D2.3 Architecture Demonstrator

CitiSim

Smart City 3D Simulation and Monitoring Platform

ITEA3 – Project CitiSim

Document Properties

|  |  |
| --- | --- |
| Authors | Citisim Partners |
| Date | 2018-10-10 |
| Visibility | Consortium |
| Status | Final |

History of Changes

|  |  |  |  |
| --- | --- | --- | --- |
| Release | Date | Author, Organization | Changes |
| 0.0 | 01/06/2018 | Ismael Torres, Prodevelop | TOC - Initial Document |
| 0.1 | 18/07/2018 | Cristian Trapero Mora (UCLM) | Docker |
| 0.2 | 24/07/2018 | José Ignacio Mota Ortiz (UCLM) | MQTT |
| 0.3 | 26/09/2018 | George Suciu, Muneeb Anwar (BEIA) | Review of document |
| 0.4 | 27/09/2018 | Javier Sánchez (Answare) | Visual wiki |
| 0.5 |  |  |  |
| 1.0 | 10/10/2018 | Ismael Torres, Prodevelop | Final Document |

Executive Summary

The general purpose of CitiSim is devoted to the design and implementation of a new generation platform for the Smart City ecosystem.

The D2.2 architecture demonstrator is a software deliverable. This software contains the core elements of the CitiSim Platform/Architecture, which have been defined for the project. These elements have been developed during the execution of the project.

This document is a complement to the software developments and explains the architecture defined by all partners, as well as, how to configure and install it using docker technology.

List of Figures

[Figure 1: CitiSim architecture proposal 10](#_Toc526937546)

[Figure 2: Docker stack vs VM stack 14](file:///F:\IDI\Citisim\Citisim_svn\doc\WP2\Deliverables\D2.3\D2.3_ArchitectureDemonstrator_v1-%20final.docx#_Toc526937547)

[Figure 3: CitiSim deployment in Docker 15](#_Toc526937548)

[Figure 4: CitiSim Dashboard 16](#_Toc526937549)

[Figure 5: CitiSim IceStorm GUI 17](#_Toc526937550)

[Figure 6: CitiSim Node Controller 18](#_Toc526937551)

[Figure 7: CitiSim Persistence Service 19](#_Toc526937552)

[Figure 8: Persistence service main interface 20](file:///F:\IDI\Citisim\Citisim_svn\doc\WP2\Deliverables\D2.3\D2.3_ArchitectureDemonstrator_v1-%20final.docx#_Toc526937553)

[Figure 9: Example of Property Service tree structure 21](#_Toc526937554)

[Figure 10: Structure of Docker files 22](#_Toc526937555)

[Figure 11: CitiSim IceGrid Application (Core Services Node) 28](file:///F:\IDI\Citisim\Citisim_svn\doc\WP2\Deliverables\D2.3\D2.3_ArchitectureDemonstrator_v1-%20final.docx#_Toc526937556)

[Figure 12: Definition of the Property server in the IceGrid application 30](#_Toc526937557)

[Figure 13: Adapter configuration for Property server in the IceGrid application 31](#_Toc526937558)

[Figure 14: CitiSim IceGrid Application (Applications Node) 32](#_Toc526937559)

[Figure 15: Output of the docker-compose up 36](#_Toc526937560)

[Figure 16: Output of docker ps -a 37](#_Toc526937561)

[Figure 17: CitiSim IceGrid application running 37](file:///F:\IDI\Citisim\Citisim_svn\doc\WP2\Deliverables\D2.3\D2.3_ArchitectureDemonstrator_v1-%20final.docx#_Toc526937562)

[Figure 18: Output of docker- compose down 38](#_Toc526937563)

[Figure 19: Open Citisim IceGrid application in icegridgui 41](file:///F:\IDI\Citisim\Citisim_svn\doc\WP2\Deliverables\D2.3\D2.3_ArchitectureDemonstrator_v1-%20final.docx#_Toc526937564)

[Figure 20: Node Controller server properties 41](file:///F:\IDI\Citisim\Citisim_svn\doc\WP2\Deliverables\D2.3\D2.3_ArchitectureDemonstrator_v1-%20final.docx#_Toc526937565)

List of Tables

[Table 1: CitiSim device properties 20](#_Toc526937566)

[Table 2: JSON template for MQTT messages 44](#_Toc526937567)

[Table 3: Example of MQTT message 44](#_Toc526937568)

[Table 4: CitiSim.config example 46](#_Toc526937569)

[Table 5: mqtt.config example 46](#_Toc526937570)

Table of Contents

History of Changes 3

Executive Summary 4

List of Figures 5

List of Tables 6

1. Introduction 8

1.1. Project 8

1.2. Work package 8

1.3. Document overview 8

2. Architecture – an overview 10

3. CitiSim – Docker Infrastructure 14

3.1. Introduction: Automatic deployment 14

3.2. Core services: A brief description 16

3.3. Docker configuration 21

3.4. Setting-up a CitiSim scenario with docker 35

4. CitiSim-MQTT-adapter 44

4.1. Service description. 44

4.2. Interface description and implementation details 44

4.3. Service set-up and running 45

4.4. Service example of use 46

4.5. Repository 47

5. Conclusions 48

1. Introduction
   1. Project

The general purpose of CitiSim is devoted to the design and implementation of a new generation platform for the Smart City ecosystem.

This platform will provide a powerful monitoring and control infrastructure to enable planners to make critical management decisions on tactical and strategic levels based on the knowledge provided by the specific platform developed. For a natural interaction and better understanding of the events that happen in the city, 3D visualization techniques as augmented virtuality and augmented reality will be explored.

The D2.3 architecture demonstrator is a software deliverable. This software contains the core elements of the CitiSim Architecture, which have been defined for the project. These elements have been developed during the execution of the project.

This document is a complement to the software provided in the D2.3 and explains the architecture defined by all partners, as well as, how to configure and install it using docker technology.

The next deliverable "D2.4CitiSim developers manual", currently under development, already provides a good starting point to download and to test core components of CitiSim architecture.  You could find the developers manual at <http://pike.esi.uclm.es:8012/developers_manual/> (user:

citisim, passwd: CitiD0c!). For Itea3 member committee: It is necessary a bitbucket.org account and to be integrated in the CitiSim bitbucket group so please, send an e-mail to [felix.villanueva@uclm.es](mailto:felix.villanueva@uclm.es) to get access to CitiSim group.

* 1. Work package

This document has been produced as a deliverable within the WP2: *Reference Architecture Framework.* It corresponds to the deliverable *D2.3 Architecture Demonstrator* as result of the work done in the *Task 2.2 Core Services design and implementation*.

* 1. Document overview

The Deliverable D2.3 Architecture Demonstrator contains an overview of the CitiSim Architecture (Section 2), and a description to deploy to deploy a Docker runnable environment with the services that are being developed for CitiSim project (Section 3). This section explains how to configure and deploy the core services of the architecture to run in Docker containers.

Moreover, the CitiSim-mqtt-adapter service, one of the core services of the architecture, is explained in section 4. This service is an application that connects to a provided MQTT server address, subscribes to a list of topics and forwards received MQTT events to the CitiSim event distribution service.

Finally, a conclusion section is provided at the end of the document.

* + 1. Abbreviations and Acronyms

|  |  |
| --- | --- |
| Abbreviation/Acronym | Definition |
| 3D  API  CI  GUI  IoT  MQTT  OMG  REST  RFID  SoTA  SSH | 3 Dimensional  Application Program Interface  Continuous Integration  Graphical User Interface  Internet of Things  Message Queue Telemetry Transport  Object Management Group  REpresentational State Transfer  Radio Frequency IDentification  State-of-The-Art  Secure Shell |
| YAML | Ain't Markup Language |

1. Architecture – an overview

The first version of the CitiSim architecture has been elaborated considering the partner’s vision and requirements analysis of all the partners, you could find this information in the deliverable D2.1.

The main goals of the architecture are:

1. The architecture proposal for CitiSim should be flexible enough to support different platforms (Windows, Linux, Android, etc.) and different programming environments (Java, c++, etc.). At this point we should remember the deliverable *D2.1 State of the Art analysis* where we analyzed, among other points, the development framework of different CitiSim partners.
2. The access to those services of CitiSim from third parties through the Internet should be done by APIs REST according to the current state of the art.
3. The internal middleware used in core service implementation should be efficient enough to support scalability, security issues, flexibility, etc.

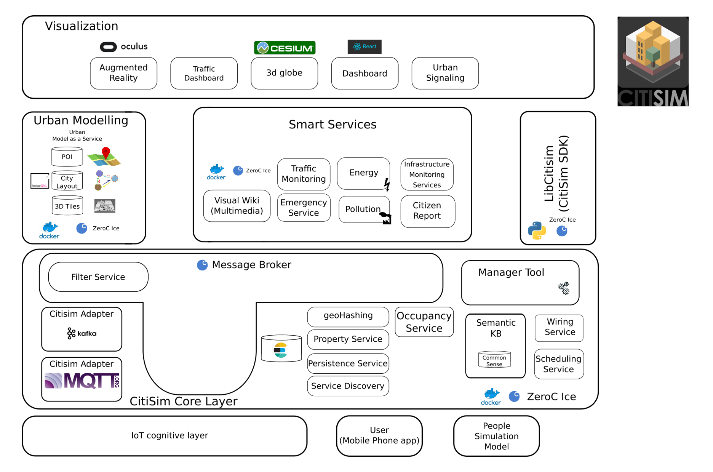
With these conclusions, the first approach to CitiSim Architecture is showed in Figure 1 . In the CitiSim architecture base, we can identify the following key components:

Figure 1: CitiSim architecture proposal

**IoT Layer**: This component is a layer where all the information from sensors/actuators is collected. Since it is a key part of the CitiSim is developed in the WP2. The sensors will register in the message broker through an interface as publisher and each sensor will call message broker interface periodically with its value and some metadata about the reading (e.g. Expiring time, Timestamp, quality of the reading, etc.). The actuators will implement a simple interface for change the environment.

In the simulation plane, the simulating algorithms will use the same method. From a practical point of view, a service in the upper layer does not distinguish among real data and simulated data.

**User mobile phone app**: This application enables the user to access to different information provided by the CitiSim platform. For example, BEIA will use this component to show information about traffic data information and TAIGER and BEIA will provide reporting capabilities to citizens.

**Core Layer:** The core layer architecture represents the CitiSim platform itself. It is supposed that if a City runs a single instance of the core layer in order to support the rest of the services. The key components of the core layer are:

* Message Broker:The message broker is the hearth of CitiSim due we will use for information (raw data, events, sensor information, etc.) distribution among core services and with the smart services layer. In the message broker, each service should subscribe to specific *topics* where information is published and, of course, a service also has to post the information generated. A *topic* is a logical bus where information is published. The types of the messages are defined by an interface. For example, from the IoT layer, several sensors sending information related with true/false information will use the following interface: *void notify (bool value, string source, Metadata data);* All the services interested in this type of information will implement the method *notify* so the message broker will call all subscribed services by calling each *notify* method of each service.
* Filter Service: In order to scale properly, a filter service is defined in order to subscribe to a specific topic with some filter options. In this way, only information that passes all the filters defined by each service will be communicated. The filters will be setup on subscription phase.
* Property Service: The property service is a service devoted to store static/semi static properties of devices/services in an instance of CitiSim platform. For example, the position of a smoke sensor, the last revision of an extinctor, the manufacturer of specific actuators are examples of information stored/accessed through the property service. The property service will store all this information in a data store.
* Persistent service: This service is subscribed to all topics in the message broker and store, in the data store. This persistent service will store and compact the information about the city.
* Semantic KB: this knowledge database will store basically three types of information:
  + The vocabulary and relations of concepts in a current city. This semantic information is common to any developed city. The services will use this vocabulary in its metadata, interfaces, etc.
  + The rules about how a city works regarding traffic and pedestrian.
  + A service description of the instances running in this instance of CitiSim.
* Scheduling service: This experimental service will orchestrate complex behaviours according to the services deployed and a new desire expressed by a user or service. For example, the authorization (literally open the door) to go into a building can be done by several methods (facial recognition, RFID tag, PIN code, etc.) according to the TIC infrastructure deployed. This scheduling service could link the access methods with the Authorization/Access service dynamically under an access request done by a user.
* Semantic Service: This service will manage information at semantic level and it will integrate other domains with CitiSim domain.
* Manager Tool: A tool for monitoring the CitiSim platform (state of the services/devices)
* Adapters: These modules will interconnect CitiSim (together with the semantic service) with other domains (e.g. MQTT devices, Kafka based platforms, Sofia2 services, etc)

**Urban modelling layer:** This layer will store structural information about the city. The key components of this layer are:

* Urban model as a service: This component is a repository with different models with information about urban furniture, street layout, 3D building models, supplies model (energy grid layout, water grid layout, etc.). The three primary repositories will be a Point of Interest repository with specific information about monuments, cultural buildings, etc. devoted to tourist applications. A street layout with a graph with the streets and pedestrian paths and a 3D tiles repository for 3D virtual world construction. The idea is to offer this information through an API to build visualizers accessing remotely to the information and also to provide information updates by a push model (e.g. street cut by some unusual event, water leak and affected area, etc.)

**Smart service layer:** The smart service layer takes the information collected in the core layer and by using the urban model layer provides a service related to stakeholders of a smart city. The facilities planned in CitiSim smart layer are extracted from the use case defined in the consortium. Third party services can be modeled and implemented attending to different use cases. From the currently defined use cases, the following smart services will be implemented:

* Pollution, Energy and Infrastructure monitoring service. A generic monitoring service provides with information related with a specific domain and enables the control of devices related to such field. To illustrate this type of services, in CitiSim we will monitor pollution, energy and Infrastructure. Smart Energy service will provide the capability to provide historical and future (forecast) estimated data of electricity usage.
* People monitoring: The key idea under this service is to estimate, from different sources of information (e.g. video analysis, sensors, access services, etc.), the occupancy of different areas at specific times. This service will store, with different accuracy, the number of people of specific areas.
* Traffic monitoring: In similar way that people monitoring, this service will collect information about the traffic conditions in the street layout.
* Emergency service: This smart service will follow a set of steps when a specific emergency is detected. This emergency service will provide with evacuation paths according to the type of emergency and the status of the infrastructure.
* Citizen Reporting: A smart service to allow citizens reporting any kind of security risk issue and/or situation, e.g. a fire extinguisher in bad conditions. By the use of a mobile app, they will be able to take a photo of the situation and it will be send a message to the platform, together with the geolocalization. The image will be processed in order to identify the risky item and to provide additional data, adding intelligence and enriching the previous message. Heatmaps and specific dashboards/widgets will display important metrics.
* Semantic Search Service: capable to offer advance semantic analysis of data gathered from sensors to provide with an improved search feature, e.g. using smart filtering, according to user preferences.
* Visual Wiki: This service provides information (including multimedia) about the city in different formats.

The set of services of the smart service layer is not closed, according to the evolution of the project, it could be possible to split services in more functional one or implement new services if needed.

Finally, the visualization layer provides to final users (mayor, citizens, companies, etc.) with information regarding different aspects of the information managed in a CitiSim instance. In the project, the different partners will work with various “visualizers” ranging from physical elements (information panels, kiosks, etc.) to virtual worlds where real information is showed (augmented reality) passing through virtual reality glasses for virtual interaction. The idea of the consortium is to develop different proofs of concept to visualize information.

These demos will be:

* Dashboard: The target of this visualizer is to show tendencies at strategic level from historical data.
* 3D globe: This visualizer will show a 3D virtual world where current information is presented. The key idea of this service is to show, in an intuitive way, all the information regarding specific domain and/or event. This visualizer implements the concept of augmented virtuality, a virtual world where real information is presented.
* Augmented reality is a service where virtual information is added over glasses. The user who wears the glasses will visualize the real environment with virtual information added regarding specific event (emergency, cultural agenda, etc.)
* Traffic dashboard: this citizen devoted visualizer will show personalized information related with traffic.
* Urban signaling: this service enables the control of the information showed in distributed information panels.

The layers will be defined/implemented in deep through CitiSim project. The deliverable *D5.1 common framework definition and service design* will specify how a service is designed/implemented for CitiSim.

1. CitiSim – Docker Infrastructure

This section presents a detailed description to deploy a Docker runnable environment with the services that are being developed for CitiSim project. First, we will understand how the Docker architecture works. Next, we will list CitiSim core services and see how to configure them to run in Docker containers. Finally, we will see how to deploy a CitiSim instance and add new services to the containers.

* 1. Introduction: Automatic deployment
     1. What is Docker?

Docker is a container **platform** used to develop, deploy and run containerized applications, allowing a clear separation between applications and infrastructure.

Containers are isolated from each other, they can communicate between them using well defined channels. The isolation allows to run many containers simultaneously on the same host. Containers are more lightweight than virtual machines because all the containers run directly within the same host machine’s kernel.

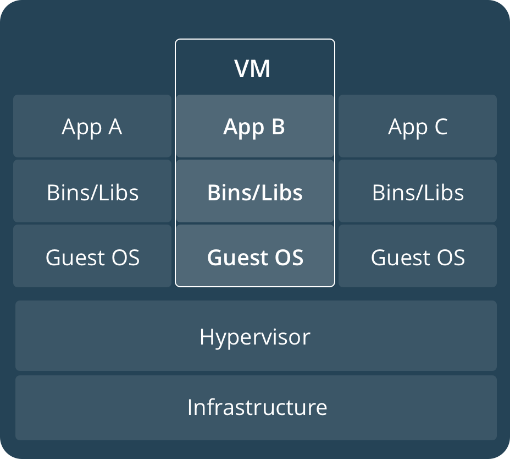
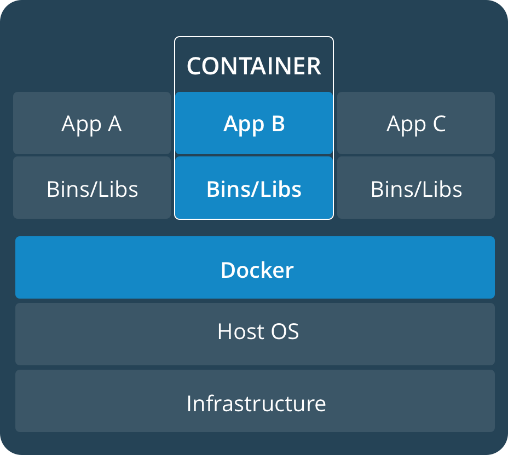


Figure 2: Docker stack vs VM stack

Containers are runtime instances of images, that includes all the needed content to run an application.

This information has been obtained from the official docker webpage. If you need to know more about Docker's architecture, visit these links:

* <https://docs.docker.com/engine/docker-overview/>
* <https://docs.docker.com/get-started/>
  + 1. CitiSim Docker deployment

In order to deploy the services developed in the CitiSim project, a Docker environment will be defined through which these services will be started **independently of the operating system or infrastructure** used as a host, thus facilitating deployment in production environments.

Most of the core services of CitiSim are developed with the middleware ZeroC Ice, so to facilitate the management of them, we will use the tools provided by this middleware like IceGrid. **IceGrid** is a location and activation service for Ice applications.

The company ZeroC provides some of the middleware services as **Docker images**, so we can use them to perform our deployment: <https://hub.docker.com/r/zeroc/> .

In our case, we have chosen to group the **CitiSim core services** in an **IceGrid node** to manage the activation or deactivation of them, obtain metrics and access to service logs, among others. This IceGrid node will be defined with a Docker image with its respective configuration files and will be executed in a single container.

On the other hand, we will have another Docker image in which we will define a new IceGrid node that will execute the occupancy service application and dummies clients to generate synthetic information that will allow us to validate the correct functioning of the core services.

And finally, these two IceGrid nodes will be registered in an **IceGrid Registry** that will run in another container Docker.

Therefore, we will have **3 Docker containers** to deploy a CitiSim instance. These three containers will be orchestrated using **Docker Compose**. In the figure we show the architecture deployment.

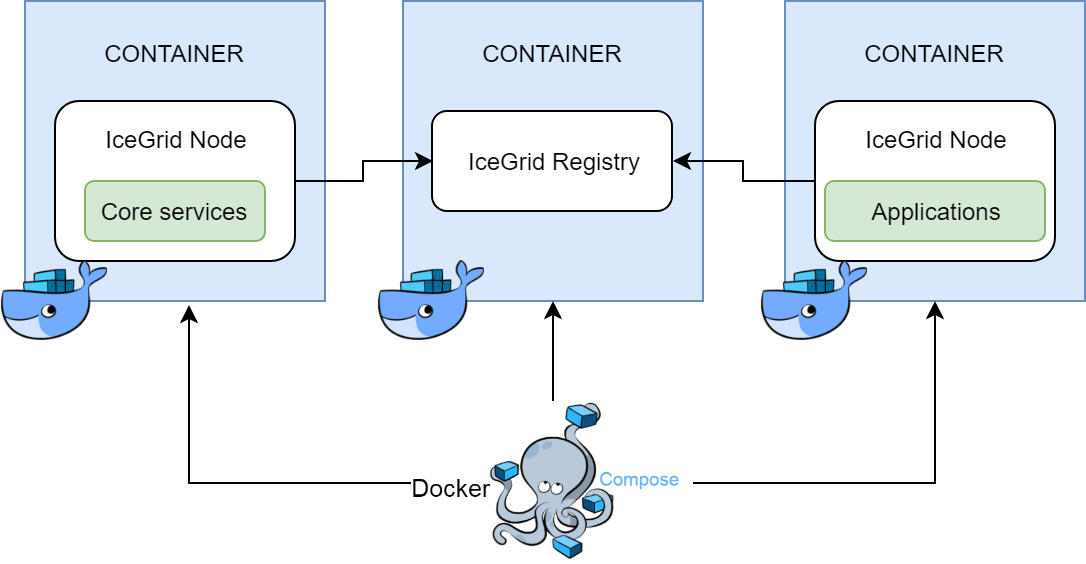


Figure 3: CitiSim deployment in Docker

* + 1. What is Docker Compose?

Compose is a tool for defining and running **multi-container Docker applications**. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration. Compose works in all environments: production, staging, development, testing, as well as CI workflows.

The features of Compose that make it effective are:

* Multiple isolated environments on a single host;
* Preserve volume data when containers are created;
* Only recreate containers that have changed;
* Variables and moving a composition between environments.

If you need to know more about Docker compose, visit these links: <https://docs.docker.com/compose/overview/>

* 1. Core services: A brief description

In this chapter, we will give a brief description of all CitiSim core services. At the time this document is written, the following services are available.

* + 1. Dashboard

Web application as a dashboard for the CitiSim Core Services. It will display node list and status, service state, events, and so on.

The service is compose by:

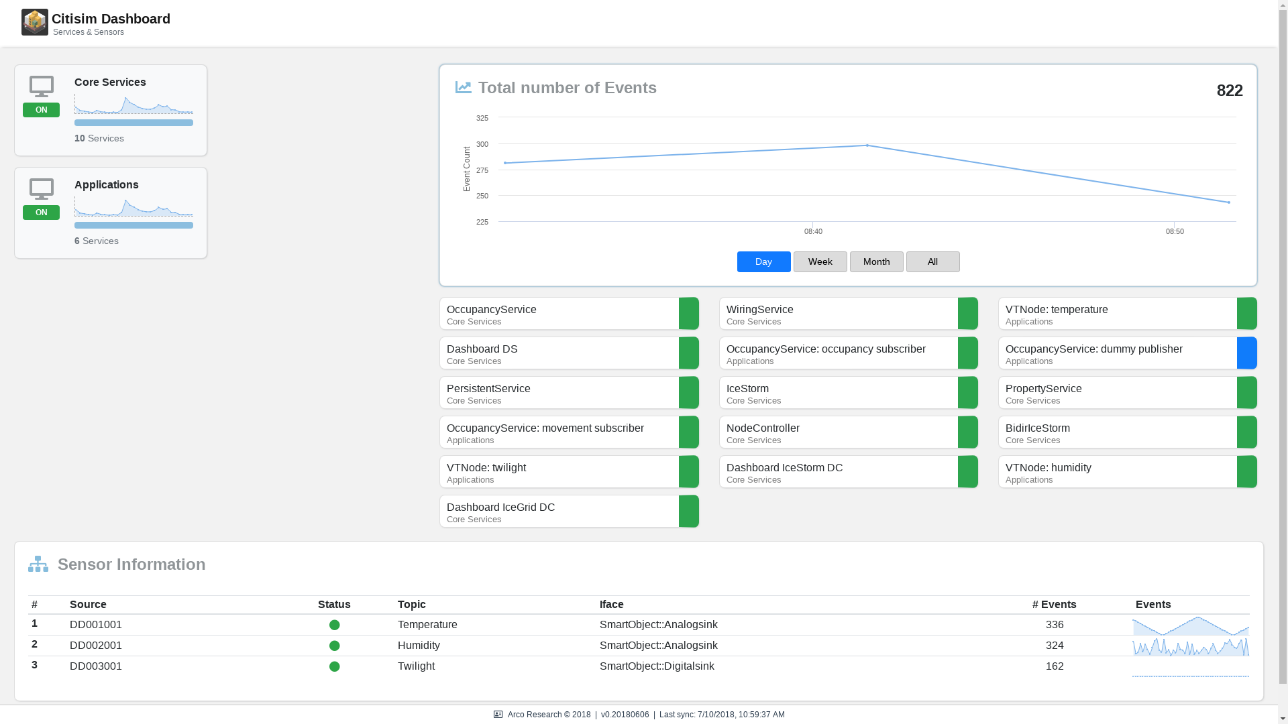
* **IceGrid Data Collector:** IceGrid Data Collector, retrieves information about services and nodes from an IceGrid Registry, and pushes them to the dashboard data sink,
* **IceStorm Data Collector**: IceStorm Data Collector, retrieves data about events and sensors from an IceStorm instance, and pushes them to the dashboard data sink,
* **Dashboard Data Sink**: Listens for incoming invocations from data collectors and updates the dashboard database.

Figure 4: CitiSim Dashboard

**Source code**: <https://bitbucket.org/arco_group/prj.citisim/src/master/src/dashboard/>

* + 1. IceStorm GUI

Web application that provides a Graphical User Interface (GUI) to manage the list of topics, create and destroy topics, and to get the topic statistics and subscribers information.

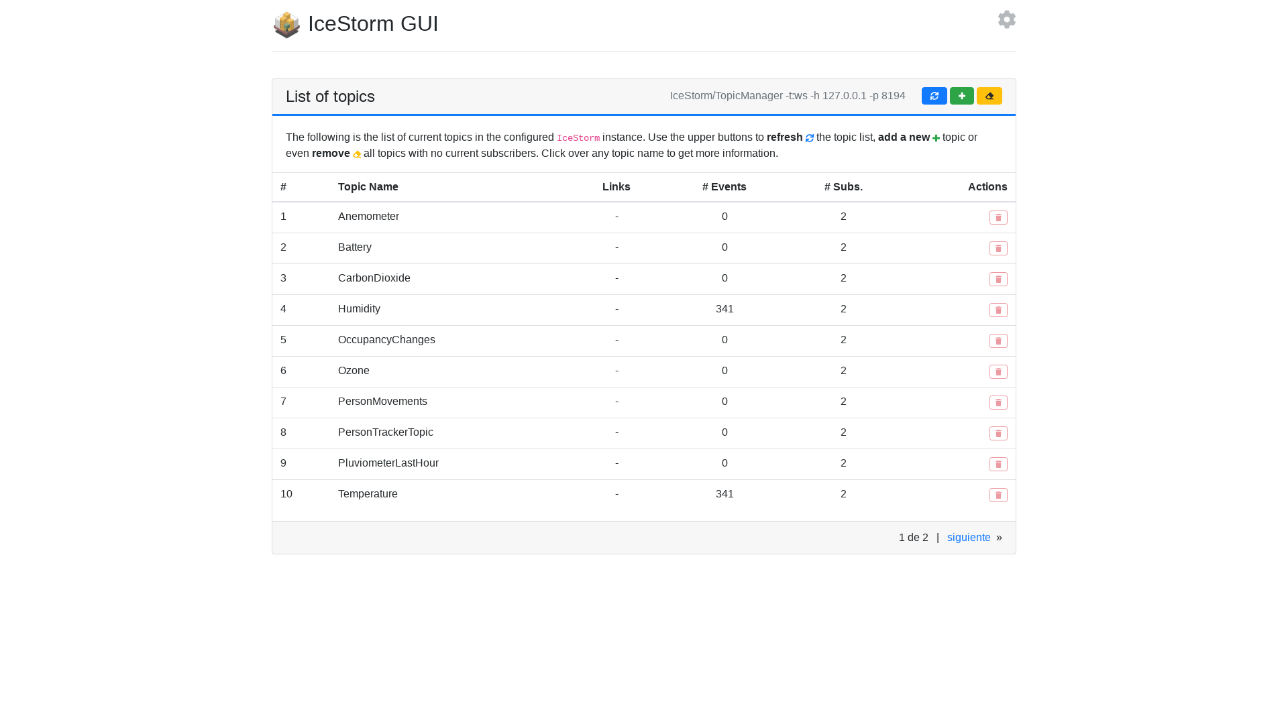


Figure 5: CitiSim IceStorm GUI

**Source code**: <https://bitbucket.org/arco_group/prj.citisim/src/master/src/icestormgui/>

* + 1. Node Controller

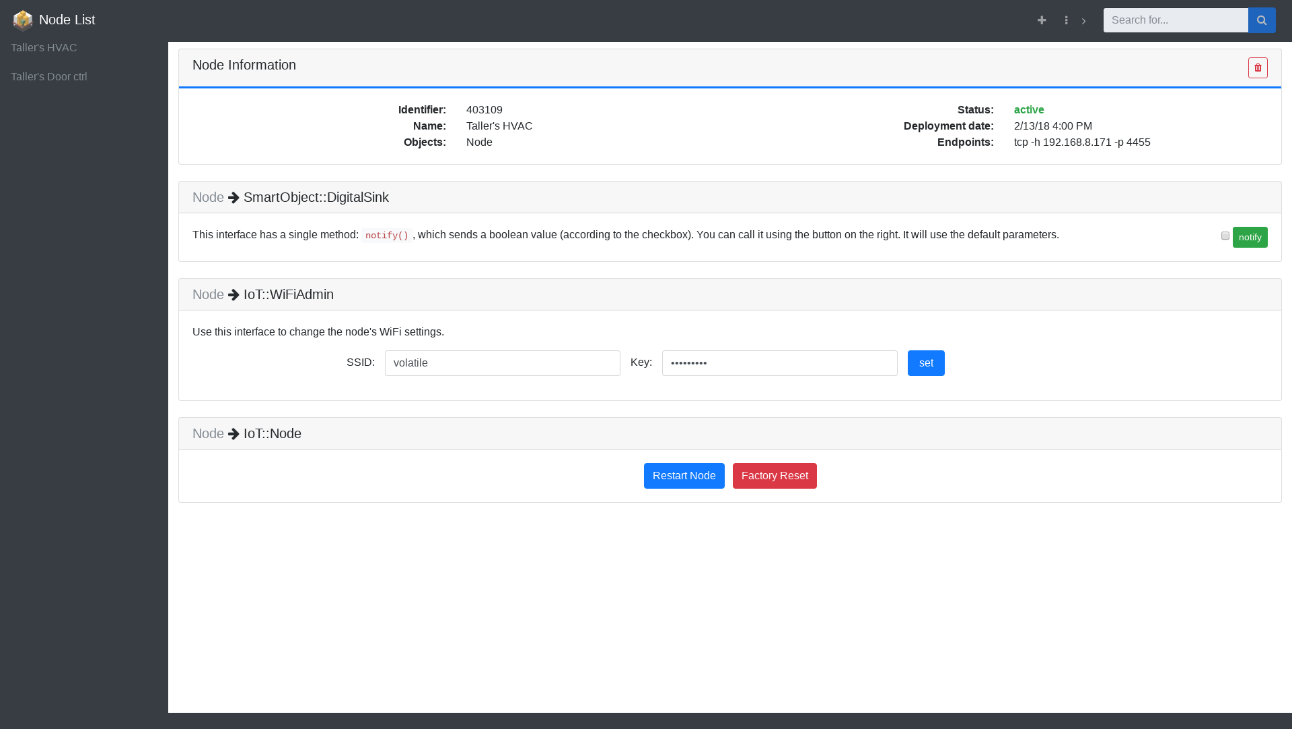
Web application to manage the different nodes that are deployed, usually very constrained devices that cannot run an IceGrid node, or even an operating system.

Figure 6: CitiSim Node Controller

**Source code**: <https://bitbucket.org/arco_group/prj.citisim/src/master/src/node-controller/>

**Documentation**: <http://pike.esi.uclm.es:8012/recipe/citisim_manager_tool_getting_started/>

* + 1. Persistence Service

Service that receives and stores events from IceStorm topics, to be able to process later. It listens on a list of topics (given by its names) and uses a common database to store the event itself and some more meta-information related to the event (like the receive timestamp). This service subscribers will only use CitiSim interfaces.

Figure 7: CitiSim Persistence Service

This service provides a web interface which by default is set on port 8000 of the machine. From this interface, it is possible to add new events tot he database, to edit already existing events and to check the history for a specific event.

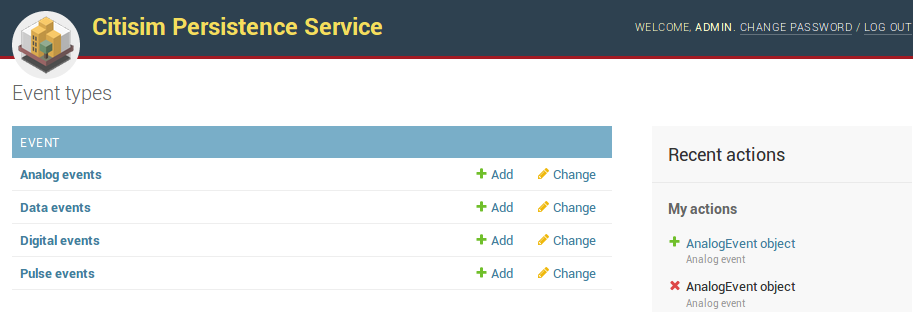


Figure 8: Persistence service main interface

**Source code**: <https://bitbucket.org/arco_group/prj.citisim/src/master/src/persistence-service/>

* + 1. Property Service

Property Service is a simplified version of the OMG Property Service, based on a dictionary interface. Therefore, property (key, value) dictionaries are created, in which value can also be another dictionary, creating this way a tree structure.

This can be used to query data from a specific device or to select or filter devices that satisfy a certain condition regarding their properties.

The next table describes the available properties for devices in CitiSim:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Type** | **Values** | **Description** |
| units | str | 1 | units of the readings |
| hw-range | float | 2 | [min, max] allowed sensor values |
| position | float | 3 | [longitude, latitude, altitude] |
| orientation | float | 3 | [x, y, z] |
| manufacturer | str | 1 | device manufacturer |
| deployment-state | str | 1 | physical deployment date: 2001-01-01 |
| last-revision-date | str | 1 | maintenance last revision date: 2001-01-01 |
| software-version | str | 1 | running software version |
| hardware-version | str | 1 | device HW version |
| Table 1: CitiSim device properties | | | |

The next figure shows an example of the tree structure for the properties of a sensor:



Figure 9: Example of Property Service tree structure

**Source code**: <https://bitbucket.org/arco_group/ice.property-service-simple/src/default/>

**Documentation**: <http://pike.esi.uclm.es:8012/recipe/property_service_usage/>

* + 1. Wiring Service

Provides the mechanisms necessary for doing automatic state propagation for services using the event service. It will allow single and multi-observer setups.

**Source code**: <https://bitbucket.org/arco_group/citisim.wiring-service/src/default/>

* + 1. Occupancy Service

Service that estimates the occupation of a certain area of a building. As a result, different events announcing changes in the occupation of a specific room, as well as the movements of people between rooms, will be published on channels.

**Source code**: <https://bitbucket.org/arco_group/occupancy-service>

**Documentation**: <http://pike.esi.uclm.es:8012/recipe/occupancy_service_using_the_service/>

* 1. Docker configuration

Once we have seen the CitiSim core services available, we will detail the different parts of the **Docker environment** needed to perform the scenario deployment shown in Figure 2.

Each of the containers is an **isolated environment** in which one or more services are executed. For the definition of this environment, we use what is known as **Dockerfile**. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. So, for each of the containers we will have to define a Dockerfile.

To define the containers to be executed, the ports to be exposed for each container, the volumes to manage the persistence of data and the connections between the containers, we will define the **docker-compose.yml** file. This file tells to **Docker Compose** how to manage the CitiSim containers.

First, we will explain in detail each of the Dockerfile and its associated configuration files for each of the containers. And finally, we will detail the docker-compose.yml file that will orchestrate these containers.

The file directory would be structured as follows:

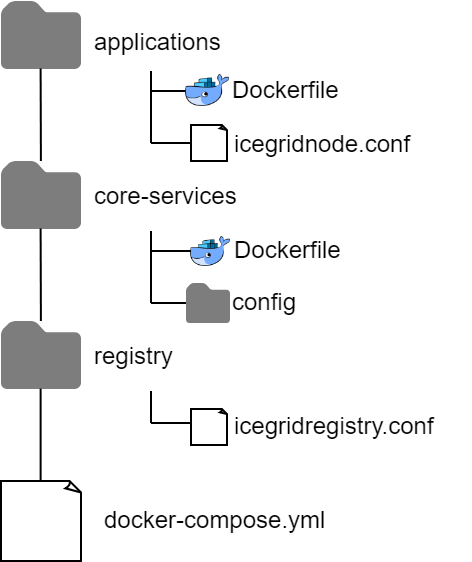


Figure 10: Structure of Docker files

* + 1. Registry Container

As we saw in Section 1.2, ZeroC provides Docker images to execute containers of different Ice services. One of the images allows the execution of an **IceGrid Registry service**. The image is this: [https://hub.docker.com/r/zeroc/IceGridRegistry/](https://hub.docker.com/r/zeroc/icegridregistry/).

There are different tags with their respective associated Dockerfiles to be able to use each of the Ice versions available. In this case, we will use **ZeroC Ice version 3.6.4**, so select the [zeroc/IceGridRegistry:3.6.4 image](https://github.com/zeroc-ice/ice-dockerfiles/blob/master/3.6/icegridregistry/Dockerfile).

To be able to execute this image Docker, it is necessary to provide a **configuration file** for the IceGrid Registry. In this configuration file we will put each of the parameters required by the Registry, such as the name of the IceGrid instance and the endpoints to which the IceGrid nodes will be connected among others.

In our case we have defined the configuration in the file **icegridregistry.conf**:

IceGrid.InstanceName=Citisim-IceGrid

IceGrid.Registry.Client.Endpoints=tcp -p 4061 -h citisim-registry : ws -p 4071 -h citisim-registry

IceGrid.Registry.Server.Endpoints=tcp -h citisim-registry

IceGrid.Registry.Internal.Endpoints=tcp -h citisim-registry

IceGrid.Registry.Data=/var/lib/ice/icegrid/registry

IceGrid.Registry.PermissionsVerifier=Citisim-IceGrid/NullPermissionsVerifier

IceGrid.Registry.AdminPermissionsVerifier=Citisim-IceGrid/NullPermissionsVerifier

IceGrid.Registry.DefaultTemplates=/usr/share/Ice-3.6.3/templates.xml

Ice.UseSyslog=1

Ice.ProgramName=icegridregistry (Citisim-IceGrid Master)

Ice.StdOut=/var/lib/ice/icegrid/registry/std.out

Ice.StdErr=/var/lib/ice/icegrid/registry/std.err

IceGrid.Registry.Trace.Node=1

IceGrid.Registry.Trace.Replica=1

There are different options to configure, so in this document will describe the most important only. The first one is the *InstanceName*, which must be unique to distinguish it from other IceGrid instances.

Following, one of the most important parts of the file is the definition of the Endpoints, since they will tell us the name of the container and the port in which the Registry will be listening. In this case the **ports *4061*** and ***4071*** of the container named ***citisim-registry*,** are the ones that will be listening.

The Data option defines the locations where the IceGrid data will be stored. The *PermissionsVerifier* and *AdminPerimissionsVerifier* have been **set as null** to not ask for user and password when accessing to the IceGrid service. All other configurations are optional, but we recommend leaving them as defined in the file.

With the Docker image and the configuration file we could already run the container. To do this, we will execute the following command:

$ docker run --name citisim-registry -v /path/to/config/:/etc/icegridregistry.conf:ro -d zeroc/icegridregistry:3.6.4

Where */path/to/config* is the configuration file location.

As each container is isolated from the host on which it is executed, it is necessary to do a **port forwarding** so that we can access the Registry from localhost. This redirection can be defined at the time of executing the container. If you want to make the container visible to the localhost, you must add the option ***-p 4061:4061 -p 4071:4071*** to the docker run command.

With everything we've seen so far, we could deploy a Docker container with the IceGrid Registry.

* + 1. Core services container

In this section we will see all the steps we have followed to build a Docker container that will house an IceGrid node in which all the CitiSim core services will be executed.

As we saw in the previous section, we will need a Docker image or write our Dockerfile to build the environment where the services will be executed. In this case, most services will require one or more configuration files to be executed.

On the other hand, it is necessary to define an **IceGrid application** that allows us to configure the Ice servers to start up and manage each of the services.

* + - 1. Dockerfile

In this section, we will detail the instructions that we have defined in the Dockerfile to be able to execute the core services shown in Section 2.

First of all, choose the base image on which the services will run. In this case, we have chosen to use a [Debian stretch image](https://hub.docker.com/_/debian/) because all the services developed to support this operating system. To do this we only add the instruction in the Dockerfile:

**FROM** debian:stretch

Next, we will install all the necessary packages to run the services, such as web servers, Python libraries or a process manager among others.

**RUN** apt update \

&& apt -y install zeroc-ice-utils \

zeroc-IceGrid \

zeroc-ice-slice \

zeroc-ice-compilers \

zeroc-icebox \

libzeroc-ice-dev \

mercurial \

nginx \

git \

uwsgi \

uwsgi-plugin-python3 \

python3-zeroc-ice \

python3-wrapt \

python3-django \

python3-chartkick \

python3-flask \

python3-dateutil \

python3-ephem \

build-essential \

staticsite \

supervisor \

&& rm -rf /var/lib/apt/lists/\* /tmp/\* /var/tmp/\*

Next, we copy the configuration files of the services from outside the container to the container and the SSH key that we will use to download the repos of the services from Bitbucket.

**COPY** config/IceGridnode2.conf /etc/

**COPY** config/icestormgui.conf /etc/nginx/sites-enabled/

**COPY** config/persistence-service.conf /etc/nginx/sites-enabled/

**COPY** config/dashboard.conf /etc/nginx/sites-enabled/

**COPY** config/persistence-service.ini /etc/uwsgi/apps-enabled/

**COPY** config/dashboard.ini /etc/uwsgi/apps-enabled/

**COPY** config/supervisord.conf /etc/

**COPY** config/citisim.xml /etc/

**COPY** config/properties.json /tmp/

**COPY** .ssh /root/.ssh

Following, we give the root user permission to read the SSH keys and generate the IceGrid directory where the services are going to be executed. We also add the file necessary to run the node controller service.

**RUN** chmod 0600 /root/.ssh/audit-key

**RUN** mkdir -p /var/lib/ice/IceGrid/node2 \

&& chown ice:adm /var/lib/ice/IceGrid/node2 -R \

&& mkdir -p /var/lib/ice/.config/node-controller/

**COPY** config/node-controller.conf /var/lib/ice/.config/node-controller/settings.conf

**RUN** chown ice:ice /var/lib/ice/.config/ -R

Bellow, we move to the */tmp/* directory where we will **clone the repositories of the services**. Once the repositories are cloned, we will generate the necessary symbolic links to the service interfaces. We will also change the owner of the repositories so that the ice user can read the files.

**WORKDIR** /tmp/

**RUN** git clone ssh://bitbucket.org:/arco\_group/prj.citisim \

&& git clone git@bitbucket.org:arco\_group/prj.citisim-libcitisim.git \

&& hg clone ssh://bitbucket.org:/arco\_group/ice.property-service-simple \

&& git clone ssh://bitbucket.org:/arco\_group/occupancy-service \

&& git clone ssh://bitbucket.org:/arco\_group/kinect-person-tracker \

&& hg clone ssh://bitbucket.org:/arco\_group/citisim.wiring-service \

&& git clone ssh://bitbucket.org:/arco\_group/bidir-icestorm.git \

&& hg clone ssh://bitbucket.org:/arco\_group/iot.node \

&& ln -sf /tmp/iot.node/slice /usr/share/slice/iot \

&& make -C /tmp/bidir-icestorm/src all \

&& ln -fs /tmp/prj.citisim/slice /usr/share/slice/citisim \

&& mkdir -p /usr/share/slice/PropertyService \

&& ln -sf /tmp/ice.property-service-simple/src/PropertyService.ice \

/usr/share/slice/PropertyService

**RUN** chown ice:adm /tmp/\* -R

Next, we configure the applications Django dashboard and persistence service. The django-admin shell < /tmp/persistence-service-adduser.py command adds the username and password defined in the script to both applications to configure a default user. To change both parameters, you must modify the script.

**RUN** export PYTHONPATH="/tmp/prj.citisim/src/persistence-service/src/webapp" \

&& export DJANGO\_SETTINGS\_MODULE=webapp.settings \

&& django-admin collectstatic --no-input \

&& django-admin migrate \

&& chown www-data:www-data /tmp/prj.citisim/src/persistence-service/src/ -R \

&& chmod a+rw /tmp/prj.citisim/src/persistence-service/src/webapp -R \

&& django-admin shell < /tmp/persistence-service-adduser.py

**RUN** export PYTHONPATH="/tmp/prj.citisim/src/dashboard/webapp" \

&& export DJANGO\_SETTINGS\_MODULE=dashboard.settings \

&& django-admin collectstatic --no-input \

&& django-admin migrate \

&& chown www-data:www-data /tmp/prj.citisim/src/dashboard/webapp -R \

&& chmod a+rw /tmp/prj.citisim/src/dashboard/webapp -R \

&& django-admin shell < /tmp/persistence-service-adduser.py

Finally, we copy the necessary files to generate a web page from a Markdown file with the information of all the services that are going to be displayed in the container.

*# Generate doc site*

**COPY** config/README.md /var/www/docsite/content/

**COPY** config/settings.py /var/www/docsite/

**RUN** ssite build /var/www/docsite \

&& mv /var/www/docsite/web/\* /var/www/html/ \

&& chown www-data:www-data /var/www/html/ -R

Finally, to configure the container as an executable or as a service you must add the **CMD** or **ENTRYPOINT** directive. According to the Docker philosophy, only one service can be performed per container. In our case, as we have to execute several services inside the container, we will use a process manager called

**ENTRYPOINT** ["/usr/bin/supervisord", "-c", "/etc/supervisord.conf"]

**upervisord**[[1]](#footnote-1). This process manager will allow us to launch different processes from a configuration file.

Inside the file **supervisord.conf** we will list all the processes that will be launched. In our case we execute the following services:

1. **IceGrid node for CitiSim** **core services**
2. **Nginx server**
3. **Persistence service uwsgi application**
4. **Dashboard uwsgi application**

And run the following commands:

1. **icegridadmin to add the CitiSim application**
2. **icegridadmin to run bidirIcestorm server**

These six commands allow you to sequentially launch the Icegrid node, boot the web servers from the persistence server and the dashboard, load the Icegrid application, and run the application servers.

[supervisord]

nodaemon=true

[program:IceGridnode]

command=IceGridnode --Ice.Config=/etc/IceGridnode2.conf

priority=1

user=ice

[program:nginx]

command=/usr/sbin/nginx -g "daemon off;"

priority=3

[program:uwsgi-persistence-service]

command=uwsgi --ini /etc/uwsgi/apps-enabled/persistence-service.ini

priority=4

[program:uwsgi-dashboard]

command=uwsgi --ini /etc/uwsgi/apps-enabled/dashboard.ini

priority=5

[program:IceGridadmin]

command=IceGridadmin -u user -p pass -e "application add /etc/citisim.xml"

priority=20

[program:bidiricestorm]

command=IceGridadmin -u user -p pass -e "server enable BidirIceStorm"

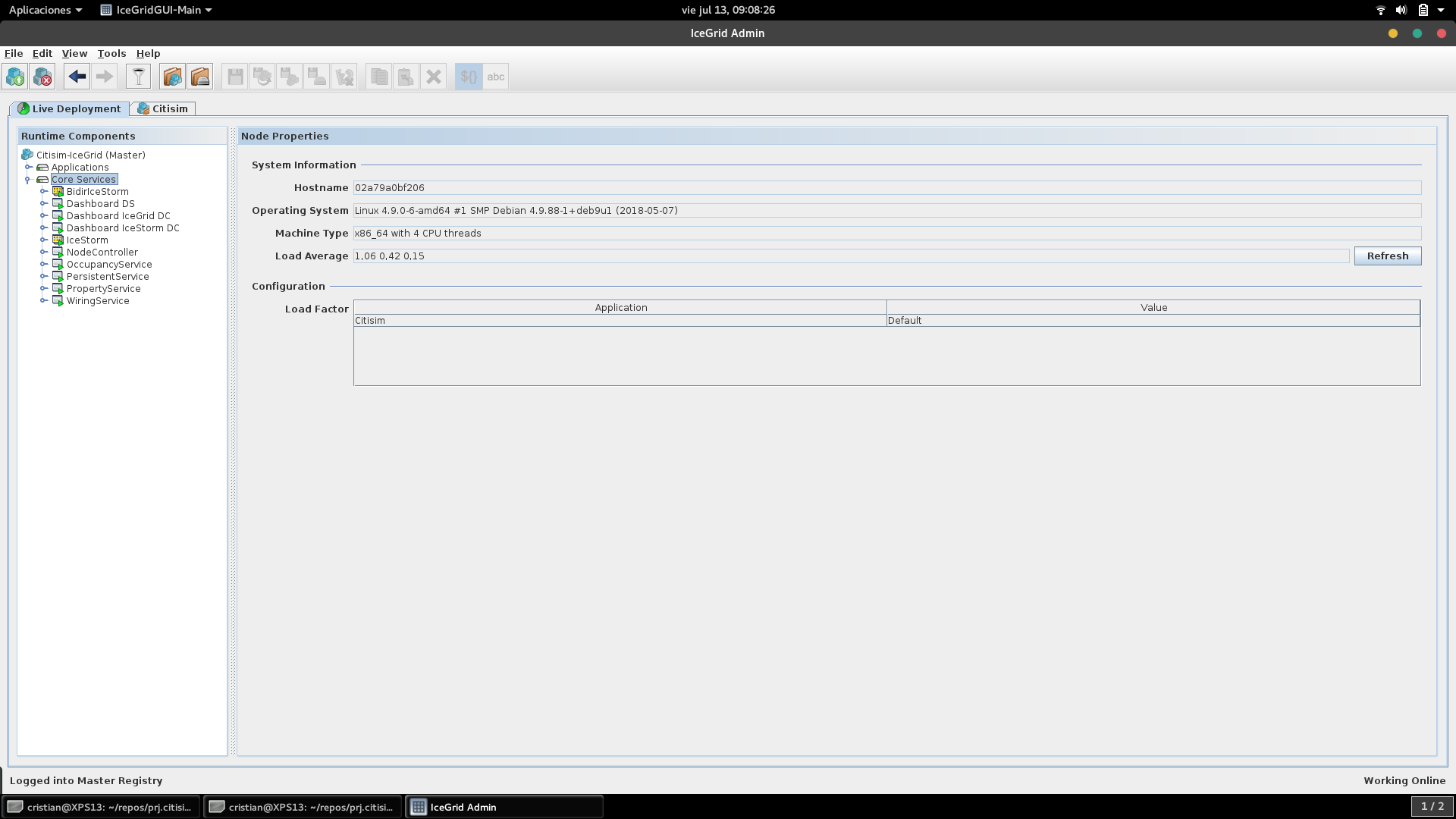
priority=40

With all that we have seen so far, we could already execute the container with the services. But for this to work, it is necessary to have **an instance of the Registry container** to connect to. Next, we will look a little more in detail at the service configuration files.

* + - 1. Configurations folder
* **README.md:** Markdown file that contains all information about the services deployed whit his respective configurations parameters as users, passwords or endpoints among others. This file is displayed on a web page.
* **citisim.xml:** IceGrid application that defines all services that will be executed in the Docker containers.
* **dashboard.conf:** Nginx configuration for dashboard app.
* **dashboard.ini:** Uwgsi configuration for dashboard app.
* **icegridnode2.conf:** IceGrid node configuration. This file defines the Registry to which the node will be connected, in our case, to the Registry container.
* **icestormgui.conf:** Nginx configuration for IceStorm GUI app.
* **node-controller.conf:** Node controller service configuration file. This file defines the property server to which the node controller app will be connected.
* **persistence-service-adduser.py:** Python script that allows you to add a default username and password to applications.
* **persistence-service.conf:** Nginx configuration for persistence service app.
* **persistence-service.ini:** Uwsgi configuration for persistence service app.
* **properties.json:** JSON test file for uploading data to the property service.
* **settings.py:** Python script to define the metadata of the help website
* **supervisord.conf:** Supervisord configuration file.
  + - 1. CitiSim IceGrid application

Once we have the Dockerfile and the service configuration files, the next step is to define the IceGrid application that will manage the IceGrid nodes and the servers that will run on each of them.

Figure 11: CitiSim IceGrid Application (Core Services Node)



In our case we will run the following 10 servers in the IceGrid Core Service node:

1. **BidirIceStorm**
2. **Dashboard DS**
3. **Dashboard IceGrid DC**
4. **Dashboard IceStorm DC**
5. **IceStorm**
6. **NodeController**
7. **OccupancyService**
8. **PersistentService**
9. **PropertyService**
10. **WiringService**

Describing the configuration of each of the servers would make this document very extensive, so we will only give an example of one server.

For example, for the Properties Service we will define the following properties:

* **Server ID**: PropertyService
* **Properties:**
  + Ice.StdOut = $(server.distrib)/std.out
  + Ice.StdErr = $(server.distrib)/std.err
  + PropertyService.Data = $(server.distrib)/db.json
* **Path to executable**: ./property-server
* **Working directory**: /tmp/ice.property-service/src
* **Activation mode:** always

The reserved name***[[2]](#footnote-2)*** ***$(server.distrib)*** define the pathname of the enclosing server distribution directory. So, the output and error files will be located in the server's own directory.

The **Working directory** property specifies the location of the repository containing the service to be run. In our case, in the Dockerfile we have cloned the repository in the */tmp/* folder, so we will define the path in that property.

In the **Path to executable** property we will define the file to be executed by the server, in this case ./*property-server*.

With the **Activation mode** property set to **always**, we define that the server will start automatically when we load the application in the IceGrid Registry, this way we tell IceGrid to start the core services automatically.

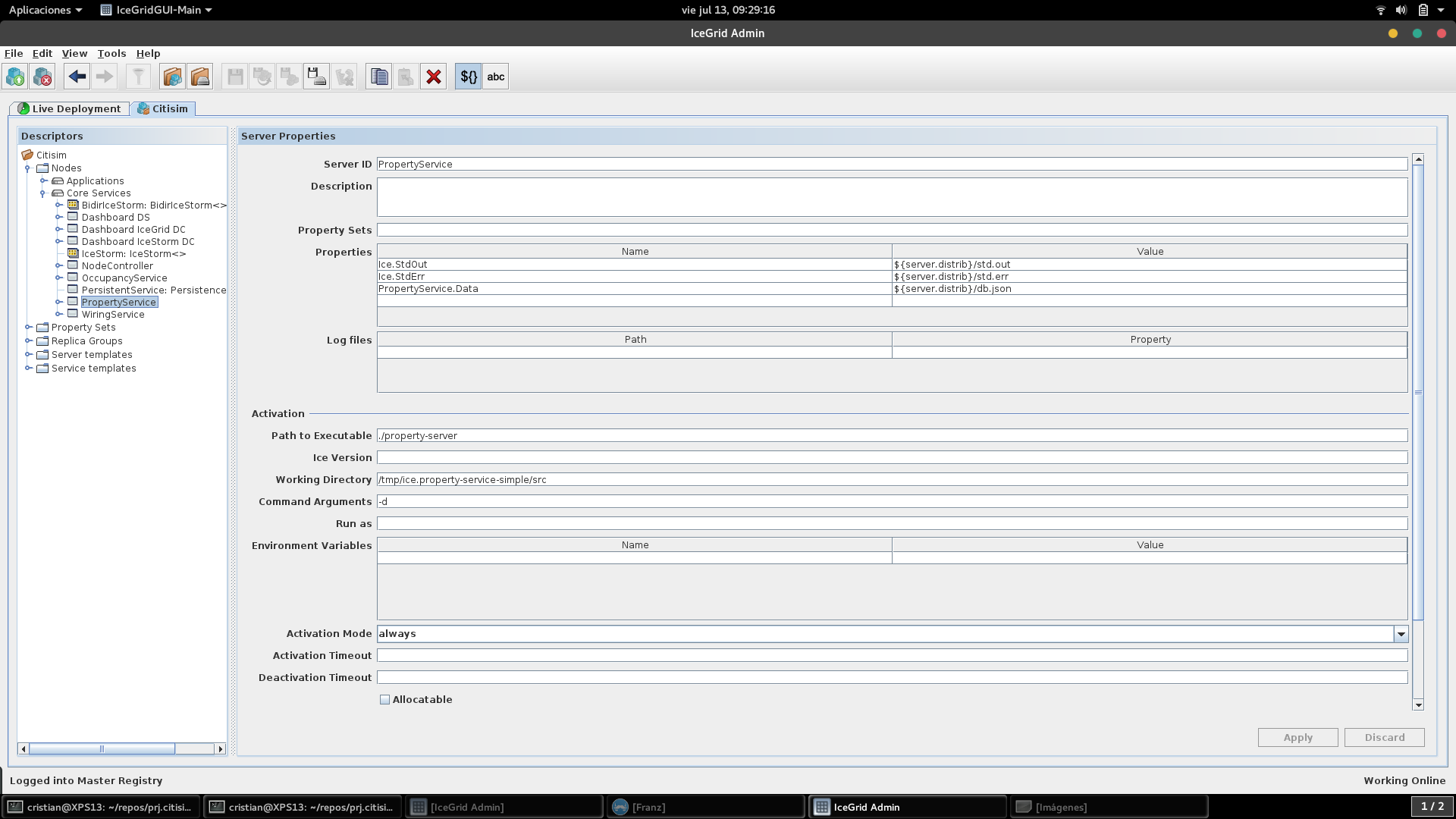


Figure 12: Definition of the Property server in the IceGrid application

On the other hand, we must define the adapter of the server where the service is going to listen. For this service, we will have an endpoint on **port 4334** and another published one. For the published endpoint we must define the hostname of the Docker container. In this case, the core services container will be named as **citisim-core-services**.

In addition to this we will declare the service proxy as a **well-known object**. A well-known proxy refers to a well-known object, that is, its identity alone is sufficient to allow the client to locate it. The Property Service server will be identified as **PropertyServer**.

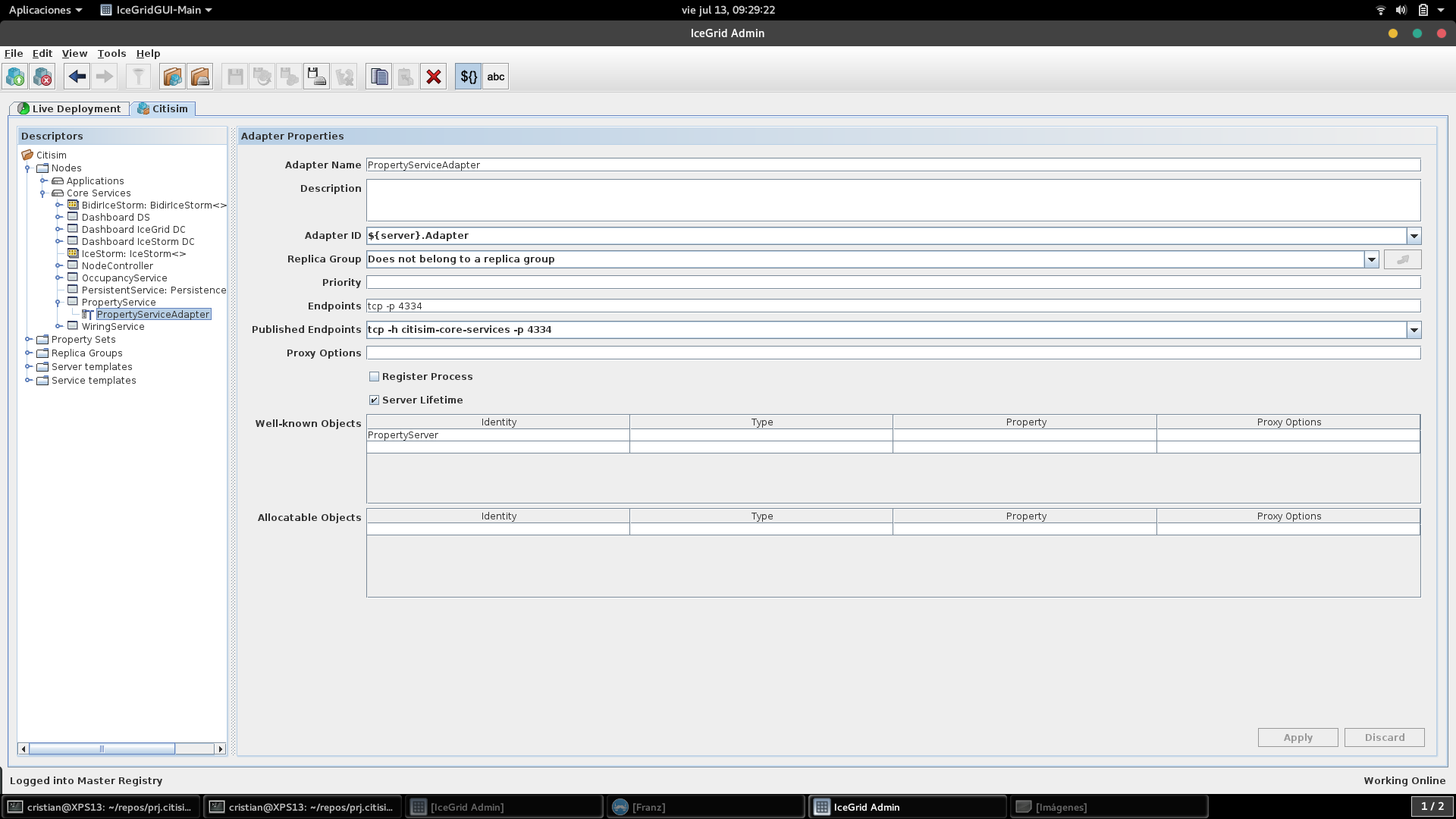


Figure 13: Adapter configuration for Property server in the IceGrid application

With this example we can get an idea of how to configure a server in IceGrid. Some of the services will require more configuration properties, but this document is not intended to detail it.

* + 1. Applications container

The **Applications container** has a similar structure to the Core Service container, but in this case, only the IceGrid node service will be executed. In this container, we will only execute services that publish events in the IceStorm of the Core Services node. In addition to these, the Occupancy Servicewill also be executed, which will allow us to obtain data on the movement of people in an environment, from data simulated by a dummy application.

About the Dockerfile, it has a structure similar to that of Core Services Dockerfile except that it only installs the necessary packages to run the applications and clones their respective repositories. In addition, the ENTRYPOINT command launches the IceGrid node instead of the supervisord process manager.

**ENTRYPOINT** [ "icegridnode", "--Ice.Config=/etc/icegridnode1.conf"]

The only configuration file for this container is the **icegridnode1.conf.** This file defines the IceGrid Registry locator to which the IceGrid node must be connected. In this case, this locator is defined in the file we saw in Section 3.1 above. Additionally, it also establishes the node name, the output and error files, and the endpoint of the node among others.

In the same application shown in Section 3.2.3 the Application node was defined with all its servers. As we can see in Figure 11 we have **6 servers**:

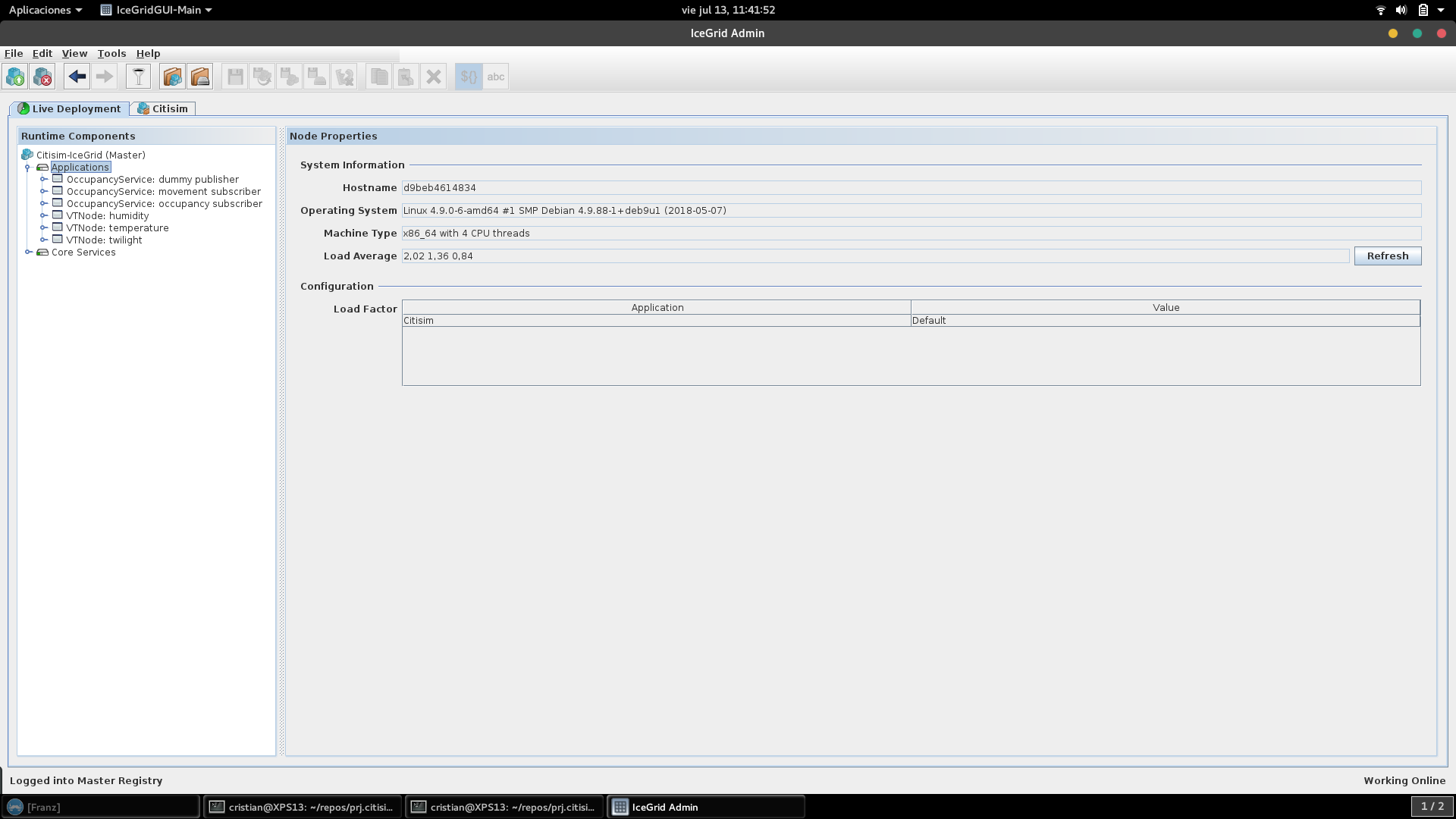
* ***OcuppancyService: Dummy publisher*:** Simulates the motion events generated by Kinect.
* ***OccupancyService: Movement subscriber:*** Keeps track of the movements taking place in a given space.
* ***OccupancyService: Occupancy subscriber:*** Keeps track of the number of people in a given space.
* ***VTNode: humidity:*** Publishes humidity data in the IceStorm server.
* ***VTNode: temperature:*** Publishes temperature data in the IceStorm server**.**
* ***VTNode: twilight:*** Publishes twilight data in the IceStorm server.

Figure 14: CitiSim IceGrid Application (Applications Node)

As was the case with the Core Services container, to execute this container, it is necessary to have a running instance of the Registry container in order to connect the node and execute the services. This container also has a dependency on the Core Services container because the data generated by the applications depends on the services that are performed in the IceGrid Core Services node.

Therefore, to launch all the nodes, we must follow an order of container execution, which will be defined by the Docker Compose orchestrator.

* + 1. Docker-compose.yml

This file will allow us to orchestrate the Docker application defined for CitiSim. As we saw in Figure 2, we will have 3 containers:

1. **IceGrid Registry Container**
2. **IceGrid node for CitiSim Core Services**
3. **IceGrid node for CitiSim Applications**

For the correct functioning of the IceGrid application, we must follow the order of execution of the containers. This order will be described in the docker-compose.yml file thanks to the **depends\_on** instruction. Next, we will list the file and explain each of the corresponding parts.

version: '2'

services:

**citisim-registry**:

image: zeroc/icegridregistry:3.6.4

container\_name: citisim-registry

volumes:

- ./Registry/icegridregistry.conf:/etc/icegridregistry.conf:ro

- citisim-registry:/var/lib/ice/icegrid

ports:

- "4061:4061" # IceGrid Registry

- "4071:4071" # ws IG-Registry

networks:

- citisim

**citisim-applications**:

build: ./applications

container\_name: citisim-applications

depends\_on:

- citisim-registry

volumes:

- citisim-applications:/var/lib/ice/icegrid

networks:

- citisim

**citisim-core-services**:

build: ./core-services

container\_name: citisim-core-services

depends\_on:

- citisim-registry

volumes:

- citisim-core-services:/var/lib/ice/icegrid

ports:

- "80:80" # README

- "4334:4334" # PropertyService

- "7160:7160" # WiringService

- "8080:8080" # IcestormGUI

- "8081:8081" # PersistenceService

- "8082:7154" # NodeController

- "8083:80" # README

- "8084:8084" # Dashboard

- "8192:8192" # IceStorm-TopicManager

- "8193:8193" # IceStorm-Publish

- "8194:8194" # ws IS-TopicManager

- "8195:8195" # ws IS-Admin

- "8292:8292" # BiDir IS-TopicManager

- "8293:8293" # BiDir IS-Publish

networks:

- citisim

volumes:

citisim-registry:

citisim-applications:

citisim-core-services:

networks:

citisim:

As we can see we have three services: **citisim-registry**, **citisim-applications** and **citisim-core-services**.

The **citisim-registry** uses the public **zeroc/icegridregistry:3.6.4** image pulled from the Docker Hub Registry while the other two services use images that are built from the **Dockerfile** of their corresponding directories.

Following, we specify the custom container name for each service. In this case, they will be called the same as the service.

Now in the services **citisim-applications** and **citisim-core-services** we define the dependency on the **citisim-registry** thanks to the instruction **depends\_on**. This instruction has two effects:

1. docker-compose up starts services in dependency order. In our case, citisim-registry is started before citisim-applications and citisim-core-services.
2. docker-compose up SERVICE automatically include SERVICE’s dependencies. In our case, docker compose up citisim-core-services also create and start citisim-registry.

Once the dependencies between services have been resolved, we define the **volumes** that each of the containers will use. These volumes will allow us to provide persistence to the application and to mount folders relative to the host to load files.

For the citisim-registry service, we load the file icegridregistry.conf from a **relative host path** to the container path where it will use that file to launch the Icegrid Registry. Additionally, we generate a volume on the host to persistently store the data generated in the Icegrid Registry. For the other two services, we also generate a volume. The declaration of all these volumes is given at the end of the docker-compose.yml file.

The next thing to the volume definition is the **port forwarding** from the container to the host. This way we can expose the services that are executed inside the containers to the localhost. We will do as many port forwardings as services require.

Finally, we will specify the **network** called **citisim** to be able to connect the services together. This instruction will allow us to create more complex topologies and specify custom network drivers when exposing the application in a production environment.

With everything we have seen so far, we can now execute a CitiSim instance in a very simple way. In the following chapter we will see how to deploy the application with all the configuration described in this chapter.

* 1. Setting-up a CitiSim scenario with docker

In this chapter we will see how to deploy a CitiSim instance, how to destroy it and how to add new services to the CitiSim Icegrid application.

First of all, **clone the CitiSim repository** in your computer:

git clone <https://bitbucket.org/arco_group/prj.citisim.git>

Inside the repository, we will find a directory named **docker** that contains all the files of the application with the same structure that we saw in Figure 7.

Next, we must **install Docker** and **Docker Compose** if we haven't already done so. To do this, go to <https://docs.docker.com/install/> and select the installation procedure according to your operating system.

To install Docker in **Debian host**, we must follow these steps:

$ sudo apt-get update

$ sudo apt-get install \

apt-transport-https \

ca-certificates \

curl \

gnupg2 \

software-properties-common

$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -

$ sudo add-apt-repository \

"deb [arch=amd64] https://download.docker.com/linux/debian \

$(lsb\_release -cs) \

stable"

$ sudo apt-get update

$ sudo apt-get install docker-ce docker-compose

If you use Windows, you can install Docker from the following link: <https://store.docker.com/editions/community/docker-ce-desktop-windows>

Once we have installed Docker and Docker Compose we can now perform the deployment of the instance of CitiSim. To facilitate both the deployment and the destruction of the docker containers we have a Makefile with two instructions: **up** and **down**.

* + 1. Deploy CitiSim instance

On Linux, you must open a terminal in the docker directory of the prj.citisim repository and execute the following command to deploy a CitiSim instance:

$ make up

This command executes another command, but we have defined it to simplify the deployment process. If you have another operating system that does not have the *make* tool, you can **run the command**:

$ docker-compose -p citisim up --build

This command builds, (re)creates, starts, and attaches to containers for a service. Unless they are already running, this command also starts any linked services.

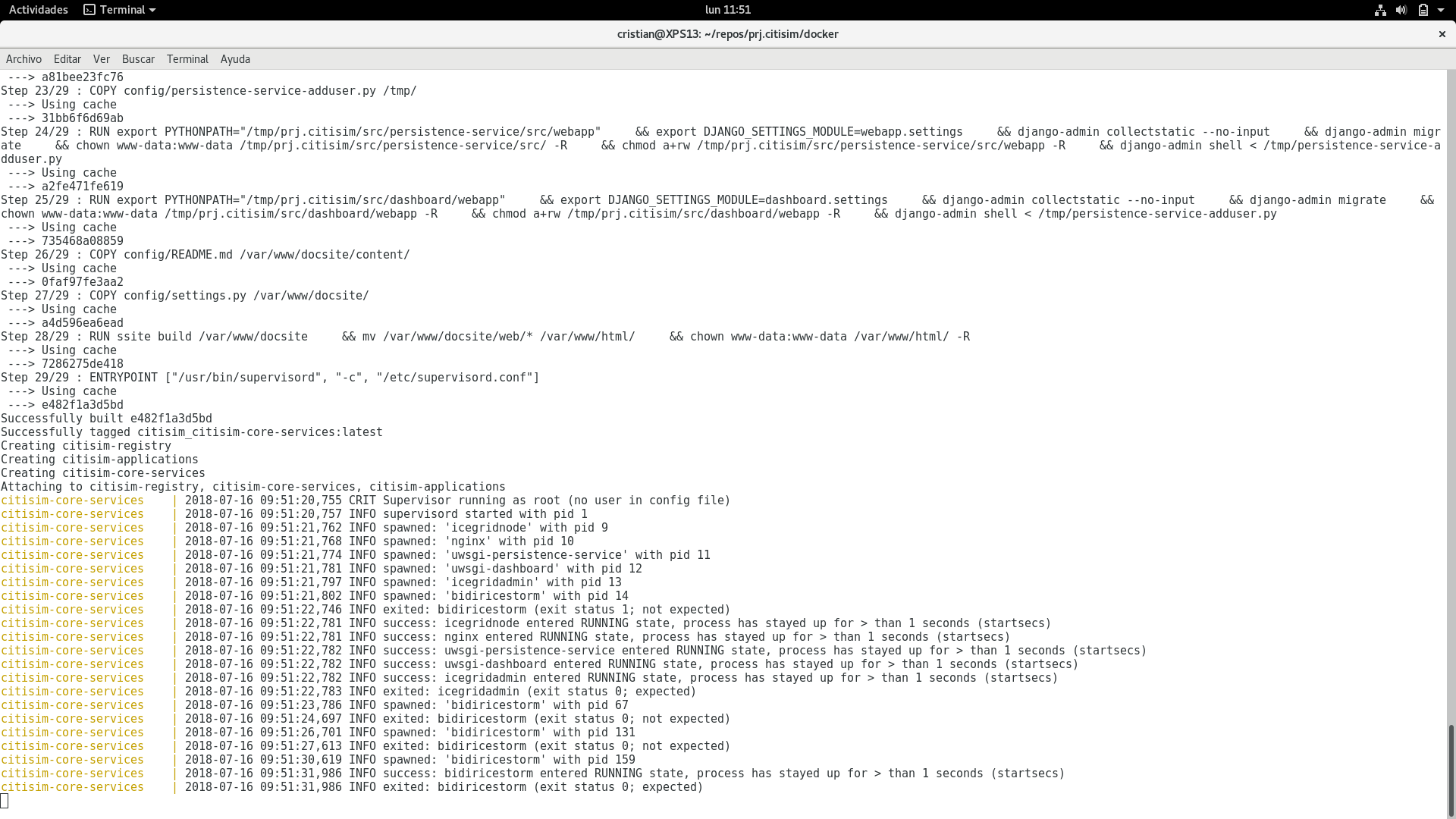
After executing the command, you will have to wait a while and you will get an exit like this in your terminal:

Figure 15: Output of the docker-compose up

To **check** **that all three containers are running properly**, we can open a new terminal and execute the following command:

$ docker ps -a

We should get an output like this:

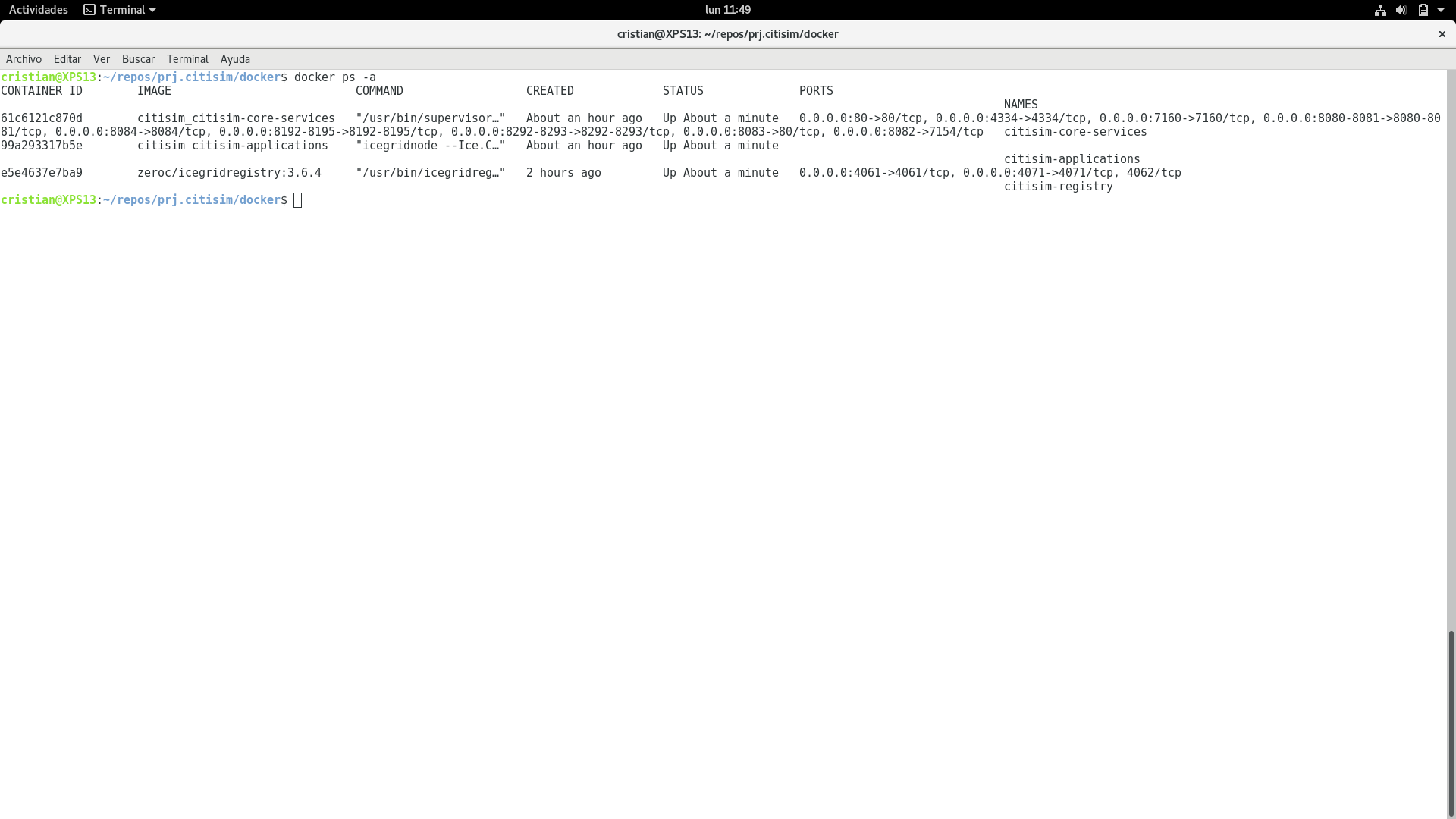


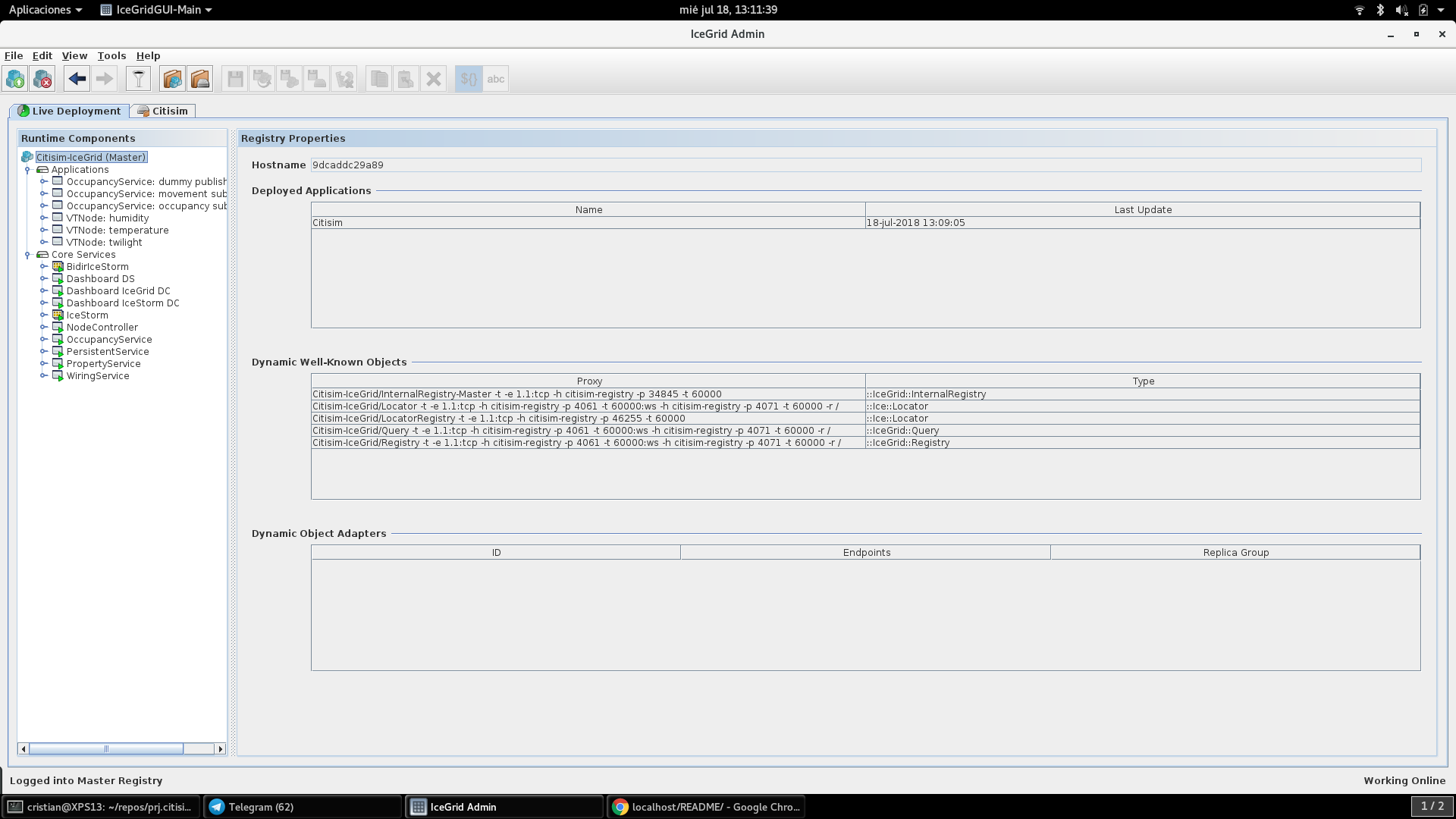
Figure 16: Output of docker ps -a

As we can see, the three containers of the CitiSim instance are listed with their corresponding port redirection. Which means **we have successfully deployed all CitiSim services**.

If you browse to <http://localhost/README>[, you will see a web page where some of the services deployed with their respective credentials and settings are displayed.](http://localhost/)

You can check that all services are running by accessing the IceGrid Registry. To access it we can use the icegridgui tool and configure the parameters as defined in the section *IceGrid-Registry (Citisim-IceGrid)* in the README web page.

Figure 17: CitiSim IceGrid application running



* + 1. Destroy CitiSim instance

To stops containers and removes containers, networks, volumes, and images created by docker compose up, you must type this in a terminal:

$ make down

Or if you don't have make installed:

$ docker-compose -p citisim down --volumes

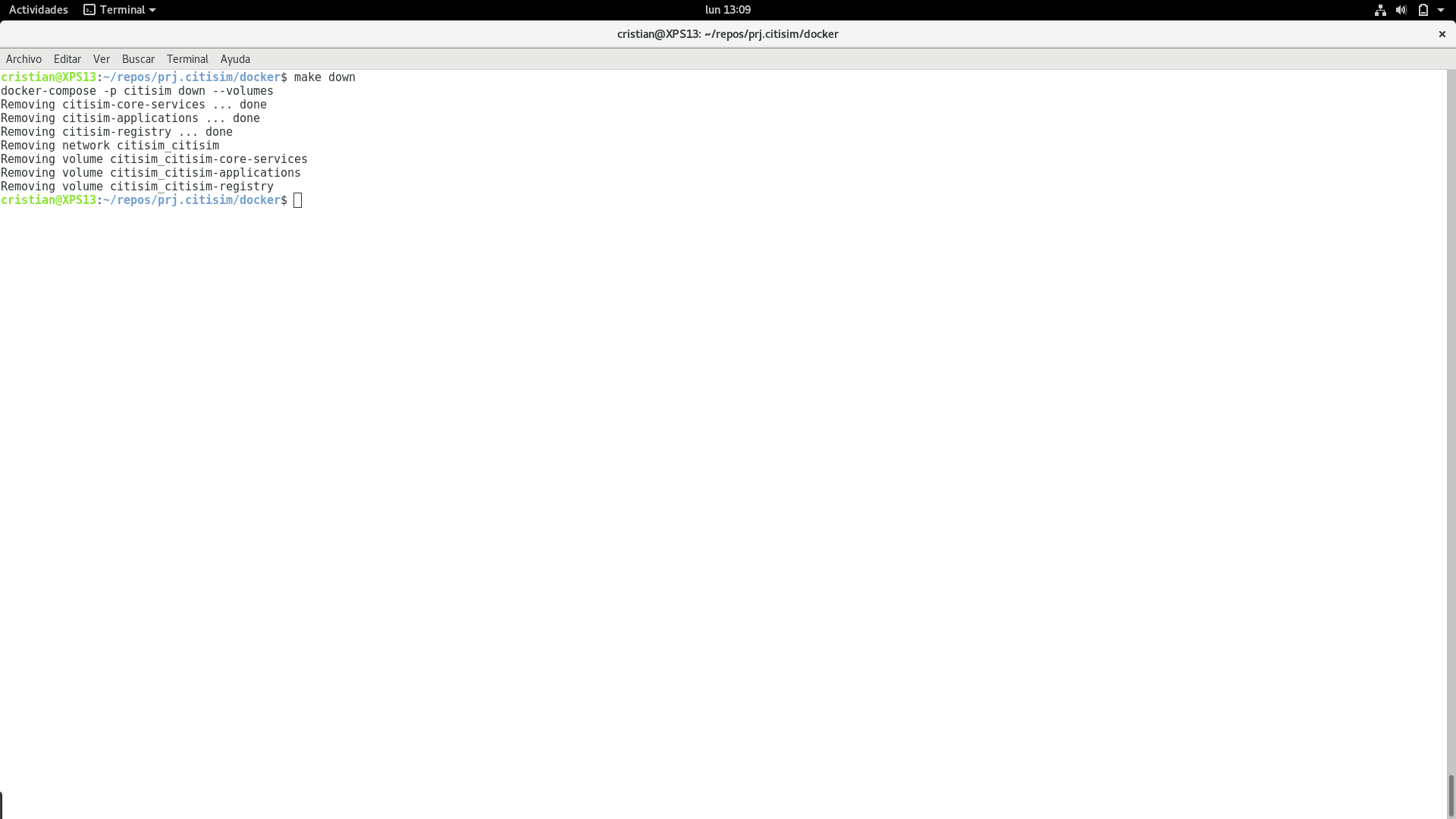
We should get an output like this:

Figure 18: Output of docker- compose down

With this simple command **we have already destroyed the instance of CitiSim**. If you want to keep the volumes with the container data, you must remove the --volumes option from the command.

* + 1. Add service to CitiSim instance

In case we want to add a new service to the CitiSim Docker, we have to **edit several files**. The steps to follow to add a service should be as follows:

1. Install the dependencies and download the service repository to the corresponding Dockerfile file.
2. Write the service configuration file if required and add it to the container with the COPY directive in the location required for the service.
3. Edit the citisim.xml file to add the new server that will run the service.
4. Add the port forwarding used by the service to the docker-compose.yml file.

To illustrate the procedure described above, we are going to add a new server that will run the node controller service configured with a new properties server. This server will run on the Icegrid node that contains the CitiSim core services.

* + - 1. Edit Dockerfile

Since we have already included a Node Controller service in the container, it is not necessary to include the dependencies and the service repository again, but we will review the Dockerfile instructions that must be added for the service to work.

To know the dependencies that the service has, we must go to the control file of the debian folder of the service. This file is located at the following address in the repository: <https://bitbucket.org/arco_group/prj.citisim/src/master/src/node-controller/debian/control>

As we can see, it has dependencies with the following Debian packages: ***python3, python3-flask, python3-zeroc-ice, property-service-simple, iot-node*** and ***citisim-slice***. So, first of all, it would be to install these dependencies in the container, for that we simply write the following instruction in the Dockerfile:

**RUN** apt update \

&& apt -y install python3 \

python3-flask \

python3-zeroc-ice

The other three packages can be downloaded from the pike repository, but for that we have to add it as specified in this link: <http://pike.esi.uclm.es/>. These in packages what they do is to clone the interfaces of the services from CitiSim to the container, so we can do without this step, and clone the repositories and make a symbolic link of these as we have done in the previous Dockerfile.

First, we download the repositories in a directory known as /tmp/ and then we create the directories related to the interfaces and create the symbolic links to the interfaces.

**RUN** git clone ssh://bitbucket.org:/arco\_group/prj.citisim \

&& hg clone ssh://bitbucket.org:/arco\_group/ice.property-service-simple \

&& hg clone ssh://bitbucket.org:/arco\_group/iot.node \

&& ln -sf /tmp/iot.node/slice /usr/share/slice/iot \

&& ln -fs /tmp/prj.citisim/slice /usr/share/slice/citisim \

&& mkdir -p /usr/share/slice/PropertyService \

&& ln -sf /tmp/ice.property-service-simple/src/PropertyService.ice \

/usr/share/slice/PropertyService

With these steps, we would have already solved the dependencies of the Node Controller service. As we have also downloaded the prj.citisim repository, which includes the Node Controller service, we could already run it.

The Node Controller service requires a configuration file that tells you the proxy of the Property Server you need to connect to retrieve the endpoints of the IceC objects. The Property Server proxy will be defined in a file outside of the Dockerfile. So we'll have to load this file with the COPY directive.

**COPY** config/node-controller.conf /var/lib/ice/.config/node-controller/settings.conf

* + - 1. Add service configuration file

Once the dependencies are resolved, the service repository is cloned, and the configuration file is added to the Dockerfile, we must write the proxy of the Properties Server in the configuration file. In this example, we are going to connect the Node Controller to the CitiSim Property Server located at pike.esi.uclm.es.

The Node Controller service configuration file, named as ***node-controller.conf,*** contains the following:

ps\_proxy = "PropertyServer -t:tcp -h pike.esi.uclm.es -p 4334"

* + - 1. Edit CitiSim application

So far, we would have completed steps 1 and 2 of *Section 4.3*. Now let's edit the CitiSim application to create a new server on the node. This server will run the Node Controller service. For the edition of the CitiSim application we can use a text editor or the Icegrid Graphic Interface, also called icegridgui. In this example, we will use icegridgui to add the new server.

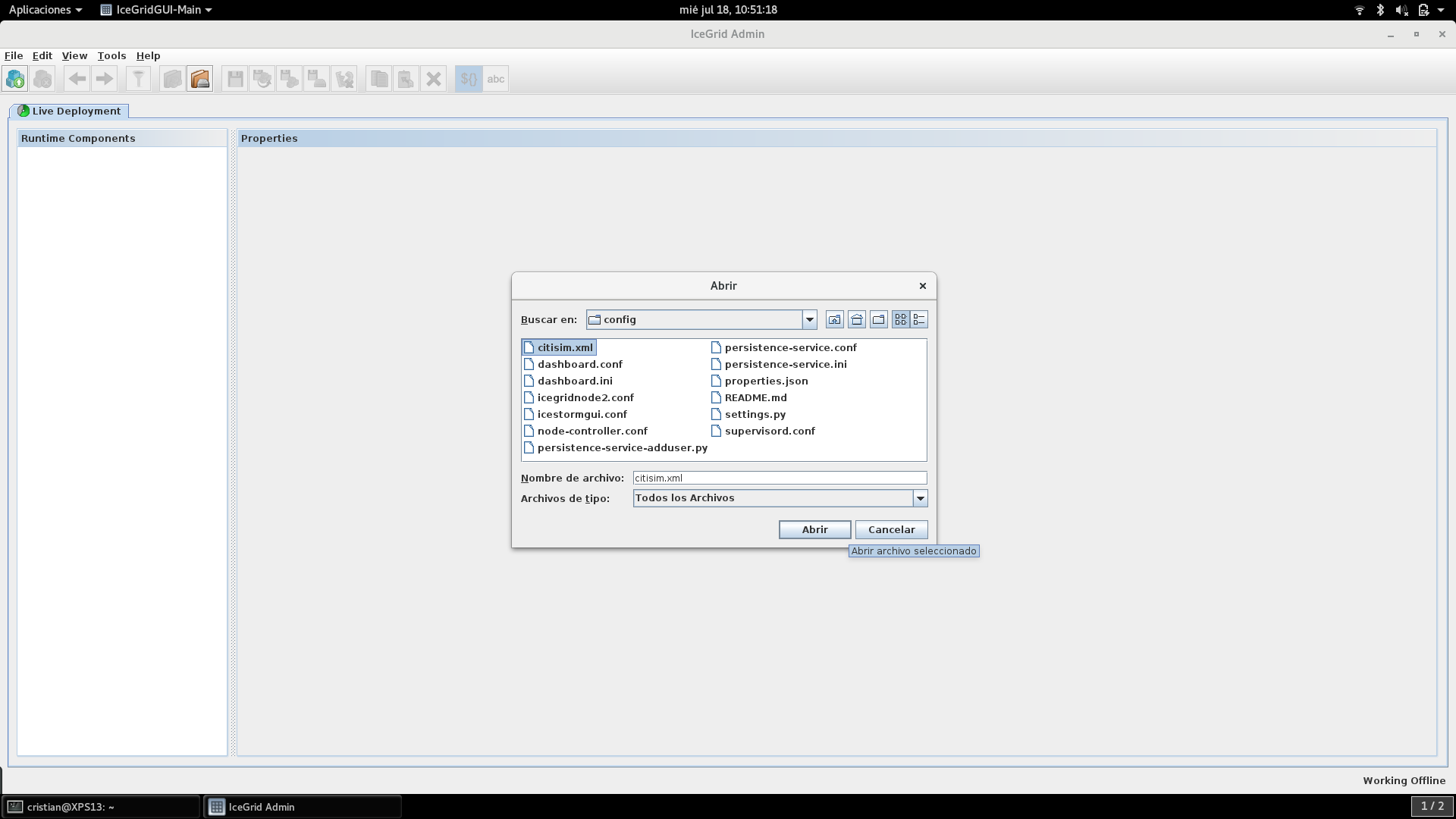
1. First of all, **open icegridgui** and **load the application** citisim.xml.

Figure 19: Open Citisim IceGrid application in icegridgui

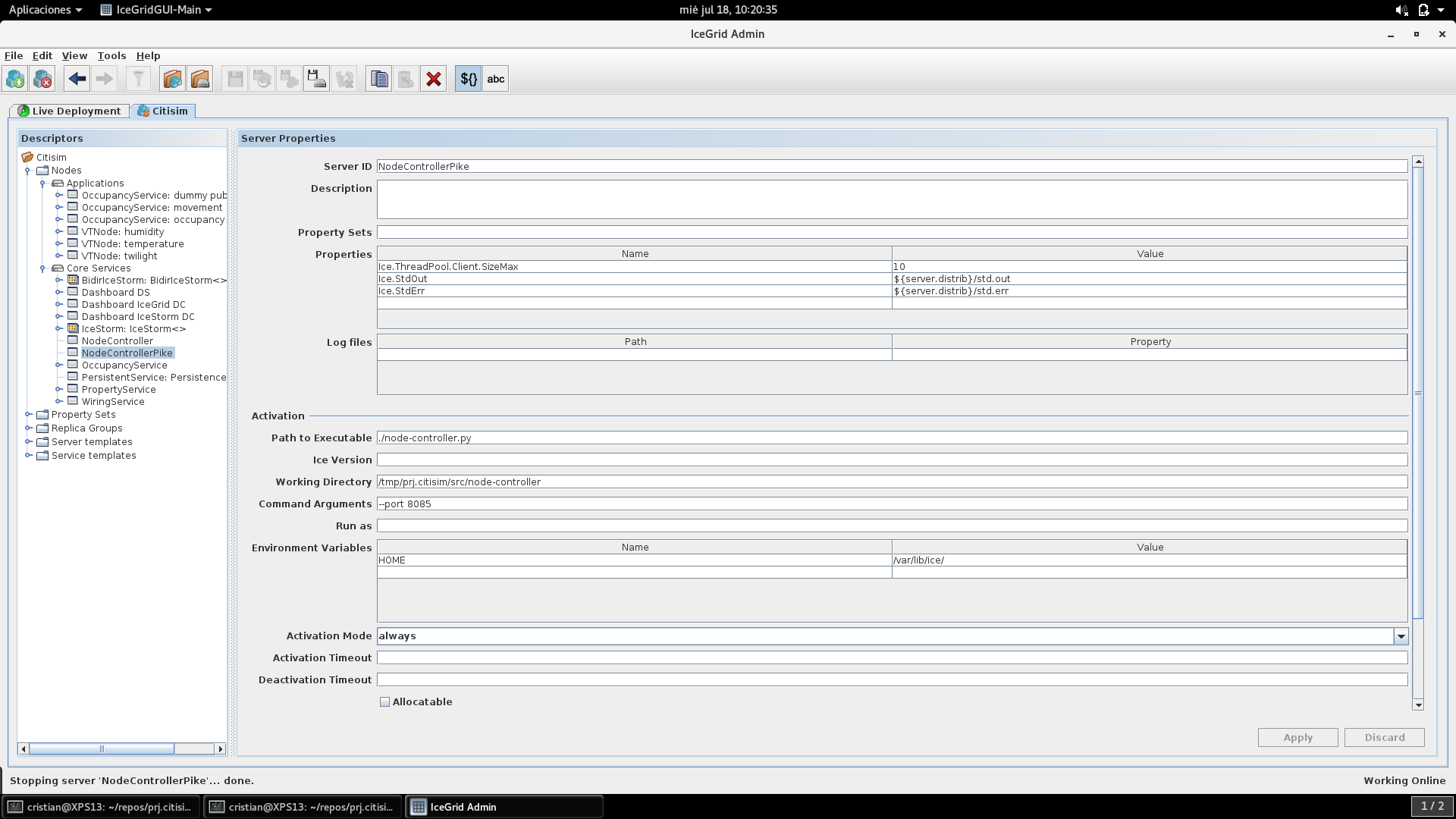
1. Select **New Server** option in the **Core Services node** and add the server with the following properties.

Figure 20: Node Controller server properties

As you can see, the most important properties directly related to the container are those of the **Activation section**. In the Working Directory option, we define the place where is placed the source code of the service. In this case, as we clone the prj.citisim repository in the /tmp/ directory, so we can find the source code in the **/tmp/prj.citisim/src/node-controller**

The **executable** of the service to be executed is: **./node-controller.py**

As an **additional argument** to the executable we have defined **--port 8085** to expose the web interface of the Node Controller in the port of our choice. This is because we already have another Node Controller running on the default port (7154) and it is not possible to have two services running on the same port.

Additionally, we have defined the **HOME environment variable** to the corresponding directory where the configuration file is located that tells the Node Controller the proxy of the Properties Server to which the service must connect. As we saw in the COPY directive of the Dockerfile, we copied the configuration file in the directory /var/lib/ice/.config/node-controller/settings.conf but since the service itself already has hardcoded the directory .config/node-controller/settings.conf, we only have to select the prefix /var/lib/ice/ice/

The last thing to define in this section is the **Activation mode**, in our case we will define it as **always** so that the server starts when loading the application in the Icegrid Registry.

Once all these options have been configured, the only thing left to do is to define the service properties to indicate the location of the output and error files and the name of the server.

1. Finally, we **save the changes** made in the IceGrid application in the same file from where we opened it.
   * + 1. Edit docker-compose.yml

Once the server is configured to be started when we start the container, the only thing left to configure is the **redirection of the service ports** to expose it to the outside of the container. To do this, just edit the docker-compose file and add the option "**8085:8085**" in the ports section of the **citisim-core-services** service:

citisim-core-services:

build: ./core-services

depends\_on:

- citisim-registry

container\_name: citisim-core-services

volumes:

- citisim-core-services:/var/lib/ice/icegrid

ports:

- "80:80" # README

- "4334:4334" # PropertyService

- **"8085:8085" # NodeControllerPike**

..........

With this last step, **we have already completed the process of adding a new service** to the container. Each of the services requires a specific configuration, but we hope that this example will help us to include new services.

1. CitiSim-MQTT-adapter
   1. Service description.

Citisim-mqtt-adapter is an application that connects to a provided MQTT server address, subscribes to a list of topics and forwards received MQTT events to the CitiSim event distribution service.

It parses received messages, extracting the data needed to publish new events to CitiSim making use of the **LibCitiSim** library.

* 1. Interface description and implementation details

CitiSim-mqtt-adapter connects to a given MQTT server, subscribes to a provided list of topics and parses received messages. These messages follow the JSON template described below:

|  |
| --- |
| {  "id": "ID",  "id\_wasp": "ID\_WASP",  "id\_secret": "ID\_SECRET”,  "sensor": "SENSOR",  "value": "VALUE",  "timestamp": "TS("c")"  } |
|  |

Table 2: JSON template for MQTT messages

|  |
| --- |
| {  "id": "64132",  "id\_wasp": "SCP2",  "id\_secret": "42177063D9374202",  "sensor": "BAT",  "value": "7",  "timestamp": "2018-06-05T18:02:20+03:00"  } |
|  |

Table 3: Example of MQTT message

It also interfaces with CitiSim making use of the libcitisim library, which acts like a broker.

It uses the *get\_publisher* method for each configured topic, and then publishes the corresponding event by means of the *publish*method:

* **get\_publisher (topic\_name, source, meta)** : returns the publisher object for a provided topic name.
* **publish (topic\_name, value, source, meta)** : publishes the provided value in the presented topic, along with source and metadata.

**Topic\_name** is a string corresponding to the magnitude being measured. A correspondence is stablished between the “value” specified in the MQTT message and the topic names defined in CitiSim.

**Source** must be an 8 byte string that uniquely identifies a sensor. Usually in MQTT, topics are federated, so there is a topic for each individual sensor. This way a correspondence between the topic and the 8 byte string can be stablished.

Currently this string is assigned to a combination of *MeshliumID*, *WaspmoteID*, and *Sensor,* which are specified in the MQTT topic, but it can be applied to any other format as long as the topic makes reference to a unique sensor.

The following is an example of a topic containing this information:

* meshliumf958/SCP2/BAT

Where “meshliumf958” is the Meshlium gateway ID, “SCP2” is the ID for the Waspmote and “BAT” refers to the sensor.

**Value** is obtained from the MQTT message.

**Meta** is built from metadata contained in the MQTT message. In this case, this metadata consists uniquely on the timestamp of the event itself.

* 1. Service set-up and running

Apart from a working ZeroC ICE installation, the libcitisim package is needed, which can be installed from the following repository: <http://pike.esi.uclm.es/>

In Debian-based systems, this can be achieved by running the following command:

**$ sudo apt-get install python3-zeroc-ice libcitisim**

Paho-mqtt is also needed for the adapter to run, this can be installed with pip3:

**$ sudo pip3 install paho-mqtt**

Once installed, the repository must be cloned:

**$ git clone git@bitbucket.org:arco\_group/citisim-mqtt-adapter.git**

In this directory three main files can be found:

* **mqttAdapter.py** : Service executable file.
* **citisim.config** : Contains the libcitisim broker configuration.
* **mqtt.config** : Contains the MQTT broker configuration and correspondences with CitiSim.

|  |
| --- |
| # -\*- mode: config; coding: utf-8 -\*-  Ice.Default.Locator = IceGrid/Locator -t:tcp -h pike.esi.uclm.es -p 5061  TopicManager.Proxy = IceStorm/TopicManager |
|  |

Table 4: CitiSim.config example

Where **Ice.Default.Locator** is the endpoint of the CitiSim locator and **TopicManager.Proxy** is the proxy for the Topic Manager that will be used by the LibCitiSim broker.

|  |
| --- |
| {  "client" : "CitisimUCLM",  "broker\_addr" : "mqtt.beia-telemetrie.ro",  "topic\_list" : "meshliumf958/#, meshliumf956/#",  "sensor\_ids" :  {  "meshliumf958/SCP1/BAT" :"00000001",  "meshliumf958/SCP1/TC" :"00000002",  "meshliumf956/SCP1/PRES" :"00000003",  }  ,  "icestorm\_topics" :  {  "BAT" :"Battery",  "PRES" :"Pressure",  "TC" :"Temperature",  }  } |
|  |

Table 5: mqtt.config example

Where:

* **client** is the name of the client that will be used to connect to the MQTT broker.
* **broker\_addr** is the address of the MQTT broker.
* **topic\_list** is a list of topics that will be subscribed
* **sensor\_ids** is a correspondence between MQTT topics and the unique 8-byte identifier used in CitiSim
* **icestorm\_topics** is a correspondence between the “sensor” field in the MQTT messages and CitiSim defined topics.
  1. Service example of use

Executing the service is really simple: once citisim.config and mqtt.config are properly configured, all that is needed to do is running mqttAdapter.py with both configuration files as arguments:

**$ python3 mqttAdapter.py citisim.config mqtt.config**

This will connect the service to the configured MQTT broker and wait for events to arrive, locking the console.

For each message received, it will publish the corresponding event to CitiSim.

If a sensor ID is not defined in mqtt.config, events will still be published, but with source = **“MISSING\_ID: <MQTT\_TOPIC>”**  instead of the 8 byte sensor ID.

Also, if the “sensor” field is not defined in mqtt.config, events will still be published, but since the corresponding icestorm topic is not known, they will be published to a topic called **“Unconfigured”**.

* 1. Repository

https://bitbucket.org/arco\_group/citisim-mqtt-adapter/

1. Conclusions

This document is a complement to the *D2.2 Architecture demonstrator (Software)*. It explains the architecture of the CitiSim platform and how-to config and deploys it using docker technology.

The core components and the services components will evolve during the execution of the project. You could access to the last version of them through a software repository.

A Full description of the architecture and a developer manual will be provided in the deliverable “*D2.3 Reference architecture developer* *manual*” by M28.

1. <http://supervisord.org/> [↑](#footnote-ref-1)
2. <https://doc.zeroc.com/ice/3.6/ice-services/icegrid/using-descriptor-variables-and-parameters#id-.UsingDescriptorVariablesandParametersv3.6-SpecialDescriptorVariables> [↑](#footnote-ref-2)