PROGRAMMING LANGUAGE RESEARCH GROUP
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

DEPARTMENT

UNIVERSITY OF CALIFORNIA, IRVINE

VIGILIA DOCUMENTATION

Table of Contents

1. Getting started

This is a short (but hopefully comprehensive) documentation of Vigilia. Vigilia is a research programming environment for smart home IoT devices.
The main features of Vigilia are:

- new programming model that involves distributed processes (i.e. controller/app, and device drivers),
- programming environment that supports Java and C++; each controller/app or device driver can be written in Java or C++,
- strong security features that assume a threat model that allows attackers to know the SSID and password; these security features involve: 1) strict firewalling through code instrumentation, and 2) process jailing.

For more background information, please read our published paper on the Vigilia system.

2. Hardware Installation

The Vigilia programming model has been tested using Raspberry Pi's. This section is dedicated to explain the installation process of a Vigilia-ready Raspberry Pi, which includes a few unusual tweaks.

2.1.Basic Hardware

This most basic setup consists of the following hardware as router and compute nodes:

1. one Raspberry Pi 2 Model B as the master,
2. one Raspberry Pi 2 Model B as the slave, and
3. one router NETGEAR Nighthawk X4S R7800.

2.2.Basic Raspbian Installation

The default **Raspbian** can be downloaded here. The latest version that has been tested for our Raspberry Pi hardware was **Raspbian Stretch 2017-11-29** with kernel version **4.9**. However, for Vigilia, we normally need to recompile the kernel because the security module Tomoyo Linux is not enabled by default (see below). The basic installation procedure can be found here.

2.3.Tweaks

We do the following setup for our Raspberry Pi. We have been using Raspberry Pi 2 Model B for our experiments.

Tomoyo Linux

1. The Raspbian that we can download here usually does not come with security modules built-in. We need to recompile Raspbian to get a kernel that has Tomoyo Linux enabled. Basically, we can follow the build process here to choose the right options for kernel compilation for Tomoyo. Then, we can cross compile Raspbian using the instructions here. Basically, we can choose the option **Security options** and enable the options for Tomoyo Linux and, perhaps, also **Default security module** and choose Tomoyo as the default option.

   Link to Raspbian source code:
   https://github.com/raspberrypi/linux
   Note: We can execute the following commands to recompile the kernel and patch it into the SD card that already has Raspbian installed in it.
   Command listing:

git clone -b rpi-4.9.y git://github.com/raspberrypi/linux.git linux-32
cd linux-32
KERNEL=kernel7
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j3 bcm2709_defconfig
menuconfig #Then activate the security options for Tomoyo Linux here!
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j3
mkdir -p mnt/fat32 && mkdir -p mnt/ext4
sudo mount /dev/sdc1 mnt/fat32 && sudo mount /dev/sdc2 mnt/ext4
sudo make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j3
INSTALL_MOD_PATH=mnt/ext4 modules_install
sudo cp mnt/fat32/$KERNEL.img mnt/fat32/$KERNEL-backup.img
sudo scripts/mkknlimg arch/arm/boot/zImage mnt/fat32/$KERNEL.img

Note: This link is useful when you have build errors.

Account Setup

2. First and foremost, change the password for the default account **pi**. In one of our experiments that involved placing our devices outside of our campus firewall, the Raspberry Pi's were infected by a trojan/virus. We can replace the default password, which is **raspberry**, with anything else that is not the default password itself.

3. Activate SSH service. SSH is deactivated in **Raspbian Stretch 2017-11-29** with kernel version **4.9**.
Command Listing
sudo /etc/init.d/ssh start

Note: Alternatively, to avoid doing this for every reset, we can put this command inside **/etc/rc.local**.

4. Create a new user that is called **iotuser**. This can be any username but by default, Vigilia is set to use this username. Using the default account pi:
    - execute *sudo visudo*,
    - add two lines for **iotuser** with privileges exactly the same as **pi**,
    - these lines are the following:
    iotuser ALL=(ALL:ALL) ALL
    iotuser ALL=(ALL) NOPASSWD: ALL

5. Login using **iotuser** and go to **/home/iotuser/**.
Command Listing
su iotuser # login as iotuser

6. Change the hostname from **raspberrypi** to something else (e.g. **raspberrypi1** for master, **raspberrypi2** for the first slave, **raspberrypi3** for the third slave, etc.). For this, we need to change **/etc/hostname** and the IP address mapping in **/etc/hosts** (more instructions here and here).

   Note: We can reboot the Raspberry Pi after changing its hostname.

7. The first time we boot up all the compute nodes, **master** has to synchronize its SSH public key with all other **slave** nodes and **Vigilia router** (see the **Router section** to get more information about its installation and SSH access through dropbear).
   We need to perform the instructions listed here to generate a new key for **master** and copy it to the **slave** nodes. Then we can try to execute SSH to access the slave node from the master node. If this is successful without the slave node asking for any password, then the SSH has been set up correctly.
   Command Listing
   ssh iotuser@<slave-node-ip-address>

   Note: We have to also do this process for the **master** node itself since it needs to distribute processes also to itself using SSH.

Back to Tomoyo

8. We can **install** Tomoyo through executing the following commands.
   Command Listing:
   apt-cache search tomoyo # make sure that we have tomoyo-tools
   sudo apt-get install tomoyo-tools # put security=tomoyo in /boot/cmdline.txt and reboot
   sudo /usb/lib/tomoyo/init_policy # initilialize policy file
   sudo cat /etc/tomoyo/profile.conf # check Tomoyo profile

   We can do Step 9 from here to initialize and check out Tomoyo profile (the last 2 commands in the listing above). After that, put the option **security=tomoyo** in **/boot/cmdline.txt** then reboot the Pi once again.

9. A few useful commands to play around with Tomoyo's setup. We can learn more extensively about Tomoyo Linux from here.
   sudo /usr/sbin/tomoyo-editpolicy
   sudo /usr/sbin/tomoyo-savepolicy
   cat /sys/kernel/security/tomoyo/stat

   Note: Tomoyo policy file is stored in **/sys/kernel/security/tomoyo/domain_policy**

Vigilia Code

10. Download Vigilia code from our website. This might take a while depending on the quality of your internet connection. Alternatively, we can also do **git clone** on a faster machine on a faster machine and do **scp** to transfer the entire **iot2** folder onto the Raspberry Pi or just use a USB drive.
    Command Listing
    git clone git://plrg.eecs.uci.edu/vigilia.git

11. Then to compile the files:
    - Go into **iot2/iotjava/** and execute **make**. This will compile the main components of Vigilia, i.e. runtime system, compiler, and installer.

    Note:
    - For a **slave** node, we just need to compile the **runtime**.
    - The Vigilia runtime has both Java and C++ parts for the slave. This **make** execution will compile both slaves and put them in **iot2/bin/iotruntime/slave** (please see Section **3.10** to know more about the C++ compilation for Vigilia).

    Command Listing
    cd iot2/iotjava/ # we go into this folder
    make # we run make to compile the files

Master Node

12. If this is the **master** Raspberry Pi, we need to install **mysql**.
    Command Listing
    sudo apt-get install mysql-client
    sudo apt-get install mysql-server
    sudo apt-get install libmysql-java

13. Then we need to set up the password for the **root** account of **mysql**.
    Command Listing
    sudo mysql -u root # then we get into mysql prompt
    SET PASSWORD FOR 'root'@'localhost' = PASSWORD('1234'); # default root password
    *# Now we want to enable access without having to **sudo***
    USE mysql;
    UPDATE user SET Plugin='' WHERE User='root';
    FLUSH PRIVILEGES;
    quit;

    Note: We can try again and execute *mysql -u root -p*. When it asks for password then we can type in *1234* and we should get access to the **mysql** server this time without **sudo**.

14. Now, we should dump our existing Vigilia database into the **mysql** server.
    Command Listing
    sudo mysqladmin create IoTMain -p
    cd iot2/others/Mysql/ # go into the Mysql directory in our iot2 repository
    gunzip < IoTMain.gz | sudo mysql IoTMain -p # copy the IoTMain database into mysql

    Note: At this point we can check if the database has been installed correctly by checking it out in our **mysql** server.
    mysql -u root -p # then input the password
    USE IoTMain;
    SHOW TABLES; # we will be able to see the following

```
+---------------------------+
| Tables_in_IoTMain         |
+---------------------------+
| Alarm                     |
| AlarmSmart                |
| AmcrestCameraAddCM1       |
| AmcrestCameraAddCM2       |
...
| WeatherForecastAddWF1     |
| WeatherForecastAddWF2     |
| WeatherGateway            |
| WeatherGatewaySmart       |
| WeatherPhoneGatewayAddGW1 |
| WeatherPhoneGatewayGW1    |
+---------------------------+
96 rows in set (0.00 sec)
```

15. If this is the **master** Raspberry Pi, then we need also to go to the folder **benchmarks** and execute **make**.
    For a **master** node, we compile the **runtime** and the **benchmarks**.
    Command Listing
    cd iot2/benchmarks/ # we go into this folder
    make # we run make to compile the files

    Note:
    - Vigilia benchmarks have the C++ version of drivers for **LabRoom** and **LifxLightBulb**---due to time and resource limitation, we could not rewrite the other drivers from Java to C++ ourselves. So, hopefully you can help us create more C++ drivers after reading about Vigilia and this documentation. :)

- This **make** execution will compile the C++ drivers as well and put them into **iot2/bin/iotcode/** (please see Section **3.10** to know more about the C++ compilation for Vigilia).

2.4.Router

We use the NETGEAR Nighthawk X4S R7800 router for our experiments. For this router, we have a custom-build firmware that was originally from LEDE/OpenWrt. This is an informational page about LEDE/OpenWrt for this router. We made a few changes in our custom build that supports:

1. Iptables and its additional hooks that are rarely used, e.g. *CHECKSUM*.
2. Advanced features of hostapd that required us fixing hostapd, e.g. *disable_dgaf* and *proxy_arp* options.
3. Our configuration files that have been adjusted for Vigilia.

To compile and install the firmware onto the router, we have to follow these steps:

1. Download the code from our website using git, then switch branch to *lede-17.01*.
   Command Listing
   git clone git://plrg.eecs.uci.edu/lede.git
   git branch -a # you will see the branch remotes/origin/lede-17.01
   git checkout lede-17.01 # switch to that branch

2. Copy the compilation configuration file **vigilia.config** into **.config** file in the main folder. We do this in menuconfig by choosing **< Load >**, and loading **vigilia.config**. Then, we need to build it.
   Command Listing
   ./scripts/feeds update -a # get all the latest package definitions defined in feeds.conf / feeds.conf.default
   ./scripts/feeds install -a # install symlinks of all of them into package/feeds/
   make menuconfig # load vigilia.config here, then save it
   make world -j3 # make world -j<number-of-proc>

   Note: If we are missing the **zlib** library and **awk**, we have to install them. This is a good solution.

3. After the compilation, the image will be found in **lede/bin/targets/ipq806x/generic/lede-ipq806x-R7800-squashfs-factory.img.** We can flash this image onto the router using the instructions here.

Note: Our installation works without setting any password for the TFTP software when clicking on the *Upgrade* button, but it depends on the state of the router when flashing. On our router, the LED turns *blinking white* (**not** green) after *blinking orange*.

4. After flashing and booting for the first time, we need to set up the router using *RJ45* cables. No password is needed for the first access. To set up network configurations:
    - **Option 1:** We can access the router by connecting a device to the router, opening a browser, and typing the default router IP address: **http://192.168.1.1**. This will open the IUCI configuration page and we can set up the router now. We can set up the router according to the configuration files in **lede/vigilia_setup/config**.
    - **Option 2:** When your laptop is connected to the router using the *RJ45* cable, you can always do SSH connection with it; no password is required for the first time. Then we can use SCP to copy the configuration files, namely *dhcp*, *hostapd-psk*, *network*, and *wireless*, directly from **lede/vigilia_setup/config** into **/etc/config**.

   For the *firewall* file, we have to rename it, e.g. *firewall.bak*, so that the router will not install the firewall rules there.
   Command Listing
   mv /etc/config/firewall /etc/config/firewall.bak # rename firewall

   Note:
   We typically need to change:
    - the IP address of the router in the *network* file, namely *config interface 'lan'* (the default is *128.195.204.94*), which is the outside interface of the router,
    - the list of devices with their MAC addresses (and perhaps the IP addresses) in both *dhcp* and *hostapd-psk* depending on the hardware that we have.

5. Set up a password for the system and reboot.
   Command Listing
   passwd # set up a new password
   reboot

6. We also want to do the same thing that we did with SSH in the Raspberry Pi installation, so that the **master** Raspberry Pi can access the router using its key. The LEDE/OpenWrt firmware uses dropbear to manage these SSH and key.
   Command Listing
   ssh root@192.168.1.1 "tee -a /etc/dropbear/authorized_keys" < ~/.ssh/id_rsa.pub
   # we do this from the master Raspberry Pi; 192.168.1.1 is the router's IP address

7. Set **net.bridge.bridge-nf-call-iptables=1** in **/etc/sysctl.conf** to make iptables work in bridge mode. Additionally, we can also set **ip6tables** and **arptables** to work in the **bridge** mode, but we will not use them in Vigilia. We then run **sysctl -p** to apply the change to **sysctl.conf**.

Command Listing
vim /etc/sysctl.conf # then change the value net.bridge.bridge-nf-call-iptables
sysctl -p

8. Create **/root/vigilia_setup/register** and copy the scripts in **lede/vigilia_setup/register/version_3** into it, e.g. using SCP. These Shell scripts work with the Android app that registers and deletes devices to and from the router. We also need to copy **lede/vigilia_setup/setup** into **/root/setup**, and **lede/vigilia_setup/rc.local** into **/etc** to replace the router's **/etc/rc.local**.
   Command Listing
   scp -r lede/vigilia_setup/register/version_3/ root@<router-ip>:/root/vigilia_setup/register*
   scp -r lede/vigilia_setup/setup root@<router-ip>:/root
   scp -r lede/vigilia_setup/rc.local root@<router-ip>:/etc
   Note:
   A few scripts that we can use inside **/root/setup**:
   - **startup.sh** will be run by the rc.local script when LEDE is booting up. This script contains the initial firewall rules for Vigilia router and a number of workaround rules to fix a few issues when disable_dgaf and proxy_arp options are activated, i.e. hostapd checksum bug (for disable_dgaf feature).
   - **clean** and **nat** scripts are for cleaning and activating NAT(Network Address Translation) on the system.
   - **dhcp** shows the IP address assignments to different connected devices.
   - **show** shows the active iptables rules.
   - **transfer** contains commands to transfer files through the SCP command.

9. Reboot the system and we will have a working LEDE router for Vigilia system.

2.5.Smart Device Installation

To install smart home IoT devices, they have to be installed on the router, and on the database using Vigilia phone application.

Router Side

To prepare the phone application to work well with the router, we need to do the following steps first.

1. Connect the phone to the router that was set up in Section **2.4**.
2. Using **Android Studios** that was downloaded in section 4, create a new project with the gradle provided in **iot2/others/lede-gui**.
   - Under **res** → **values** → **constants.xml**, the **default_rssid, default_routerip**, **mysql_hostip**, **default_rpwd**, **mysql_hostpassword**, and any other variables that are different from what have been specified there need to be changed to the

respective values associated with your router and RaspberryPi (see Master Node Section **2.4**).
    - You can find the IP of your router and Pi using the command **ifconfig**. You will be using the IP under *wlan0* for the Pi and *eth0* for the router.
3. Launch the application on the phone that was connected to the lede router.
4. You are now ready to register, view, and delete devices on the router.

Main Menu

To register a device to the router, we need to perform the following steps:

Figure 2.5.1. Main Menu

1. Figure 2.5.1 shows the main menu for the application. The various functionalities of the application such as registration of devices and maintenance of the database can be utilized here. To begin, select the **REGISTER DEVICE** button. This will take you to a new page and begin the registration of devices with Vigilia.

Note:

- In general, the app needs some time for processing after we click on a button. For connecting to the database and saving new information, it could take up to **30 seconds** to **1 minute** because of the SSH connection that is made from the smartphone to the router. Since this is a research app, some of the screen transitions are not accompanied by any notice/warning messages.
- To make sure that the app is still working well, we can observe the messages printed out on the **Android Studios** console when the app is running.

Router Registration

Figure 2.5.2. Registering a new IoT device

2. After clicking **REGISTER DEVICE** in the Main Menu, you will be taken to the page displayed in Figure 2.5.2.
3. Initially, the **WiFi** will be temporarily down as the password is being changed. Please wait until the **WiFi** is re-connected, do not press any buttons until it is.
4. Once the **WiFi** is connected, please press the **LOAD** button on the screen so that the application can see the current registered devices.
5. After pressing **LOAD**, connect the new IoT device to the router using the generated password shown on the screen.
   Note:
   - In this step, we need to connect the device to the router by using the default app that comes from the device manufacturer and by inputting the generated random password.
   - To do this, we need to copy the generated random password and switch screen to open the device manufacturer app for the device installation, e.g., the WeMo app for a WeMo switch installation.
6. After the device is connected to the router, please click the **CHECK** button to check if the device was successfully connected to the router. If this does not take you to the next page, please try reconnecting the IoT device onto the router and clicking **CHECK** again.

7. If you wish to cancel the registration process, press the **BACK** button to return to the main menu, this will change the wifi password back to the default value and will take some time for the application to reconnect to the router.
   Note:
   If you do not press the **BACK** button, the **WiFi** password will remain the random string so it is important to utilize the **BACK** button, or you may need to restart your entire process.

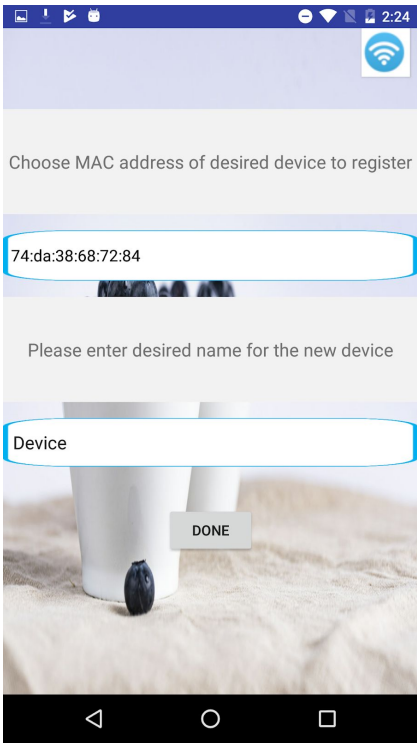


Figure 2.5.3. Setting the New Device Information

8. The next step in the registration process is to select the MAC address of the the device you wish to register and give it a name.
9. Selecting the first box will reveal a dropdown box of available MAC addresses, please choose the MAC address corresponding to the new device that you wish to register on the router.
10. You may then type the desired name into the second box.
    Note: The name should be a single string, i.e., no spaces.

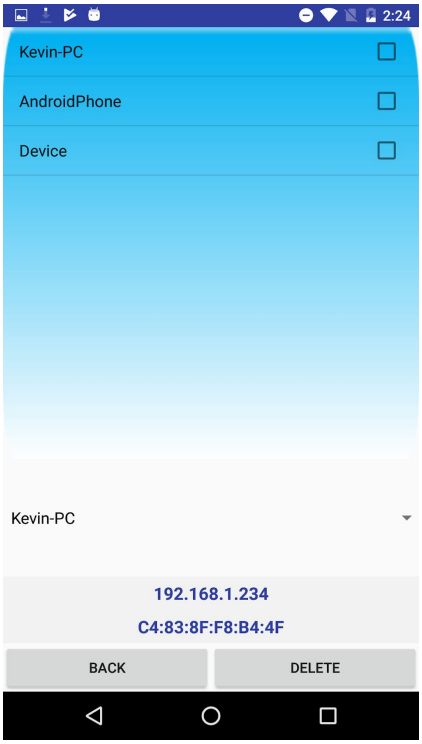


Figure 2.5.4. Viewing/Deleting Registered Devices

11. Clicking on **DEVICE LIST** within the Main Menu (see Figure **2.5.1**) will take you to the page shown in Figure 2.5.4. It will be populated by the registered devices.
12. Clicking a device will show you the IP and MAC of the selected device.
13. If you want to delete the registered device, click the **DELETE** button. This will delete the device off of the network, so if you want to use that device in the future you will need to re-register the device.

2.6.Smart Device Database Management

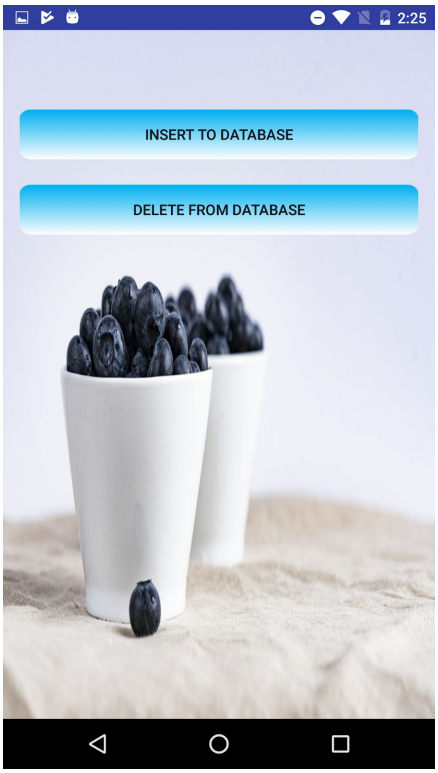


Figure 2.6.1. Inserting/Deleting on Database

Figure 2.6.1 shows the page that you will be taken to when clicking **DATABASE** from the Main Menu (see Figure **2.5.1**). This will be the page that you will use to get registered devices on and off of the database for use in your home environment.

Insertion

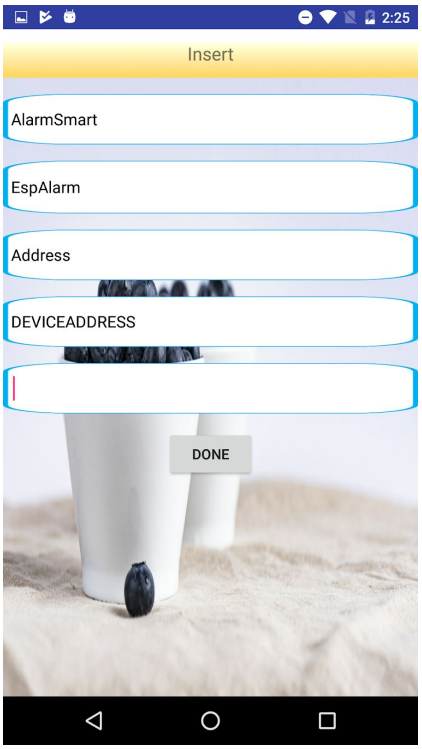


Figure 2.6.2. Insert to Database Page

1. When you have successfully registered the device(s) on the router, you are now able to insert those devices onto the database to create environments with those devices.
2. First you will choose the type of device that you would like to insert into the database, like **AlarmSmart**, **CameraSmart**, etc.
3. Then you will choose the subtype or brand of the device, like **EspAlarm**, **AmcrestCamera**, etc.
4. Depending on the type and subtype of the device chosen, you will need to input device and address information for the device.
5. Once you are finished inputting all of the information prompted for the device, you can click done to complete inserting that device into the database (see Section **2.3** step **14** to see the database content).

Note:

- The **DlinkAlarm** subtype needs to be added into the **.config** files and it is not available in the current version of the phone app, i.e., **it is set aside for the tutorial section (see Section 6)**.
- To run our first benchmark, i.e., a test benchmark called **Lifxtest**, in Section **2.7**, we only need to register, at least, **one light bulb**.

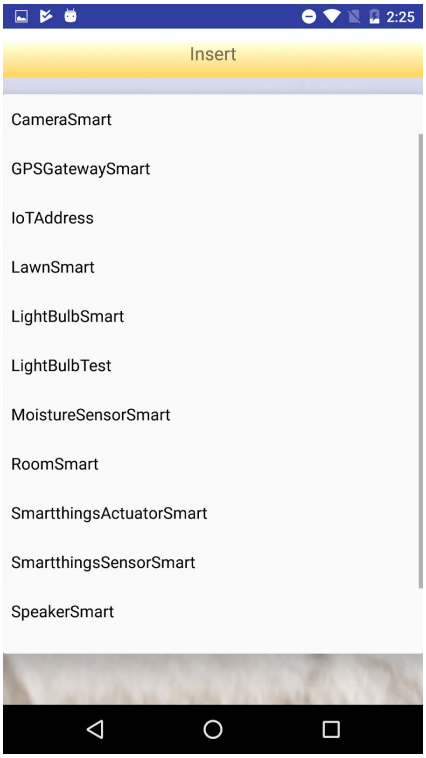


Figure 2.6.3. List of Supported Devices “out of the box”

6. Figure 2.6.3 shows some of the types of devices that **Vigilia** supports (per July 2018).

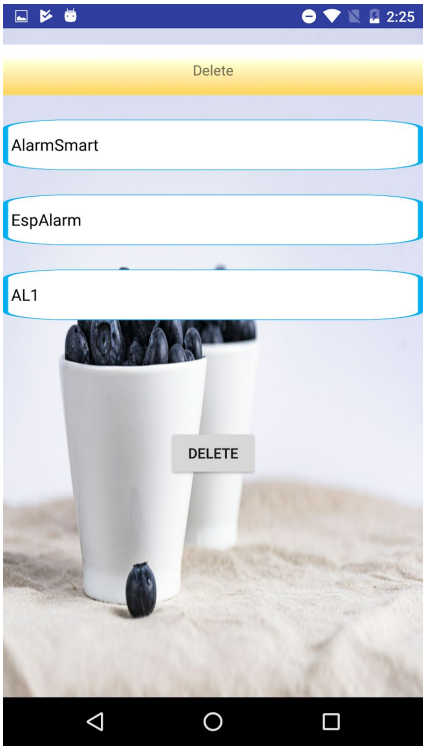


Figure 2.6.4. Deleting From Database Page

Deletion

7. Figure 2.6.4 displays the page that you will see when clicking **DELETE FROM DATABASE** within the **DATABASE** tab (see Figure **2.6.1**).
8. You will need to know the **type** and **subtype** of the device that you would like to remove from the database.
9. After choosing the type and subtype of the device, you will need to know the device ID that you're deleting.
   Note: If you do not know the device ID of the device that you want to delete, please find the device ID within the MySQL database. (see Section **2.3** step **14** to see the database content).

Application Configuration

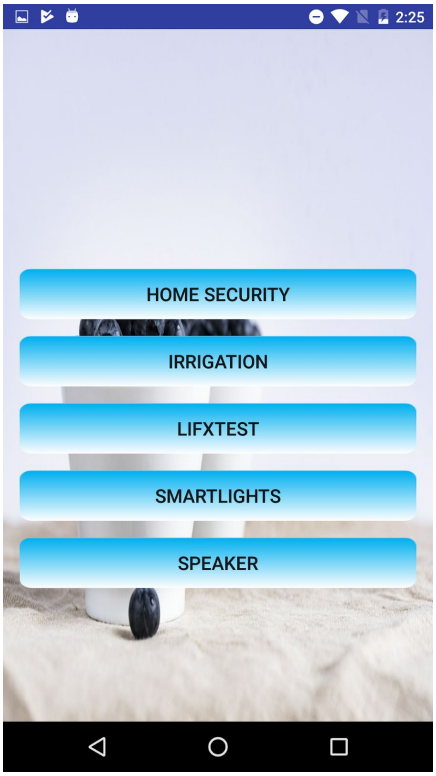


Figure 2.6.5. Supported Environments

10. Figure 2.6.5 shows the supported environments within the **APPLICATIONS** tab (see Figure **2.5.1**).

11. This tab is where you will be able to set the connections between devices that are within the database, for specific home environments. As per July 2018, we have 5 **off-the-shelf** applications that can be directly configured.

Home Security Example

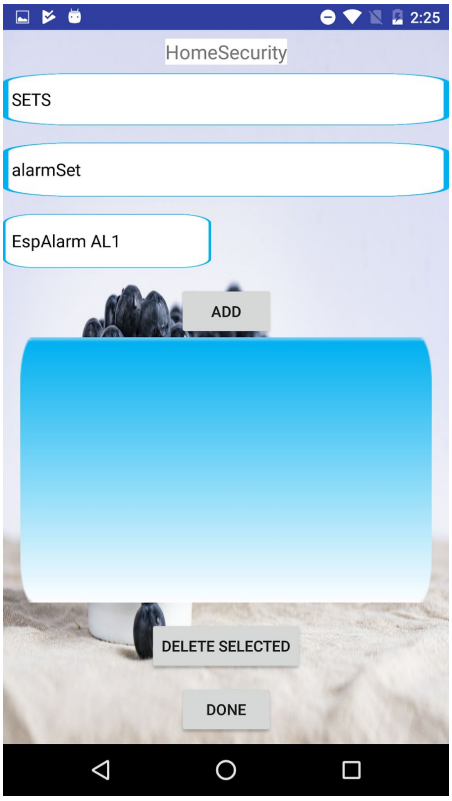


Figure 2.6.6. Home Security Example

12. Figure 2.6.6 shows the Home Security setup page.
13. You need to completely select sets and relations that correspond to the specific environment. If you missed selecting one set/relation or more, then clicking **DONE** would not take any effect.
    - For Home Security, we must select an **alarm**, **door lock**, and **IoT cloud server** for the sets as well as a **room-sensor** and **room-camera** relationship.
    - For every set and relation, we can put one or more members and the app allows for flexible and out-of-order insertions.

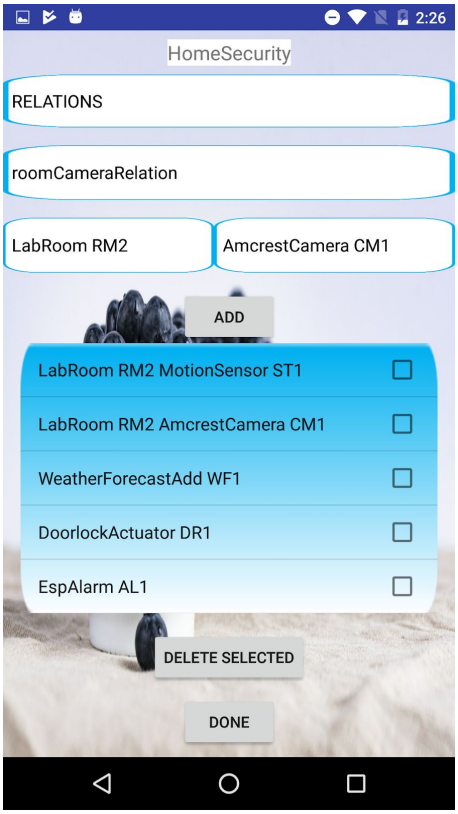


Figure 2.6.7. Completed Home Security Environment

14. A complete **HOME SECURITY** environment is shown in Figure 2.6.7.
15. All of the devices that you want to be selected must be already inserted into the database.
    - If a device that you are looking for is not shown as one of the options of devices please go back to Insertion Section.
16. If you accidentally selected a device, you can remove the selected device by clicking on the device(s) and then clicking **DELETED SELECTED**.
17. Once you have completely finished selected all of the devices for the sets and relations, you can click **DONE** to finish setting up the specific environment.

Device Driver Configuration

Figure 2.6.8. Selecting a Driver Device Type

18. After clicking on drivers on the main menu, you will be taken to the page displayed in figure 2.6.8.
    - On this page, you will be able to select the device type that you would like to set up the drivers for.

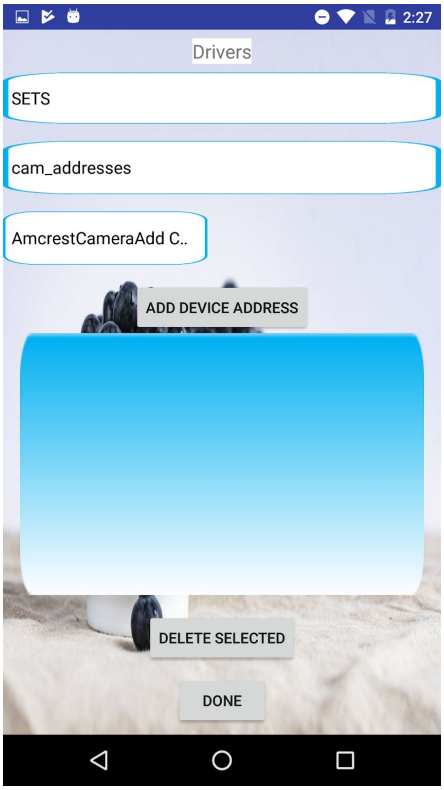


Figure 2.6.9. Choosing Devices

19. Once you have chosen the device, you can then choose the devices that are affected within the page displayed in Figure 2.6.9.
20. You can choose multiple devices by choosing them within the dropdown menu and then clicking **ADD DEVICE ADDRESS**.
    Note:
    - At this point, the app allows accidental insertion (i.e., **no warning!**) of devices of the wrong type into a set, e.g., light bulbs into the **cam_addresses** set that is a camera set.
    - This would just create a SELECT statement that would return an empty set because of searching for a wrong device type.
21. You will then click done when you have chosen the devices.
22. For Section 2.7 below, we have to do this Application Configuration and Device Driver Configuration, at least for the **LIFXTEST** application (see Figure 2.6.5).

Note:

- The phone app talks to the router and the Master Node when the user tries to install devices, and set up sets and relations.
- For sets and relations, it particularly modifies the **.config** files in the folder **iot2/localconfig/mysql**.
- Executing **make** in **iot2/iotjava** would copy this folder entirely into **iot2/bin/iotruntime**; a Vigilia application would instantiate device drivers, sets, and relations at runtime using the queries stored in **iot2/bin/iotruntime/mysql**.

- Therefore, we need to execute **make** in **iot2/iotjava** or copy the **.config** files manually from **iot2/localconfig/mysql** into **iot2/bin/iotruntime/mysql**, every time we use the phone app to change applications and device drivers settings.

2.7.Running the First Application

To test our initial setup, we need at least one LIFX Color 1000 light bulb as smart home IoT devices (two light bulbs would be ideal). We have to follow the instructions in Section 2.5. to install the hardware. Then, we can go and run the script in **run.bash**.

Command Listing

cd iot2/bin/iotruntime
vim run.bash # check that we are running Lifxtest (the first application as a test)
./run.bash # run the test application
./cleanrun.bash # clean up as we are done

Note: If this is successful, then we will be able to see the light bulbs blinking and changing colors/intensity one after the other. **Congratulations on running the first application!** :)

3. Software Installation

3.1.Running Applications

We have 4 (four) existing applications that we created to test and evaluate Vigilia, namely:

1. smart lights,
2. irrigation,
3. smart music, and
4. home security.

The easiest way to run the applications is by using **run.bash**. We can open **run.bash** and modify it to choose the application that we wish to run. Another alternative to run the applications is by directly typing the command.

Command Listing

cd iot2/bin/iotruntime
vim run.bash # choose the application that we wish to run
./run.bash # run the test application
./cleanrun.bash # clean up as we are done

Command Listing

cd iot2/bin/iotruntime
java -cp .:/usr/share/java/:./../:./../iotcode/ iotruntime.master.IoTMaster* ***<application-name>*** *# we can run 1 application by typing the name of the main application class, e.g. SmartLightsController*
java -cp .:/usr/share/java/:./../:./../iotcode/ iotruntime.master.IoTMaster* ***<application #1> <application #2> … <application #n>*** *# we can run multiple applications by typing the application names sequentially as command line options*

3.2.Smart Lights Application

This is the first complete application that we built for Vigilia. The **Smart Lights** application attempts to save energy by turning lights off in unoccupied spaces, and to improve sleep by adjusting brightness and color temperature to match the sun's color. Aside from the Raspberry Pi's and router that we have installed, we need to have the following smart home IoT devices installed to make this benchmark work.

1. two LIFX Color 1000 light bulbs, and
2. two Amcrest IP2M-841 ProHD 1080P cameras.

If the two have been installed correctly, we need to check them.

Command Listing

cd /root/setup # do this on the router
./dhcp.sh # see the DHCP mapping for devices to see our 2 light bulbs and 2 cameras
ping <ip-address> # ping the light bulbs and the cameras

In our testbed, we have the following light bulbs (*192.168.1.232* and *192.168.1.126*) and cameras (*192.168.1.91* and *192.168.1.83*).

```
1519839712 9c:8e:cd:0f:e9:d8 192.168.1.91 AMC0001BN0U96Z0XMR *
1519802799 9c:8e:cd:0f:f1:60 192.168.1.83 AMC0007LD7VK4U6B04 *
1519833306 d0:73:d5:02:41:da 192.168.1.232 * *
1519791645 d0:73:d5:12:8e:30 192.168.1.126 * *
```

Then we can try to ping them from the router and also the **master** Raspberry Pi to make sure that all the connections are good.

If all the connections are good, then we can either run the **Smart Lights** application using **run.sh** or a direct command line.

Command Listing

```
cd iot2/bin/iotruntime
vim run.bash # choose the Smart Lights application that we wish to run
# alternatively: java -cp .:/usr/share/java/*:./../:./../iotcode/ iotruntime.master.IoTMaster SmartLightsController
./run.bash # run the application
./cleanrun.bash # clean up as we are done
```

Note: We can monitor a running application from the log files in **iot2/bin/iotruntime/log**.

Command Listing

```
tail -f log/SmartLightsController.log # monitor the application log file from iot2/bin/iotruntime/
```

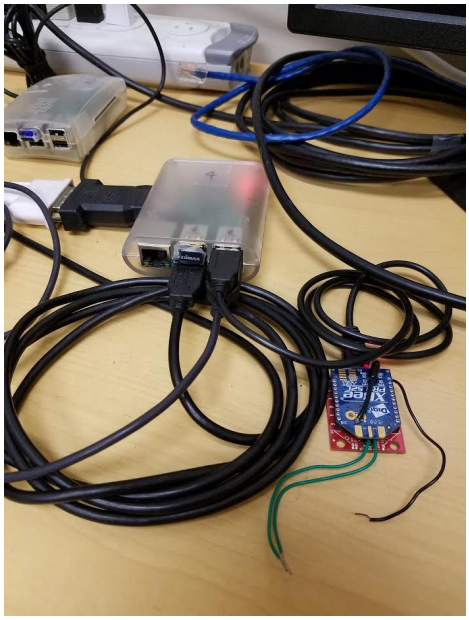
If the application runs correctly, then we can try to turn on the light bulbs by triggering the cameras. One camera is associated with one light bulb, so a trigger to a camera will turn on a light bulb.

3.3.ZigBee Gateway

Before we can run the next application, namely **Irrigation** application, we need to install a Vigilia ZigBee gateway.

In our testbed, we use the following hardware:

1. one Raspberry Pi 1 Model B+ as the ZigBee gateway, and
2. one XBee Pro S2C RF module (see the following picture).



Please follow the following instructions to install a Vigilia ZigBee gateway:

1. First we need to install Raspbian on the Raspberry Pi (see Section **2.2.**).
2. Then please follow the instructions in Section **2.3.** from **step 2-7**. We do not need the setups for Tomoyo Linux, Vigilia code, etc. on this Raspberry Pi.
3. We can download the entire Vigilia code, but all we need is **iot2/benchmarks/other/XbeePythonDriver/xbee_driver.py**.
4. We need to also extract **iot2/benchmarks/other/XbeePythonDriver/XBee-2.2.3.tar.gz** and we need to install it.
   Command Listing
   tar -zxvf XBee-2.2.3.tar.gz # extract
   cd XBee-2.2.3 # go into the directory
   sudo python setup.py install # install the xbee library
5. There are a few things that might need adjustments in **xbee_driver.py**.
   - **UDP_RECEIVE_PORT**: UDP receiver port (default is **5005** for Vigilia).
   - **SYSTEM_MASTER_ADDRESS**: the address of our master Raspberry Pi.
   - **LOCAL_ADDRESS**: the address of the local Raspberry Pi.
6. Run the XBee Python gateway program.
   Command Listing
   sudo python xbee_driver.py

Thereafter, we can test ZigBee devices that are used in our Vigilia applications. So, far we have integrated 5 devices into our applications:

- Spruce soil moisture sensor,
- Samsung SmartThings motion sensor,
- Samsung SmartThings multipurpose sensor,
- Samsung SmartThings water-leak sensor, and
- Kwikset SmartCode 910 ZigBee door lock.

To test the ZigBee devices, please follow the following instructions:

1. After running the Python driver code, join the devices into the ZigBee gateway's network, e.g. re-join network for Spruce sensor.
2. Go to **iot2/benchmarks/other/ZigbeeTest**.
3. Depending on our ZigBee devices, router, and compute node setup, we will need to modify **MY_IP_ADDRESS**, **DEVICE_MAC_ADDRESS** (ZigBee long address), and **GATEWAY**.
4. Execute **make** or **make all** to compile the test files.
5. We can run each of the tests depending on the device that we install, i.e. **make run-moisture**, **make run-motion**, **make run-multipurpose**, **make run-waterleak**, or **make run-doorlock**.

Note:

- We can run all the 5 tests simultaneously.
- We can run the tests multiple times without resetting any of the devices after joining the ZigBee network. However, after a few iterations of restarting the tests, the driver and XBee Pro S2C will become less stable, e.g. messages to and from the ZigBee get stuck, and the test application will loop forever.
- The Spruce moisture sensor is perhaps the most chatty ZigBee device that we have. It is advisable that we run the applications in the following order: **motion / multipurpose / waterleak / doorlock**, then **moisture sensor**. Otherwise, the ZigBee gateway driver will become unstable and mess up the messages to and from the devices.

4. Vigilia Applications

4.1.Phone Application

We have 3 Android phone applications for our applications. They are all located in **iot2/benchmarks/other/PhoneInterface**.

Phone App	**Application**	**Mechanism**
Control	Home security	An app that controls the alarm through our secure cloud application
Irrigation	Irrigation	An app that controls the input parameters, i.e. rainfall, duration, zip code, etc. through a phone gateway
SpeakerLocator	Smart music	An app that uses an indoor positioning library to locate and correlate the positions of speakers and the phone

Steps to install phone applications:

1. Download Android studio on your computer and set it up---we used Android studio on Ubuntu 16.04 and here is how to install it.
2. Open the Android studio and the phone app, then choose **Build** and **Make Project**.
   Note: If there is any error regarding build tools, then we need to ensure that the build tools version is correct (please see this posting). Please also make sure that the SDK platform API version is correct. During the write-up of this documentation, we use:
   - **SDK API** Version **27**
   - **Build Tools** Version **27.0.3**

   Depending on which smartphone (Android version) we use, there is going to be an update on the SDK API to compile and run the APK on the phone, e.g. *our Android 7.1.2 needed SDK API Version 25*.
   While running the APK, we can disable the auto-run mode if we hit into this error.

4.2.Irrigation Application

This is the second complete application that we built for Vigilia and it involves a ZigBee gateway, a smartphone gateway, and a smartphone Android app. The **Irrigation** application optimizes watering to conserve water. It regulates watering using a number of factors:

- moisture sensor,
- camera,
- weather forecast, and
- statistics/calculations.

To run this application we need the following hardware to be installed.

1. one Amcrest IP2M-841 ProHD 1080P cameras,

2. one Blossom sprinkler controller,
3. one Google Nexus 5X smartphone, and
4. one Spruce moisture sensor.

The Spruce moisture sensor will be connected to the Vigilia ZigBee gateway, so we can try pinging the gateway.

Command Listing

cd /root/setup # do this on the router
./dhcp.sh # see the DHCP mapping for devices
ping <ip-address> # ping the devices

In our testbed, we have the following DHCP entries on the router.

*1520105936 3c:ef:8c:7f:c0:19 192.168.1.31 AMC000BS82YUUW06A0 **
1520105932 3c:ef:8c:6f:79:5a 192.168.1.134 AMC0002CL8K0V80ZJZ *
1520105845 28:c2:dd:47:17:b6 192.168.1.129 Blossom-17B6 *
1520105155 64:bc:0c:43:3f:40 192.168.1.108 android-f1d388bc2c991bfa 01:64:bc:0c:43:3f:40
*1520025797 9c:8e:cd:0f:e9:d8 192.168.1.91 AMC0001BN0U96Z0XMR **
*1520025797 9c:8e:cd:0f:f1:60 192.168.1.83 AMC0007LD7VK4U6B04 **
1520106945 74:da:38:0d:05:56 192.168.1.192 raspberrypi4 01:74:da:38:0d:05:56
*1520069948 d0:73:d5:02:41:da 192.168.1.232 * **
*1520091279 d0:73:d5:12:8e:30 192.168.1.126 * **
1520068521 74:da:38:68:72:8a 192.168.1.198 raspberrypi1 01:74:da:38:68:72:8a
1520107651 74:da:38:0d:05:55 192.168.1.191 raspberrypi2 01:74:da:38:0d:05:55

Then we can try to ping them from the router and also the **master** Raspberry Pi to make sure that all the connections are good.

Note: The lines printed in bold are the devices needed for this application.

To run the application:

1. Run the phone application **Irrigation** and make sure that the *Gateway IP* field contains the IP address of the phone gateway, i.e. *WeatherPhoneGateway*---to know this, we need to wait until Vigilia finishes running. Please have a look at **iot2/benchmarks/drivers/Java/WeatherPhoneGateway** for more information.
2. Turn on the ZigBee gateway and then the Spruce soil moisture sensor. When the Spruce sensor is connected to the gateway, the Python program will print out "*get Report attribute*".
3. For Blossom sprinkler, we also need to change the DNS setup for dnsmasq inside **/etc/hosts** so that *home.myblossom.com* will point to the location of our driver.

   Command Listing

   vim /etc/hosts # then add an entry "192.168.1.191 home.myblossom.com" on the router
   # 192.168.1.191 is the address of our BlossomSprinkler driver process
   reboot # need to reboot the router after adding this entry
   Note:

- We also need to update the CHANNELID field in the database for our Blossom sprinkler. It is usually different and this information can be obtained by sniffing into the packets (using Wireshark/tcpdump) or running our **BlossomSprinkler.java** driver. When the sprinkler controller tries to talk to our driver, which also acts like a server, it will print out the GET message that is coming from the device,
  i.e. *GET /api/device/v1/sc/**<channel-id>**/activateschedule/ HTTP/1.1*.
- Blossom sprinkler uses a list of struct for the ZoneState object; in Java there is a limitation that we can only use **public static** format to be able to access the class properties if we do not want to use *setter* and *getter* methods. Since it is a static struct, we can only have one of these per JVM. Hence, we stick to one zone for now.
- We can test that the driver can contact the sprinkler controller by running it as a standalone process (see the commented **main** function in **BlossomSprinkler.java**). To do this, we need to manually fill out the variables that contain addresses, i.e. **IoTDeviceAddress** and **IoTZigbeeAddress**.

4. If all the connections are good, then we can either run the **Irrigation** application using **run.sh** or a direct command line.
5. Then the application will stop at printing out "*DEBUG: Waiting for the phone to send weather information*"
   Note: We can monitor a running application from the log files in **iot2/bin/iotruntime/log**.
   Command Listing
   tail -f log/IrrigationController.log # monitor the application log file from iot2/bin/iotruntime/

Note: This is a tough application to run so, sometimes the ZigBee gateway and the Spruce moisture sensor are not working properly, or the BlossomSprinkler driver could not contact the sprinkler controller. Thus, we have to make sure that we have tested every component separately and they are able to function individually.

4.3.Smart Music Application

This is the third complete application that we built for Vigilia and it involves a smartphone gateway, and a smartphone Android app. The **Smart Music** application tracks people using WiFi-based indoor localization of their cell phone and plays music from the closest speakers (we use a library from here that has still been usable until Android 7.0.0---so far it has not been perfectly accurate but this is not our main focus of attention).
To run this application we need the following hardware to be installed.

1. two iHome iWS2 AirPlay stereo speakers, and
2. one Google Nexus 5X smartphone.

Command Listing
cd /root/setup # do this on the router
./dhcp.sh # see the DHCP mapping for devices
ping <ip-address> # ping the devices

In our testbed, we have the following DHCP entries on the router.

1520190580 c8:d5:fe:e6:ad:96 192.168.1.119 * 01:c8:d5:fe:e6:ad:96
1520190551 c8:d5:fe:e6:a2:d8 192.168.1.234 * 01:c8:d5:fe:e6:a2:d8
1520186605 64:bc:0c:43:3f:40 192.168.1.109 android-f1d388bc2c991bfa 01:64:bc:0c:43:3f:40
...
1520106945 74:da:38:0d:05:56 192.168.1.192 raspberrypi4 01:74:da:38:0d:05:56
1520068521 74:da:38:68:72:8a 192.168.1.198 raspberrypi1 01:74:da:38:68:72:8a
1520107651 74:da:38:0d:05:55 192.168.1.191 raspberrypi2 01:74:da:38:0d:05:55

Then we can try to ping them from the router and also the **master** Raspberry Pi to make sure that all the connections are good.
Note: The lines printed in bold are the devices needed for this application.

To run the application:

1. If all the connections are good, then we can either run the **SpeakerController** application using **run.sh** or a direct command line.
2. Run the phone application **SpeakerLocator**. After Vigilia finishes its process and we see "*All Speakers done loading*" in the **SpeakerController.log** file, we can map our 2 locations:
   - go to one room, input 0, and press MAP,
   - go to the other room, input 1, and press MAP, then
   - press start.

   Note: At this point, the phone is sending our location to the **SpeakerController** process as we move from one room to the other.
   Please have a look at **iot2/benchmarks/drivers/Java/GPSPhoneGateway** for more information about the phone gateway for this application.

Note:

- For most runs, we can only see that when the speaker is triggered, the **IHome.java** driver is going to print out HTTP messages that it is sending to its respective speaker, but we probably could not hear the music playing. This is because we are trying to load the MP3 file and play it using the RMI mechanism using the slow Raspberry Pi.
- We can try to play the music directly without using RMI by just compiling and running the **IHome** driver individually (see the commented main function in **IHome.java**).

4.4.Secure Cloud Server

Aside from the phone gateway, we have also developed a secure cloud library that allows our application to have an encrypted and oblivious storage data structure on the cloud. We have to first set up this cloud server, before we can use it with our phone application in the fourth Vigilia application. The source code of this cloud server will be released separately on our website.

1. Clone the **git** repository,
   Command Listing
   git clone ssh://plrg.eecs.uci.edu/iotcloud.git # clone the source code
   cd iotcloud/version2/src/server
   make
2. We need to first install **Fast CGI** and **Apache2** server on the server machine. The secure data structure is implemented using both mechanism.
3. Then, we need to place the compiled **Fast CGI** code in the right location and start the Apache server.
   Command Listing
   # cd to the location where the code was compiled on the server machine
   sudo rm /usr/lib/cgi-bin/iotcloud.fcgi # replace the Fast CGI file with our generated file
   sudo cp iotcloud.fcgi /usr/lib/cgi-bin/iotcloud.fcgi
   sudo service apache2 restart
   Note:
   - The server is now ready to serve requests from applications that use the secure cloud library.
   - Before every run of our application, we need to make sure that the directory for the data structure is empty. Hence, we must delete old data for every run.
     Command Listing
     *sudo rm -rf /iotcloud/test.iotcloud/**

4.5.Home Security Application

This is the fourth complete application that we built for Vigilia and it involves a smartphone Android app that connects to a secure cloud database. The **Home Security** application consists of multiple sensors that can detect intrusions/anomalies and sound an alarm.

To run this application we need the following hardware to be installed.

1. one Amcrest IP2M-841 ProHD 1080P cameras,
2. one Google Nexus 5X smartphone,
3. one Samsung SmartThings motion sensor,
4. one Samsung SmartThings multipurpose sensor,
5. one Samsung SmartThings water-leak sensor, and
6. one Kwikset SmartCode 910 ZigBee door lock.

The ZigBee devices (3 sensors and 1 door lock) will not be seen in the DHCP table, but we basically have to make sure that the ZigBee gateway is well-connected.
Command Listing
cd /root/setup # do this on the router
./dhcp.sh # see the DHCP mapping for devices
ping <ip-address> # ping the devices

In our testbed, we have the following DHCP entries on the router.

1520532984 c4:12:f5:de:38:20 192.168.1.4 DCH-S220 01:c4:12:f5:de:38:20
...
1520497423 3c:ef:8c:6f:79:5a 192.168.1.134 AMC0002CL8K0V80ZJZ *
...
1520508537 74:da:38:68:72:8a 192.168.1.198 raspberrypi1 01:74:da:38:68:72:8a
1520507329 74:da:38:0d:05:55 192.168.1.191 raspberrypi2 01:74:da:38:0d:05:55
1520512916 74:da:38:0d:05:56 192.168.1.192 raspberrypi4 01:74:da:38:0d:05:56

Then we can try to ping them from the router and also the **master** Raspberry Pi to make sure that all the connections are good.
Note: The lines printed in bold are the devices needed for this application.
To run the application:

1. If all the connections are good, then we can either run the **Home Security** application using **run.sh** or a direct command line.
2. While the *phone is on a different network*, run the phone application **Control**. We have to let the phone talk to the cloud server via the secure cloud library (see Section **3.7**). In our lab, we dedicate one machine as a cloud server that has an individual public IP address. In our testbed, we use the machine that we named *dc-6.calit2.uci.edu* (IP address *128.195.136.167*).
   Note:
   - Before the application starts, we will see "**OFF**" status printed in red on the phone application. This means that the alarm is still in "**OFF**" condition.
   - Before every run of our application, we need to make sure that the directory for the data structure is empty. Hence, we must delete old data for every run.
     Command Listing
     *sudo rm -rf /iotcloud/test.iotcloud/**
3. We need to turn on the ZigBee gateway Python program and let the devices connect to it.
   Note:
   - The best way to do this is by pairing all devices with the gateway first and do the tests (Section **3.3**).
   - Then, before every run, we disconnect the batteries of all devices, and reconnect them again to make the devices join the ZigBee network.
   - We need to connect the motion, multipurpose, and water-leak sensors first, before the door-lock---this seems to be the most stable way to do it with the current version of the ZigBee gateway.
   - Then, we can quickly run the application.
4. If all the connections are good, then we can either run the **Home Security** application using **run.sh** or a direct command line.
   Note:
   - At some point, when Vigilia finishes the instantiation of the processes, and the devices, i.e. sensors, camera, and door-lock, are all initialized, we should see

"*DEBUG: Initialized all devices! Now starting detection loop!*" being printed in **iot2/bin/iotruntime/log/HomeSecurityController.log**.

- The phone app will report that the alarm is now "**OFF**" and we can make it go off by triggering the sensors or the camera.
- When something is triggered properly, the application will *turn on the alarm* and *lock the door*. In this case, we have to wait until the alarm is really triggered---the default setting in the alarm driver is for the alarm to just **blink red** continuously and not sound anything (we can change this in the driver *DlinkAlarm* if needed).
- Then we will see on the phone app that the alarm is now "**ON**". At this point, we press the button to turn it "**OFF**" and it will turn off the alarm and the application will go back into the detection loop.
- We can repeat this process multiple times.

Note:

- One caveat is that after we trigger the alarm to go off and the door is locked, it seems that the ZigBee gateway program will stuck at serving only the door, and it no longer responds to the callbacks from the sensors that report any events.
- Before the door-lock is triggered, we can still trigger the other sensors to make the alarm go off.

4.6.Remarks on Applications

The four applications have been heavily tested individually with all the features that Vigilia has. Vigilia does have a feature to run multiple applications at once. We can run all the four applications simultaneously, but this is hard to test since we have to reset/setup a lot of components at once before we run multiple applications. However, we have not heavily tested this---**Smart Lights** and **Speakers** are quite easy to run, but **Irrigation** and **Home Security** are a lot harder to set up, let alone run the four of them simultaneously.

Command Listing

java -cp .:/usr/share/java/:./../:./../iotcode/ iotruntime.master.IoTMaster HomeSecurityController SmartLightsController SpeakerController IrrigationController # or we can uncomment this in run.bash*

4.7.C++ Applications - Preliminaries

Aside from Java applications, we can also develop C++ applications on Vigilia. Section 2.3 explains about the compilations of the C++ version of the runtime and applications.

A few important notes about this:

- We have the C++ slave and other runtime libraries in **iot2/iotjava/iotruntime/cpp/**.
- Inside **iot2/benchmarks/Cpp/** we have a C++ test application and **iot2/benchmarks/drivers/Cpp/** we have C++ drivers.
- The Vigilia slave for C++, along with the test application and the drivers for C++ were developed using **gcc 4.9.3**.

- Therefore, it is advisable to download gcc 4.9.3 from the Raspbian repository, install it, and compile the C++ source code using it on every Raspberry Pi that we have installed.

Command Listing

```
sudo apt-get update
sudo apt-get gcc-4.9 g++-4.9
# to use g++ 4.9.3 we simply have to run g++-4.9
# invoking just g++ will give us gcc version 6.3.0 20170516 as the default gcc for this Raspbian
```

Note: The files can be compiled using the current **gcc** compiler (**v.6.3.0**) for the current version of Raspbian, but we have to adjust the implementation that will be out of the scope of this documentation.

4.8.Running C++ Application

Section 2.3 explains about the compilations for the files for both the C++ version of Vigilia runtime, application, and drivers. Due to our limited time and resources:

- We built the complete set of runtime features, e.g. router policy, Tomoyo, etc. for both Java and C++.
- We rewrote two of our drivers from Java to C++, i.e. **LabRoom** and **LifxLightBulb**. With these two we could test the runtime features that we have, also for C++.
- We have not developed the ZigBee features for C++ since we need the relevant drivers to test them.
- With both Java and C++, we can choose between the 2 when we run our specific drivers and controllers. Please see **LabRoom.config** and **LifxLightBulb.config** for examples of the language feature.

```
LANGUAGE=Java # The main language is Java for either a controller or a driver
LANGUAGE=C++ # The main language is C++ for either a controller or a driver
```

- We can also define more fine grained specification in the language feature. This way we can combine different instances of controller and drivers in either C++ or Java.

```
LANGUAGE_LifxLightBulbLB1=Java # The language for this specific instance of LifxLightBulb driver is Java
LANGUAGE_LifxLightBulbLB2=C++ # The language for this specific instance of LifxLightBulb driver is C++
```

The C++ side has not been heavily tested, but we have one application that can be run.

1. After the compilation of the Vigilia runtime and applications (Section 2.3), we can run a test application called **Lifxtest** in **iot2/bin/Lifxtest**.
    - The Java version of this application creates a set of LifxLightBulb instances and exercises the driver's functions, i.e. turning on/off, temperatures, and colors.

- The C++ version of this application creates a set of LifxLightBulb and a set of LabRoom instances. Then, the application creates a relation object to relate the two types of instances, e.g. LabRoom 1 is connected to LifxLightBulb 1, LabRoom 2 is connected to LifxLightBulb 2, etc. This version also exercises the same driver's functions.

Note:

- The C++ version has not been heavily tested, but one combination that has been tested is the C++ Lifxtest controller, accompanied with the Java LifxLightBulb driver instances, and the C++ LabRoom instances.
- The LifxLightBulb driver for C++ has been tested to run on itself and also using the RMI stub and skeleton. However, this driver seems to have a problem with its *Shared Object (.so)* version, e.g. using the C++ LifxLightBulb driver instance in the application does not really turn on the light bulb for some reason although the standalone run of the same LifxLightBulb driver works really well.

2. To test out the combination, we have to:
   - Check and modify the LANGUAGE option in **iot2/bin/Lifxtest/Lifxtest.config** into "**C++**".
   - Check and modify the LANGUAGE option in **iot2/bin/iotcode/LifxLightBulb/LifxLightBulb.config** into "**Java**".
   - Check and modify the LANGUAGE option in **iot2/bin/iotcode/LabRoom/LabRoom.config** into "**C++**".

Then we can execute the commands in the listing below to run the application.

Command Listing

cd iot2/bin/iotruntime
*vim run.bash # choose the application **Lifxtest** that we wish to run*
./run.bash # run the test application
./cleanrun.bash # clean up as we are done

Command Listing

cd iot2/bin/iotruntime
java -cp .:/usr/share/java/:./../:./../iotcode/ iotruntime.master.IoTMaster **Lifxtest***

Note: When the runtime gets stuck due to the different interactions of threads, we can simply execute **cleanrun.bash** and **run.bash** again to rerun it.

5. Vigilia Compiler

5.1.Compiling the Compiler

The make command that we execute in **iot2/iotjava/** should compile the Vigilia compiler as well, i.e. *make tree*, *make parser*, and *make compiler*. This compiler is written fully in Java. The parser has been automatically generated by using the Java CUP infrastructure. Please have a look at this and this for more information about the Java CUP.

5.2.RMI Stubs/Skeletons Generation for One Driver

In general, the Vigilia compiler has the following specifications. We can take the *AmcrestCamera* as an example.

1) It takes **policy** (.pol) and **requirement** (.req) files to generate. Please see **iot2/localconfig/iotpolicy/AmcrestCamera** that contains
    - amcrestcamera.pol,
    - smartlightscam.req,
    - motiondetection.pol, and
    - motiondetection.req.

    Basically, the class *MotionDetection* is the callback class for the *AmcrestCamera* class.
2) Three distinctive features of the Vigilia compiler are the followings:
    - It can **generate** stub/skeleton classes for **both C++** and **Java**. Its API library can transport method invocations and method parameters between the two languages.
    - It allows programmers to **select required methods** based on object capabilities model, e.g. please see *amcrestcamera.pol* (this defines the available methods and capabilities that cluster the methods into categories) and *smartlightscam.req* (this gives the ability to select only the required capabilities, i.e. methods).
    - It does **static analysis** to ensure that the callback class is guaranteed to be not passed around between remote objects, i.e. drivers and controllers, because this would generate runtime errors as this would violate the router firewall and Tomoyo policies.
3) The compiler gets the specification of the interfaces, namely *Camera* and *CameraCallback* from the **amcrestcamera.pol** and **motiondetection.pol** files. From this it is going to generate 2 new interfaces in addition to the 2 specified interfaces:
    - Camera,
    - CameraCallback
    - CameraSmart, and
    - CameraSmartCallback.

    Basically, the compiler is going to do **static analysis** on the interfaces and it is going to swap the callback class name that acts as an input to the callback function in the *Camera* class. Thus, initially we have the method "*public void registerCallback*

(CameraCallback _callbackTo)" in the *Camera* class. After the static check, using the additional **requirement** (.req) files, the compiler generates:

Interface	**Callback Method**
Camera	*public void registerCallback(CameraSmartCallback _callbackTo)*
CameraCallback	*N/A*
CameraSmart	*public void registerCallback(CameraCallback _callbackTo);*
CameraSmartCallback	*N/A*

For example, we use the *Camera* object for the *SmartLightsController* application. When the runtime runs and instantiates the objects:

- On the *Camera* driver side, we will have the interfaces *Camera* and *CameraSmartCallback*. The actual *Camera* object will reside here and it will be contained by the skeleton *Camera_Skeleton* class. This class will call the *CameraSmartCallback* class, contained by the stub *CameraSmartCallback_Stub* class, whenever the *Camera* object needs to do a callback to the actual *CameraCallback* object.
- On the *SmartLightsController* side, we will have the interfaces *CameraSmart* and *CameraCallback*. Here, every time the controller needs to call a method on the actual *Camera* object, it will use the stub *CameraSmart_Stub* that will connect to the skeleton *Camera_Skeleton*. The actual callback object *CameraCallback* is located here and it is contained by the skeleton *CameraCallback_Skeleton*. Every time there is a callback from the *Camera* object side using the stub *CameraSmartCallback_Stub*, this stub will connect to the skeleton *CameraCallback_Skeleton*.

An example of basic run of the compiler can be done with the following command (taking the *AmcrestCamera* class as an example). Basically we use the policy and requirement files to generate the interface, stub, and skeleton classes. We also need to specify the **driver** and **controller** class names using the options "*-cont*" and "*-drv*" to let the compiler generates package names correctly.

Command Listing

```
cp ../localconfig/iotpolicy/AmcrestCamera/*.pol ../bin/iotpolicy/
cp ../localconfig/iotpolicy/AmcrestCamera/*.req ../bin/iotpolicy/
cd ../bin/iotpolicy; java -cp .:..:../../jars/java-cup-bin-11b-20160615/*:../../bin iotpolicy.IoTCompiler
-cont SmartLightsController amcrestcamera.pol smartlightscam.req -drv AmcrestCamera
motiondetection.pol motiondetection.req -drv AmcrestCamera -java Java
```

5.3.RMI Stubs/Skeletons Generation for One Application

As an example, we can run the stub and skeleton generation for the application *SmartLightsController* by running "*make run-compiler-smartlight*" from *iot2/iotjava/*.

Command Listing

```
make run-compiler-smartlight          # we can simply run this make command or the full list
```

Command Listing

```
cp ../localconfig/iotpolicy/LifxLightBulb/*.pol ../bin/iotpolicy/
cp ../localconfig/iotpolicy/LifxLightBulb/*.req ../bin/iotpolicy/
cp ../localconfig/iotpolicy/AmcrestCamera/*.pol ../bin/iotpolicy/
cp ../localconfig/iotpolicy/AmcrestCamera/*.req ../bin/iotpolicy/
cp ../localconfig/iotpolicy/Room/*.pol ../bin/iotpolicy/
cp ../localconfig/iotpolicy/Room/*.req ../bin/iotpolicy/
cd ../bin/iotpolicy; java -cp .:..:../../jars/java-cup-bin-11b-20160615/*:../../bin iotpolicy.IoTCompiler
-cont SmartLightsController lifxlightbulb.pol smartlightsbulb.req -drv LifxLightBulb
amcrestcamera.pol smartlightscam.req -drv AmcrestCamera motiondetection.pol
motiondetection.req -drv AmcrestCamera room.pol roomsmart.req -drv LabRoom -java Java
```

After we run these commands:

1. We can see the generated files in *iot2/bin/iotpolicy/output_files*.
2. The directory *interfaces* outside contains the original *interfaces* of the application.
3. In the folder Java, we can see 3 directories, i.e. *controller*, *drivers*, and *interfaces*.
4. The inside *interfaces* directory contains all the new interfaces that are generated as a result of the compiler's static checking.
5. The *controller* directory contains all the stub/skeleton classes that will be on the controller side at runtime. This will usually be stub classes or skeleton classes for callback classes.
6. The *drivers directory contains all the drivers involved in the compilation*. Each driver directory, e.g. LabRoom, will usually contain skeleton classes or stub classes for callback classes.

6. Tutorial for Developer

This is a tutorial for any developers who wish to include a new subtype into the database and, thus, the Vigilia phone app. This tutorial uses the **DlinkAlarm** subtype that comes with the **iot2** repository. At this point, this subtype is used in the **HomeSecurity** application, but the driver's, along with the phone app's, config files have not yet been modified for it. In other words, the **DlinkAlarm** subtype would not appear on the phone app interface unless one does the following steps to add it into the config files and the Vigilia database.

Note:

- This tutorial assumes that the developer has written the driver and application code altogether, and needs to integrate them with the database and phone app.
- At the moment, we would suggest that if one wish to become a Vigilia developer, they would need to do the following steps:
    - read the publication on the Vigilia system;
    - try to follow the pattern from our existing applications, starting from the **Lifxtest** application, which is the simplest one;
    - basically a Vigilia application needs the following components: 1) main application (that contains **IoTSet** and **IoTRelation**), 2) device drivers (that contains IoTSet for **IoTDeviceAddress** and/or **IoTZigBeeAddress**), 3) RMI stubs and skeletons (can be generated using the Vigilia compiler - see Section **5**) and 4) new application/driver **.config** files along with adjustments for the existing **.config** files.
    - after understanding **Lifxtest**, the **SmartLightsController** would be the next best example that combines **IoTSet** and **IoTRelation**, while the **IrrigationController** is great to understand more about the ZigBee drivers and devices.

6.1.Adding Subtype to Vigilia

First we need to go to **iot2/benchmarks/drivers/Java** and create a new directory with the name of the subtype that you're adding. Within this directory you will need to place the **.config** and **.java** files that are associated with that subtype.

6.2.Modifying .config File for the Subtype

Note: Some subtypes will not have any **IoTSet<IotDeviceAddress>** or **IoTSet<IoTZigbeeAddress>** variables, in this case you can just skip this section. Also some subtypes already have **FIELD** information within the **.config** file, you can also skip this step if that is the case.

1. When adding a new subtype, we need to modify the **.config** file using the **.java** file of the subtype with the field information of the device.
2. Open up the **.java** file associated with the subtype, then look for **IoTSet<IoTDeviceAddress>** or **IoTSet<IoTZigbeeAddress>** datatypes.
    - Make note of how many variables are associated with IoTSet.
    - Record the variable names of all of the variables with this data type.
    - Also make note if a variable is a ZigBee address or a regular device (MAC) address.
3. Once you have collected the information in the previous step, go into the **.config** file associated with the subtype. We will be adding the **FIELD** information within this file.
    - First place how many variables you found, with **FIELD_NUMBER=#**, where **#** is the number of **IoTSet<IoTDeviceAddress>** and **IoTSet<IoTZigbeeAddress>** variables you recorded.
4. For each **IoTSet<IoTDeviceAddress>** and **IoTSet<IoTZigbeeAddress>** we will be making a block of **FIELD** information.
    - For each variable, put **FIELD_#**, where **#** is the current variable block equaling the variable name. For example, if we recorded two **IoTSet<IoTDeviceAddress>** variables with name **devUdpAddress** and **devZigbeeAddress** we would have this:
        - **FIELD_NUMBER=1**
        - **FIELD_0=devUdpAddress**
        - **FIELD_1=devZigbeeAddress**
5. For each **FIELD_#** we need to put the field class, field type, and field independent.
    - **FIELD_CLASS_#** is if the IoTSet is of type **IoTDeviceAddress** or **IoTZigbeeAddress**.
    - **FIELD_TYPE_#** would be equal to IoTSet since that is the datatype of the variables.
    - **FIELD_INDEPENDENT_#** would be equal to true if the the **FIELD** belongs to the independent set, usually is noted within the **.java** file. If not then the default should be **TRUE**.
    - A finished **.config** file would look like this:
        - **FIELD_NUMBER=2**
        - **FIELD_0=devUdpAddress**
        - **FIELD_CLASS_0=IoTDeviceAddress**
        - **FIELD_TYPE_0=IoTSet**
        - **FIELD_INDEPENDENT_0=TRUE**
        - **FIELD_1=devZigbeeAddress**
        - **FIELD_CLASS_1=IoTZigbeeAddress**
        - **FIELD_TYPE_1=IoTSet**
        - **FIELD_INDEPENDENT_1=TRUE**

Note:
In our case, for **DlinkAlarm.config** in **iot2/benchmarks/drivers/Java/DlinkAlarm**, we would have the following.

```
# Skeleton/original interface
INTERFACE_CLASS=Alarm
# Stub
INTERFACE_STUB_CLASS=AlarmSmart

# Language
LANGUAGE=Java

# Phone app/C++ instrumentation
FIELD_NUMBER=1
FIELD_0=alm_Addresses
FIELD_CLASS_0=IoTDeviceAddress
FIELD_TYPE_0=IoTSet
FIELD_INDEPENDENT_0=TRUE

# Tomoyo
ADDITIONAL_MAC_POLICY=No
```

6.3.Adding Subtype into SupportedDevices

1. Within the **.java** file associated with your added subtype, record the number **IoTDeviceAddress** and **IoTZigbeeAddress** variables there are. There may be none, it is also important to note if there is none. Also look at the **.java** class constructor and take note of any parameters that do not have the **IoTSet** datatype.
2. Within **iot2/localconfig/SupportedDevices/AddressInformation.config** you will make a new **.config** file that is associated with your new subtype that you're adding.
3. At the top of **.config** file type in the number of **IoTDeviceAddress** and **IoTZigbeeAddress** variables that you recorded, with the tag **ADDRESSES** and **ZBADDRESSES**.
    - For example, in **IHome** there are 5 variables that have the datatype **IoTDeviceAddress** and 0 variables that have the datatype **IoTZigbeeAddress** so it would look like this:
        - **ADDRESSES=5**
        - **ZBADDRESSES=0**
4. For each Address we are going to make a **DEVICEADDRESS_#**, **PORTNUMBER_#**, and **PROTOCOL_#** fields.
    - If you want the user to input either the device address(MAC address), port number, or protocol being used then you will set the field to be equal to "**USER**".
    - If you would like to autogenerate the fields in the database, then you will put the specific information in the field. An example of this might look like:
        - **DEVICEADDRESS_1=USER**
        - **PORTNUMBER_1=1024**
        - **PROTOCOL_1=tcp**

Note:
Some subtypes might need a **SOURCEWILDCARD_#** and **DESTWILDCARD_#** for each address if applicable within the **.java** file.

5. Then you will need to insert the device information that was obtained by looking at the constructor of the .java class associated with your new subtype.
6. For example, **AmcrestCamera**'s class constructor has username and password as parameters variables that are not **IoTSet** datatypes.
    - This means that the file for **AmcrestCamera** would have:
        - **DEVICEINFO=2**
        - **USERNAME=USER**
        - **PASSWORD=USER**
7. Again you will place "**USER**" if you want the user of the Android application to insert their information into the application.

Note:
In our case, for **DlinkAlarm**, we would have the following inside **iot2/localconfig/SupportedDevices/AddressInformation.config**.

```
TYPE_14=AlarmSmart
TAG_14=AL
SUBTYPE_14=2
TYPE_14_0=DlinkAlarm
TYPE_14_0_NUM_OF_ZBADDRESSES=0
TYPE_14_0_NUM_OF_ADDRESSES=1
TYPE_14_0_ADDRESS_FIELDS=PORTNUMBER PROTOCOL DEVICEADDRESS
PORTNUMBER_14_0_ADD_0=80
PROTOCOL_14_0_ADD_0=tcpgw
DEVICEADDRESS_14_0_ADD_0=USER
TYPE_14_0_NUM_OF_DEVICE_INFO=0
```

6.4.Adding Subtype into driversList.config

1. You will then need to place your subtype information into **driversList.config** within **iot2/localconfig/SupportedDevices/driversList.config**.
2. You will increment **FIELD_NUMBER** by one, and then add a new **FIELD_#** at the bottom with your specific subtype.
3. Then you will add the specific subtype as a field and set it equal to the **<subtype>/<subtype>.config**.
4. Once completely finished, you will need to run make again from **iot2/iotjava** to copy these files into **iot2/bin**.

Note:
In our case, for **DlinkAlarm**, we would have the following inside **iot2/localconfig/SupportedDevices/driversList.config**.

```
FIELD_NUMBER=15
COMMAND_PREFIX=cat ~/iot2/benchmarks/drivers/Java/
...
FIELD_14=DlinkAlarm
DlinkAlarm=DlinkAlarm/DlinkAlarm.config
```

6.5.Adding Subtype into Database

1. Currently, you will need to insert a copy of the subtype into the database.
2. First you will need to add your subtype to the table that the subtype falls under.
    - For example **DlinkAlarm** falls under **AlarmSmart**, and to add **DlinkAlarm** to the table you would use the command:
        - **INSERT INTO `AlarmSmart` VALUES ('AL#','DlinkAlarm');** where # is the next number in the sequence for that specific type.
            - Instead of **AlarmSmart**, **AL**, and **DlinkAlarm** you would change those values to be the values corresponding to your subtype.
3. Then you would need to make a new table for the device information and address information.
    - For **DlinkAlarm**, the device information table would be created with: **CREATE TABLE `DlinkAlarmAL#` ('DEVICEPIN' varchar(10) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin1;** instead of **DEVICEPIN** you would place the device information that you found in Section **6.4** where you're inserting the **DEVICEINFO**.
    - The Address information table would be created with: C**REATE TABLE `DlinkAlarmAddAL2` (`DEVICEADDRESS` varchar(20) DEFAULT NULL, `PORTNUMBER` int(11) DEFAULT NULL, `PROTOCOL` varchar(5) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=latin1;** again instead of **DlinkAlarmAddAL#**, you would place your subtype that you're adding as well as the ID and the number corresponding to that subtype.
4. After creating the tables, you will need to populate the tables with some temporary data. This can be deleted after the first subtype is inserted into the database successfully.
    - To populate the device information table: **INSERT INTO `DlinkAlarmAL2` VALUES ('215530');** again changing the required parameters where applicable.
    - To populate the address information table: **INSERT INTO `DlinkAlarmAddAL2` VALUES ('c4:12:f5:de:38:20',80,'tcpgw');** again changing the required parameters where applicable.

    Note:
    - Beside changing the **.config** files in the previous subsections, these database manipulation steps are necessary at the moment, for the application to see the addition of the subtype.

- Please take note that the tables **AlarmSmart**, **DlinkAlarmAL2**, and **DlinkAlarmAddAL2** already exist in the current version of IoTMain.gz database. You only have to perform steps in Sections **6.1** to **6.4** to complete this tutorial, and double check the information in the database according to Section **6.5**.