McGill University

GINI on a Cloud

Cloud computing for internet emulator

Marc Atie; David El Achkar; Simon Foucher; Mia Hochar

Supervisor: Muthucumaru Maheswaran

April 10

Abstract

GINI, an open source internet emulator, has the flaw that its backend is complicated to install and needs to run on UNIX. Our proposed improvements separate the front and back ends and have the back run on a computer cloud. This report covers the details about the current state of GINI, followed by our upgrade and our approach to implementing it. The report also provides data on performance analysis conducted on our improvements. Finally, the conclusion presents possible applications and further developments to the newly upgraded software.

Table of Contents

1. Introduction

GINI is a toolkit for creating virtual micro Internets and is primarily used for educational purposes. It was recently split into two different modules: the front end, which provides the GUI interface to create network instances and the back end, which instantiates and emulates the desired network. Our improvement, GINI on a cloud, makes use of this separation and adds cloud computing support to the emulator, using the resources of the computers of the Krieble Lab in the Lorne M. Trottier building. These computers can now host back end sessions for GINI users, which will be accessible from any Internet connection.

This report describes the approach we followed to implement this improvement. First, we will provide a more detailed description of GINI and talk about its current state. We will then describe the scripts we implemented for the communication process, and list the main improvements of the proposed addition. A brief conclusion will follow, along with recommendations and proposed directions for future work.

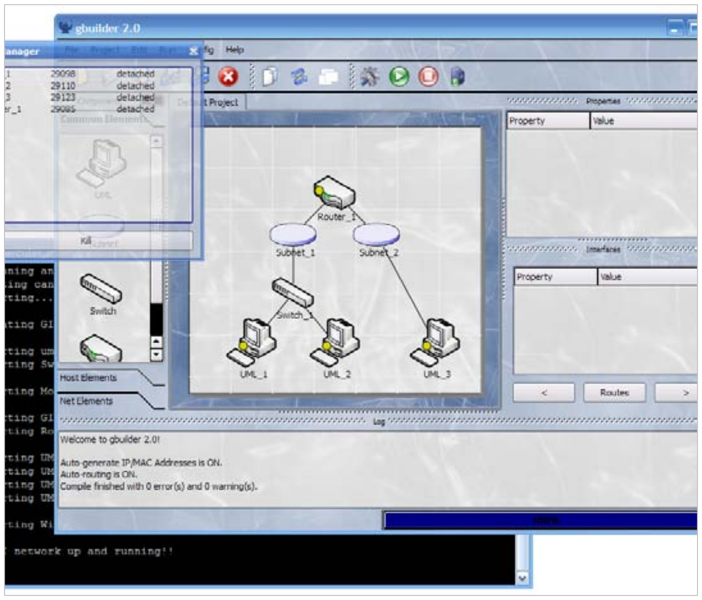
2. Current State of GINI

What is GINI?

GINI, which stands for GINI is not Internet, is a toolkit for creating virtual micro Internets.

A user can create up to moderate sized networks using GINI and can assign subnet addresses to the various components of his network without having to worry about IP and MAC addresses which are automatically assigned by GINI. Moreover, GINI is responsible for computing routing tables. Once the network is started, the process creates virtual instances of every element that is part of the user's network and creates the appropriate connections between these elements.

GINI is considered to be a viable tool for teaching and learning purposes. Computer networks students can create their own networks and observe how the different types of elements connect to each other. They can also examine how the information travels between the elements. GINI is suitable to many levels of knowledge because it provides details that can be helpful to any kind of user.



```
Welcome to gbuilder 2.0!

Auto-generate IP/MAC Addresses is ON.
Auto-routing is ON.
Compile finished with 0 error(s) and 0 warning(s).
```

```
1      29098   detached
2      29110   detached
3      29123   detached
r_1    29085   detached
```

```
network up and running!!
```

Figure 1. GINI snapshot

GINI's features

GINI's most important feature is its simplicity. It provides an easy graphical user interface (GUI) for creating network instances in order for the user to quickly adapt and master the use of this emulator. GINI is a fully open-source system. This allows users to implement new protocols or new network elements; basically, to add or remove any feature they deem necessary. Finally, GINI is a standard compliant router that works with any unmodified Linux stack.

GINI's components

GINI currently comprises six components:

- gBuilder provides the GUI interface that enables the user to create network instances. By default, all elements of a network run within the machine from where gBuilder was started; i.e. on the front end. However, gBuilder provides facilities to run the elements on a remote server.
- gLoader takes as input the topology of the network elements translated into an XML file. It launches the process that creates virtual instances of these components and of the links that connect them together.
- gRouter, as its name indicates it, is a general router. It holds all the typical functionalities of a router in a simple flexible way that can be altered by users.
- GiniLinux contains all adequate tools for a computer networking education and uses UML (User-Mode Linux) to emulate the end system.
- uSwitch is an upgraded version of uml_switch.
- WGINI is a wireless extension to GINI.

All the above information is available on the GINI project website.

3. Improvements Overview

Our improvements have separated the front end from the back end. We created a server that acts as a link between the two entities. The server is composed of three modules: the database, the dispatcher and the scheduler. As seen in Figure 3, the dispatcher communicates with the front end, the scheduler communicates with the back end, and the database acts as the meeting point where all the gathered information will be stored. The database table contains an entry for each worker. Each entry has the worker's DNS name, the IP addresses of all machines connected to that worker, the time stamp of each connection and finally the usage of every worker (registered as the uptime). Details on how this information is obtained will be explained shortly.

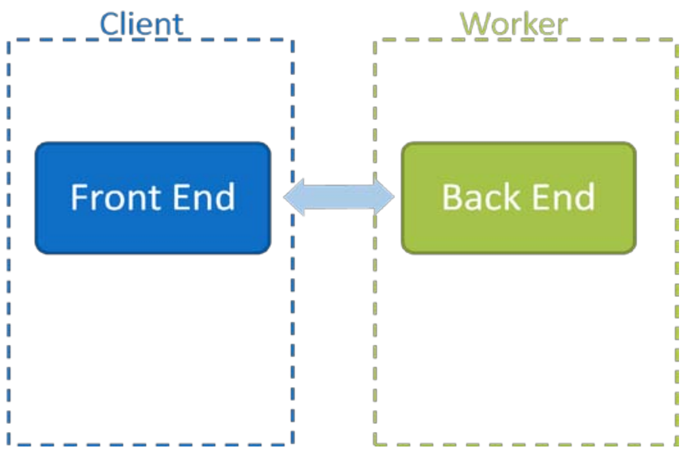


Figure 2. Current GINI Architecture

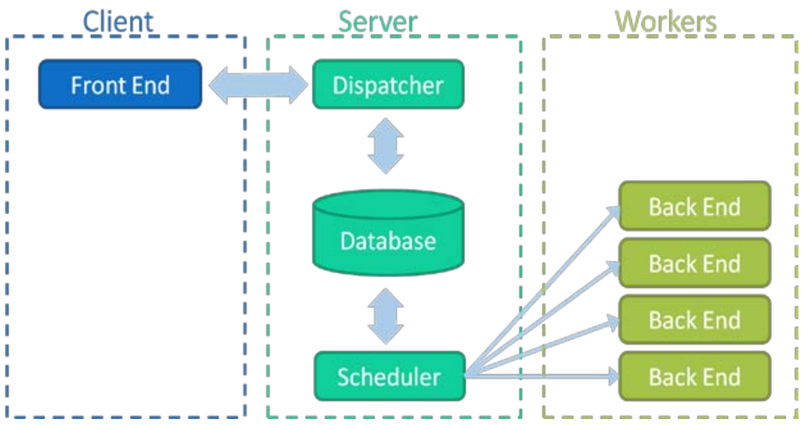


Figure 3. Proposed GINI Architecture

Resolution of GINI's drawbacks

GINI's previous state had a few disadvantages that were improved with our implementation of GINI on a Cloud. The first one is the complicated installation process that users have to go through to install GINI on a Linux machine. It is a lengthy process that requires the downloading of ten different packages.

Another drawback is that while the front end of GINI runs on Windows and Linux, its back end can only run on Linux. With our improvements the OS compatibility issues are cleared (for Windows only).

3.1.Alterations to GINI

3.1.1. Design Paradigm

One of the main design goals of this project was to minimize as much as possible alterations to the original GINI software. As such, the only file that was modified was */src/frontend/UI/Configuration.py*. All the functionalities have been implemented in an add-on component rather than evolution of GINI. This ensured that any non-trivial dependencies were left unaltered by our design. Also, if further modifications in different parts of the GINI code are to be made before it is deployed, it will be easy to add the ‘cloud’ component simply by merging a single file of code.

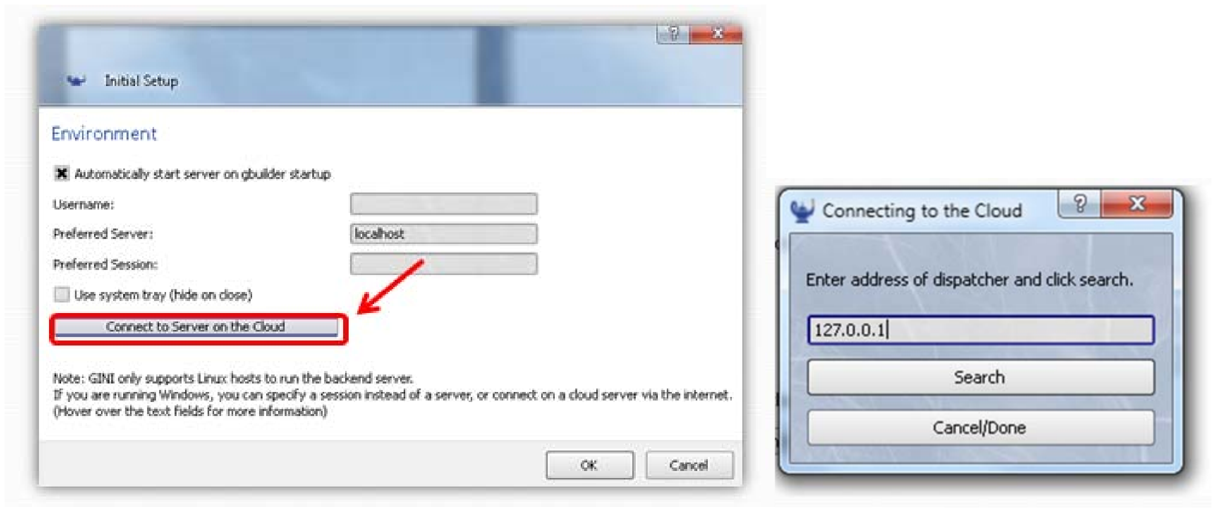
3.1.2. Networking component

The networking component has been implemented using the Python Twisted library. This choice was motivated by the fact that GINI was already built in Python, and that the Twisted library offered many easy to use high level function. The code used to link GINI with the Dispatcher was inspired by a chat server/client example presented on the twistedMatrix website. For the purpose of this project, instead of providing low level chat support, the communication functions were used to send a worker’s DNS name from the Dispatcher to GINI.

3.1.3. Additions to GINI and high level features

As mentioned above, the main design focus was to create an external component to GINI for ease of future integration. In the code, all the alterations have been centralized in the file */frontend/UI/Configurationi.py*. To make the changes active, simply download GINI, and overwrite the configuration file with the one provided with this project.

Within GINI, there are two ways to access the cloud functions: either from the initial setup wizard, or under the ‘server’ tab in the Config/Options section. In either of those places, selecting the ‘connect to cloud’ button will generate a popup window which gives access to the cloud DataBase.


Initial Setup

Environment

[✓] Automatically start server on gbuilder startup

Username:

Preferred Server: localhost

Preferred Session:

[ ] Use system tray (hide on close)

Note: GINI only supports Linux hosts to run the backend server.
If you are running Windows, you can specify a session instead of a server, or connect on a cloud server via the internet.
(Hover over the text fields for more information)

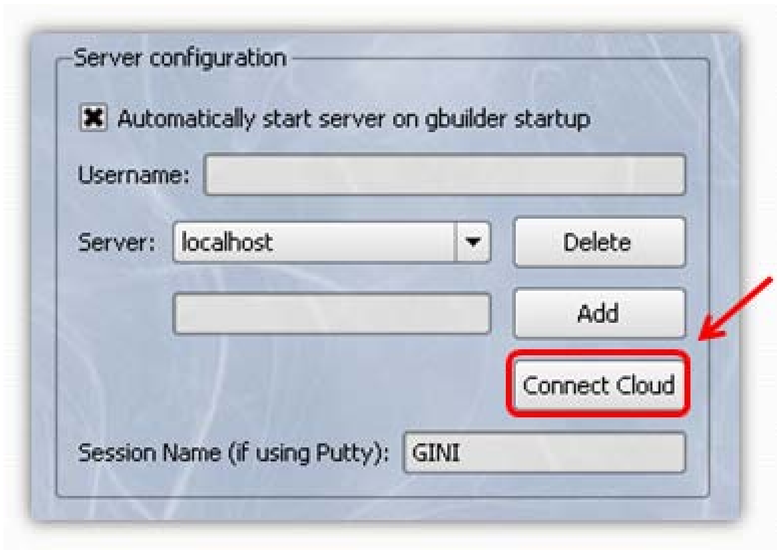
Connecting to the Cloud

Enter address of dispatcher and click search.

127.0.0.1

Figure 3. GINI Wizard Modifications

Figure 4: Access to the cloud component is provided from the initial setup Wizard (left), or from the server tab in the Config/Option dialogue (center). Pressing either of the buttons will generate a popup window with access to the cloud workforce.


Server configuration

[✓] Automatically start server on gbuilder startup

Username:

Server: localhost

Session Name (if using Putty): GINI

Figure 5. GINI Options Modifications

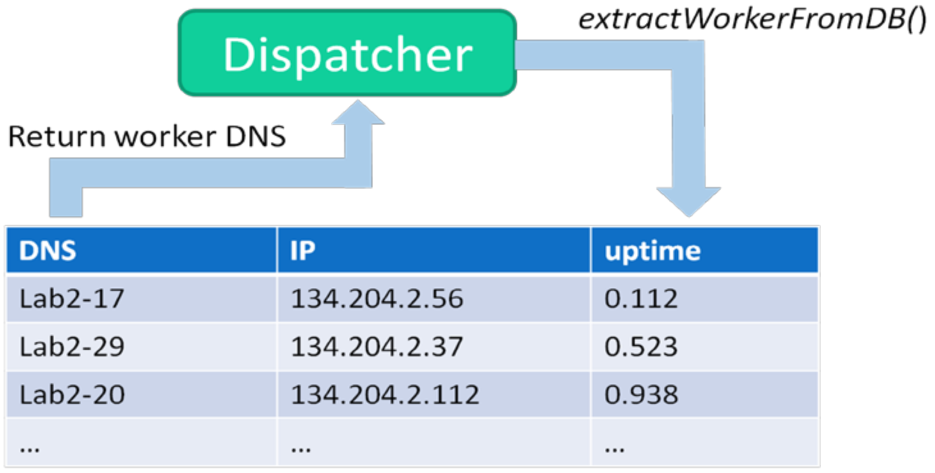
Once the FindCloudWorker object is built (the popup windows), the user has access to the cloud workers. The top text box gives the user the freedom of selecting where to connect to the Dispatcher. It was implemented as a text box during testing, and the default text value is embedded in the code and can easily be edited. In the deployment of the software, this textbox could easily be removed and replaced by a hidden static variable with the DNS name of the server hosting the Dispatcher. When the user presses the ‘search’ button, GINI sends a dummy variable to the Dispatcher and wait for a reply. In the test conducted, the reply came on average within 20ms, so the delay is negligible to the user.

If the variable returned is the same as the dummy variable sent, an error condition is recorded. Otherwise, the DNS name returned is added to the ‘server’ option field and selected as a default server. In the current implementation, the user still has the option to use this server or choose another one. In the final deployment, if the GINI backend is to run only on ‘hidden’ cloud workers, this whole operation could be migrated behind the scene and performed automatically whenever a client connects to a server. This would greatly reduce the cognitive load on the end user with regards to understanding how GINI actually works, and enable him/her to focus primarily on the simulation being performed. Before launching the server, the user will still have to enter his username to access the server. The testing has been performed using standard McGill IDs to successfully login to McGIll’s Linux machines.

Overall, the greatest advantage of this upgrade is that the user no longer needs to have any knowledge of the location and availability of Linux machines to run gServer. A user can simply fetch up-to-date information on the workforce available and proceed with launching GINI’s backend.

3.2.Dispatcher

After installing the front end on his machine, the user runs GINI; but where will the back end run? This is where the dispatcher comes in. The dispatcher is responsible for finding an available worker to run the back end. In other words, the dispatcher's main task is to allocate the available resources to all GINI connections. Whenever a new user logs on, the dispatcher goes to the database and looks for the worker best suited for the job. This means that it will choose the worker with the least CPU usage to equally allocate the work between all machines.



DNS	IP	uptime
Lab2-17	134.204.2.56	0.112
Lab2-29	134.204.2.37	0.523
Lab2-20	134.204.2.112	0.938
…	…	…

Figure 6. The dispatcher queries the database for the worker with smallest usage

Now that it has the DNS name of the appropriate worker, the dispatcher will send that information through SSH tunneling to the machine running the front end and the connection between the two ends is established.

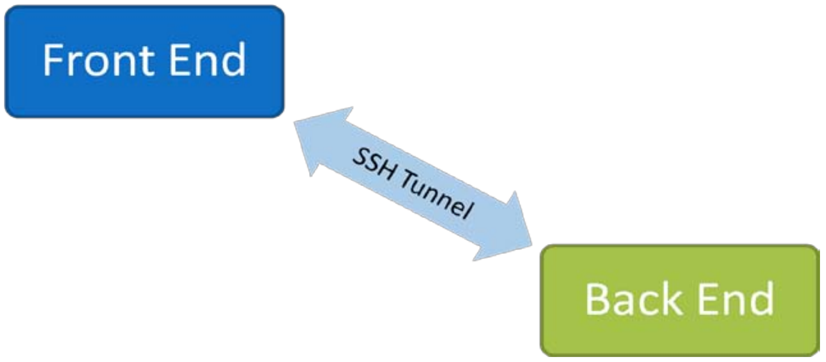


Figure 7. Connection created through SSH Tunneling

Once the connection is established, the IP address of the host will be added to the database and the dispatcher will add a time stamp to the entry. The job of the dispatcher ends here.

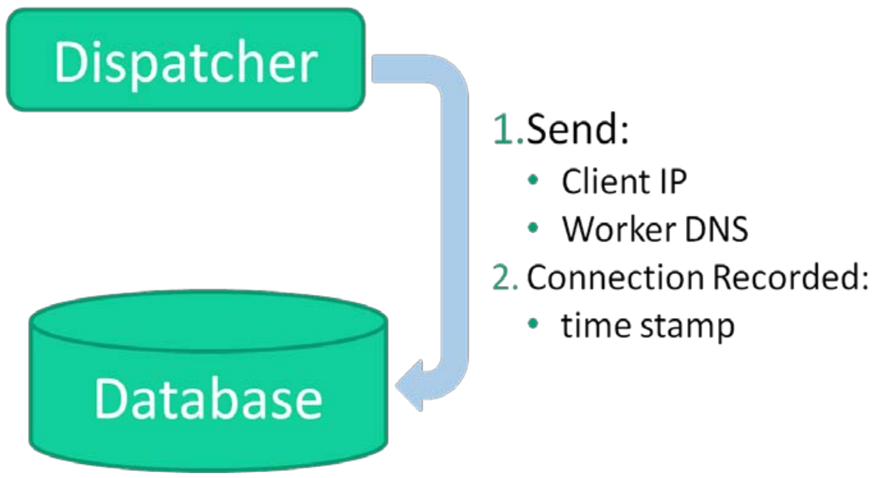


Figure 8. Dispatchers Records Each Connection

3.3.Scheduler

3.3.1. Purpose

The scheduler is the only component in our proposed improvement that communicates directly with the workers in the Krieble lab; it will relay all the required information to the dispatcher through the database. It is a Python script which should always be running on the server, and continuously polls all the workers currently in the database.

The scheduler's main function is to monitor the CPU usage of all the workers in the database. It is important to monitor the performance of the machines because we should not overload any of the workers. Therefore, each time a client requests a worker, we must find the worker with the lowest CPU usage from the database and assign it to this client.

Another function of the scheduler is to make sure all the workers are accessible and monitor the connections to these computers. While the scheduler is polling the workers, if, at any moment in time, a connection to any worker fails for some reason (if the computer crashed, or if the network connection is down), this worker will be considered as inaccessible and will immediately be removed from the database. This prevents the worker from being assigned to any further clients.

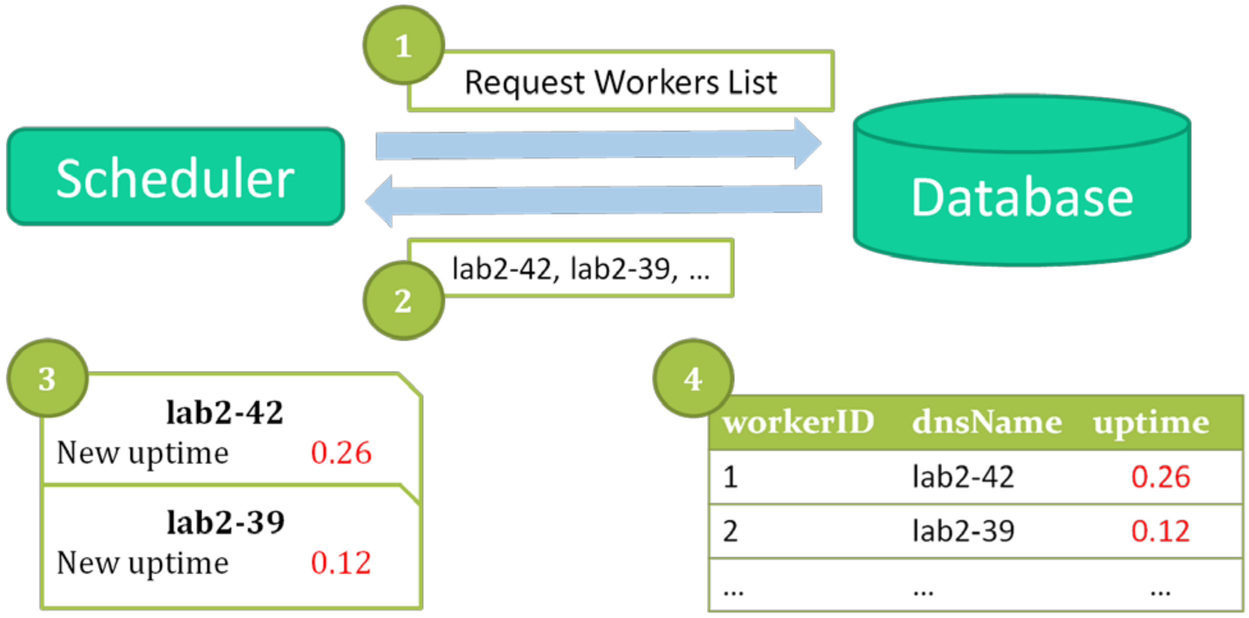


Figure 9. Scheduler Tasks Overview

3.3.2. Design Choices

There were two main design choices that we had to make while implementing the scheduler. It is important to justify these two choices before explaining the methodology.

First of all, we needed to find a way for the scheduler to communicate with all the workers. We thought a lot about what would be the best way to establish the communication between the two entities, and decided that communication through SSH would be the easiest and most efficient way to implement this; however, we ran into a lot of issues while writing the code to implement the protocol. Fortunately, we found a module online, Paramiko, which solved this problem. Paramiko is a free open-source module for Python which implements a secure SSH protocol and provides all the functions for SSH communication. With this module, the scheduler can immediately connect to a worker, send a command, read the response, and terminate the connection.

The second design choice we had to make was how to measure the CPU usage of the workers. Since Linux provides tools to monitor many different components of the CPU, there are more than 15 possible ways to monitor the CPU performance. We chose “uptime”, a command which returns the processor load average for the past 1, 5 and 15 minutes, for two main reasons: first, because the command gives a fair estimation of the actual usage of a computer; second, because it returns an average over the past few minutes. Therefore, we do not need to save each worker’s history in the database and recompute the average workload every time a database entry is updated, as we had initially planned to do; instead, we can directly store the new value for “uptime” for each worker.

3.3.3. Methodology

The scheduler runs permanently on the server in an infinite loop. At each iteration, it starts by querying the database for the list of all the workers currently available. The database responds with a list of DNS names of all workers currently stored. The scheduler then loops through all the elements in this list.

Using Paramiko, it initiates a SSH connection to each worker individually, asks for the uptime of this worker, waits for the response, then terminates the connection and updates the database entry for this worker. When all the workers in the list are updated, the scheduler goes to sleep for a minute, and start the whole process all over again.

3.3.4. Configuration

Some modifications must be made to the configuration of the scheduler when implementing GINI on a Cloud.

First of all, the scheduler needs a SOCS username and password to be able to initiate SSH connections with the workers in the lab. For our tests, we used Marc Atie’s account; however, this will need to be changed. To modify the account information, changes must be made to “Scheduler.py” at line 20.

Second, the boot sequence and shutdown sequence of all the workers must be modified. Whenever a machine is turned off, it must be removed from the list of available workers; similarly, when a machine is turned on, it should be added to the list of workers in the database. Two bash script files are provided which implement these functionalities: “addWorker.sh” and “removeWorker.sh”. Both of these scripts take one argument, which is the DNS name of the worker. Each time a computer in the Krieble lab is turned on, it must SSH into the server where the scheduler is running, and call: “$/addWorker.sh <dnsName>”. Similarly, when a computer is turned off, it must call “$/removeWorker.sh <dnsName>”. This feature makes it easy to add and remove computers from the pool of workers which will be hosting backend sessions.

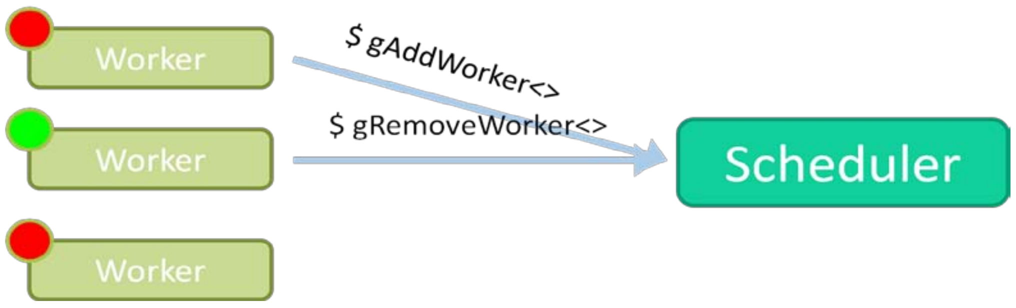


Figure 10. Communication between Workers and Scheduler

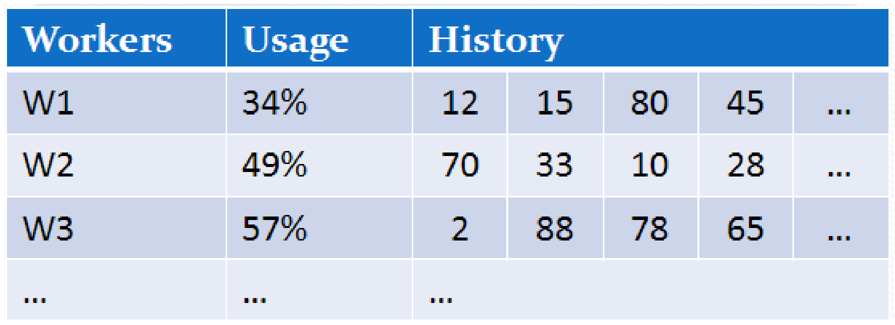
Finally, it must be ensured that there are always a few workers in the database to service incoming requests. Although this event is highly unlikely, it is possible that all the machines in the lab are shut down simultaneously. We must therefore take the appropriate precautions to always have a few computers that are turned on.

3.4.Database

The database is the link between the scheduler and the dispatcher. It holds all the necessary information about the workers and the clients connected to these workers. It consists of two tables: the workers table and the clients table. In this section, we will describe these tables, discuss the database dispatcher relationship, and explain our design choices for the database.

3.4.1. Database Contents

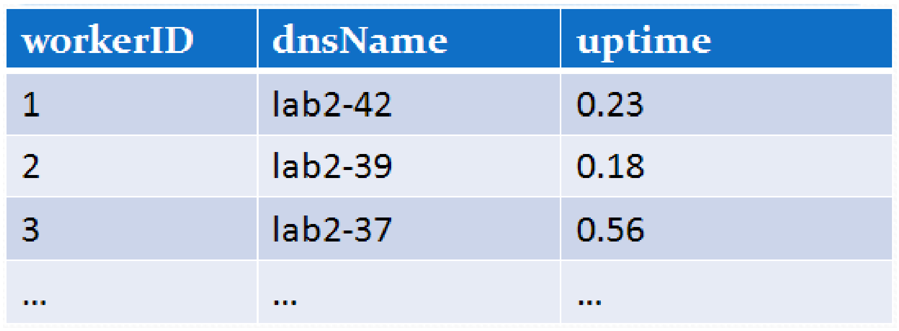
We had mentioned in our past report that the workers table would resemble the table seen in Figure 11 below. It contains the worker id, the worker's CPU usage based on a weighted average computation using the last ten entries of the usage saved in the database as the worker's history.



Workers	Usage	History				
W1	34%	12	15	80	45	...
W2	49%	70	33	10	28	...
W3	57%	2	88	78	65	...
...	...	...				

Figure 11. Initial design for the workers table

However, some changes have been made to that design. The worker's id number is still saved along with the worker's DNS name. This field is important because it is the value used for the connection of the worker with its clients. On another note, after studying the multiple possible functions that exist to compute the CPU usage, we settled on using the uptime function. It returns three values for the average load on the system's processor: one from a minute ago, one from five minutes ago, and finally one from fifteen minutes ago. In order to have a better approximation of that measure, we chose to save the value that assesses the state of the processor five minutes ago. Thus, instead of saving eleven values for the usage of the worker, we only save one and avoid having to take care of computations. Figure 12 shows the state of the worker's table in our current design.

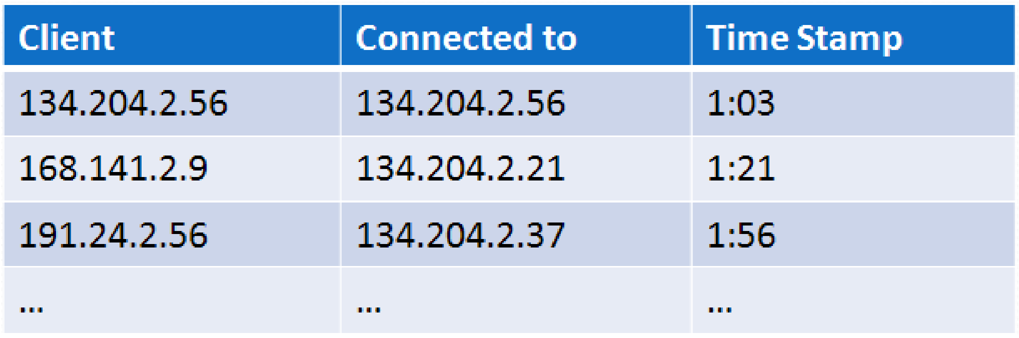


workerID	dnsName	uptime
1	lab2-42	0.23
2	lab2-39	0.18
3	lab2-37	0.56
...	...	...

Figure 12. Current design for the workers table

Every worker will host connections from different clients. The database also keeps track of all the connections that have ever been established on GINI. As mentioned earlier in the report, we have chosen not to keep connections alive for the whole time because that would create a lot of load on the worker. Since there is no way for the dispatcher to be aware of the client's IP address after the connection was established, it cannot find out whether or not the connections is still active. For this reason, the clients table holds connections that might no longer be established. We will discuss methods to handle this drawback a little further down in this report.

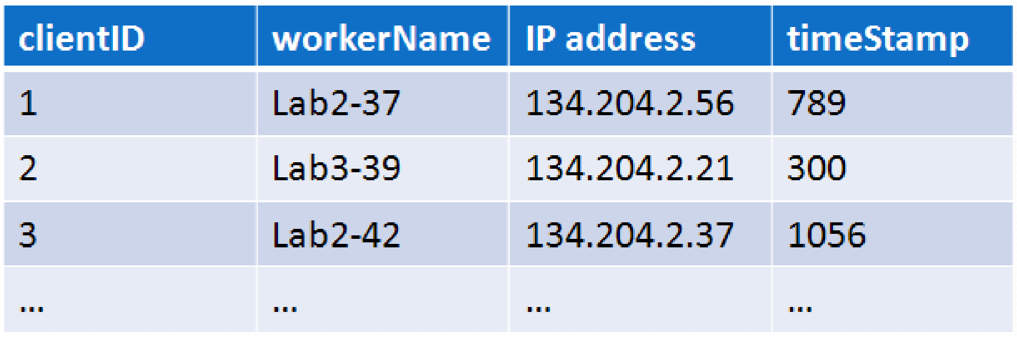
In our previous report, we described the clients table as it is seen in Figure 13. The client was denoted by its IP address, the IP address of the worker to whom the client is connected was saved, and a time stamp of the connection's establishment was the last field in that table.



Client	Connected to	Time Stamp
134.204.2.56	134.204.2.56	1:03
168.141.2.9	134.204.2.21	1:21
191.24.2.56	134.204.2.37	1:56
...	...	...

Figure 13Initial design for the clients table

Our design for the clients table has also known some changes. We notice in Figure 14 that a field for the client's id number is added. Furthermore, instead of keeping the IP address of the host, its DNS name is saved for the same reason mentioned above in the workers table. Finally, the timestamp is saved in minutes after midnight: i.e. if a connection was established at 3am, the value in the database would be 180. This method was used to simplify the implementation of a possible time out mechanism that will be explained later on.



clientID	workerName	IP address	timeStamp
1	Lab2-37	134.204.2.56	789
2	Lab3-39	134.204.2.21	300
3	Lab2-42	134.204.2.37	1056
…	…	…	…

Figure 14. Current design for the clients table

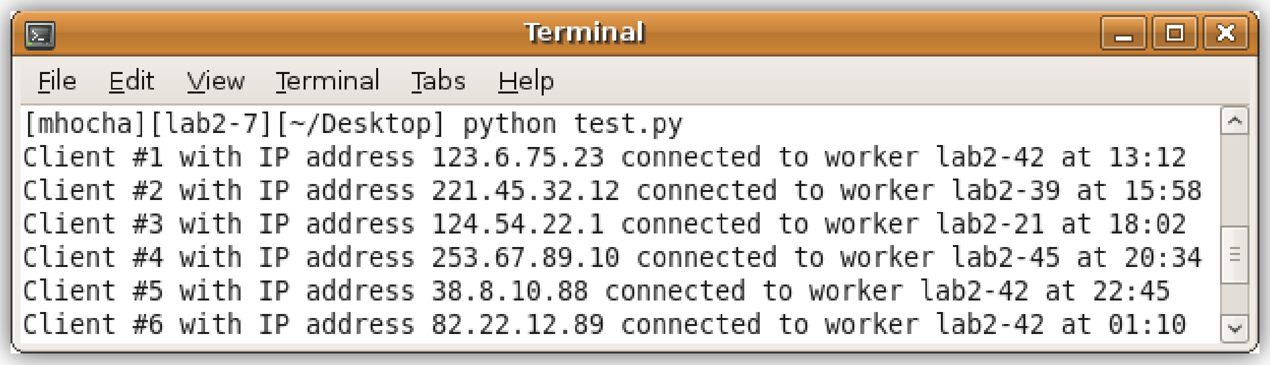
3.4.2. Database Dispatcher Relationship

The dispatcher is responsible of allocating a host for every client requesting a new connection. In order to do that, it will call the *getFreeWorker()* function in the database. This function will look through the workers table in the database and sort all existing hosts by their uptime in ascending order: from the worker with the smallest uptime to the worker with the biggest uptime. The database will return the DNS name of the first worker in the sorted list; in other words, the worker with the smallest uptime. This will allow for a fair distribution of connections among the workers.

The second part of this process is to save this request in the database log. The dispatcher will send the worker's name to the client, informing it of the host it should connect to. It will then return that value to the database in addition to the client's IP address. Finally, the database will create a new entry in the clients table; storing the IP address at the moment of the establishment of the connection as well as the host's name.

As mentioned above, all the entries in the clients table of the database remain stored since there we have not implemented a way to keep track of idle or closed connections. However, we have implemented a way for the administrator to view a log of all these connections and the ability to clear the clients table. Memory should not be an issue as storing a line of characters only necessitates a few bytes; therefore, the table can be kept untouched for years before feeling the need to clear it.

The function *displayClientsLog()* can be called to output a list of the entries of the table. A screenshot example of this can be seen in Figure 15. As shown, the time stamp is displayed in hours and minutes for the convenience of the reader. The function *clearClients()* enables the user to delete all entries of the clients table.



```
[mhocha][lab2-7][~/Desktop] python test.py
Client #1 with IP address 123.6.75.23 connected to worker lab2-42 at 13:12
Client #2 with IP address 221.45.32.12 connected to worker lab2-39 at 15:58
Client #3 with IP address 124.54.22.1 connected to worker lab2-21 at 18:02
Client #4 with IP address 253.67.89.10 connected to worker lab2-45 at 20:34
Client #5 with IP address 38.8.10.88 connected to worker lab2-42 at 22:45
Client #6 with IP address 82.22.12.89 connected to worker lab2-42 at 01:10
```

Figure 15. Current design for the clients table

3.4.3. Design choices

To implement the database, we chose SQL. The reason for that is the MySQLdb library; it is a wrapper around MySQL for the Python language and hence made the development fairly straight forward. When we were asking the SOCS helpdesk for a permission to download the library on our accounts, we were informed that McGill has already established a SQL machine and that we were allowed to have free space on it. Hence, the GINI database is stored on the machine **mysql.cs.mcgill.ca** and is called **2010GINIdb**.

Our database is initiated using the function *initDB()*. It erases all existing tables and creates new empty ones as if the database was brand new. This allows the administrator the choice to wipe the database clean if ever the need occur. Whenever the server is started, the database is set off using the *connectDB()* functions which connects to the state of the database before the server was shut down.

In our last report, we had talked about implementing a time out mechanism that would allow the dispatcher and the database to kill any passive or no longer needed connection. It would periodically poll the clients whose connection's time has exceeded a certain time out. If the client replied to the poll, the time stamp would be refreshed; otherwise, the connection would be killed and its entry removed from the database. During our implementation, we realized that doing this would cause complications: first, because we would have to change the GINI core code and variables; thus defeating the purpose of creating an add on and second because the dispatcher had no way to find the client's IP address after the connection was established.

We thought of another way to reach our end and hoped we could use the dummy variable mentioned above to send to the client its client id number from the database. That way, when the user closes the GINI window, a message would be sent to the dispatcher informing it that client X has stopped using GINI. This method was also complicated for the same reasons. Another disadvantage is that in the case of a crash, the message is not sent and the connection remains active in the eyes of the dispatcher. This would pollute the database with entries that should not be kept.

In case the administrator decides to implement a method to repair that issue, we have prepared the appropriate functions needed to access and write to the database in order to achieve this goal. They are the following:

- *getTimedOutClients()* returns a list of the IP addresses of all clients whose connections have exceeded the time out period.
- *updateClientTime(ip1, ip2, ip3, ip4)* enables the dispatcher to refresh the time stamp of the client whose IP address is ip1.ip2.ip3.ip4 in the database to the current time.
- *removeClientByIP(ip1, ip2, ip3, ip4)* allows for the removal of the entry of the client whose IP address is ip1.ip2.ip3.ip4 from the database.
- *removeClientByID(id)* allows for the removal of the entry of the client with the id sent as a parameter from the database.

4. Performance analysis

In order to conduct a performance analysis, we integrated lines of code that printed out the CPU clock in micro seconds. The first time stamps were collected on the client side, and express the delay between the moment the request is sent out, and the moment when an answer is received. It encapsulates both the network delay and the database search delay. In order to measure the delay it took to fetch data from the DB, we implemented the same 'print time' scheme on the server computer. On that machine, the first time stamp was taken as soon as the request was received, and the second one was takes just before an answer was sent back. The difference between both measurements reflects network delay.

The test were conducted on 2 computers connected on the same subnet, and due to McGill's inflexible security policies, we were not able to open TCP ports on McGill machines and establish communication via the internet at large. Therefore, the network delays as measured are at best a lower bound on what to expect. The following table outlines our findings (all numbers in microseconds).

Total Delay (µs)	DB Delay (µs)	Network Delay (µs)
28,500	1,096	27,404
15,065	898	14,167
17,696	807	16,889
18,420	912	17,508
23,695	889	22,806
12,560	935	11,625
19,323	923	18,400

Table 1. Summary of Delays Measured

5. Conclusion

Before concluding this report, there are a few recommendations we have for the further expansion and development of GINI. First, it is a known fact that the market share of Microsoft Windows is slowly decreasing as an increasing number of operating systems are penetrating the market: Mac OS market share is going up, Google is scheduled to release a new OS in the near future. To accommodate for students using these operating systems, GINI must be fully compatible with all these new platforms. Thanks to GINI on a cloud, the only work that will need to be done is making the front end compatible with the operating systems.

Finally, as GINI on a cloud has proven to be successful, it can even be taken to the next step and it can be moved to a public cloud with more resources than those offered by the computers in the Krieble lab. GINI is currently being used by students in McGill University, but the move to a public cloud could put GINI on the map and make it accessible to more students across the globe and provide them with a better educational platform