

MASTER IN HIGH PERFORMANCE COMPUTING

MASTER'S THESIS

climateprediction.net: A Cloudy Approach

Author: Diego Pérez Montes
Advisors: Tomás Fernández Pena
Juan Antonio Añel Cabanelas
June 19, 2014

Resumen

Desde los inicios de la Computación de Altas Prestaciones (High Performance Computing), la simulación de modelos climáticos ha sido una de sus principales aplicaciones. Por este motivo, tradicionalmente, los modelos climáticos han sido ejecutados sobre supercomputadores por sus requerimientos y costes. Reducir este coste fue el motivo principal del desarrollo del proyecto climateprediction.net (también llamado CPDN) [1], una iniciativa de computación distribuida cuyo propósito es ejecutar diferentes (miles) de simulaciones de modelos climáticos para investigar las incógnitas sobre determinados parámetros. Este estudio es fundamental para poder comprender cómo pequeños cambios o variaciones en los valores pueden afectar a los modelos, por ejemplo, en investigaciones sobre cambio climático. El proyecto es actualmente propiedad de la Universidad de Oxford, que usa voluntarios a través del framework BOINC (Berkeley Open Infrastructure for Network Computing) [2] (`http://www.climateprediction.net`).

Infraestructura Actual

A pesar de que el objetivo de esta tesis no es la migración total de los servidores BOINC (sólo del Data Server, cómo se verá más adelante), sino los clientes de la infraestructura actual (que es dónde se produce la carga computacional), es necesario realizar una descripción de alto nivel para comprender de manera más clara el problema que se está intentando resolver.

En la Figura 1, se muestra la arquitectura de alto nivel de BOINC. Por la parte de los servidores se encuentran los siguientes elementos:

- Task server: servidor de tareas, planifica y reparte los trabajos a ejecutar a los clientes.
- Data server: servidor de datos, usado para enviar y recibir las unidades de trabajo (workunits) de los clientes BOINC.
- Bases de datos y almacenamiento: hay 2 bases de datos, una interna a BOINC y otra para las definiciones del proyecto. También hay un almacenamiento usado por el servidor de datos.

A más bajo nivel, hasta 5 demonios son usados en los servidores de tareas (Task server) y de datos (Data server):

- Scheduler: planificador, decide que trabajos son asignados a los clientes.

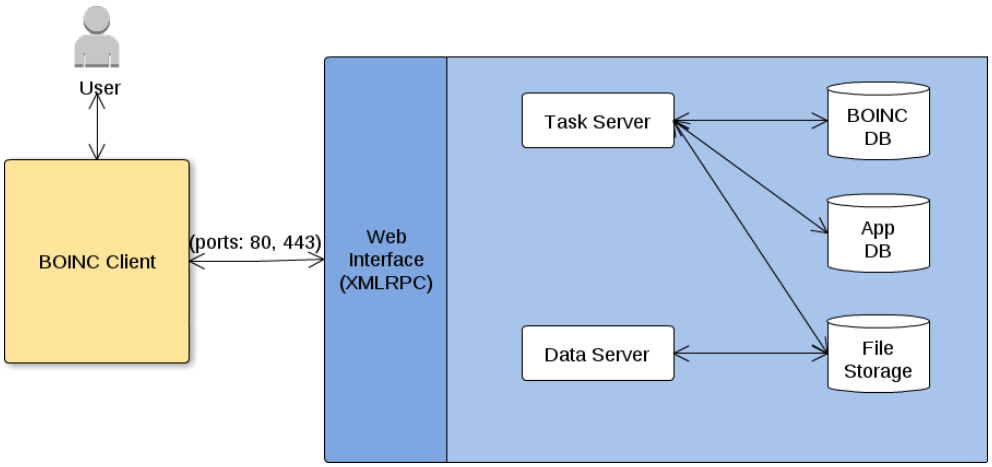


Figure 1: BOINC: Arquitectura, Alto Nivel

- Feeder: alimentador, comunica la base de datos de información con el CGI (Common Gateway Interface) del planificador, a través de memoria compartida.
- Transitioner: transicionador, controla los cambios en los estados de los resultados.
- Validator: validador, valida que los datos enviados por los clientes sean correctos.
- Assimilator: asimilador, usado para "asimilar" ficheros, esto se puede ver como una etapa (opcional) de *post-procesamiento*, por ejemplo mover los ficheros a otro almacenamiento o comprimirlo.
- File deleter: eliminador de ficheros, borra los ficheros que ya no son necesitados, esto se realiza en cuanto una workunit se finaliza de procesar.

El flujo de trabajo en una ejecución de BOINC es de la siguiente manera:

1. El cliente se conecta al servidor BOINC especificado (con unas credenciales, usuario/contraseña) y revisa el estado del proyecto. La comunicación se realiza usando HTTP(s) (Hypertext Transfer Protocol, puertos: 80 y 443) a través de RPC (Remote Procedure Calls), lo que da acceso a la planificación y procesamiento de workunits (computación a ser realizada en el cliente), dando como resultado (completo) las tareas.
2. El cliente solicita que le sean asignados trabajos.
3. Si hay algún trabajo disponible, el servidor revisa los parámetros del cliente (Sistema Operativo, arquitectura,...) y el Task Server le asigna y envía uno. Por razones de seguridad y validación cada trabajo se duplicado (y se envía a otro cliente).
4. En el lado del cliente el trabajo se recibe (los datos de entrada/ficheros son enviados por el Data Server) y la computación comienza (se realiza un fork).

5. Una vez que la workunit se ha procesado, los resultados completos (tarea) se envían de vuelta al Data Server.

6. El demonio del servidor (Validator) evalúa el resultado, revisa la integridad, y si es correcto le da crédito al cliente.

Descripción del Problema

Actualmente el proyecto se enfrenta a algunos problemas y retos:

- Debe ejecutarse una nueva versión del modelo, para la cual la actual arquitectura e infraestructura basada en BOINC no provee la mejor solución. Esto se debe a que los modelos corren sobre un entorno heterogéneo y descentralizado, cuyo comportamiento no puede ser anticipado o medido, siendo el control sobre los recursos disponibles realmente limitado.

- Siguiendo con el anterior punto, es necesario el establecimiento de algún tipo de métricas.

- Necesidad de una racionalización de los costes asociados al proyecto.

Por estas razones **esta tesis estudia, propone e implementa los siguientes objetivos:**

- Conversión total a Infrastructure as a Service (IaaS) [3], migrando a una infraestructura Cloud (Amazon Web Services [4]), incluyendo la computación y recursos de almacenamiento.

- Implementación de un Sistema Central de Control (incluyendo un *Dashboard*), que gestionará las instancias y mostrará métricas y estadísticas de alto nivel para las diferentes ejecuciones del proyecto y despliegues.

- Desarrollar todo con Software Libre para facilitar su reproducibilidad. [5].

- Documentación completa sobre el proceso (este documento).

Además, para probar el trabajo desarrollado en esta tesis, se han realizado varios experimentos con simulaciones regulares (workunits), desarrolladas por el proyecto climateprediction.net/weatherhome. Se han procesado las workunits principalmente de dos experimentos: the weatherhome UK floods (inundaciones en UK) [6] y weatherhome para Australia y Nueva Zelanda. Ambos experimentos usan para las simulaciones de (sólo) atmósfera el modelo de circulación general HadAM3P [7], originando la versión regional del mismo modelo, HadRM3P [8]. La resolución horizontal para la malla de estas simulaciones fue de 50Km. Los resultados de las workunit para el experimento de weatherhome UK han sido exitosamente usados en otros dos artículos de investigación [5, 9].

Objetivos Alcanzados

Se ha demostrado con éxito que es posible realizar simulaciones con un modelo climático en una infraestructura en la nube, algo que, aunque a priori no parece

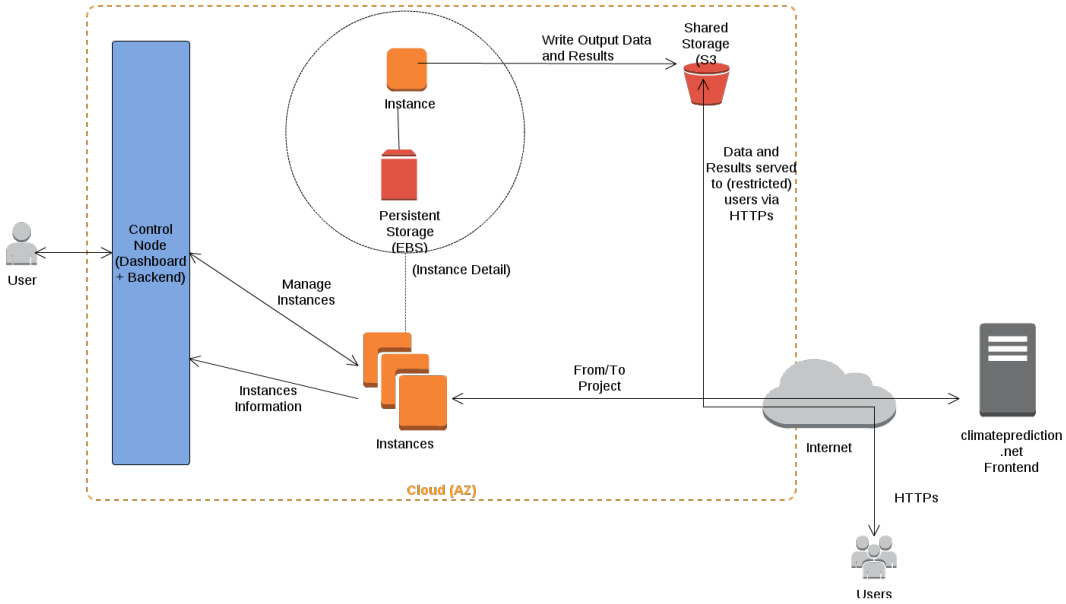


Figure 2: Infrastructura Propuesta

ser complejo, no ha sido probado por ninguna persona, al menos que el autor o directores de esta tesis tengan conocimiento.

Destacar que el proyecto aquí presentado ha logrado, en un corto período de tiempo, no sólo explorar la posibilidad incipiente del uso exitoso de los recursos de computación de alto rendimiento en la nube para la computación científica, sino que también ha ayudado a experimentos de investigación sobre el clima que se encuentran en el proceso de publicación ([5], [9]).

Añadir que este trabajo ha servido como base para obtener nuevos proyectos de investigación como parte de climateprediction.net. En concreto, un proyecto dirigido por uno de los co-directores de esta tesis ha obtenido en mayo de este año el Premio de Investigación Microsoft Azure [10], uno de los únicos 200 proyectos financiados por Microsoft Research, para estudios punteros con tecnologías en la nube. Este proyecto, se basa en el éxito demostrado en la aplicación de tecnologías y soluciones de esta tesis fin de máster.

Esquemáticamente, los objetivos alcanzados a alto nivel han sido:

- Los clientes han sido exitosamente migrados a la nube (sobre AWS EC2 y S3), como se muestra en la Figura 2.
- Distintas simulaciones han sido ejecutadas sobre la nueva infraestructura.
- Se ha desarrolado un nodo de Central (incluyendo un backend y frontend con un Dashboard), desplegado y testeado.
- Se han mostrado de manera comprensiva los costes del proyecto y la simulación, asi como métricas sobre ellos.

Posible Mejoras

La mayoría de las posibles mejoras deberían de centrarse en dar más lógica a la interacción con el estado de los clientes (usando RPC), de modo que más métricas pueden ser obtenidas y se propicie la creación de una capa de Software as a Service (SaaS). Desde el punto de vista de la infraestructura se pueden realizar las mejoras: primero, creación de una ejecución automatizada de prueba para ajustarse al precio real de los recursos antes de cada simulación, segundo, migrar totalmente la parte del servidor a la nube de manera de los costes de transferencia de datos y latencia se reduzcan bastante.

Abstract

climateprediction.net is one of the biggest projects in Earth about simulation models for climate research for more than 10 years. Nowadays, the project is facing a set of new different challenges:

- Growing and variable need for new resources, caused by the execution of a new version of the model, and the output and processing of huge amount or results that can't be processed by the current infrastructure that runs over the BOINC framework (Berkeley Open Infrastructure for Network Computing, weather@home).
- Lack (and need) of more control over the workflow (and the intermediate stages).
- Streamlining of the costs and budget.

To address these issues, this thesis makes an initial study of the current state of the project and propose/implement the next solutions:

- Re-engineer (and deploy) the client side from a volunteer computing architecture to a Infrastructure as a Service (IaaS) based on Cloud Computing (over Amazon Web Services, AWS). This new infrastructure will take care both of the computing, based on AWS EC2 (Elastic Compute Cloud) instances, and (shared) storage needs over S3 buckets.
- Design and develop a Control Node with a system composed by a RESTful backend (an Application Programming Interface (API) coded in Python over Flask, Boto to communicate with the AWS API, and SQLite3 as Database) to control the simulation executions, and a frontend to display statistics (using HTML5 and JQuery/jqPlot), information and metrics.
- Provide documentation about the process, caveats, next steps and recommendations for the project.

(Illustration by: Miles Kelly, Royalty Free)

Contents

Introduction and Problem Background

Since the beginning of the High Performance Computing, climatic models simulation has been one of its main application, because of this, traditionally they were executed in supercomputers because of their needs and costs, this is why the climateprediction.net project (also known as CPDN) was created [1] as a distributed computing initiative, which aim is to run different (thousands) climate modeling simulations in order to research the uncertainties on some parameters, which is really important to be able to understand how small changes or variations in the values can affect the models, for instance for climate change investigations. The project is currently run by the University of Oxford using volunteer computing via the BOINC framework (Berkeley Open Infrastructure for Network Computing) [2] (`http://www.climateprediction.net`).

1.1 Current Infrastructure

Although the purpose of this thesis is not the full migration of the BOINC servers (only the Data Server), but the client side of the current infrastructure (that is where all the computational load happens), it seems necessary make a high level description of it so the understanding of the problem that is being tried to solve will become clearer.

In the Figure 1.1, the high level architecture of BOINC is shown. In the server part, there are different elements:

- Task server: This server schedules and dispatches the jobs to the clients.
- Data server: This server is used to send and receive the workunits to the BOINC clients.
- Data bases and storage: There are 2 Databases, one for BOINC itself and another one for the project defined application. There is also a File storage used by the Data Server.

At lower level, up to 5 daemons are used for the Task and Data servers:

- Scheduler: decides how jobs are dispatched to the clients.
- Feeder: communicates database information to the scheduler CGI (Common Gateway Interface) via shared memory

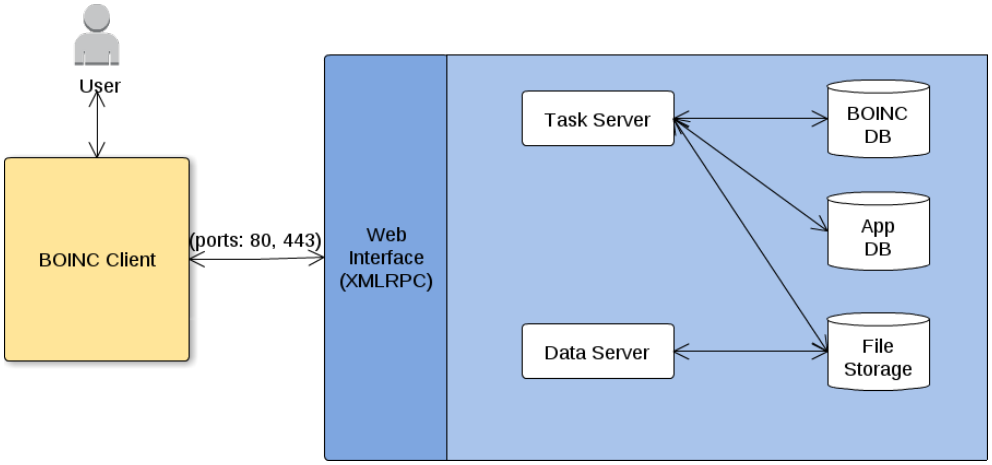


Figure 1.1: BOINC: High Level Architecture

- Transitioner: controls changes of state for results
- Validator: validates that the sent result is valid,
- Assimilator: used to assimilate the file, this can be seen as an (optional) *postprocessing* stage, for instance move to another storage system or compress it.
- File deleter: deletes files that are no longer needed after a workunit has finished

The workflow of a BOINC execution is as follows:

1. The client connects to the BOINC server (with the given credentials) and checks the status of the project. The communication is done using HTTP(s) (Hypertext Transfer Protocol, ports: 80 and 443) via RPC (Remote Procedure Calls), which gives access to the scheduling and processing of workunits (computation to be performed) giving as (completed) results the tasks.
2. The client asks for jobs.
3. If there is any job available, the server checks the client parameters (Operating System, architecture,...) and the Task Server assigns and sends the client one. For security and validation reasons every jobs is duplicated (and dispatched to another client as well).
4. In the client side the job is received (needed input data/files will be sent by the Data Server) and the computing starts (a fork is done).
5. Once the workunit is finished the completed results (called task) are sent back to the Data Server.
6. The server (validator daemon) evaluates the results, checks the integrity, and if it is correct gives credit to the client.

1.2 Problem Description

Currently, the project is facing some issues and challenges:

- A new version of the model needs to be run and the current architecture and infrastructure based in BOINC (as it is) does not provide the best solution for this purpose, and this is because the models are running over an heterogeneous and decentralized environment which behavior can not be anticipated or measured and the control over the available resources is really limited.
- Attending to the previous point, establishment of some kind of metrics is needed.
- A rationalization of the costs needs to be done.

Because of this reasons **this thesis studies, proposes and implements the next objectives**:

- Full conversion to an Infrastructure as a Service (IaaS) [3], by the migration into a Cloud Infrastructure (Amazon Web Services [4]), including Computing and (shared) Storage Resources.
- Implementation of a Central Control System (including a Dashboard), that will manage the instances, as well as display high level metrics and statistics for the different project executions and deployments.
- Make everything with Free Software to comply with science reproducibility [11].
- Complete documentation about the process (this document).

Also, for the purpose of this thesis we performed several experiments with regular simulations (workunits) developed by the climateprediction.net/weatherhome project. We processed workunits from two main experiments: the weatherhome UK floods [6] and the weatherhome Australia New Zealand project. Both experiments use for the simulations the atmosphere only general circulation model HadAM3P [7] driving the regional version of the same model, HadRM3P [8]. The horizontal resolution for the grid of these simulations was 50 km. The outputs of workuntis for the weatherhome UK floods experiment have been already successfully used in another two research papers [5, 9].

Computing Infrastructure Migration

The proposal presented in this document has been built using Amazon Web Services (AWS). This solution has been selected because of different reasons:

- available resources: at the moment of writing this document Amazon is one of the biggest world Infrastructure as a Service (IaaS) providers.
- it is well documented: which can be really useful for new features development and troubleshooting.
- migration from/to another Cloud is supported: avoiding this way a vendor lock-in in case that the project decides to use another system.

Anyway, most of the procedures here described can be reused into a generic way for another Cloud Systems, and this will be documented into the most agnostic way possible.

2.1 First Estimation and Problem Dimension

Initially, and in order to estimate the real size of the problem, test runs need to be done. For this, and after analizing the different possibilities available in AWS EC2, two representative systems were selected.

The first system, was CPU computing oriented (medium-intensive), its specifications are detailed in Table 2.1, and the second one for CPU & GPU computing (highly-intensive), as seen on Table 2.3.

Then, the BOINC client was configured into every instance and done 10 consecutive executions (this is, execute the client and request/run jobs for the same amount of time in both systems), results for system 1 are the last 2 rows of table 2.1 and for system 2 on the last 2 rows of table 2.3.

Another important aspect calculated is the cost of simulations, that can be checked on table 2.2 and 2.4 respectively.

After executing the climatic model into the two different configurations, and analyzing the result data, the next aspects can be highlighted:

- The CPU usage was always between the 98% and 100%.
- The (efficient) usage of GPUs was low (and not really shown on the logs).

CPU:	2 x Xeon E5-2650
MEM:	8GB (4 GB/Core, 50%)
GPU:	No
Running Time (avg):	59.46 hours
Time per Workunit (avg):	175.91 hours (7.32 days)

Table 2.1: System #1: m1.large

Ratio, Workunit/Hour:	0.0056 units/hour
Execution Cost (59.46 hours):	USD 1.486
Workunit Cost:	USD 4.464
Simulation Cost (36,000 workunits):	USD 160,704

Table 2.2: System #1: Ratios and costs:

CPU:	16 x Xeon X5570
MEM:	24GB (1.50 GB/Core, 6.25%)
GPU:	2 x Tesla M-Class M2050
Running Time (avg):	58.38 hours
Time per Workunit (avg):	47.89 hours (1.99 days)

Table 2.3: System #2: cg1.4xlarge

Ratio, Workunit/Hour:	0.0208 units/hour
Execution Cost (58,38 hours):	USD 122.603
Workunit Cost:	USD 100.966
Simulation Cost (36,000 workunits):	USD 3,634,776

Table 2.4: System #2: Ratios and costs:

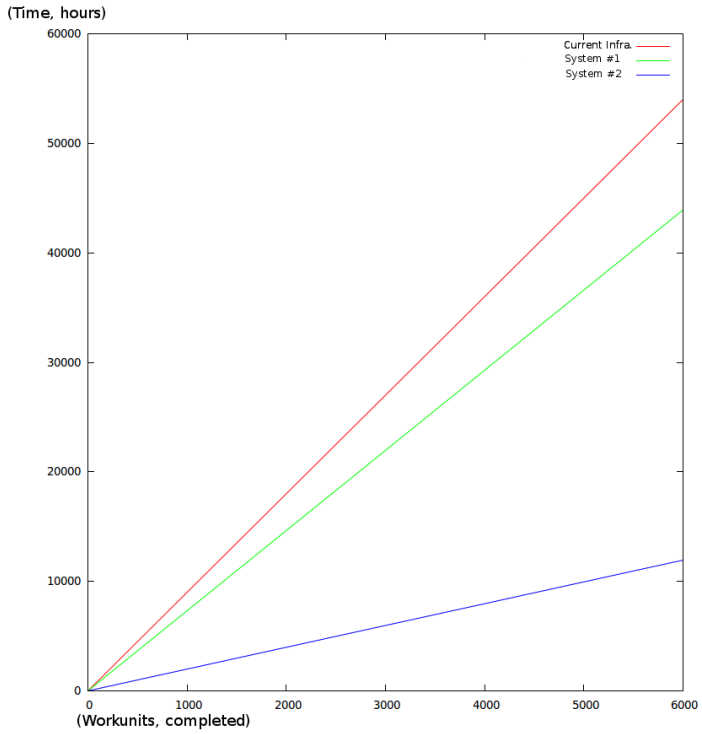


Figure 2.1: Workunit Processing Time Comparison (less is better)

- The memory usage was low (average: 1.5%, this is, wasting 98.5% of this resource).
- The storage usage was low, so a resource optimization can be done by using smaller but faster disks.
- It was detected that some parts of the model were running in 32 bit, switching to 64 bit is highly recommended.

With this we can conclude that the model is highly CPU demanding, and that the memory and disk Input/Output usage are really low and have a minor impact into the simulation.

Even though the cg1.4xlarge performed better (Figure 2.1), analyzing the data it seems that **the m1.large is more efficient** because the price per workunit is better (Figure 2.2) and the average time per workunit is still under the current one($\sim$7 days/workunit vs. $\sim$8 days/workunit), and therefore it is more suitable for the simulations. Anyway, the prices and specs may vary within a short amount of time so an automated reevaluation of the relationship workunits/hour price could be needed (prices for this evaluation were last consulted on 10/06/2014).

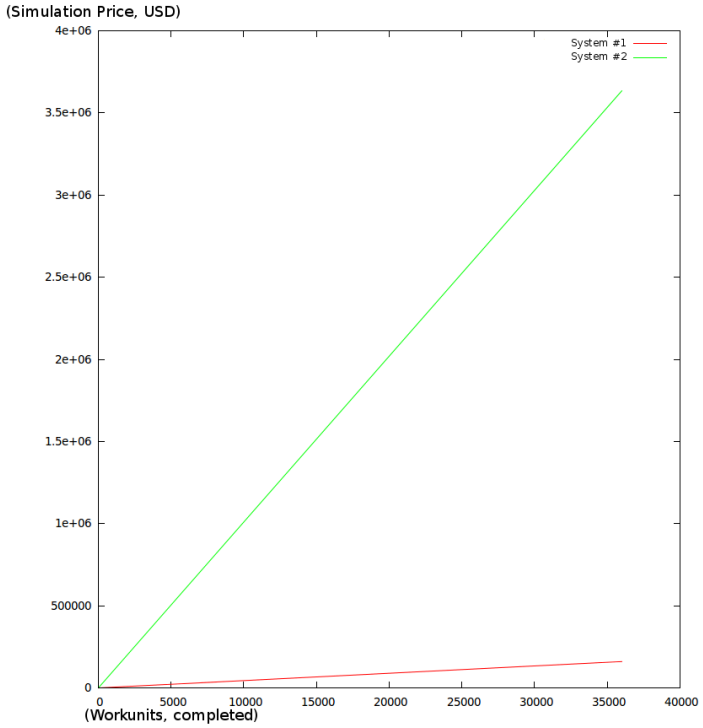


Figure 2.2: Simulation Price Comparison (less is better)

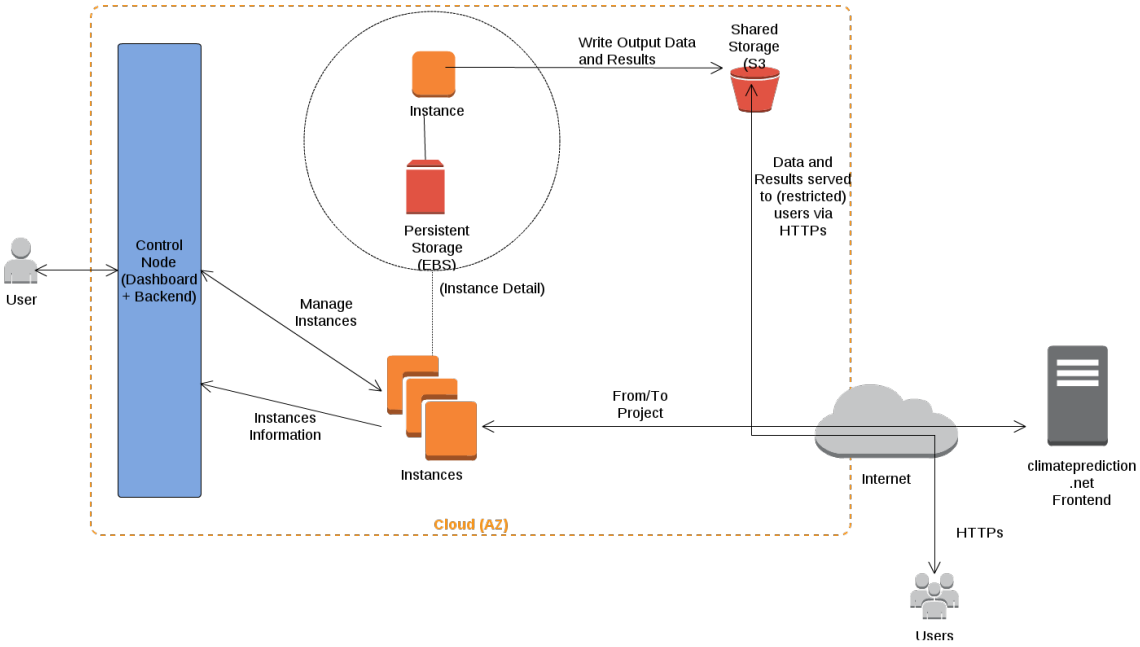


Figure 2.3: Proposed Infrastructure

2.2 Computing Infrastructure Design and Implementation

The new computing infrastructure, Figure 2.3, will be built over virtualized instances (AWS EC2). Amazon provides also *Autoscaling Groups*, that allows the user to define policies to add or remove dynamically instances triggered by a defined metric or alarm. As the purposes of this project are use the rationalization of the resources, and also have full control over them (via the Central System described on Chapter 4) this feature, as well as any type of Load Balancing or Failover, won't be used in the Cloud side but, in the control system node that serves as backend for the Dashboard.
The new workflow for a project/model execution is:

Prerequisites: Tasks have been setup in the server side and are ready to be sent to the clients, this can be currently checked into the public URL `http://climateapps2.oerc.ox.ac.uk/cpdnboinc/server_status.html`

1. The (project) Administrator user configures and launches a new simulation via the Dashboard.

2. The required number of instances are created based on a given template that contains a parametrized image of GNU/Linux with a configured BOINC client.

3. Every instance connects to the server and fetch 2 tasks (1 per CPU, as the used instances have 2 CPU, described in the section 2.1).

4. When a task is processed the data will be returned to the server, and

OS Image (AMI):	Amazon Linux AMI 2014.03.1 (64 bit)
Instance Type:	m1.large
Firewall (Security Groups):	Inbound: Only SSH (22) Accepted Outbound: Everything Accepted.
Persistent Storage:	Root 16GB (volume type: standard)

Table 2.5: Parameters for Template Instance

also stored into a Shared Storage so it will be accessible for a given set of authorized users.

5. Once there are not tasks available the Control Node will shut down the instances.

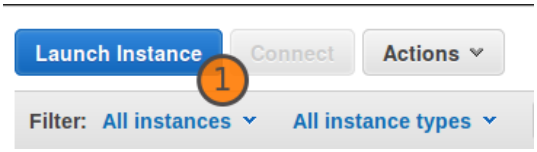
It should be noted that at any point the Administrator will be able to have real time data about the execution (metrics, costs...) as well as change the running parameters and apply them over the infrastructure.

2.2.1 Template Instance Creation

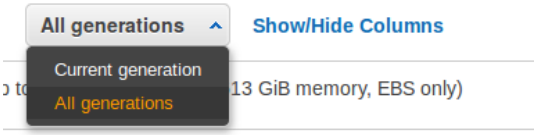
In order to be able to create an homogeneous infrastructure the first step is to create an (EC2) instance that can be used as template for the other ones.

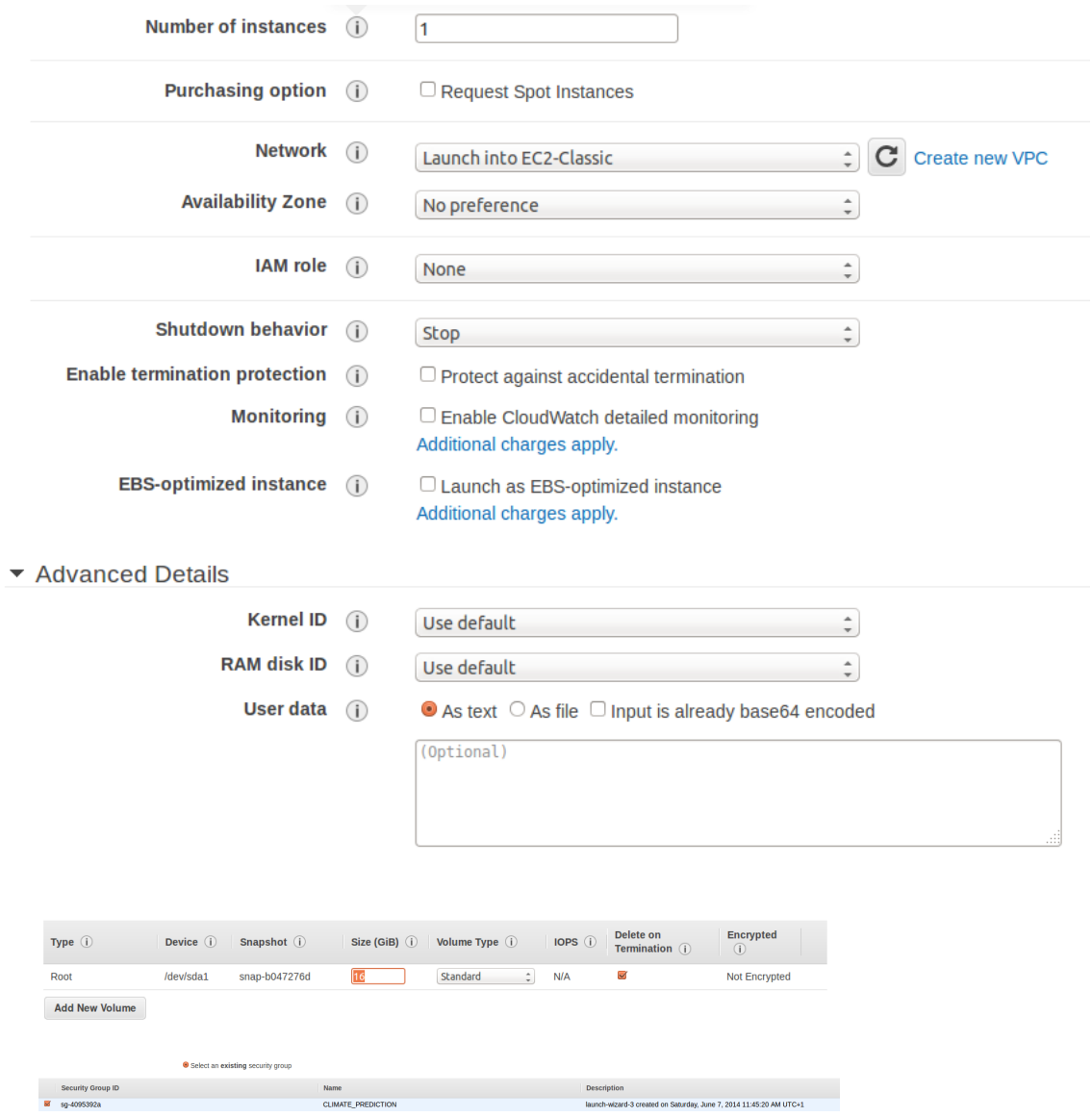
The high level steps to follow to get a Template Instance (with the parameters defined in the Table 2.5) are exposed below:

1. On the AWS dashboard click “Launch Instance”, then select the given OS image (AMI) type.



2. Select the image type (m1.large, if not in the list select the “All Generations” option).





3. Revise and set the parameters.

4. Launch the Template Instance.

Note: Remember to create a new key-pair (public-private key used for password-less SSH access to the instances) and save it (it will be used for the Central System), or use another one that already exists and is currently accessible.

Installing and testing AWS and EC2 Command-Line Interface

(Prerequisites: wget, unzip and Python 2.7.x)

This step is optional, but it is highly recommendable because this will be advanced control of the infrastructure through the shell. The follow description applies and have been tested on Ubuntu 14.04 [12], but can be reproduced into any GNU/Linux system:

First, create an *Access Key* (and secret/password), via the AWS web interface in the *Security Credentials* section. With this data the *AWS_ACCESS_KEY* and *AWS_SECRET_KEY* variables should be exported/updated, please have in mind that this mechanism will be also used for the Dashboard/Metrics Application (Section 4 of the current document):

```
$ echo "export AWS_ACCESS_KEY=<your-aws-access-key-id>" \
>> $HOME/.bashrc
$ echo "export AWS_SECRET_KEY=<your-aws-secret-key>" \
>> $HOME/.bashrc

$ source $HOME/.bashrc
```

```
#Download and install the AWS CLI
$ wget https://s3.amazonaws.com/aws-cli/awscli-bundle.zip &&\
unzip awscli-bundle.zip &&\
sudo ./awscli-bundle/install -i /usr/local/aws -b /usr/local/bin/aws

#Download EC2 API tools
wget http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip &&\
sudo mkdir /usr/local/ec2 &&\
sudo unzip ec2-api-tools.zip -d /usr/local/ec2

#Remember to update the
#PATH=$PATH:/usr/local/ec2/ec2-api-tools-<API_VERSION>/bin
$ echo "export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64/" >>
$HOME/.bashrc

$ source $HOME/.bashrc
```

Installing BOINC and its dependencies

The project executes both 32 and 64 bit binaries for the simulation so once the Template Instance is running, the needed packages and dependencies need to be installed via:

```
$ sudo yum install expat.i686 flac.i686 fontconfig.i686 freetype.i686 \
gamin.i686 glib2.i686 glibc.i686gnutls.i686 gtk2.i686 libX11.i686 \
libXau.i686 libXext.i686 libXfixes.i686  libXft.i686 libXi.i686 \
libcom_err.i686 libgcc.i686 libgcrypt.i686 libgpg-error.i686 \
libstdc++.i686 libxcb.i686 xcb-util.i686 zlib.i686 libcurl.i686 \
openssl097a.i686
```

The version of BOINC used will be the latest from git [13], to download and compile it:

```
#Install needed development tools
$ sudo yum groupinstall 'Development Tools'
```

```
$ sudo yum install git libcurl libcurl-devel openssl097a.i686 \
openssl-devel

#Fetch BOINC source code, compile and install the client
$ git clone git://boinc.berkeley.edu/boinc-v2.git boinc && cd boinc\
&& ./_autosetup && ./configure --disable-server --disable-manager \
--enable-optimize && make \
&& sudo make install

#Create user and set permissions and ownership
#(in case that we want a boinc user)
$ sudo adduser boinc
$ sudo chown boinc /var/lib/boinc
```

Once the BOINC client is installed it must be configured so it will automatically run on every instance with the same parameters:

1. Create a new account in the project:

```
$ boinccmd --create_account climateprediction.net <EMAIL>\
<PASSWORD> <NAME>
```

2. With the account created (or if already done) the client needs to be associated to the project by creating a configuration file with the user token:

```
$ boinccmd --lookup_account climateprediction.net <EMAIL> \
<PASSWORD>| grep "account key" | sed 's/\(.*\): \(.*\)/\2/g' \
| xargs boinccmd --project_attach climateprediction.net

#Status check
$ boinccmd --get_state
```

3. Make BOINC to start with the system (ec2-user will be used because of permissions):

```
$sudo echo 'su - ec2-user -c "cd /home/ec2-user/boinc-client/bin/ \
&& ./boinc --daemon"'\
>> /etc/rc.local
```

Simulation Terminator

An essential piece of software, developed for this thesis, is the *Simulation Terminator*, that decides if a node should shutdown itself in case that workunits were not processed for a given amount of time (by default 6 hours, via cron) or there are no jobs waiting on the server.

This application is provided with the digital content enclosed with this document.

To install it (by default into */opt/climateprediction/*):

```
$ sudo ./installClient
>> Simulation Client succesfully installed!
```

When an instance is powered off will be terminated (destroyed) by the *Reaper* service, that runs into the Central Control System (as described on Chapter 4).

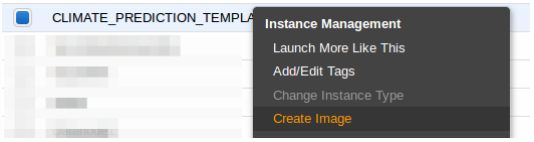
2.2.2 Contextualization

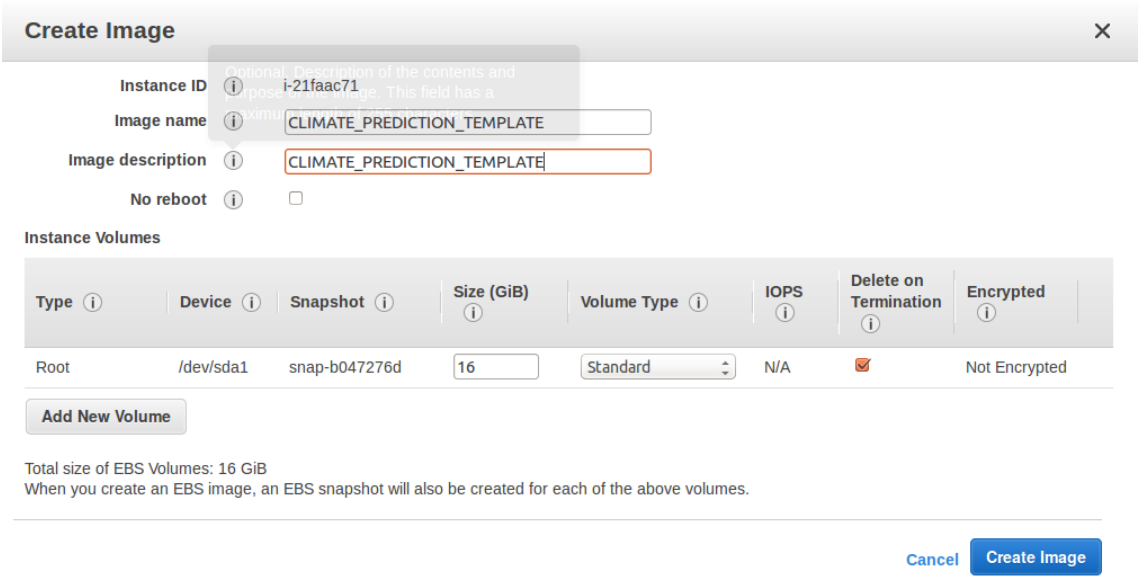
Now that the Template Instance is ready, this is that all the parameters have been configured and the BOINC client is ready to start processing tasks, the next stage is to Contextualize it. This means that a OS image will be created from it, which will give our infrastructure the capacity of being scalable by creating new instances from this new image.

Unfortunately this part is strongly related with the Cloud type, and although can be replicated into another systems will only explicitly work in this way for AWS.

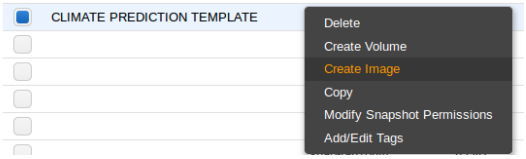
The steps to follow:

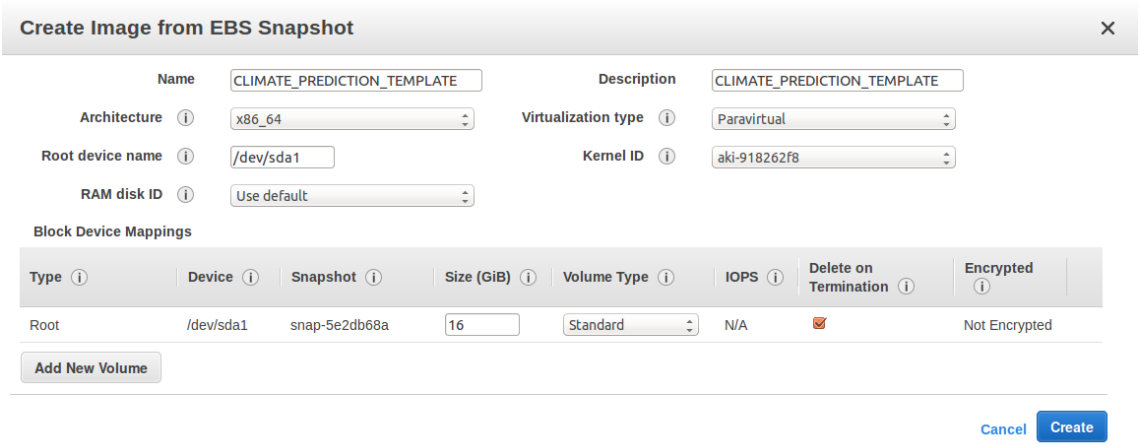
1. On the instances list (AWS Dashboard), select the Template Instance, right click and select *Create Image*, name of the image: *CLIMATE_PREDICTION_TEMPLATE*. This will create a disk Image that can be used for a full Instance Template (AMI):





2. To finally create the Instance Image (in the AWS Dashboard) go to Images→AMIs and right click on *CLIMATE_PREDICTION_TEMPLATE* and fill the parameters, at least name as *CLIMATE_PREDICTION_TEMPLATE* (the same as Image, for better identification) and match the kernel image (AKI) with the original Template Instance (currently: aki-919dcaf8). This step is very important, otherwise the new instances created for the project simulations won't boot correctly.





At this point the Computing infrastructure is ready to be deployed and scaled, this will be done trough the Dashboard described in Chapter 4.

Storage Infrastructure

Another problem to solve is the scarcity of storage space in the current infrastructure, every simulation (36000 workunits) generates ~3.6 TB of results. A solution for this can be as scalable (cloud) massive storage.

The proposed architecture is shown in Figure 3.4, where the clients send the results (tasks) to an Amazon Simple Storage Service (S3) bucket (storage endpoint). At the same time the CPDN can access this data over the Internet to run postprocessing on it (e.g. custom assimilator), this can be done by using the AWS CLI.

3.1 Upload Server

Once a client has processed a workunit the task (result) is created and sent to the defined Upload Server, that for the CPDN is `http://cpdn-upload2.oerc.ox.ac.uk/cpdn_cgi/file_upload_handler`. This needs to be done in a transparent way for the clients and without modifying the server because we don't want to affect the actual running experiments (but in the future the servers should distribute a configuration that directly points to the S3 bucket), to do this the data should be intercepted, this can be done in 2 steps/components:

- **The name resolution** should be *faked* by changing the CNAME `http://cpdn-upload2.oerc.ox.ac.uk/cpdn_cgi/file_upload_handler` point to the created S3 bucket endpoint. Bind documentation can be reviewed for this [14]
- **A web server as endpoint**, with HTTP and HTTPs support, configured to resolve the URL `http://$UPLOAD_SERVER/cpdn_cgi/` (the jobs are created to target this URL). To simplify this stage the storage provided by AWS, S3, will be used because it has a simple HTTP(s) server

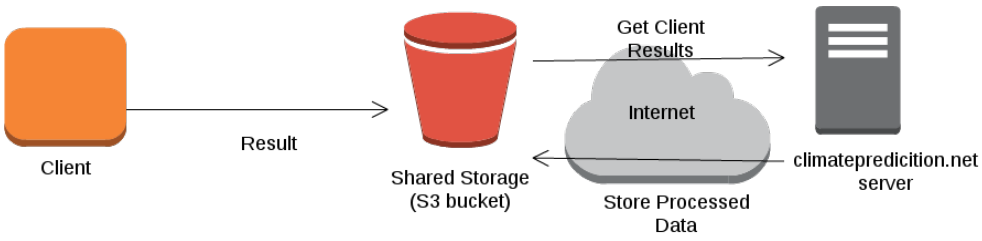


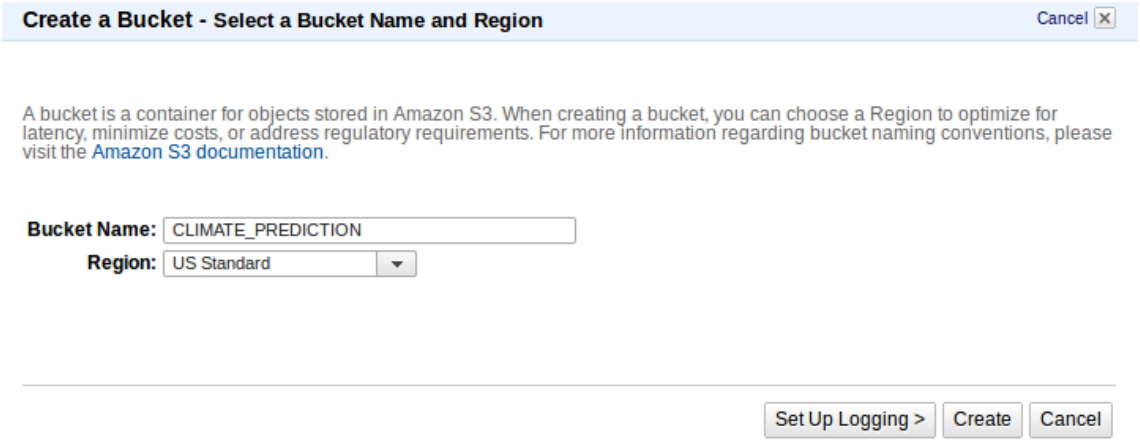
Figure 3.4: Shared Storage Architecture

that supports all the required HTTP methods (GET and POST). The (expected) content of the *file_upload_handler* must be:

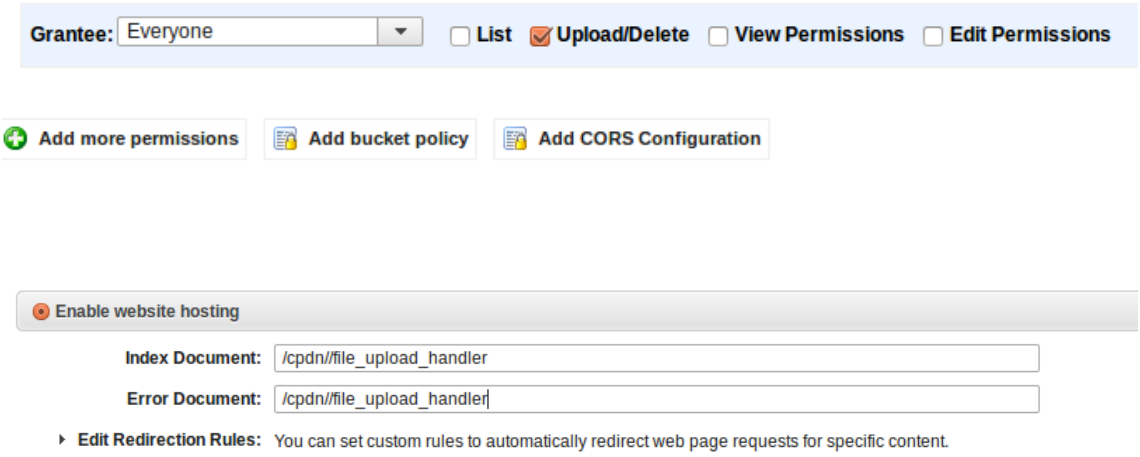
```
<data_server_reply>
    <status>1</status>
</data_server_reply>
```

1. Access to the S3 Service from the AWS dashboard.

2. Click on "Create" Bucket, the name should be *CLIMATE_PREDICTION* and must be in the same Region than the instances.



3. Activate (in the Options) the HTTP/HTTPs server.



4. To secure the bucket remember to modify the policy so only allowed IP ranges can access it (in this case only IP ranges from instances and from CPND servers).

```
{
    "Version": "2012-10-17",
    "Id": "S3PolicyId1",
    "Statement": [
        {
            "Sid": "IPAllow",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:*",
            "Resource": "arn:aws:s3:::CLIMATE_PREDICTION/*",
            "Condition" : {
                "IpAddress" : {
                    "aws:SourceIp": "ALLOWED_IP_RANGES"
                }

            }
        }
    ]
}
```

Central Control System and Dashboard

4.1 Application Architecture

The goal of the Central Control System is to give more consistency to the view of the project as an IaaS by providing a simple interface (both backend and frontend).

As shown in Figure 4.1, it has two main components:

- *Backend:* gives the user a RESTful API that gives basic functionalities: simulation information and management. It is intented to give a (even more) agnostic access to the Cloud.
- *Frontend:* makes it easier to communicate with the API in a very intuitive and simplistic way.

4.1.1 Backend and API

The backend of the Central System consists into:

- A *RESTful (Representational state transfer) API* over Flask (a Python web microframework [15]) that controls the Infrastructure (with Boto, a Python interface to Amazon Web Services [16]) that will be configured in Section 4.2.
- A *Simple Scheduler*, that will be in the background and will take care that the simulation is running with the given parameters (e.g. all the required instances are up).
- The *Reaper*, a subsystem of the Simple Scheduler that is some sort of garbage collector and will terminate powered off instances in order to release resources.

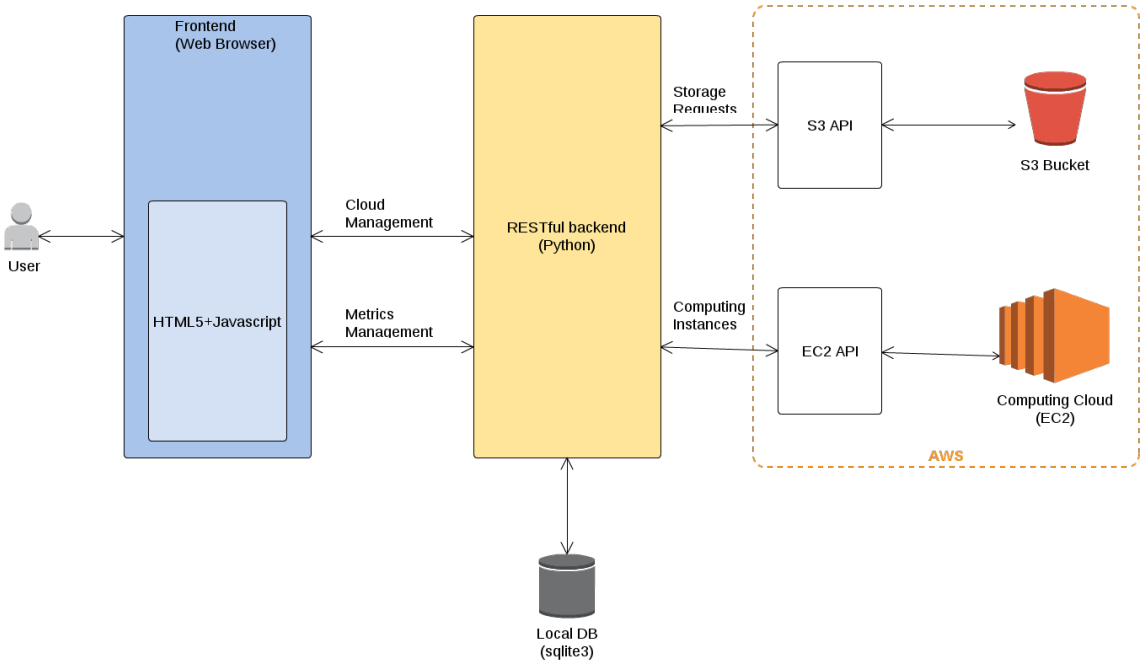


Figure 4.1: Dashboard and Metrics Application Architecture

API Reference

The backend can be resused and integrated into another systems in order to give the full abstraction over the project. The available requests (HTTP) are:

- **Get Simulation Status:**
  Call: *status*
  Request Type *GET*
  Returns: *JSON* object with current simulation status:

```
{
        requestedInstances: <int:REQUESTED_INSTANCES>
        executionCost: <float:EXECUTION_COST_IN_USD>,
        workunitCost: <float:WORKUNIT_COST_IN_USD>,
}
```

- **Get Metric:**
  Call: *metric/METRICNAME*
  Request Type *GET*
  Returns: *JSON* object with metric (time series):

```
{
        <METRIC_VARIABLE>: [{<int|float:METRIC_DATA>,
                            <timestamp:TIMESTAMP>}],
}
```

- **Set/Modify Simulation:**
  Call: *simulation*
  Request Type *POST*
  Input: *JSON* object with simulation parameters, if already running will modify it:

```
{
        instances: <int:REQUESTED_INSTANCES>
}
```

  Returns: *JSON* object with result (0=fail, 1=succesful):

```
{
        result: <0|1>
}
```

- **Stop Simulation:**
  Call: *simulation/stop*
  Request Type *GET*
  Returns: *JSON* object with result (0=fail, 1=succesful):

```
{
        result: <0|1>
}
```

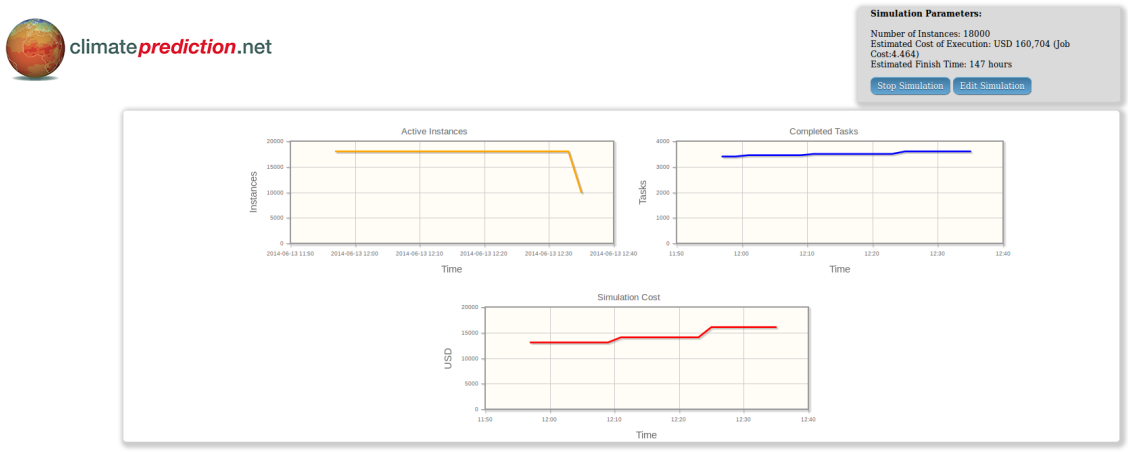


Figure 4.2: Dashboard Interface

4.1.2 Frontend

A simplistic (and functional) GUI (Graphical User Interface) has been designed to make it more understandable the execution of the simulation on the Cloud.

Two control actions are available:

- Start/Edit Simulation: sets the parameters (cloud type, number of instances...) of the simulation and runs it.
- Stop Simulation: forces all the instances to Terminate.

There are 3 default metrics (default time lapse: 6 hours):

- Active Instances: number of Active Instances.
- Completed Tasks: number of workunits successfully completed.
- Simulation Cost: accumulated Cost for the Simulation.

4.2 Installation and Configuration

The applications are intented to run at any GNU/Linux The only requirements are (apart from Python 2.7) Flask and Boto, that can be easily installed into any GNU/Linux :

```
$ pip install flask virtualenv boto pysqlite daemonize
```

First configuration and Run

For this step the file from this thesis controlSystem.tar.gz ,that contains all the software and configurations for the Central System, needs to be uncompressed into /opt/climateprediction/, then just:

```
$ cd /opt/climateprediction
$ ./firstRun.py
>> Connector Type? aws
>> AWS Key: <TYPE_YOUR_AWS_KEY>
>> AWS Secret: <TYPE_YOUR_AWS_KEY_SECRET>

>> Dashboard Username: <TYPE_DASHBOARD_USERNAME>
>> Dashboard Password: <TYPE_DASHBOARD_PASSWORD>

[OK] Central System Ready to Run! Type ./run.py to start.

#Start the service...
$ ./run.py

#... and Central System starts resolving backend and frontend requests
```

Optionally the configuration can be set manually by editing the file *Config.cfg* (parameters in <>):

```
#Main Configuration
[main]
connector=<CLOUD_CONNECTOR>
pollingTime=<REFRESH_TIME_IN_SECONDS>

[HTTPAuth]
user=<DASHBOARD_USERNAME>
password=<DASHBOARD_PASSWORD>

#AWS Credentials Configuration
[AWSCredentials]
KEY=<AWS_KEY>
PASSWORD=<AWS_SECRET>

#AWS Connector Configuration
[AWS]
AMI=<AMI>
instanceType=<INSTANCE_TYPE>
securityGroup=CLIMATE_PREDICTION
keyPair=<CLIMATE_PREDICTION_KEYPAIR>
```

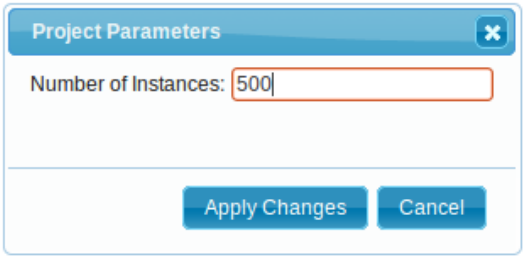
4.3 Use and Project Deployment

Now that the Central System has been installed and configured it will be listening and accepting connections into any network interface (0.0.0.0) on port 5000, protocol HTTP, so it can be accessed via web browser, Firefox or Chromium are recommended because of Javascript compatibility.

4.3.1 Launch a New Simulation

When starting a simulation the number of instances will be 0, this can be changed by clicking "Edit Simulation", set the number into the input box and

click on *Apply Changes*, within some minutes (defined in the configuration file, in the *pollingTime* variable) the system will start to deploy instances (workers).



4.3.2 Modify a Simulation

If the number of instances needs to be adjusted when a simulation is running the procedure is the same than launching a new simulation (*Edit Simulation*). **Please be aware that if the number of instances is reduced unfinished workunits will be lost** (the scheduler will stop and terminate them using a FIFO).!

4.3.3 End Current Simulation

When a simulation wants to be stopped click *Stop Simulation*. This will reduce the number of instances to 0, copy the database as *SIMULATION-<TIMESTAMP>* for further analysis and reset all the parameters and metrics.

Conclusions

5.1 Objectives Achieved

It has been successfully demonstrated that it is possible to run simulations with a climatic model using an infrastructure in the cloud, something that, while a priori does not seem to be complex, has not been tested by anyone, to the best of our knowledge.

It should be noted that the project here presented has achieved in a short period of time, not only explore the incipient possibility of successful use of high performance computing resources in the cloud for scientific computing, but which also has helped already experiments of climate research that are in the process of publication ([5],[9])

In addition, this work has served as a basis for obtaining new research projects as part of climateprediction.net, in particular a project led by one of the co-directors of the thesis has achieved in May this year the Microsoft Azure for Research Award [10], one of only about 200 projects funded by Microsoft Research, for state-of-art studies with Cloud Computing technologies. This project, is based on the demonstrated success in the application of technologies and solutions of this master final project.

Schematically, the achieved high level objectives were:

- The client side was successfully migrated to the Cloud (EC2+S3).
- Different simulations were run over the new infrastructure.
- The Control Node (and the Dashboard: frontend and backend) was developed, deployed and tested.
- A comprehensive cost of the project and the simulation was shown, as well as metrics over them.

5.2 Possible Improvements

Most of the possible improvements should focus into giving more logic to the interaction with clients status (via RPC calls) so more metrics can be pulled from them and maybe creating a new Software as a Service (SaaS layer). From the infrastructure point of view two main improvements can be done: first, a probe/dummy automated execution will be needed to adjust the price to a real one before every simulation, second fully migrate the server side into the cloud so the costs of data transfer and latency will be dramatically reduced.

Bibliography

[1] M. Allen. Do it yourself climate prediction. *Nature*, (401):642, 1999.

[2] University of California. Berkeley Open Infrastructure for Network Computing. 2014. `http://boinc.berkeley.edu/`.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] Amazon Elastic Compute Cloud. Amazon Web Services. *Retrieved November*, 2011.

[5] Añel et al. The extreme snow accumulation in the western spanish pyrenees during winter and spring 2013. *Submitted to Bull. Am. Meterol. Soc.*, 2014.

[6] climateprediction.net. Weather@home 2014: the causes of the uk winter floods. 2014. `http://www.climateprediction.net/weatherathome/weatherhome-2014/`.

[7] C. et al. Gordon. The simulation of sst, sea ice extents and ocean heat transports in a version of the hadley centre coupled model without flux adjustments. *Climate Dyn.*, (16):147–168, 2000.

[8] Pope, V. D., M. L. Gallani, P. R. Rowntree, and R. A. Stratton. The impact of new physical parametrizations in the hadley centre climate model: Hadam3. *Climate Dyn.*, (16):123–146, 2000.

[9] Schaller et al. Thee heavy precipitation event of may-june 2013 in the upper danube and elbe basins. *Submitted to Bull. Am. Meteor. Soc.*, 2014.

[10] Microsoft. Azure research awards. 2014. `http://blogs.msdn.com/b/azure_4_research/archive/2014/05/08/announcing-47-new-azure-research-awards.aspx`.

[11] Juan A. Añel. The importance of reviewing the code. *Communications of the ACM*, 5(54):10.1145/1941487.1941502., 2011.

[12] Canonical Ltd. Ubuntu. 2014.

[13] Linus Torvalds. Git: free and open source distributed version control system. 2014. `http://www.git-scm.com`.

[14] howtoforge.com. Bind installation on centos. Mar 2010. http://www.howtoforge.com/bind-installation-on-centos.

[15] Miguel Grinberg. Designing a RESTful API with Python and Flask. May 2013. http://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask.

[16] Mitch Garnaat. boto: A Python interface to Amazon Web Services. 2010. http://boto.readthedocs.org/en/latest/.