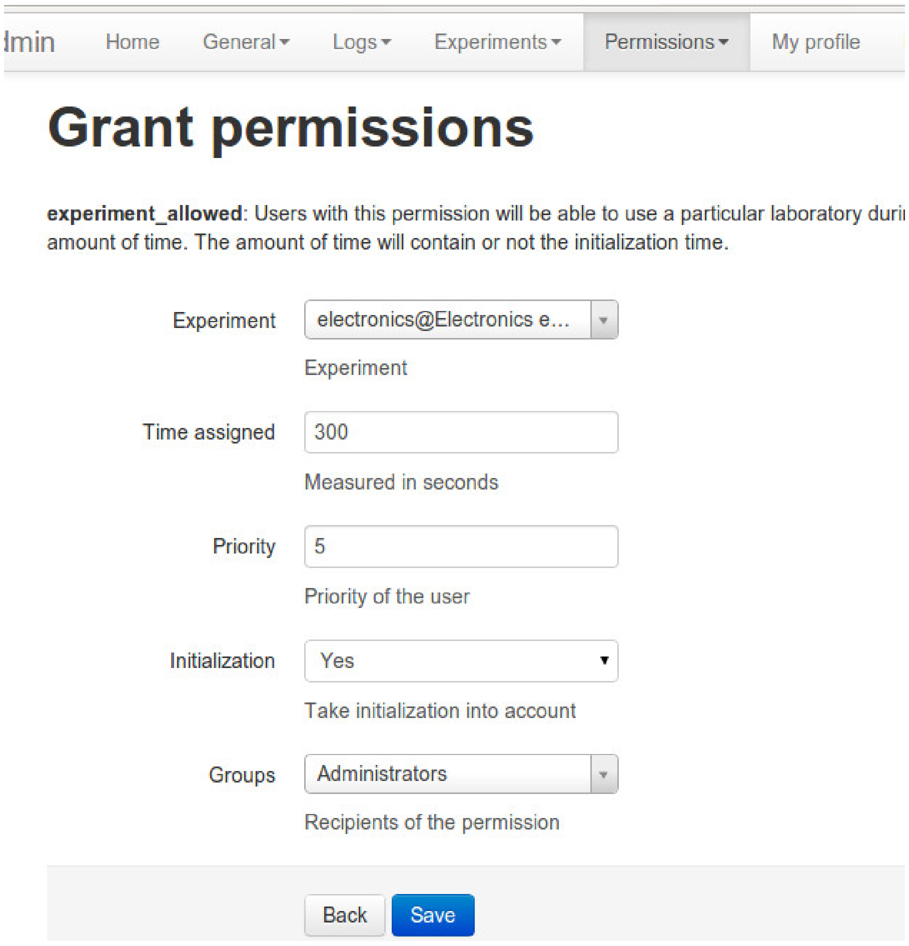


Grant permissions

Select what kind of permission you are going to grant, and to who (a group, a role, a particular user):

Permission type:	experiment_allowed
Type of recipient:	Group

And then you will be able to grant permissions on the developed laboratory to a particular group (such as Administrators):

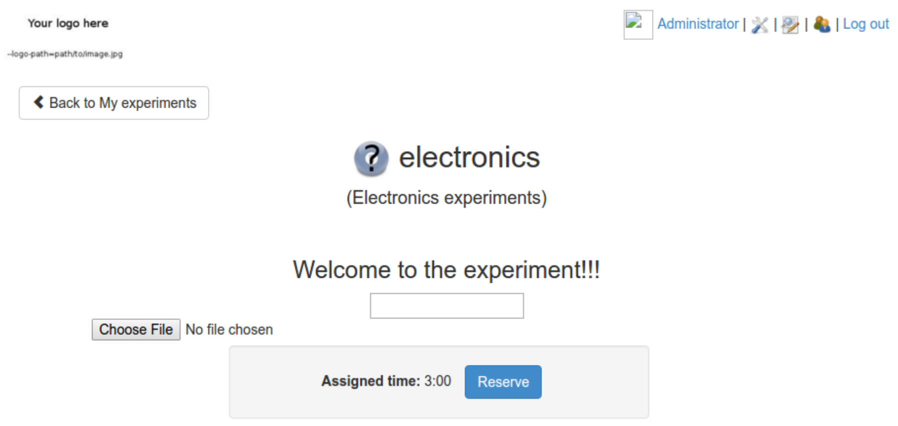


Grant permissions

experiment_allowed: Users with this permission will be able to use a particular laboratory duri
amount of time. The amount of time will contain or not the initialization time.

Experiment	electronics@Electronics e...
	Experiment
Time assigned	300
	Measured in seconds
Priority	5
	Priority of the user
Initialization	Yes
	Take initialization into account
Groups	Administrators
	Recipients of the permission

From this point, you will be able to use this experiment from the main user interface. If you see this:


Your logo here
--logo-path=path/to/image.jpg
Administrator | | | | Log out
Back to My experiments
electronics
(Electronics experiments)
Welcome to the experiment!!!
Choose File No file chosen
Assigned time: 3:00
Reserve

And once you click on reserve you're sending commands to the experiment and receiving them back, everything is fine.

However, you should create your own client, and you have to configure it in the page where you added the lab. You can edit it by going to `Experiments` and clicking on the edit button next to the lab you have just created. However, the particular configuration depends on the approach taken:

- If you are developing a managed laboratory (regardless if you are using Python or an XML-RPC experiment), jump to *Configuring the client in a managed laboratory*.
- If you are developing an unmanaged laboratory, jump to *Configuring the client in an unmanaged laboratory*.

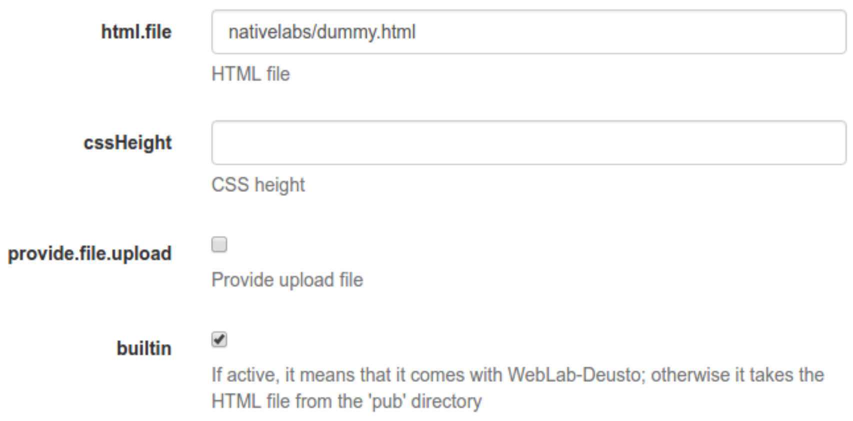
Configuring the client in a managed laboratory

We strongly encourage you to develop clients in JavaScript. However, we also support Adobe Flash (while most mobile devices do not support it) and Java applets (while most web browsers do not support them nowadays). This section explains how to modify the configuration to support the three options:

- *JavaScript*
- *Java applets*
- *Flash*

JavaScript

By default, in the previous steps we selected that the client would be `js`, which is fine if you are developing a JavaScript laboratory. However, we also selected the `builtin` option and the `nativelabs/dummy.html` html file:



html.file	nativelabs/dummy.html HTML file
cssHeight	 CSS height
provide.file.upload	☐ Provide upload file
builtin	✓ If active, it means that it comes with WebLab-Deusto; otherwise it takes the HTML file from the 'pub' directory

The `builtin` option (which by default is false) reports that the file (`nativelabs/dummy.html`) is provided by WebLab-Deusto, so it finds it in its local directories. However, when you create a WebLab-Deusto instance, there is a directory called `pub`. Whatever you put in this directory will be publicly available to the Internet on `/weblab/web/pub/`.

You can try to make a file called `example.txt` and put it in this directory. Going to http://localhost:12345/weblab/web/pub/example.txt or http://localhost/weblab/web/pub/example.txt (depending on if you're using Apache or the development server) should show you the file. Furthermore, in the Administration Panel, in `System` -> `Public directory` you can also modify the files (while this feature is only fully functional when using Apache).

Whenever you disable the `builtin` option, WebLab-Deusto will search for the file in this directory (unless in `html.file` you put something that starts by `http://` or `https://`, in which case the absolute url will be used; for example if you put the files in a different server).

So, for starting, the best option is to copy the `dummy.html` example to this directory. So you might go to dummy.html in github and click on the `Raw` button to access the raw file, and download to the `pub` directory.

Once downloaded, they will be in `/weblab/web/pub/dummy.html`. However, the internal includes to other JavaScript files will not work. In particular, the following code is not correct:

```
<script type="text/javascript" src="../js/jquery.min.js"></script>
<script type="text/javascript" src="../weblabjs/weblab.v1.js"></script>
```

Since those two directories (`../js/jquery.min.js`) do not exist anymore. So either you change it by an absolute URL:

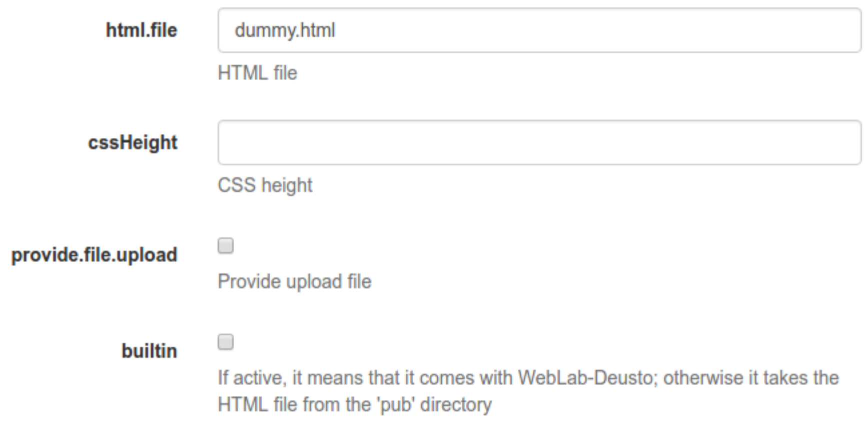
```
<script type="text/javascript" src="/weblab/static/js/jquery.min.js"></script>
<script type="text/javascript" src="/weblab/static/weblabjs/weblab.v1.js"></script>
```

or you replace it by a proper relative path:

```
<script type="text/javascript" src="../static/js/jquery.min.js"></script>
<script type="text/javascript" src="../static/weblabjs/weblab.v1.js"></script>
```

Note: This is assuming that you are locating `dummy.html` in the `pub` directory directly. If you move it to a directory inside `pub` (e.g., `electronics/dummy.html`), don't forget to modify the paths accordingly (e.g., `../../static...`) or use absolute ones.

Once you have changed those paths, you can safely edit the experiment in the Administration Panel. To do so, deactivate the `builtin` option and change the `html.file` to `dummy.html`:



html.file	dummy.html HTML file
cssHeight	 CSS height
provide.file.upload	☐ Provide upload file
builtin	☐ If active, it means that it comes with WebLab-Deusto; otherwise it takes the HTML file from the 'pub' directory

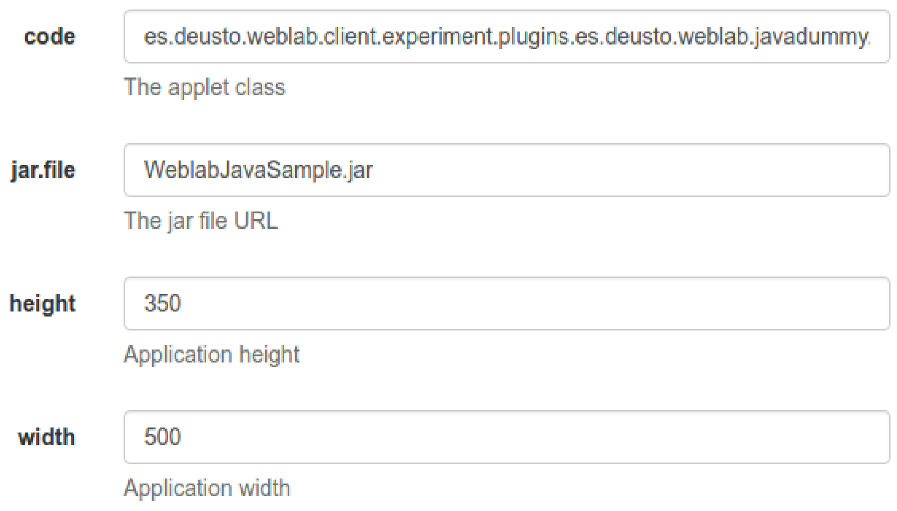
Now you can change the `dummy.html` or create other HTML from scratch and follow these steps to add it to the `pub` directory and use it in other laboratories. You can now go to the *Summary*.

Java applets

Nowadays most web browsers do not support Java applets. For this reason, we highly recommend not using Java applets for the development of remote laboratories. However, if you have a limited and controlled audience and an existing remote laboratory in Java, you can still use WebLab-Deusto.

First, you must compile the GWT client, as explained in gwt.

Then, you have to select the `java` client from the client list when editing an experiment. However, so as to load the laboratory, additional parameters must be configured, such as where is the JAR file, what class inside the JAR file must be loaded, and the size of the applet. An example of this configuration would be:



code	es.deusto.weblab.client.experiment.plugins.es.deusto.weblab.javadummy. The applet class
jar.file	WeblabJavaSample.jar The jar file URL
height	350 Application height
width	500 Application width

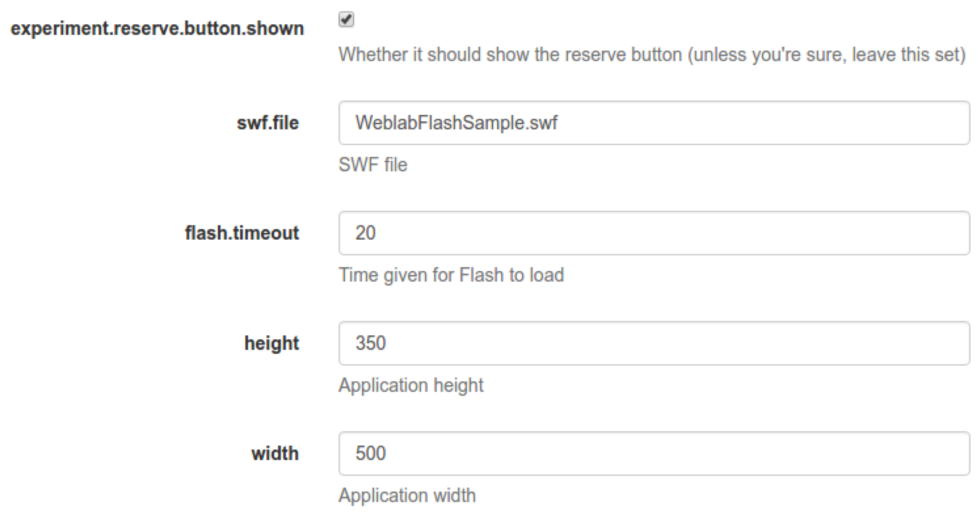
Those JAR files should be located in the `public` directory (see here), which will require you to re-compile and re-run the `setup` script.

Flash

Nowadays most mobile web browsers do not support Flash, and the Flash support is decreasing in regular web browsers. For this reason, we highly recommend not using Flash apps for the development of remote laboratories. However, if you have a limited and controlled audience and an existing remote laboratory in Flash, you can still use WebLab-Deusto.

First, you must compile the GWT client, as explained in gwt.

In the case of Flash applications, the client from the list must be `flash`. However, so as to load a particular laboratory, some additional parameters must be configured, such as where is the SWF file, the size of the application, or the maximum time that WebLab-Deusto will wait to check if the Flash applet has been connected -e.g., 20 seconds-, since sometimes the user uses a flash blocking application or a wrong version of Adobe Flash. An example of this configuration would be:



experiment.reserve.button.shown	✓	Whether it should show the reserve button (unless you're sure, leave this set)
swf.file	WeblabFlashSample.swf	SWF file
flash.timeout	20	Time given for Flash to load
height	350	Application height
width	500	Application width

Those SWF files should be located in the `public` directory (see here), which will require you to re-compile and re-run the `setup` script.

Configuring the client in an unmanaged laboratory

In the case of the unamanged laboratories the process is very simple. When editing the experiment in the administration panel as detailed above, simply select the `redirect` client:

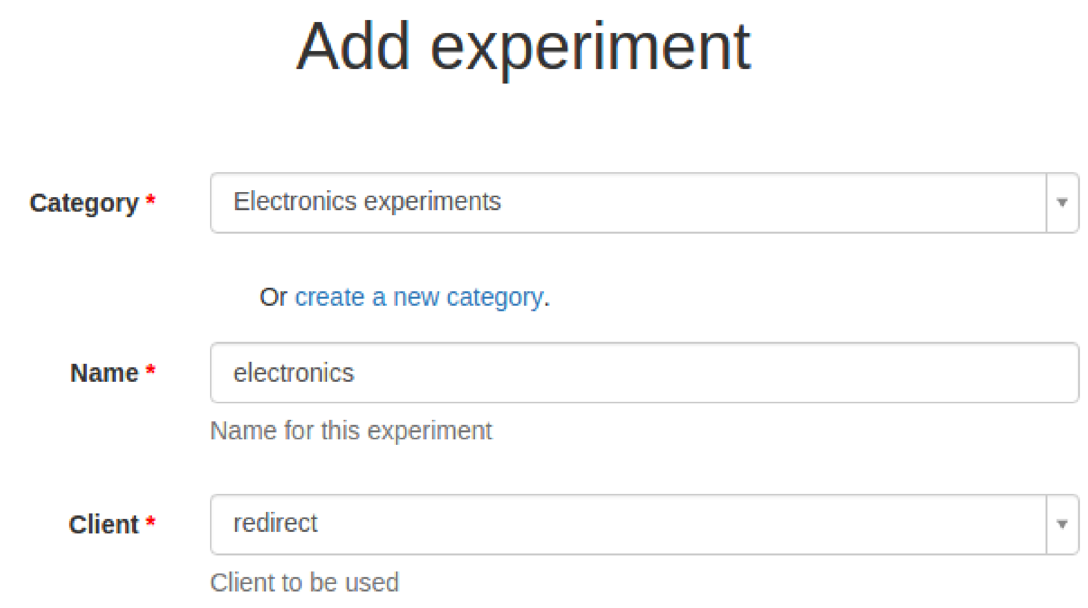
This way, whenever the user can access the laboratory, it will be redirected to it automatically. You don't need to deal with any client-side code.

3.2.6 Summary

Congratulations! WebLab-Deusto requires four actions to add a new experiment, explained in this section and on this figure:

These four actions are registering the new Experiment server, modifying the configuration of the Laboratory server and the Core server and adding the experiment to the database using the Admin panel.

After doing this, you may start sharing your laboratories with other WebLab-Deusto deployments, as stated in the *following section*.



Add experiment

Category * Electronics experiments

Or create a new category.

Name * electronics

Name for this experiment

Client * redirect

Client to be used

Fig. 11: Configure the experiment as `redirect`.

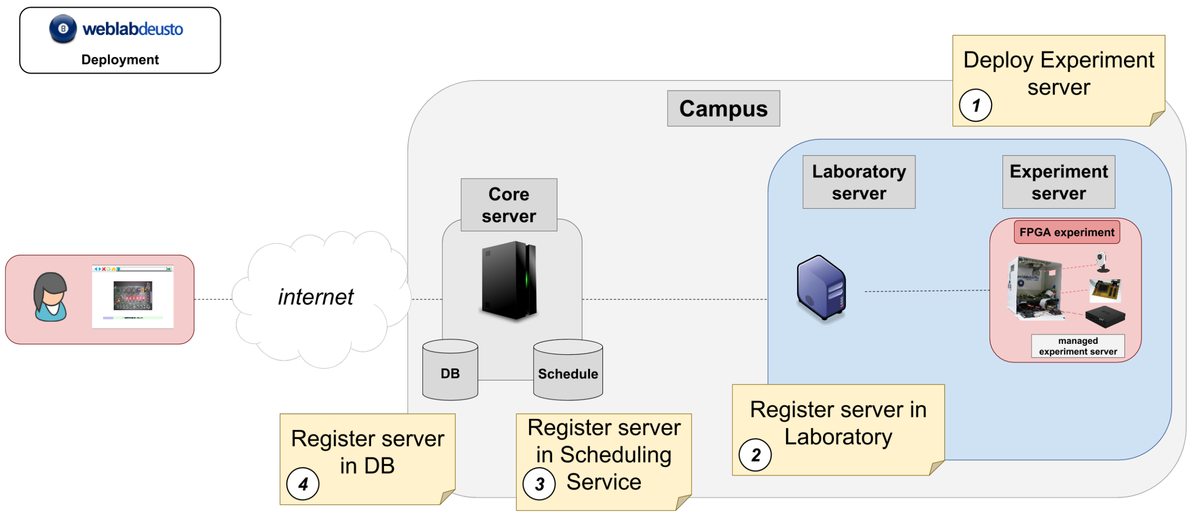


Fig. 12: Steps to deploy a remote laboratory in WebLab-Deusto.

3.3 Remote laboratory sharing

Table of Contents

3.3.1 Introduction

WebLab-Deusto supports *federation*. This means that one WebLab-Deusto instance can share its laboratories with other instances, as well as consume them.

Let's imagine that there are two universities, `UniA` and `UniB`. If `UniB` is a provider, it will have registered one special type of user (e.g., `uni-b`, with role `federated` (instead of `student` or `administrator`). `UniB` will guarantee permissions to this user as if it was any other type of user. It can even be part of a group. For example, it may grant access to this user to an experiment called `exp1`.

Then, `UniA` can configure that it will use `UniB` with that user (and password) only to access `exp1`. It may also define that `exp1` at `UniB` is called `experiment1` in `UniA`, so the name does not really matter. Furthermore, `UniA` can re-share `exp1` to a third University, called `UniC`, using the same approach (creating a new user, etc.).

During the entire process, `UniB` will not need to know who are those students coming from `UniA`, since `UniB` trusts `UniA` and `UniA` trusts those students. Also, the consumer system, once the user has finished, will be able to know what the user did, so if the administrator goes to the stored Logs in the administration panel, he will see what commands were sent.

In this case, `UniA` is acting as a consumer, and `UniB` as a provider. It is also common that both act as consumers and providers at the same time, sharing different laboratories each other. If it happens that both have copies of the same laboratory (e.g., the VISIR remote laboratory is available in at least 6 universities in Europe), they can even define that they will use the other system whenever their local resources are full. If `UniA` has 3 copies and `UniB` has 2 copies, and 6 students come in any of them, the sixth student will be waiting for any of the 5 students to finish their session.

Since all the relations are defined as users, the administrators can also change the priority in the queue. For instance, this enables that `UniB` defines that, in case of queue, their students will go first (and those from `UniA` later).

This section explains the technical details of how to do this. We will assume that the experiment in `UniB` is called `visir@Visir experiments`, and in `UniA` we want to call it `ud-electronics@Electronics experiments`.

Warning: If you want to use this (or any of the WebLab-Deusto laboratories), *contact us*, and we will create you an account for your university. Please do not use the `weblabfed` user for anything but testing.

3.3.2 Consuming other remote laboratories

So as to consume a remote laboratory, the first step is that the external system creates a federated user. So as to test this, we will use the WebLab-Deusto system in production with the following public credentials: `weblabfed` and `password`. If you go there with your web browser, you will see the laboratories available for that account, and you can even use them as a regular user.

Then, the process is quite similar to *deploying a new laboratory*, but without interacting with the Laboratory server or the Experiment server, since they are already configured in the provider system.

Basically, we have to:

1. *Registering a scheduling system for the experiment*
2. *Add the experiment server to the database and grant permissions*

Note: As in all the steps that require changing the configuration of the server, you will need to restart the WebLab-Deusto instance after applying all the changes.

Registering a scheduling system for the experiment

We have to configure the Core server to manage this remote laboratory. As explained in *Step 3: Registering a scheduling system for the experiment*, the entire configuration of the Core server related to scheduling is by default in the `core_host_config.py` file. It is placed there so if you have 4 Core servers in different instances (*which is highly recommended*), you have the configuration in a single location. In this file, you will find information about the database, the scheduling backend, etc.

There is one variable called `core_scheduling_systems`, which by default is as follows:

```
core_scheduling_systems = {
        'dummy_queue'       : ('PRIORITY_QUEUE', {}),
        'robot_external'    : weblabdeusto_federation_demo,
}
```

There, we have to add a new scheduler called `external_electronics`. We can do it directly:

```
core_scheduling_systems = {
        'dummy_queue'       : ('PRIORITY_QUEUE', {}),
        'robot_external'    : weblabdeusto_federation_demo,
        'external_electronics' : ('EXTERNAL_WEBLAB_DEUSTO', {
                                  'baseurl' : 'https://weblab.deusto.es/weblab/',
                                  'username' : 'weblabfed',
                                  'password' : 'password',
                                  'experiments_map' : {'ud-electronics@Electronics␣
↪experiments' : 'visir@Visir experiments'}
                         })
}
```

Or, more commonly, create other variable for that:

```
electronics_federation = ('EXTERNAL_WEBLAB_DEUSTO', {
                                  'baseurl' : 'https://weblab.deusto.es/weblab/',
                                  'username' : 'weblabfed',
                                  'password' : 'password',
                                  'experiments_map' : {'ud-electronics@Electronics␣
↪experiments' : 'visir@Visir experiments'}
```

(continues on next page)

(continued from previous page)

```
                        })


core_scheduling_systems = {
        'dummy_queue'       : ('PRIORITY_QUEUE', {}),
        'robot_external'    : weblabdeusto_federation_demo,
        'external_electronics' : electronics_federation,
}
```

There, what we are detailing is that the scheduler identified by `external_electronics` will rely on the external server with the URL and credentials defined in the other variable. Note that there is a variable called `experiments_map`, which maps local names with names in the foreign system. In this case, we are defining that when using this scheduler for the local `ud-electronics@Electronics experiments`, it will instead call the foreign system asking for `visir@Visir experiments`. If this variable is not provided or is empty (`{}`), it will simply ask for the same name as local (in this case, it would call `ud-electronics@Electronics experiment`, which would not exist in the foreign system).

Now we have to register that we actually want to use this scheduler. For local experiments, there is a local variable explained in *Step 3: Registering a scheduling system for the experiment*, which defines which Laboratory servers manage which Experiment servers:

```
core_coordinator_laboratory_servers = {
    'laboratory1:laboratory1@core_machine' : {
            'exp1|dummy|Dummy experiments' : 'dummy1@dummy_queue',
        },
}
```

However, in the federated environment, there is no such concept, since this mapping is already managed by the remote system. What we need is to use other variable as follows:

```
core_coordinator_external_servers = {
    'external-robot-movement@Robot experiments'     : [ 'robot_external' ],
    'ud-electronics@Electronics experiments'    : [ 'external_electronics' ],
}
```

This is basically defining that the `ud-electronics@Electronics experiments` will be managed by the scheduler `external_electronics` that we just defined.

Note: This configuration maps an identifier to a *list* of schedulers. This means that you can add multiple scheduler if the particular laboratory was deployed in more than one system. For instance, it could define:

```
core_coordinator_external_servers = {
    'external-robot-movement@Robot experiments'     : [ 'robot_external' ],
    'ud-electronics@Electronics experiments'    : [ 'electronics-deusto', 'electronics-
↪uned' ],
}
```

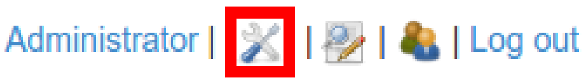
And your system will use both universities (as long as you have the credentials for both configured in the schedulers variable).

Furthermore, this configuration is not incompatible with local laboratories. If you had the `core_coordinator_laboratory_servers` configured using the same identifier pointing to a local scheduler, the system will use first the local resources, and if they are in a queue it will use the remote resources. This is how you can implemented distributed load balancing.

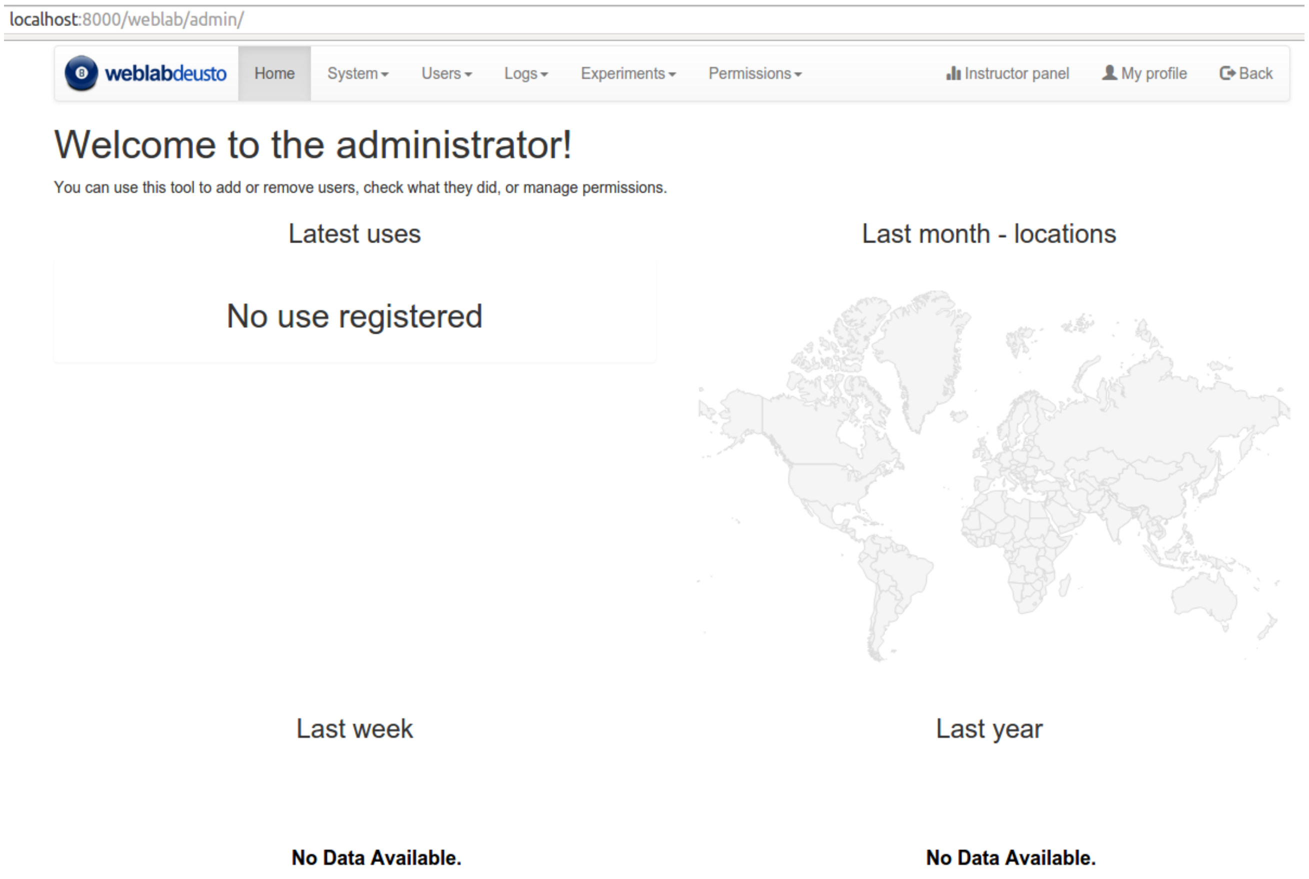
Add the experiment server to the database and grant permissions

The last step is to add the laboratory to the database and grant permissions to the students. This process is exactly the same as defined in *Step 4: Add the experiment server to the database and grant permissions*.

Go to the administrator panel by clicking on the top right corner the following icon:



You will see this:



Welcome to the administrator!

You can use this tool to add or remove users, check what they did, or manage permissions.

Latest uses

No use registered

Last month - locations

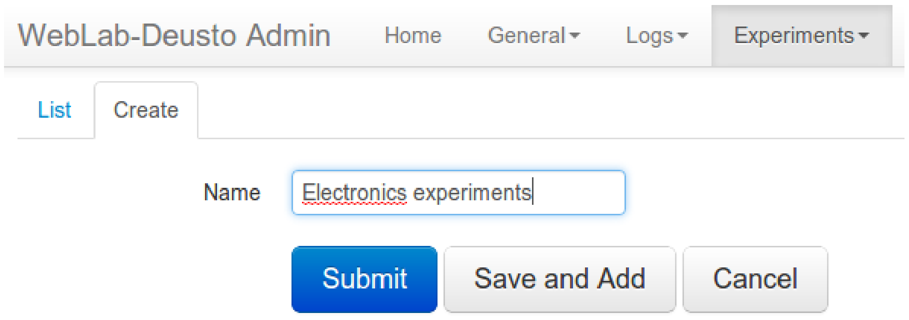
Last week

No Data Available.

Last year

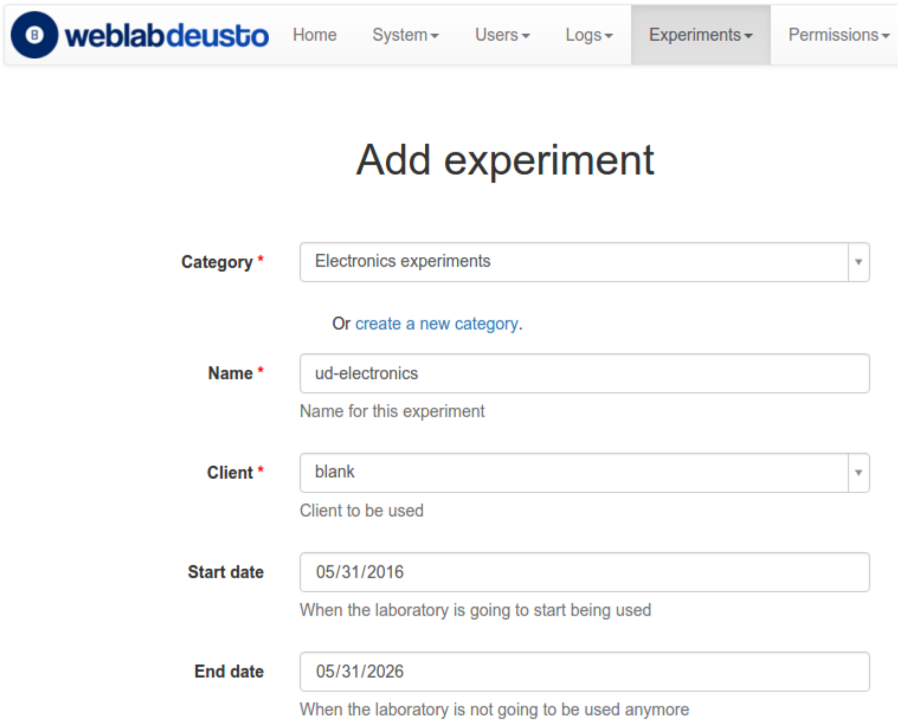
No Data Available.

On it, go to `Experiments`, then on `Categories`, and then on `Create`. You will be able to add a new category (if it did not exist), such as `Electronics experiments`, and click on Submit:



Name: Electronics experiments

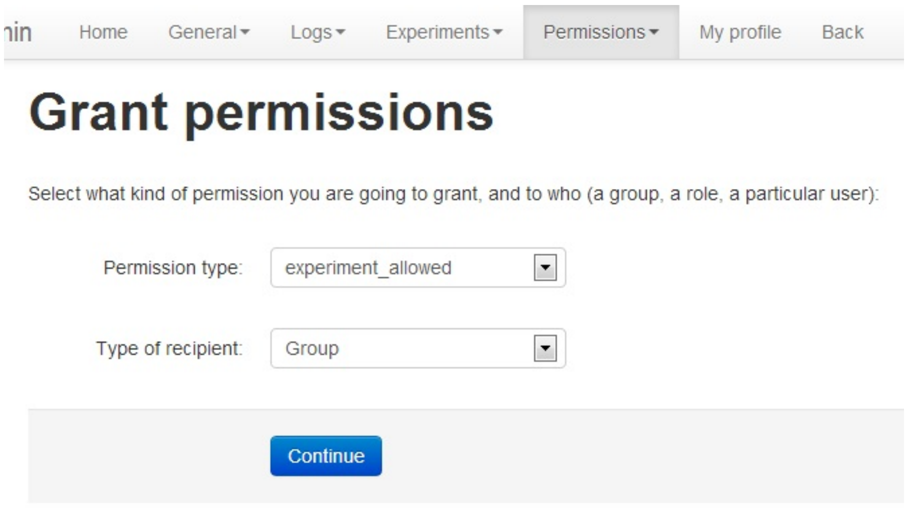
Then, go back to `Experiments`, then `Experiments`, and then on `Create`. You will be able to add a new experiment, such as `ud-electronics`, using the category just created. The Start and End dates refer to the usage data. At this moment, no more action is taken on these data, but you should define since when the experiment is available and until when. You can provide your own client if you want to provide further instructions in the beginning, but typically here you will want to leave the `blank` client:



Add experiment

Category *	Electronics experiments
	Or create a new category.
Name *	ud-electronics
	Name for this experiment
Client *	blank
	Client to be used
Start date	05/31/2016
	When the laboratory is going to start being used
End date	05/31/2026
	When the laboratory is not going to be used anymore

At this moment, the laboratory has been added to the database. Now you can guarantee the permissions on users. So as to do this, click on `Permissions`, `Create`. Select that you want to grant permission to a Group, of permission type `experiment_allowed`.

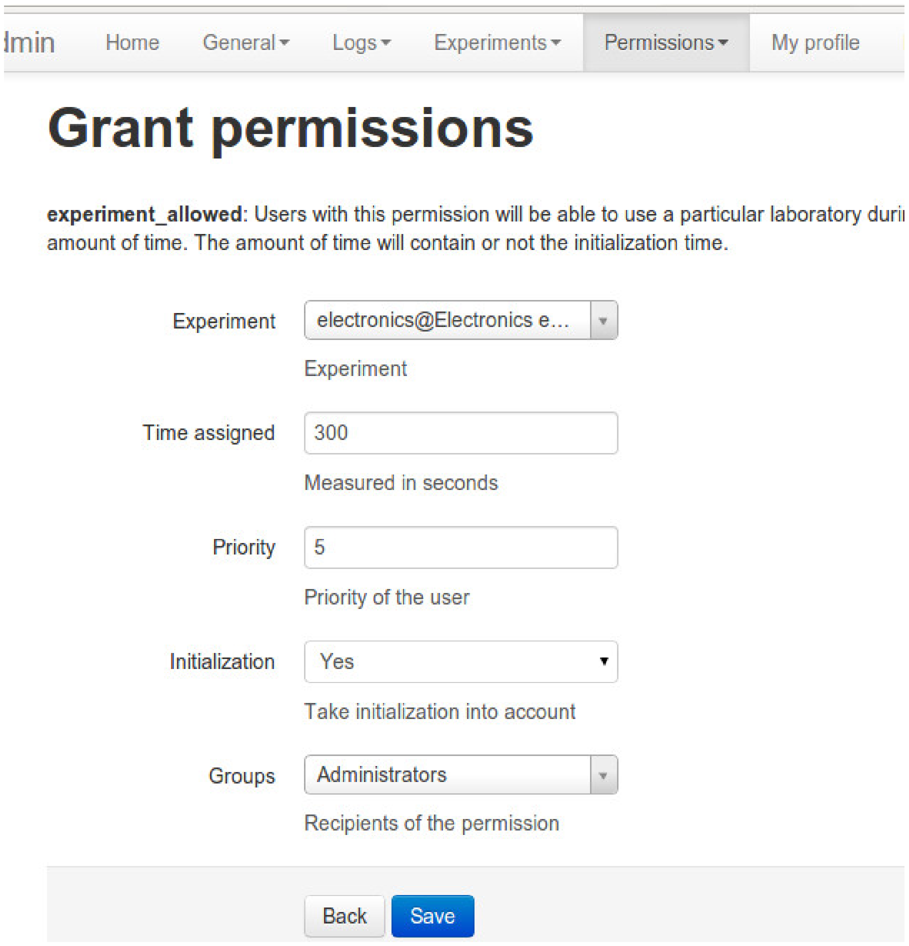


Grant permissions

Select what kind of permission you are going to grant, and to who (a group, a role, a particular user):

Permission type:	experiment_allowed
Type of recipient:	Group

And then you will be able to grant permissions on the developed laboratory to a particular group (such as Administrators):



Grant permissions

experiment_allowed: Users with this permission will be able to use a particular laboratory duri
amount of time. The amount of time will contain or not the initialization time.

Experiment	electronics@Electronics e...
	Experiment
Time assigned	300
	Measured in seconds
Priority	5
	Priority of the user
Initialization	Yes
	Take initialization into account
Groups	Administrators
	Recipients of the permission

From this point, you will be able to use this experiment from the main user interface.

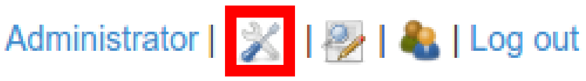
3.3.3 Sharing your remote laboratories

Sharing a WebLab-Deusto laboratory is much easier than consuming one. You only need two steps:

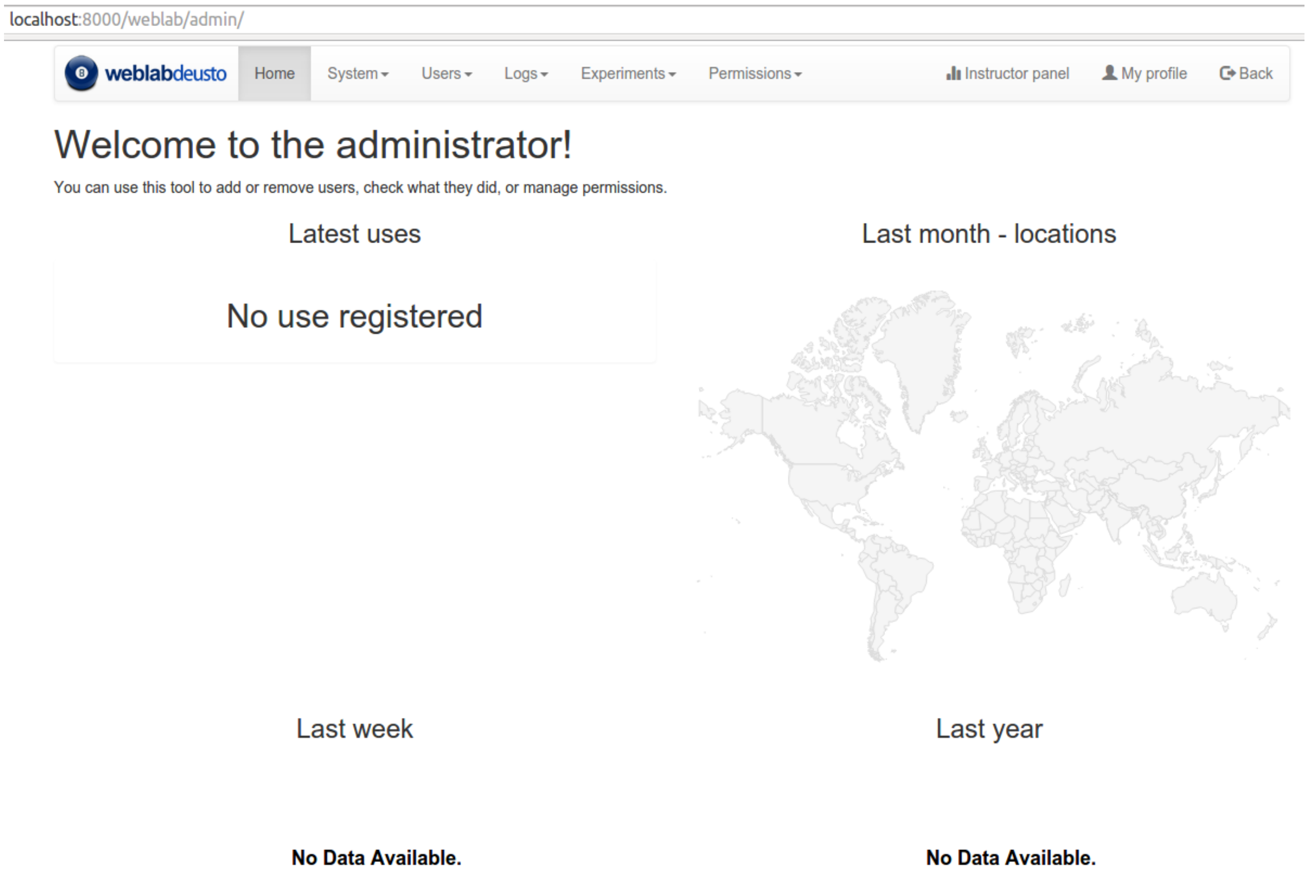
1. *Create a user for the foreign entity*
2. *Grant permissions on that laboratory to this entity*

Create a user for the foreign entity

Go to the administrator panel by clicking on the top right corner the following icon:



You will see this:



Welcome to the administrator!

You can use this tool to add or remove users, check what they did, or manage permissions.

Latest uses

No use registered

Last month - locations

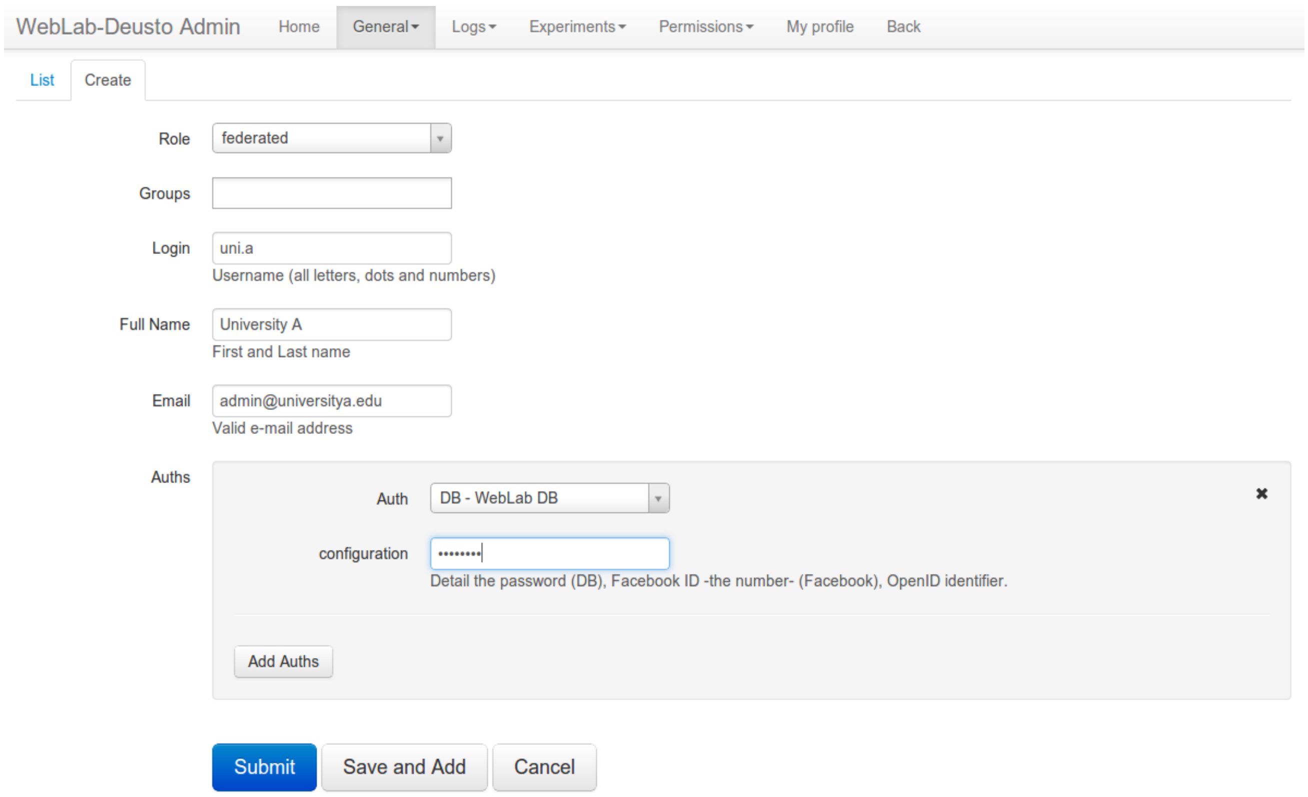
Last week

No Data Available.

Last year

No Data Available.

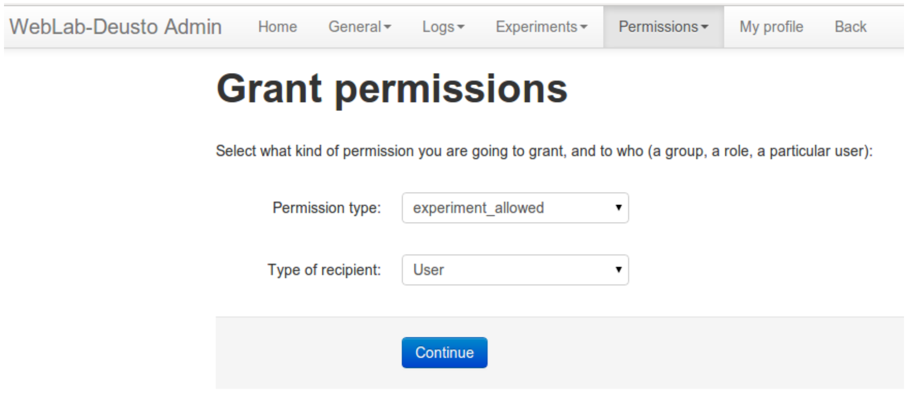
There, go to `General` -> `Users` -> `Create`, and add a user using the Role `federated`, and providing a password (click on "Add Auths" and select `DB`):



Role	federated
Groups	
Login	uni.a
	Username (all letters, dots and numbers)
Full Name	University A
	First and Last name
Email	admin@universitya.edu
	Valid e-mail address
Auths	
Auth	DB - WebLab DB
configuration	••••••••
	Detail the password (DB), Facebook ID -the number- (Facebook), OpenID identifier.

Grant permissions on that laboratory to this entity

Still in the administration panel, click on `Permissions`, `Create`. Select that you want to grant permission to a User instead of a group, of permission type `experiment_allowed`.

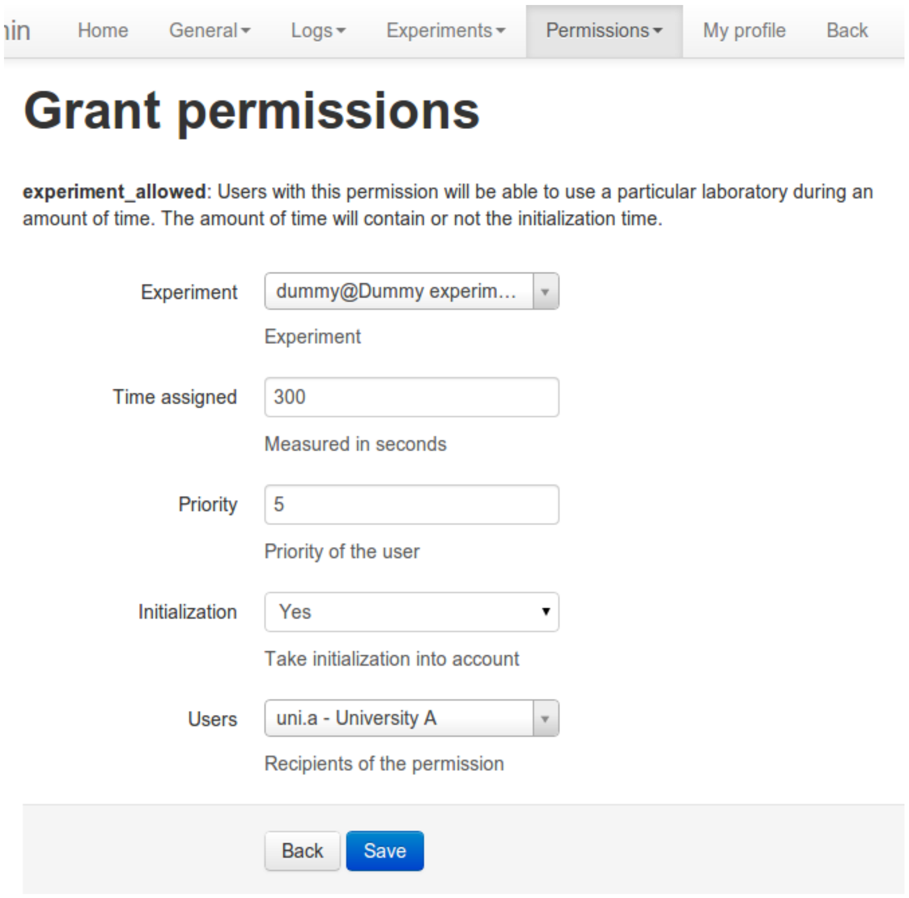


Grant permissions

Select what kind of permission you are going to grant, and to who (a group, a role, a particular user):

Permission type:	experiment_allowed
Type of recipient:	User

Then, select the laboratory you wish to grant access to, select the user, and select the rest of the arguments.



Grant permissions

experiment_allowed: Users with this permission will be able to use a particular laboratory during an amount of time. The amount of time will contain or not the initialization time.

Experiment	dummy@Dummy experim...
	Experiment
Time assigned	300
	Measured in seconds
Priority	5
	Priority of the user
Initialization	Yes
	Take initialization into account
Users	uni.a - University A
	Recipients of the permission

You may for instance establish that you allow 3600 seconds (1 hour) to the laboratory, but then the consumer side system may establish that one particular group will only have permission to use it for 10 minutes. However, the opposite is not possible, since even if the consumer system establishes that they can use it for one hour, when the consumer system contacts the provider system, it will define that they only have permissions for 10 minutes.

3.3.4 Exchanging or selling accesses to your labs on LabsLand

LabsLand enables you to exchange lab accesses with other institutions, for free or for profit. LabsLand is the spin-off of the WebLab-Deusto project, and it aims to support people to consume remote lab accesses in a daily basis. To this end, it provides a marketplace for both providers and consumers of remote laboratories, where remote laboratory providers can sell or share accesses to their remote laboratories, and consumers will have a clear idea of what they can really consume, and at what cost.

If you develop a remote laboratory with WebLab-Deusto, we encourage you to access LabsLand and joining the network.

CHAPTER 4

External systems

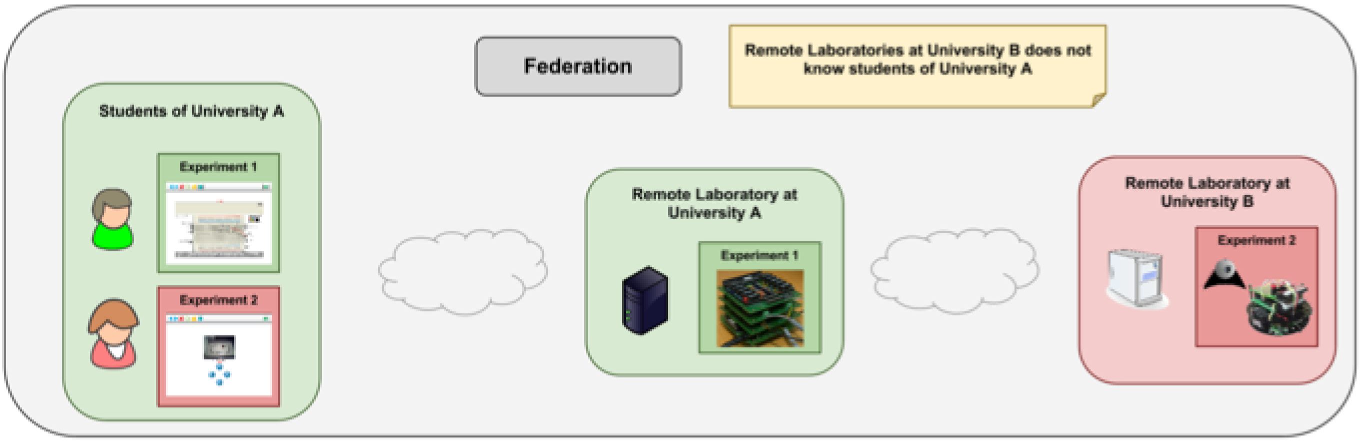
This section is intended for people who is going to integrate WebLab-Deusto in external systems, such as Learning Management Systems, Content Management Systems, OAuth providers or similar.

4.1 External tools integration through federation

The easiest way to make it work from external tools is using the federation API. We currently provide APIs for various programming languages (.NET, PHP, Python), and there are external efforts for other languages (Ruby).

4.1.1 Background

As described in the *federation section*, WebLab-Deusto supports federation. This means that two WebLab-Deusto systems can exchange remote labs without dealing with particular users, as seen in the following figure:

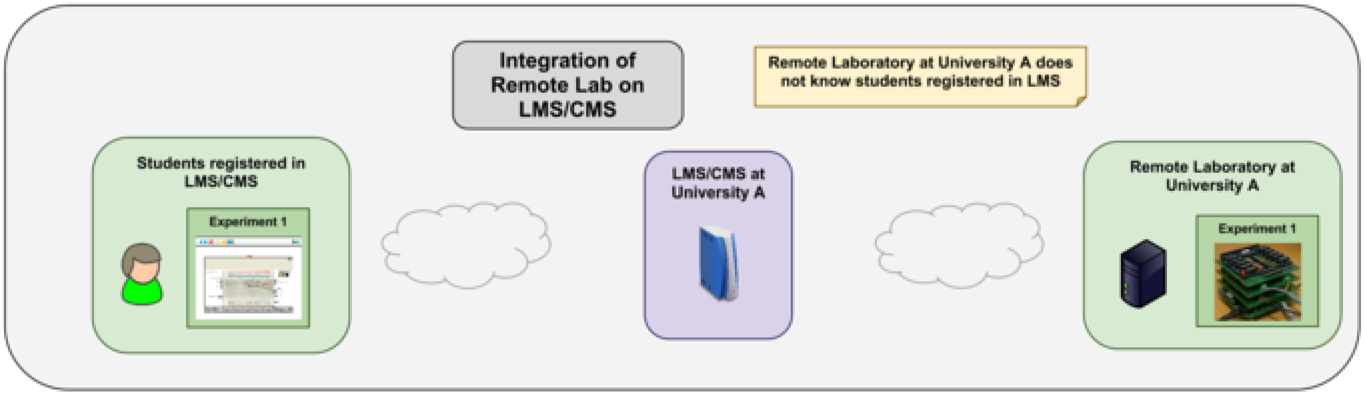

Federation
Remote Laboratories at University B does not know students of University A
Students of University A
Experiment 1
Experiment 2
Remote Laboratory at University A
Experiment 1
Remote Laboratory at University B
Experiment 2

This basically means that in the example, the system deployed in `University A` manages authentication and authorization, so its the one who knows who are the users, etc. The contract between the systems of `University`

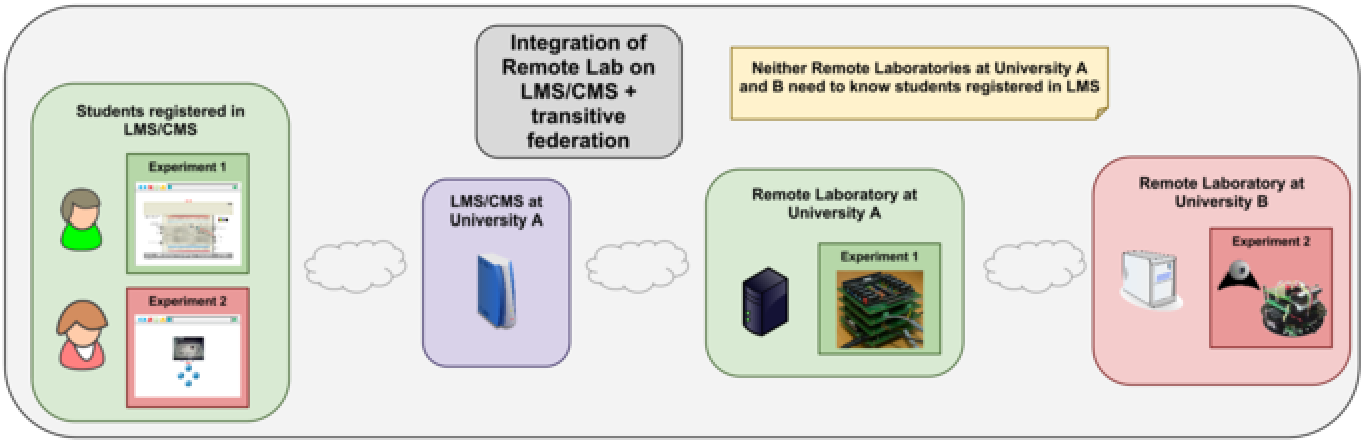
A and University B does not need to deal with users or groups: if University A can access three different laboratories, then it's University A who must choose which users can access each of these laboratories.

However, this idea is essentially what is aimed when integrating a remote laboratory in a Learning or Content Management System (LMS/CMS). It is the LMS the one that chooses what authentication mechanisms must be used, what students are in which courses, and which courses should use which laboratories.

Therefore, using the federation APIs enable this second scheme, where the LMS/CMS is basically a WebLab-Deusto consumer, and uses the same federation protocol that other LMS/CMS use:


Integration of Remote Lab on LMS/CMS
Remote Laboratory at University A does not know students registered in LMS
Students registered in LMS/CMS
Experiment 1
LMS/CMS at University A
Remote Laboratory at University A
Experiment 1

If the federation protocol does not support transitivity (this is, a consumer may not re-share the laboratories to a third-party consumer), the LMS/CMS would need to be configured with each external remote laboratory, which is not an ideal situation. Typically, remote laboratory managers are the ones who deal with this type of contracts. However, WebLab-Deusto supports transitivity (see the *federation section*), so it is possible to have a local WebLab-Deusto instance, configure it in the LMS, and configure in this WebLab-Deusto instance as many external federated instances as required.


Integration of Remote Lab on LMS/CMS + transitive federation
Neither Remote Laboratories at University A and B need to know students registered in LMS
Students registered in LMS/CMS
Experiment 1
Experiment 2
LMS/CMS at University A
Remote Laboratory at University A
Experiment 1
Remote Laboratory at University B
Experiment 2

4.1.2 How does it work?

Let's take the example of the .NET consumer. It provides a set of data classes, but the main class is *WebLabDeusto-Client*. Once you create an instance of it, you can create a session by passing custom credentials:

```
WebLabDeustoClient weblab = new WebLabDeustoClient("http://localhost/weblab/");

SessionId sessionId = weblab.Login("user", "password");
```

And you can use this session identifier to retrieve the list of available laboratories:

```
foreach(ExperimentPermission permission in weblab.ListExperiments(sessionId))
    Console.WriteLine("I have permission to use {0} of category {1} during {2} seconds
↪", permission.Name, permission.Category, permission.AssignedTime);
```

Which will print in the console something like:

```
I have permission to use ud-logic of the category PIC experiments during 150 seconds
I have permission to use submarine of the category Aquatic experiments during 150␣
↪seconds
[...]
```

In addition, and more importantly, you can use the session identifier to perform a reservation. For instance, if you want to create a reservation of the `ud-logic` laboratory, you can provide the following the laboratory and the category:

```
Reservation reservation = weblab.ReserveExperiment(sessionId, "ud-logic", "PIC␣
↪experiments", consumerData);
```

Now, the fourth argument is `consumerData`, which represents additional information that the consumer system (e.g., a LMS) will provide. This includes statistics information like the user-agent (i.e. what web browser is the student using?), the referer (i.e. where did he come from?) or the IP address, but also information about the reservation itself: who is the user, what is the maximum time that he will have for the laboratory (e.g., the consumer may have 150 seconds, but still the consumer can restrict it to 50 seconds for a group of students), or a certain priority:

```
//            consumerData["user_agent"]    = "";
//            consumerData["referer"]       = "";
//            consumerData["mobile"]        = false;
//            consumerData["facebook"]      = false;
//            consumerData["from_ip"]       = "...";

//
// Additionally, the consumerData may be used to provide scheduling arguments,
// or to provide a user identifier (that could be an anonymized hash).
//
            consumerData["external_user"]                = "an_external_user_
↪identifier";
//            consumerData["priority"]                     = 3; // the lower, the␣
↪better
//            consumerData["time_allowed"]                 = 100; // seconds
//            consumerData["initialization_in_accounting"] = false;
```

Finally, the consumer will generate a URL that can safely be forwarded to the student. It includes a reservation identifier, which can only be used for actions related to that reservation. For instance, the student can not use that reservation_id to obtain the list of laboratories or create new reservations:

```
Console.WriteLine(reservation);

string url = weblab.CreateClient(reservation);

Console.WriteLine(url);
```

4.2 lms4labs

lms4labs is a joint open effort towards creating a generic tool for integrating different remote laboratories in different

Learning Management Systems. At this moment, contributors are from the University of Deusto, Massachusetts Institute of Technology and UNED.

The system right now works with WebLab-Deusto, and ongoing work is focused on supporting MIT iLabs. In the current version, it supports Moodle 1.9, Moodle 2.x and .LRN, but it is easily extensible for other management systems. Right now, ongoing work is focused on supporting the IMS LTI standard.

Refer to the lms4labs documentation for further information. The approach taken from the WebLab-Deusto perspective, is the one described in the *federation for external tools section*.

4.3 Other approaches

So as to consume WebLab-Deusto laboratories from external tools, the best and recommended way is to use the *federation system*. However, it is also possible to support external tools through other schemes, such as OpenID or OAuth 2.0 for authentication. Documentation regarding these systems is available in the *authentication section*.

CHAPTER 5

WebLab-Deusto Development

This section is intended for people who is going to contribute to the WebLab-Deusto system itself, rather than remote laboratories on top of it.

5.1 Contribute

Table of Contents

5.1.1 Introduction

This section is focused on enabling external people to contribute the project. WebLab-Deusto is an Open Source and non for profit project. Help us to improve the system. You do not need to be a software developer to contribute!

5.1.2 Documentation

Documentation is not as maintained as we would like. However, in every page in this documentation you'll see a `Show source` button that redirects you to this page in Github. If you find something that could be better described in a different way, feel free to click there or go to our github doc repository, and then click on edit (given a file, search for "History" and next to it you'll find the Edit button). You will need to create a github account first and be logged in with that account. Once you do it and save changes, we are notified and can apply the changes (but it will be clearly acknowledged in github that you're the person doing the change).

5.1.3 Translations

Translations are very welcome, and nowadays it's pretty simple to contribute a translation. In the github repository, you may go to the:

```
client/src/es/deusto/weblab/client/i18n
```

directory and find a file called `IWebLabI18N.properties`. If you're in github, click on `Raw` to see the file and you can save it. Make sure that the extension of the file is `.properties` when you download it (and not `.txt`). This is very important in Microsoft Windows, where certain browsers will change the extension calling it `.properties.txt`: if you double click the file and it opens it with the text editor automatically, you should change the extension. The other approach is to download the whole repository (you may have done it before) as detailed in *Download using git*. Then, the file is located in the directory explained above, with the proper `.properties` extension.

Once you have the file, you may use the Google Translator Toolkit. You should take the file, open it, replace all the '' by '. Then, you should go to the Google Translator Toolkit and upload the file. You will be able to select the original language (english) and the target language (the one you want to translate it to). Then, it will show you an interactive environment where Google has tried to translate most of the sentences. Many of them will be wrong, but it is much easier to correct the file than to start from scratch. Furthermore, the tool is collaborative, so you may add other translators and split the sentences among them.

Then, please submit the file to the WebLab-Deusto developers (*Contact*) to incorporate it to the project.

5.1.4 Issue reporting

WebLab-Deusto has bugs (it may work wrong in certain circumstances), as well as many things to improve in many ways. If you find a bug, or if you think of particular things that should be changed (e.g., *I miss a documentation page for this*, *this tool is not generating what I expect here*, *I would like to be able to do this*), please, tell us, we are eager to hear you.

You may do this in public by reporting an issue in our issue tracker, which is in github. Or if you prefer doing this in private, just *contact us*.

5.1.5 Bug fixes

If you find a bug, and you think you can fix it, you can do three things:

- Just publish it in the public mailing list or notify the developers (*Contact*).
- Create an account in *github <http://github.com/>*, fork the project by clicking on `Fork`, and in your copy of the project, modify whatever you need. Then, create a pull request through github. We will be notified, review the code and apply the changes.

5.1.6 New functionalities

If you want to create a new functionality not present in WebLab-Deusto, you are very welcome. Feel free to discuss it with us in the mailing lists, or do a prototype. Also, refer to the *WebLab-Deusto development* section.

5.1.7 Add remote laboratories to the network

You have made a super cool remote lab using WebLab-Deusto? Please, *contact us* to do any (or all) of the following:

- Add the code to the WebLab-Deusto repository. We will make sure that if we change anything, the laboratory is still compliant.
- Advertise it in the documents.
- Share it with other universities and schools.
- Add it to the demo account in the main WebLab-Deusto repository.
- Add it to the default account created when you create a new WebLab-Deusto repository.

5.2 WebLab-Deusto development

Table of Contents

5.2.1 Introduction

This section covers documentation about how to work on the WebLab-Deusto development. If you want to develop a remote laboratory instead of working in the middle layers, go to the *Remote laboratory development* section.

5.2.2 Setting up the development environment

This section assumes that you have successfully used the steps refered in the *Installation: further steps* section.

Server side

When developing the server side, it is best to create a new environment on which WebLab-Deusto is not deployed. To do so, create a new virtualenv as explained in *Installation*, and install all the requirements, but do not run the `python setup.py install` command.

So as to deploy the testing database (required to pass the tests), you need to run the following in the `weblab/server` directory:

```
python develop.py --deploy-test-db --db-engine=mysql --db-create-db --db-ask-admin-
↪passwd
```

Once you do this, you will be able to launch the server side tests, by running:

```
python develop.py
```

If you are developing and you think you want to do some static analysis of your code, run the following:

```
python develop.py --flakes
```

Finally, the `develop.py` script comes with many more options. Run the following to see them:

```
python develop.py --help
```

Sample environment

Note: To be written, but you may go to the `server/launch` directory and find many testing deployments. The `sample` one is especially interesting, since whenever you make a change in the Python code, it is automatically restarted.

5.2.3 Contributing

There are plenty of issues in the issue tracker at github. You may add new ones if you find things to change, but you may also take the existing ones an fix them by your own.

Take a look at *Contribute* to find other ways to contribute to the project.

CHAPTER 6

Appendixes

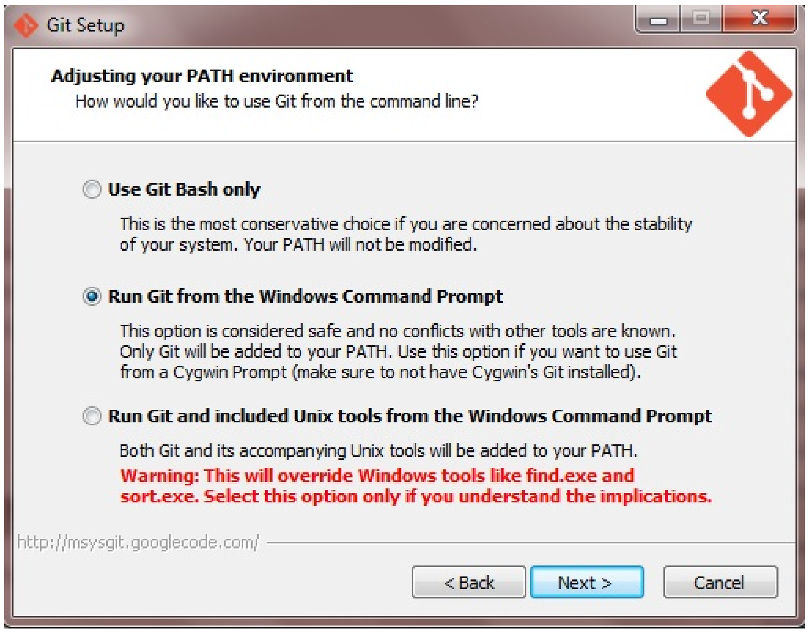
6.1 Download using git

Table of Contents

6.1.1 Microsoft Windows

Git is a source control system. It enables you to download the code, keep track of what you have changed (if you make any change), and you can easily download the latest code in the repository. There are different visual and command line tools to use Git. Use the tool you're more familiar with. Here we are going to detail the most basic one, which is the standard system (command-line based).

So as to download git for Microsoft Windows, go to the git official page. An installer will be downloaded. The installation process is straightforward: you just need to click on "Next" except for one point, which it says "Adjusting your PATH environment". In that step, select the second option ("Run Git from the Windows Command Prompt"):



Adjusting your PATH environment
How would you like to use Git from the command line?

( ) **Use Git Bash only**
This is the most conservative choice if you are concerned about the stability of your system. Your PATH will not be modified.

[✓] **Run Git from the Windows Command Prompt**
This option is considered safe and no conflicts with other tools are known. Only Git will be added to your PATH. Use this option if you want to use Git from a Cygwin Prompt (make sure to not have Cygwin's Git installed).

( ) **Run Git and included Unix tools from the Windows Command Prompt**
Both Git and its accompanying Unix tools will be added to your PATH.
Warning: This will override Windows tools like find.exe and sort.exe. Select this option only if you understand the implications.

http://msysgit.googlecode.com/

Once the installation process is finished, you need to open the command prompt ("Start menu" -> "Run" -> type "cmd" and press enter or "Windows menu" -> type "cmd" and press enter). On it, you may run the following:

```
C:\Users\John> cd \
C:\> git clone https://github.com/weblabdeusto/weblabdeusto.git weblab
Cloning into 'weblabdeusto'...
remote: Counting objects: 43259, done.
remote: Compressing objects: 100% (7927/7927), done.
remote: Total 43259 (delta 31828), reused 42870 (delta 31439)
Receiving objects: 100% (43259/43259), 47.13 MiB | 315 KiB/s, done.
Resolving deltas: 100% (31828/31828), done.
Checking out files: 100% (2729/2729), done.
```

From this point, you have downloaded the latest version of WebLab-Deusto. If you later wanted to upgrade the system to a new release, you must use the following command:

```
C:\Users\John> cd \weblab
C:\weblab> git pull
Already up-to-date.
C:\weblab>
```

Feel free to try TortoiseGit (a graphical tool), which is open source and you can download it for free here, and apply the same steps. Github also provides a useful tool for Microsoft Windows, while it requires registration in the system.

After downloading it you can go to the next step: *Installing the requirements*.

6.1.2 Linux

You need to install git. In most Linux distributions, a package is available. For instance, in Ubuntu, you may run:

```
user@machine:~$ sudo apt-get install git
```

If unsure, go to the GNU/Linux downloads page and follow the instructions. Once installed, you only need to download the source code:

```
user@machine:~$ git clone https://github.com/weblabdeusto/weblabdeusto.git weblab
```

After downloading it you can go to the next step: *Installing the requirements*.

6.1.3 Mac OS X

You need to install git. By going to the official page, you will get an installer and instructions. Once installed, you may open a terminal and run:

```
$ git clone http://github.com/weblabdeusto/weblabdeusto.git weblab
```

After downloading it you can go to the next step: *Installing the requirements*.

6.2 Boole-Deusto y Weblab-Deusto

6.2.1 Introducción

En esta guía se describe cómo utilizar Boole-Deusto con Weblab-Deusto. Las características de integración añadidas a Boole-Deusto hacen que sea posible y sencillo diseñar un circuito combinacional o un autómata, y probarlo prácticamente al momento en un equipo real provisto por Weblab-Deusto.

Boole-Deusto soporta dos tipos distintos de circuitos:

- Circuitos combinacionales
- Autómatas

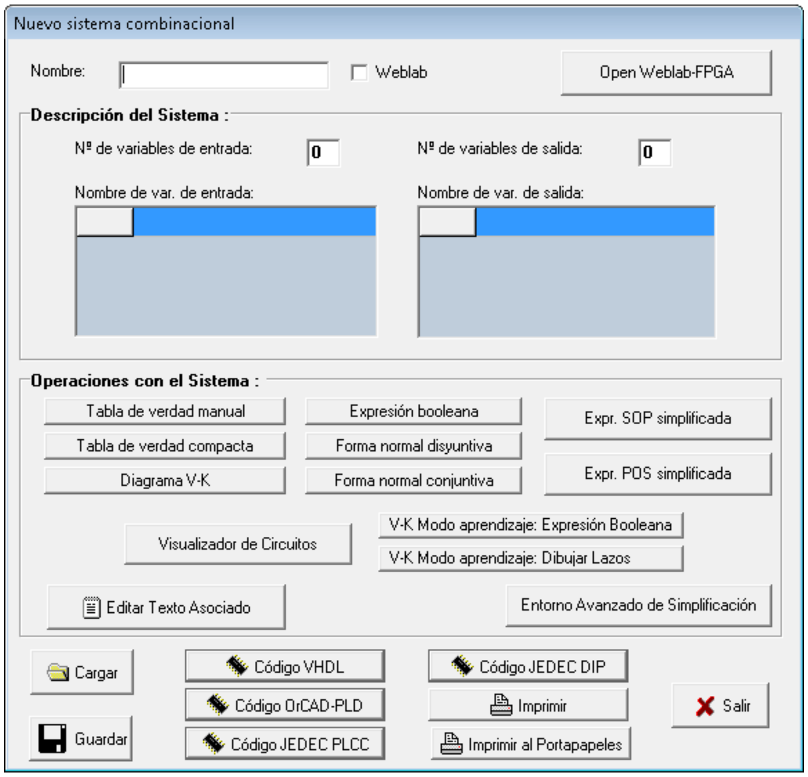
Puesto que existen diferencias significativas entre ambos tipos, se dedicará a cada uno una sección distinta de esta guías.

6.2.2 Circuitos Combinacionales

Introducción

La mayor parte de características relacionadas con la creación de circuitos combinacionales no ha sufrido cambios con respecto al Boole-Deusto original.

El Boole-Deusto modificado tiene este aspecto:



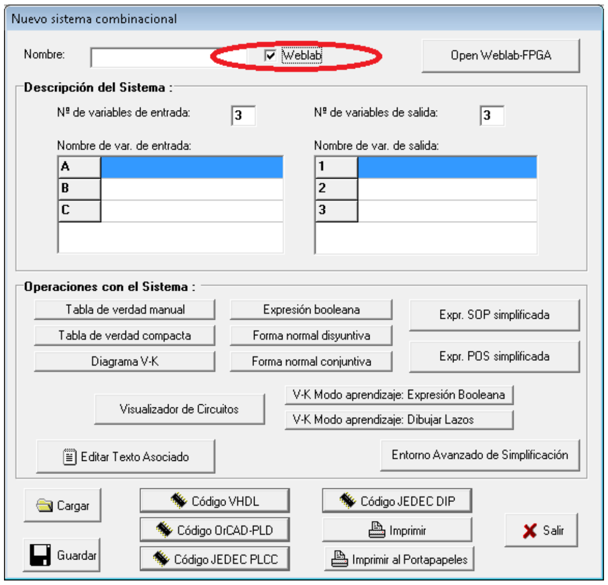
Como puede observarse, principalmente se han añadido algunos controles relacionados con Weblab a la parte superior izquierda de la ventana.

En las secciones posteriores se describirá brevemente el propósito de estos nuevos controles, y se incluirá una guía rápida paso a paso para crear y probar un sistema combinacional.

Controles de Weblab

Los controles añadidos a Boole-Deusto son dos, cuyo propósito se describirá brevemente a continuación.

Activación / desactivación de Weblab



Nombre: [✓ Weblab]

Nº de variables de entrada: 3

Nº de variables de salida: 3

Nombre de var. de entrada	
A	
B	
C	

Nombre de var. de salida	
1	
2	
3	

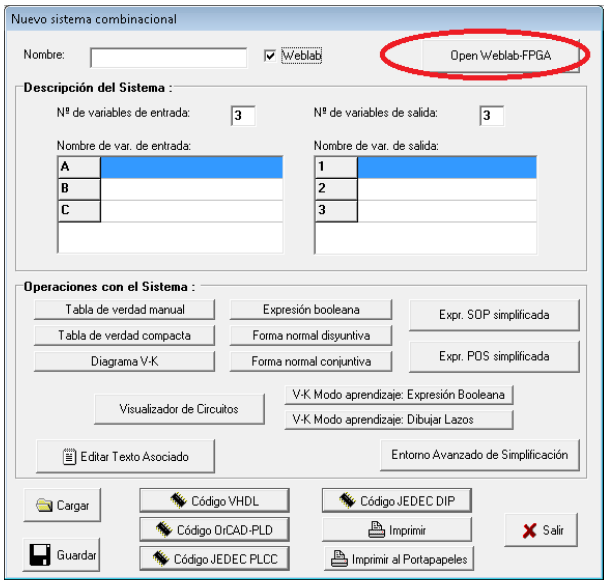
Este control sirve para activar o desactivar el *modo weblab*. El *modo weblab* puede ser activado o desactivado en cualquier momento. Cuando está desactivado, Boole-Deusto se comporta exactamente como el Boole-Deusto original. Cuando está activado, sin embargo, se producen los siguientes efectos:

- Las tablas de entradas / salidas permiten elegir los nombres correctos, que se corresponden con las entradas / salidas en Weblab.
- El código VHDL que Boole-Deusto genere será diferente al que normalmente generaría, ya que tendrá diversos cambios orientados a hacerlo directamente compatible con el experimento FPGA de Weblab.

Warning: El sistema permite, al igual que el Boole-Deusto original, pero incluso en modo weblab, asignar nombres arbitrarios de entradas y salidas, o incluso repetir nombres existentes. Si bien el sistema en general funcionará de forma predecible al hacer ésto, los programas generados no serán compatibles (al menos, sin previa modificación) con Weblab-Deusto. Por eso, para facilitar el uso conjunto, se recomienda utilizar siempre nombres de la lista de entradas/salidas y nunca repetirlos.

Warning: En este momento existe un bug conocido que impide en ocasiones, estado en modo weblab, que aparezcan las sugerencias de entradas / salidas de Weblab. Debido a ciertos motivos, esto tiende a suceder siempre que se hace click por primera vez en la primera celda de la tabla de entradas y de salidas. Para evitarlo, se recomienda hacer siempre click primero en otra celda. Es decir, en una celda que no sea la primera.

Botón de apertura de Weblab



El botón "Open Weblab" abrirá una ventana del navegador que esté configurado por defecto, y generalmente tras dar al usuario la posibilidad de autenticarse, accederá directamente al experimento FPGA, lo que permitirá al usuario subir inmediatamente el código VHDL que genere y probarlo de forma rápida y sencilla.

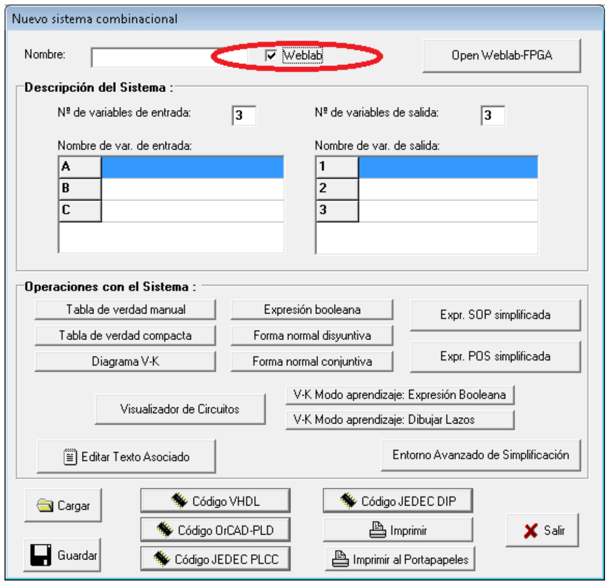
Note: En este momento, el experimento Weblab-FPGA, que es el utilizado para probar el código VHDL, requiere un usuario registrado en Weblab que tenga ciertos privilegios. Sin dichos privilegios no será posible probar el código. En caso de que se necesiten obtener tales privilegios, se recomienda ponerse en contacto con el equipo de Weblab-Deusto, o con quien corresponda.

Guía: Creando y probando un sistema combinacional

En el transcurso de esta breve guía, crearemos con Boole-Deusto un nuevo sistema combinacional, que después probaremos directamente en WebLab utilizando las nuevas características de integración.

Para esta guía, se asume que el usuario está algo familiarizado con los sistemas combinacionales, y con el Boole-Deusto original.

1. Comenzamos la creación de un sistema combinacional.
2. Ahora, activaremos el modo Weblab, habilitando el control que se aprecia en la siguiente figura, y que nos permitirá asignar fácilmente los nombres necesarios de las entradas/salidas, así como generar código VHDL compatible con Weblab.



Nombre: [✓ Weblab]

Nº de variables de entrada: 3

Nº de variables de salida: 3

Nombre de var. de entrada:

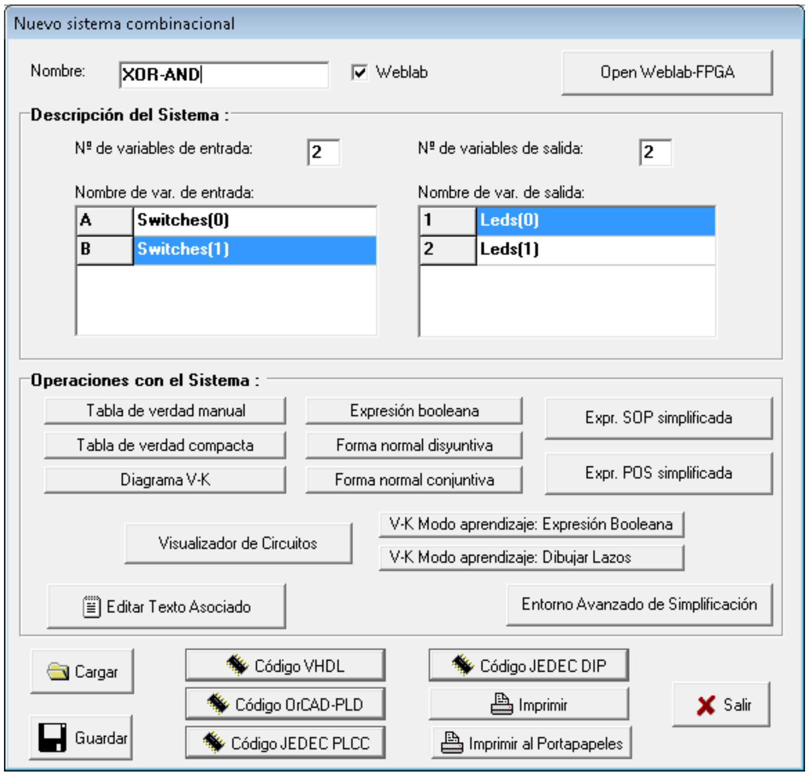
A	
B	
C	

Nombre de var. de salida:

1	
2	
3	

3. Con el modo Weblab habilitado, procedemos a dar un nomber al sistema, que en este caso será XOR-AND, ya que, como veremos enseguida, calcular el XOR y el AND será su tarea.

4. Definimos dos entradas y dos salidas, y les asignamos en la tabla los nombres. En nuestro caso, las entradas se corresponderán con los dos primeros "switches", mientras que las salidas se corresponderán con los dos primeros "leds". Es importante que los nombres utilizados sean exactamente los sugeridos por Boole-Deusto al estar en modo Weblab, ya que es el nombre el que los relacionará posteriormente con los componentes físicos reales (switches, leds, etc) de los que consta Weblab. Queda así:



Nombre: XOR-AND [✓ Weblab]

Nº de variables de entrada: 2

Nº de variables de salida: 2

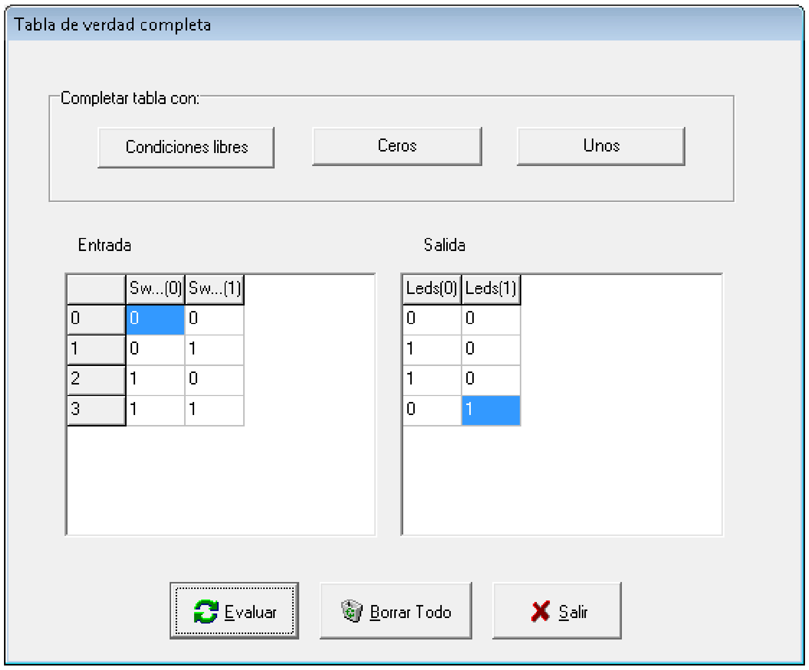
Nombre de var. de entrada:

A	Switches(0)
B	Switches(1)

Nombre de var. de salida:

1	Leds(0)
2	Leds(1)

5. Hecho esto, definiremos normalmente la tabla de verdad para nuestro sistema. Es imprescindible hacer click en "evaluar" tras definirla. La tabla que utilizaremos será la siguiente:



Entrada

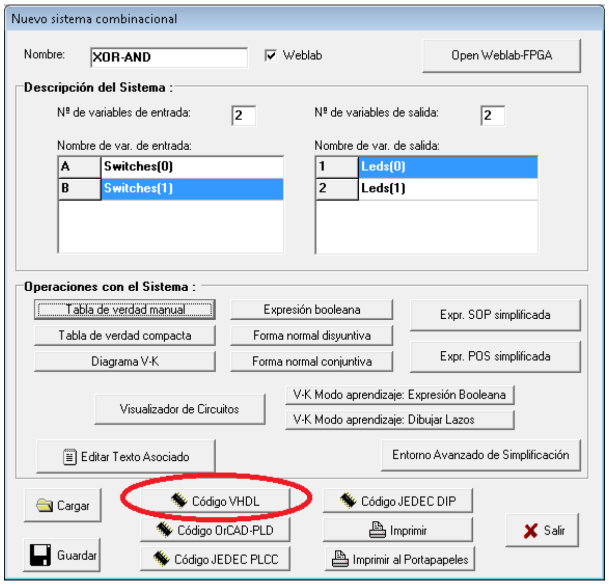
	Sw...(0)	Sw...(1)
0	0	0
1	0	1
2	1	0
3	1	1

Salida

Leds(0)	Leds(1)
0	0
1	0
1	0
0	1

6. Una vez definida la tabla de verdad, podemos, si así lo deseamos, hacer uso de las múltiples características que ofrece Boole-Deusto, tales como visualizar el circuito o los diagramas que le corresponden.

7. Para poder probar nuestro sistema combinacional en Weblab-Deusto, deberemos primero generar el código VHDL. Es **imprescindible** asegurarse de que antes de generar el código, el modo Weblab esté habilitado. El código que se genera por defecto (en el modo estándar) no es directamente compatible. Para generarlo, como en el Boole-Deusto tradicional, deberemos utilizar el botón que se observa en la figura siguiente. Podemos nombrar al archivo VHDL como deseemos.



Nombre: XOR-AND ☑ Weblab

Nº de variables de entrada: 2

Nº de variables de salida: 2

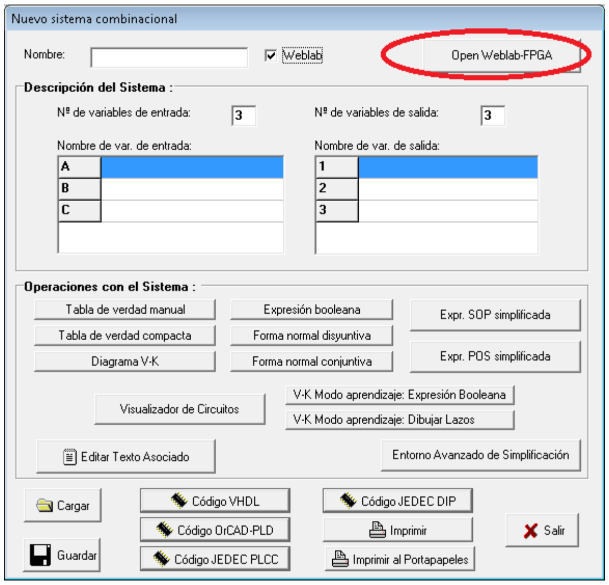
Nombre de var. de entrada:

A	Switches(0)
B	Switches(1)

Nombre de var. de salida:

1	Leds(0)
2	Leds(1)

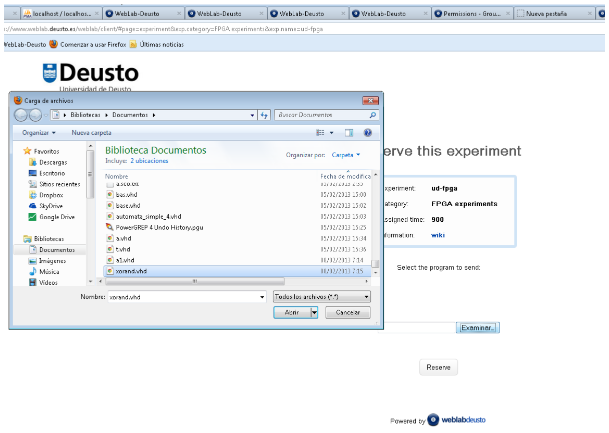
8. Con el código generado, ya estamos prácticamente listos para probar el sistema combinacional. Si lo deseamos, podemos echar un vistazo al código que hemos generado, o incluso modificarlo, siempre que respetemos ciertas reglas impuestas por Weblab(principalmente, relacionadas con los nombres de entradas y salidas). Para probarlo, haremos click en el botón "Open Weblab-FPGA", que abrirá un navegador:



Nº de variables de entrada: 3
Nº de variables de salida: 3

Nombre de var. de entrada:	Nombre de var. de salida:
A	1
B	2
C	3

9. Una vez abierto el navegador en la página de Weblab, si no lo hemos hecho ya, deberemos autenticarnos. Una vez autenticados, llegaremos a la pantalla del experimento Weblab-FPGA, en la cual deberemos elegir el archivo VHDL que hemos previamente generado, de tal forma:



s://www.weblab.deusto.es/weblab/client/#page=experiment&exp.category=FPGA experiments&exp.name=ud-fpga

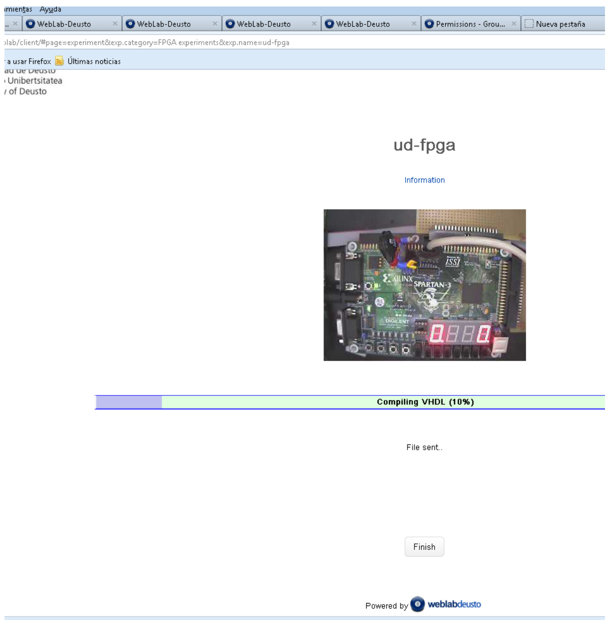
Nombre	Fecha de modifica
a.sco.txt	05/02/2013 2:35
bas.vhd	05/02/2013 15:00
base.vhd	05/02/2013 15:02
automata_simple_4.vhd	05/02/2013 15:03
PowerGREP 4 Undo History.pgu	05/02/2013 15:25
a.vhd	05/02/2013 15:34
t.vhd	05/02/2013 15:36
a1.vhd	08/02/2013 7:14
xorand.vhd	08/02/2013 7:15

Nombre: xorand.vhd
Todos los archivos (*.*)

xperiment: ud-fpga
ategory: FPGA experiments
ssigned time: 900
formation: wiki

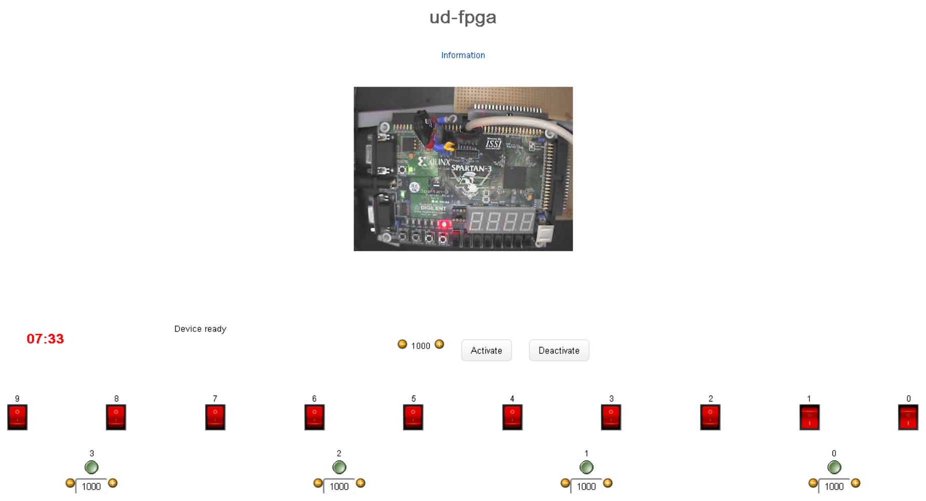
Select the program to send:

10. Tras seleccionar el archivo, deberemos darle a “reserve”, que reservará el experimento. Dependiendo del estado de Weblab-Deusto, y de la la existencia o no de una cola de usuarios, transcurrirá más o menos tiempo antes de que la reserva concluya. La figura a continuación es la pantalla que veremos una vez realizada la reserva.



11. Mientras esté presente la barra de progreso, el sistema estará, o bien sintetizando el código VHDL, o programando la placa. Puesto que especialmente la sintetización es un proceso lento, pueden llegar a transcurrir varios minutos antes de que termine. Si la barra se detuviese con un error, se recomienda consultar la advertencia que se incluye al final de esta sección. El resto de la guía asume que tanto la sintetización como la programación son correctas.

12. Una vez que el programa ha sido correctamente sintetizado, y la placa correctamente grabada, veremos algo similar a lo siguiente:



13. Finalmente, vemos que disponemos en primer lugar de una webcam, por la que podemos ver la placa física, que está actualmente ejecutando nuestro sistema combinacional. Podemos ver también los leds, que actúan como salidas, y diversos interruptores virtuales, que actúan como entrada física real a la placa, y mediante los cuales podemos interactuar. En este caso, vemos que con los interruptores a "0-1" está encendido el segundo LED, y apagado el primero, tal y como hemos definido en nuestra tabla de verdad.

14. Disponemos de un tiempo limitado para probar el sistema. Una vez que el tiempo expire, el sistema automáticamente volverá a la pantalla de reserva. Si necesitamos realizar más pruebas, deberemos reservar de nuevo.

Note: Los leds (*Leds<0-8>*), los interruptores (*Switches<0-9>*) y los botones (*Buttons<0-3>*) se ordenan de derecha a izquierda. Esto implica, por ejemplo, que el *Switch<0>* en Boole-Deusto se corresponde con el interruptor de más a la derecha, mientras que el *Switch<1>* sería el inmediatamente a su izquierda.

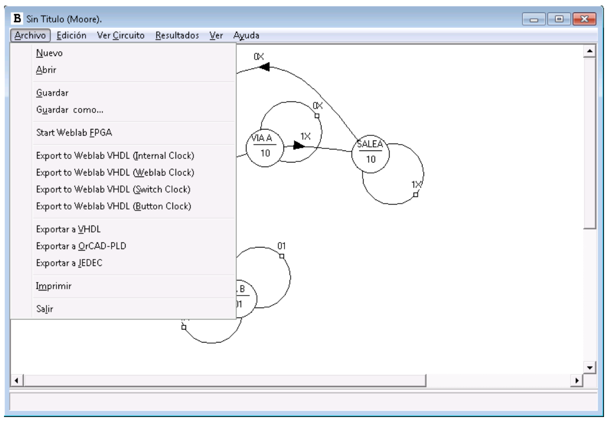
Warning: Si la barra se detuviese con un "compiling error" o con un "programming error", significaría, en el primer caso, que el proceso de sintetización ha fallado (quizás debido a un error de sintáxis), y en el segundo, que el proceso de programación de la placa ha fallado. Si el error es del primer tipo (compiling error) se recomienda:

- Comprobar que se ha seleccionado el VHDL correcto.
- Comprobar que el VHDL se ha generado en modo Weblab.
- Comprobar que todas las entradas y salidas utilizan nombres válidos de la lista de entradas y salidas de Weblab, y que por tanto no se han incluido entradas/salidas con nombres originales, ni entradas/salidas con nombres repetidos.
- Comprobar que no se hayan realizado modificaciones manuales al VHDL, o que en caso de que se hayan realizado, no contengan errores.

Si con las diversas comprobaciones anteriores no se consigue resolver el problema, o si el error es de programación (grabación

- Esperar un tiempo, y volver a intentarlo más tarde.
- Contactar con los administradores de Weblab-Deusto.

6.2.3 Autómatas



Introducción

El segundo tipo de circuito con el que Boole-Deusto puede trabajar son los autómatas. Esta característica, que ya existía en el boole-deusto original, ha sido extendida añadiendo capacidad de integración inmediata con Weblab-Deusto y en concreto sus experimentos FPGA. Tras esta extensión, es ahora posible diseñar y definir un autómata gráficamente, e inmediatamente observarlo en funcionamiento (e interactuar con él) en un FPGA físico real en Weblab-Deusto.

Aspectos básicos con Weblab-Deusto

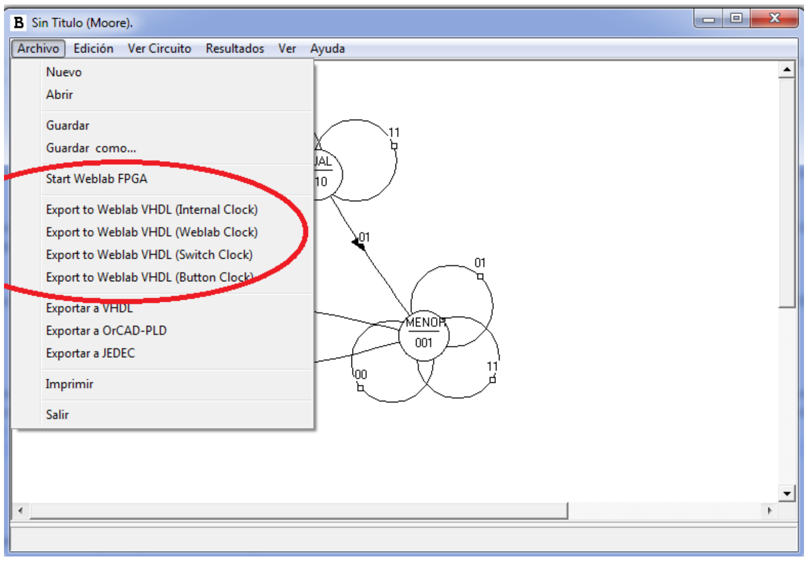
En su versión actual, para promover la simplicidad, esta parte de Boole-Deusto no requiere (ni permite) elegir las correspondencias entre las salidas y entradas de los estados, y las salidas y entradas de Weblab-Deusto. En vez de eso, se ha de saber que la correspondencia entre entradas y salidas siempre es la misma:

Las entradas son siempre los interruptores. De este modo, si tenemos por ejemplo un estado S0 que "recibe" dos entradas, estas dos entradas se corresponderán con los últimos dos interruptores (los de más a la derecha).

Las salidas son siempre los LEDs. De este modo, si tenemos un estado S0 que tiene dos salidas, estas dos salidas se corresponderán con los últimos dos LEDs (los de más a la derecha). Si la salida del estado es por ejemplo "01", el LED de más a la derecha estará encendido, y el LED inmediatamente a su izquierda apagado.

Existe un botón que devuelve al autómata a su estado inicial. Este botón es siempre el último botón de Weblab (el de más a la derecha).

Controles de Weblab



Los controles añadidos a Boole-Deusto en la integración con Weblab-Deusto son principalmente de dos tipos:

Exportación a código Weblab-VHDL

Mediante el uso de estas opciones, es posible generar código VHDL que sea inmediatamente compatible con el experimento FPGA de Weblab-Deusto. Para conseguir esta compatibilidad, el código generado utilizará nombres de entradas y salidas compatibles con las de Weblab-Deusto (definidas en un UCF).

Podemos observar que existen varias opciones para generar el código. Cada una de esas opciones corresponde a un tipo de reloj diferente, que utilizará el autómata. Los relojes disponibles son los siguientes:

Reloj Interno (Internal Clock)

Utiliza el reloj interno de la FPGA. Su frecuencia es bastante alta, lo que lo hace poco adecuado para aquellos casos en los que el comportamiento del autómata requiera de alguna clase de sincronización de las entradas y salidas con el reloj.

Reloj Weblab (Weblab Clock)

Algo más lento que el reloj interno. Está provisto por el propio Weblab-FPGA, y su frecuencia puede ser controlada de forma limitada, mediante un *slider* en el propio experimento.

Reloj Interruptor (Switch Clock)

El último interruptor (el de más a la izquierda) actúa como reloj. Esto lo hace muy adecuado para aquellos casos en los que se desee probar el autómata teniendo un absoluto control sobre el reloj. Esto podría incluir, por ejemplo, aquellos casos en los que es necesario sincronizar las entradas con él.

Reloj Botón (Button Clock)

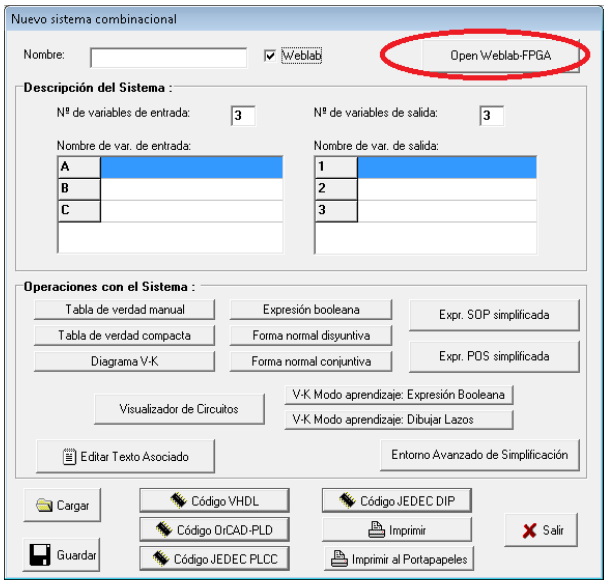
Similar al anterior, en este caso el botón de más a la izquierda actúa como reloj. De nuevo, muy adecuado para aquellos casos en los que se desee probar el autómata teniendo un absoluto control sobre el reloj.

Warning: Debido a limitaciones presentes ya en el Boole-Deusto original, se recomienda comprobar el autómata antes de intentar generar el código. Ciertos errores, como el no asignar salidas a un estado, pueden hacer que el programa deje de responder.

Warning: Si se utiliza la generación de código VHDL estándar de Boole-Deusto (la que no hace mención sobre relojes, ni sobre Weblab), el VHDL generado NO será compatible con Weblab-Deusto. Al menos, sin aplicar manualmente grandes modificaciones.

Note: **Nota de implementación** En la práctica, el reloj a utilizar no lo determina el VHDL en sí, sino el archivo de restriccines (UCF) que se utilice. Boole-Deusto añade una directiva como comentario al VHDL, como por ejemplo `` `%%%CLOCK:SWITCH%%%` ``. Esta directiva, en absoluto original de Xilinx, es detectada por el propio Weblab-Deusto, que sintetizará el VHDL provisto utilizando un UCF u otro. Cuando se utiliza Weblab-Deusto FPGA también es posible usar dichas directivas en código VHDL escrito manualmente.

Apertura de Weblab



El botón "Open Weblab" abrirá una ventana del navegador que esté configurado por defecto, y generalmente tras dar al usuario la posibilidad de autenticarse, accederá directamente al experimento FPGA, lo que permitirá al usuario subir inmediatamente el código VHDL que genere y probarlo de forma rápida y sencilla.

Note: En este momento, el experimento Weblab-FPGA, que es el utilizado para probar el código VHDL, requiere un usuario registrado en Weblab que tenga ciertos privilegios. Sin dichos privilegios no será posible probar el código. En caso de que se necesiten obtener tales privilegios, se recomienda ponerse en contacto con el equipo de Weblab-Deusto, o con quien corresponda.

Note: Esta sección del manual, y el propio Boole-Deusto en lo relacionado a estos aspectos, se encuentran actualmente en desarrollo.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search