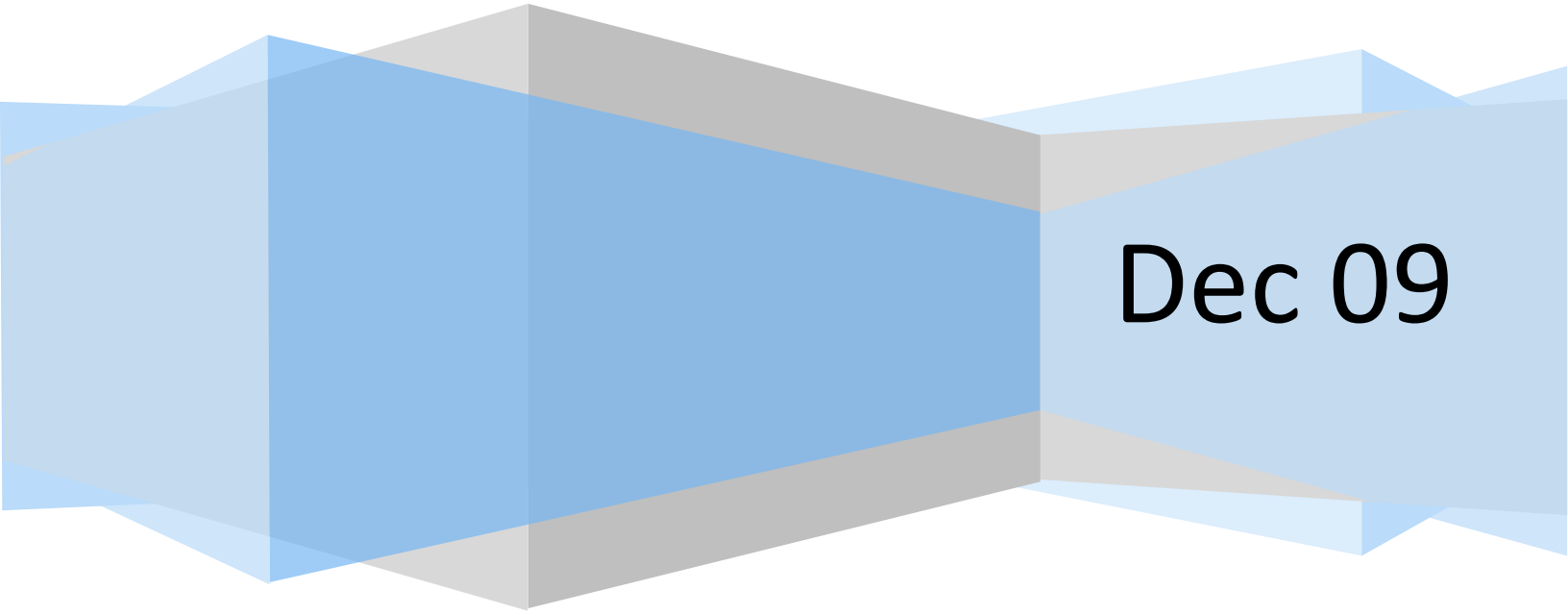
McGill University

GINI on a cloud

Cloud computing for internet emulator

Marc Atie; David El Achkar; Simon Foucher; Mia Hochar
Supervisor: Muthucumaru Maheswaran



Dec 09

Abstract

GINI, an open source internet emulator, has the flaw that its backend is complicated to install and needs to run on UNIX. We propose to separate the front and backends and have the back run on a computer cloud. This report will cover the details about the current state of GINI, followed by our proposed upgrade and our approach to implementing it. The conclusion will present possible applications and further developments to the newly upgraded software. Finally, the appendix will include the documentation of major GINI source files.

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 proposed improvement, GINI on a cloud, makes use of this separation to add cloud computing support to the emulator, using the resources of the computers of the Krieble Lab in the Lorne M. Trottier building. These computers will host back end sessions for GINI users, which will be accessible from any Internet connection.

To make this possible, we will first need to review the source code for GINI to make sure that the two entities are completely independent, and then implement a communication mechanism between the two components.

This report describes the approach we propose to follow to solve the problem. First, we will provide a more detailed description of GINI and talk about its current state. We will then describe the scripts we will implement 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.

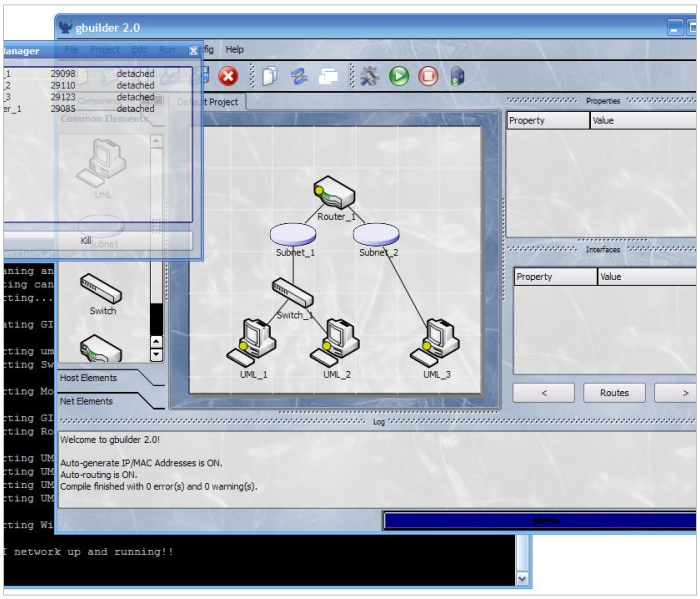


Figure 1. GINI snapshot

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.

GINI's drawbacks

The state of GINI today has a few cons that we are hoping to improve 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. This creates OS compatibility issues and is an issue to consider because of the current market penetration of new operating systems such as Mac OS. Finally, GINI also knows a network protocol compatibility issue because its router is not compliant with those of Cisco Systems. This is an issue that we could address in the future if time allows us.

3. Proposed Improvements

In order to address all the issues discussed in the previous section, we propose to definitively separate the front end from the back end. We will create a server that will act as a link between the two entities. The server will be 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 will act as the meeting point where all the gathered information will be stored. The database table will contain an entry for each worker. Each entry will have the worker's DNS name and IP address, the IP addresses of all machines connected to that worker, the time stamp of each connection and finally the usage of every worker. Details on how this information is obtained will be explained shortly.

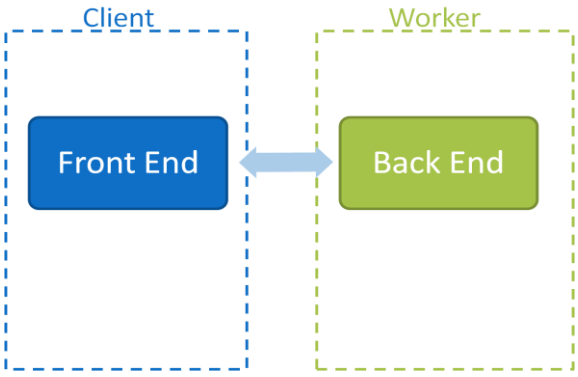


Figure 2. Current GINI Architecture

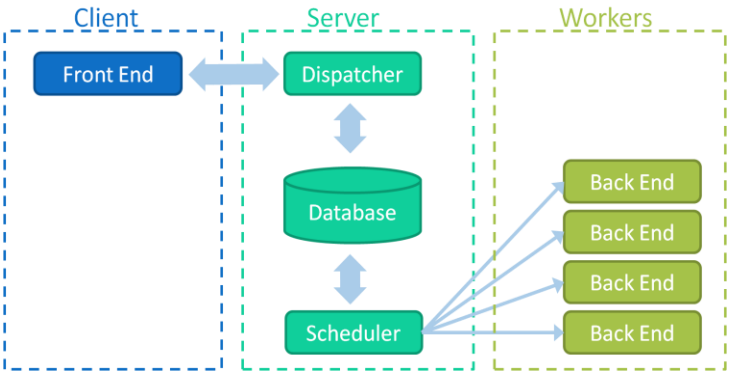
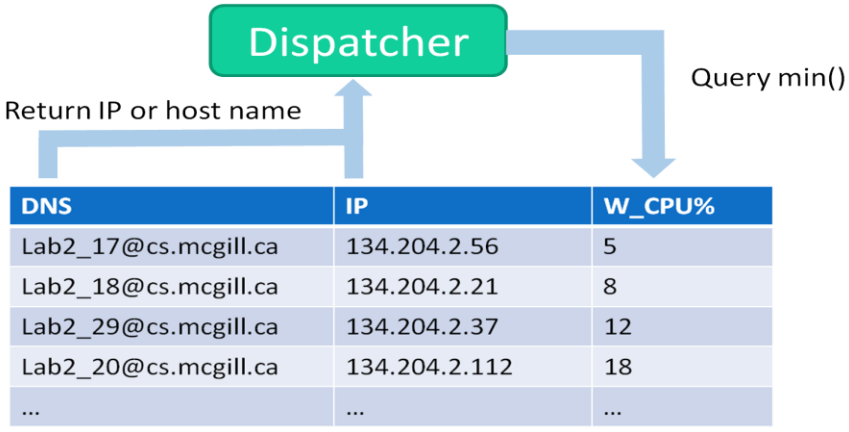


Figure 3. Proposed GINI Architecture

3.1. 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 will go to the database and look for the worker best suited for the job. This means that it will choose the worker with the least usage to equally allocate the work between all machines.



DNS	IP	W_CPU%
Lab2_17@cs.mcgill.ca	134.204.2.56	5
Lab2_18@cs.mcgill.ca	134.204.2.21	8
Lab2_29@cs.mcgill.ca	134.204.2.37	12
Lab2_20@cs.mcgill.ca	134.204.2.112	18
...	...	...

Figure 4. The dispatcher queries the database for the worker with smallest usage

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

3.2.Scheduler

The scheduler or resource manager is the server's third and final entity. Its main purposes are to manage the worker's connections and to examine their CPU usage to be sure that the dispatcher correctly allocates resources with overloading a certain machine.

Obtaining the worker's information

The scheduler will consecutively visit every worker and pipe the data on its usage using the $PS command. Once the list of all processes running on the worker is obtained, it will extract its current CPU usage. For every worker entry, the database keeps a history of the ten last CPU usages fed by the scheduler. This is the case because the read CPU usage at a certain time is not necessarily representative of the actual state of the worker. The scheduler could be catching the worker at a bad moment. The actual usage seen in the second column of Figure 5 is a weighted average computed with the data in the history.

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

Figure 5. Example of worker usage history in the database

Updating the database

The resource manager updates the database to enter the newly gathered information of every worker. This operation is done in four steps:

1. Query database for worker entry using the worker's DNS name or IP address.
2. Extract previous ten CPU usages of the worker in question.
3. Compute new weighted average with more weight on the most recent usages to better describe the actual state of the worker.
4. Store new weighted average in the database.

Time Out

As mentioned above, the scheduler manages the worker's connections. An important factor to the performance of any software is the optimization of resources; i.e. prohibiting waste or in our case idle connections. We have briefly stated that the dispatcher adds the time of creation of every connection to its entry in the database. Now, we can add that whenever the scheduler

updates the database, it will examine these time stamps and confirm that the time elapsed since the creation of the connection has not exceeded a certain value. This value called the time out value is manually inserted by the workers' administrator. If, for any certain connection, the elapsed time is greater than the time out value, the scheduler prompts the user with a "Are you still there?" message that will appear as a pop-up window on the screen.

If the user is still using the connection, the time stamp will be refreshed in the database. If, after a certain amount of time, the user has not replied, the connection will automatically be killed to make room for new connections and its entry in the database will be removed.

3.3.User Interface

The scheduler process will run in the background of the server. In order for the network administrator to be able to configure or communicate with the scheduler, we are going to update the $PATH file on the server and a few commands to allow for exchanges with the resource manager. A few commands that we have thought of are:

- $ gtimeout <timeInMinutes> to set the time out value.
- $ gkill <ipAddress> to manually kill a connection using the host's IP address.
- $ gaddworker <ipAddress> <name> to manually add a worker.
- $ gremoveworker <ipAddress> <name> to manually remove a worker.
- $ showWorkers to show a list of all workers.
- $ showActiveSessions to show a list of all active sessions.

Dynamic Update

Finally, it has come to our attention that the machines of the Krieble lab are shut down every night. This means that somebody would have to manually add every worker when the machines are turned on and to manually remove every worker when they are shut down.

To address this issue, we are going to update the boot and shutdown sequences of these machines to allow for dynamic update. This means that whenever a machine is turned one, it will automatically send a $gaddWorker command to the scheduler in the server through SSH tunneling as seen in Figure 6 below. The scheduler will then add an entry for that worker in the database. Similarly, when a machine shuts down, it will automatically send a $gremoveWorker command to the scheduler to remove its entry from the database and inform the dispatcher that this worker has logged off the network.

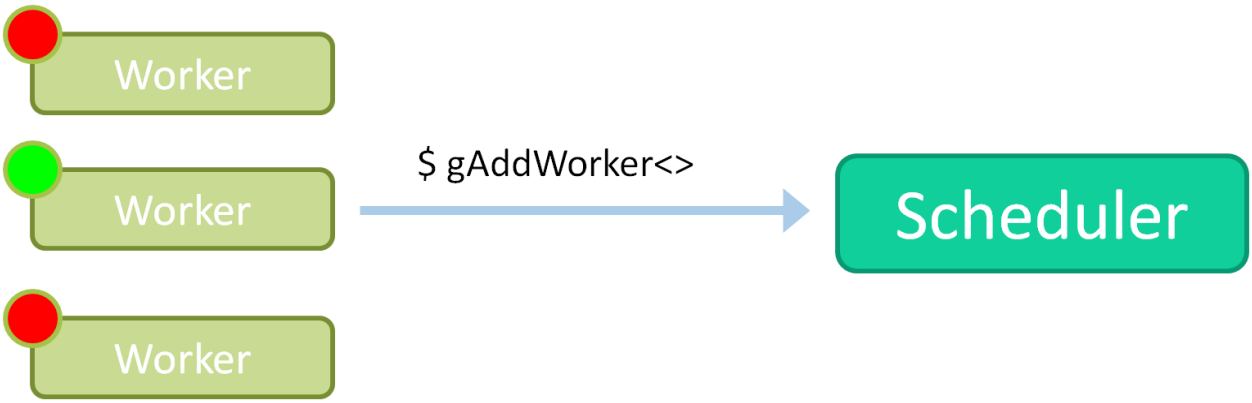


Figure 6.Worker informs scheduler that it has logged on to the network

3.4.Advantages

Until the summer of 2009, GINI was not split into front end and back end and the program had to run on one machine. The separation of the software into these two components can be used to fix many of GINI's drawbacks.

First of all, installation of GINI on a Linux machine is a very tedious process as it requires installing about 10 different packages. Installing the front end on a Windows platform is, however, very straightforward. Since GINI on a cloud would require users to have only the front end installed on their machine, the installation process is made a lot easier. Users also wouldn't need to worry about software upgrade and maintenance of the back end: once the developers update the back end on the workers, users will automatically connect to the upgraded software.
Another important advantage of the separation of back end and front end is cross-platform compatibility: the back end of GINI can only run on a Linux machine, so students with Windows machines could not run GINI from their computers. With GINI on a cloud, all these students would need to do is to install the front end of GINI and connect to a back end running on a computer in the Krieble lab. An added bonus with this feature is mobility: since the back end runs on a remote server, users can connect to a back end session from anywhere as long as they are connected to the Internet. This is very important for students since they often study in different places: on campus, at home, in a coffee shop, etc…

Finally, we must talk about the important role of the dispatcher and the scheduler. Right now, theoretically, any user can immediately connect through a SSH tunnel to a computer in the Krieble lab and run GINI without contacting the dispatcher. There are, however, a few problems with this approach: first, the user would need to know the exact address of a worker; second,

computers in the Krieble lab are often turned off, so a connection to this particular worker might not be possible! In this case, the user would have to try to SSH to all the computers in the lab until he finds one that is turned on. The final problem with this approach is the lack of resource management: we could end up with 10 different users connected to the same worker, using up all of the resources of the computer, even if there are other idle computers in the lab. The dispatcher and scheduler are always aware of exactly which workers are available and they make sure that the workload is fairly distributed among all these computers.

4. Documentation of the Code

4.1.gBuilder

gbuilder is responsible for the front end of GINI. It is divided into four main components: the Core, the Network, the Devices, and the UI.

/src/frontend/gbuilder.py

`gbuilder.py` is the main file of the gbuilder. It first checks that all the libraries are installed on the machine, then seeds the random generator and opens the main window of GINI by creating an instance of `MainWindow.py`.

/src/frontend/UI/MainWindow.py

This script file is divided into 2 main classes: 'MainWindow(Systray)' and 'DebugWindow(QtGui, Qwidget)'.

For an outline of the function and their purposes please refer to *Appendix A*.

/src/frontend/UI/Configuration.py

This script manages the configuration of the software. As we are going to add functionality, we will need to make some changes to this configuration script to add our new features. Here is a brief documentation of the file as it currently is. In order to add our backend, we are going to be mostly adding to this file. Once again, the most relevant parts for our project have ben outlined in bold.

For an outline of the functions and their purposes please refer to *Appendix B*.

/src/frontend/network/Network.py

The `Network.py` file is responsible for the communication between the front end and the back end. It initializes the connection between the two components, and defines the methods for communicating user input from the user interface to the back end session.

/src/frontend/gbuilder/core/Compiler.py

`Compiler.py` is mainly responsible for compiling all the devices in the network, computing their routing tables, and validating their properties and addresses.

/src/frontend/Devices/

This subfolder contains class descriptions for all the devices supported by GINI, namely: Bridge, Firewall, Hub, Mobile, Router, Subnet, Switch, UML and Wireless access point.

4.2.gLoader

/src/backend/gLoader/

This file is the core of the backend. It is in charge of creating a GINI instance and then terminating it once the user is done. It can activate all the components of GINI (switches, routers, UMLs, wireless access points). Similarly, it can terminate them and check their status. `GLoader.py` is also in charge of creating config files for these components.

5. Steps to implementation

In order to implement our upgrade, we will need to perform the following:

Step 1: Pre- development

- Install the GINI back end on some LINUX worker computers (at least the MAC lab on Trottier 3rd Floor)
- Designate a 'server' machine (Running LINUX), either with a static IP or a DNS record such that it can be remotely accessed via Internet (set up in the deployment phase)

Step 2: Development

- We will need at least 2 machines to develop the server/client communication; one to run the front end of GINI, and one to run the server. During testing, we can use an arbitrary LINUX machine as a server as long as it is connected on the internet, has a known IP, and can open TCP connection on a given port
- We will build on the server, a network interface (the Dispatcher), a scheduling program (the scheduler) and an implicit communication interface and a data store (SQL database)

Step 3: Deployment

- Once all the components built and integrated, we will need to install the final server software on a dedicated LINUX server
- We will need to update the boot and shut down sequences of the workers such that they dynamically and autonomously add/remove themselves from the worker's database on the server.

6. 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 would need to be done is making the front end compatible with the operating systems.

Finally, if GINI on a cloud proves 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.

Appendix A *MainWindow.py*

Here is an outline of the functions and their purpose (bold items are the ones that are of interest for this upgrade and are more thoroughly documented). More specifically, the function **startServer** will have to be altered to accommodate our modifications. We will include a check to see if the user wishes to connect to the cloud, in which case, it will first have to contact the Dispatcher on the cloud server and wait for the IP of an available worker. Once this IP is received, it will be integrated as the server associated with this client, and the connection function will proceed normally.

Class MainWindow(Systray):

- __init__(self, app): Create a main window for the given application.
- center(self): Center the window.
- startTutorial(self): Start the interactive tutorial.
- lockDocks(self): Lock the dock windows so they cannot be moved, closed or resized.
- unlockDocks(self): Unlock the dock windows.
- faq(self): Open the FAQ in the default browser.
- closeTopology(self): Close the current topology.
    1. Check if the topology is running. If so, return error
    2. Call QtGui.QMessageBox.warning with 'Save before Quit?' msg. If QtGui.QMessageBox.Yes, call the function self.saveTopology()
    3. Remove all the nodes in topology memory and call shutdown functions:
        - self.properties.clear()
        - self.interfaces.clear()
        - self.routes.clear()
    4. Returns TRUE is successful, FALSE if not
- **sendFile(self):** Start a process to select and send a file to the server.
    1. Check to see if there is a server already running by calling:
        - self.server: this computer is also server
        - self.server.poll(): returns the servers or None if none running
    2. Make sure we are in a client state by checking 2 functions: self.client, to ensure twe are in a client state, and self.client.isConnected(), to ensure that we are in a connected state. If we are not in a client state, initiate a client state by calling: self.startClient()
    3. Finally send the file by calling self.sendWindow.setFilename(filename) and self.sendWindow.show()
- newScene(self): Close the current topology and create a new one.
- expandScene(self): Expand the scene based on number of expansions.
- newProject(self): Create a new project for device sharing.
- openProject(self): Load an existing project for device sharing by calling loadProject()
- **loadProject(self):** Load project file data into options. As the options are loaded in a while loop, we should not have to modify this function; only the option file.
- closeProject(self): Close the current project.

- def export(self): Open an export window to generate an image from the canvas by calling self.exportWindow.show()
- startBackend(self): Start the backend server by calling self.startServer()
- setRecovery(self, recovery): Set the recovering state of the topology by setting self.recovery = recovery
- isRunning(self): Returns whether a topology is running or not.
- **startServer(self):** Start the server backend of gbuilder, which controls running topologies. Most of the parameters used here are updated in /frontend/UI/Configuration.py
    1. Check to see if a server instance is already running, either locally (self.server), or remotely (self.server.poll).
    2. Create the following strings containing commands. The words in quotation marks have values assigned in /src/frontedn/UI/Configuration.py. These values are stored in the array option["stringIndex"]. The words without quotation marks are just plain text.
        - *base* = [ssh –t "username " @ "server "]
        - *tunnel* = [-L "localPort" :localhost: "remotePort"]
        - *server* = [bash -c -i 'gserver "remotePort" ' || sleep 5]
    3. Build the actual command that will be sent via ssh to the server. This command is stored in the string 'command' and have many versions, based on which OS the client is using and weather the server is local or remote. If using Windows, a putty session can be used instead of a server.
        - Windows w/ session: command = [putty - load "session" –l "username" -t
        - Windows w/o session: command = [*base tunnel* -m " + startpath\]
        - LINUX: command [xterm -T gserver -e *base tunnel* \server\
    4. A subprocess is created by sending 'command' via ssh (or putty) to the server (whether it be local or remote). The process is recorded in 'self.server'
- **startClient(self):** Start the client of gbuilder, which communicates with the server. This function needs not to be modified, but the client function called by this function might be.
- compile(self): Compile the current topology.
- run(self): Run the current topology.
- stop(self): Stop running the current running topology.
- stopped(self): Handle a fully stopped topology.
- loadFile(self, filetype): Load a file through a file dialog.
- loadTopology(self): Loads a topology.
- saveFile(self, filetype): Save a file through a file dialog
- saveTopologyAs(self): Save a topology under a given filename.
- saveTopology(self): Save a topology.
- copy(self): Copy selected text from the log into the paste buffer.
- config(self): Open the options window.
- arrange(self): Rearrange the topology based on the distance between nodes.
- about(self): Show the about window.

- **createActions(self):** Create the actions used in the menus and toolbars. This will need to be modified in order to add a 'log on the cloud' option
- **createMenus(self):** Create the menus with actions. We will also need to include the menu option to run the action of logging on to the cloud
- **createPopupMenu(self):** Customize the popup menu so that it is visible. We will have to call this function whenever the scheduler sends an 'are you still there' message to users.
- createToolBars(self): Create the toolbars with actions.
- createStatusBar(self): Create the status bar.
- createProgressBar(self): Create the progress bar.
- getDeviceCount(self, alive=False): Return the interfaceable device count, or the alive ones if alive id TRUE.
- pdateProgressBar(self): Update the progress bar.
- createConfigWindows(self): Create the options window.
- createDockWindows(self): Create the dock windows (dropbar, log, properties, interfaces, routes.)
- createPopupWindows(self): Create the different popup windows.
- keyPressEvent(self, event): Handle specific shortcut keys.

Appendix B *Configuration.py*

class LineEdit(QtGui.QLineEdit):

- __init__(self, text=QtCore.QString()): Create a custom LineEdit so that the context menu is visible.
- contextMenuEvent(self, event): Customize the context menu so that it is visible

Class ServerPage(QtGui.QWidget):

- __init__(self, parent=None): Create a server configuration page. Most of the elements entered in this configuration dialogue will be called when startServer loads up. Here are some of the elements to be configured here:
    - Offers the option to automatically start the server on GINI startup
    - Enter user name
    - Enter server name
    - Add/Remove servers
    - Session name for putty
    - Ssh port configuration (local and remote, gives the option to randomize the port)
- addServer(self): Add a server to the list and write it to file.
- delServer(self): Delete a server from the list.
- randomizeLocalPort(self): Randomize local port field stored into self.localPortLine to a random port between 1024 and 65535.
- randomizeRemotePort(self): Randomize remote port field stored into self.remotePortLine to a random port from (1024, 65535)
    - saveOptions(self): Save options handled by this page in the option[] array, indexed by a string. More specifically, the following items are saved: options["autoconnect", "username", "server", "session", "localPort", "remotePort"]
- updateSettings(self): Update the page with current options. (same options as outlined in the previous bullet point)

Class UpdatePage(QtGui.QWidget): Has a single __init__ def that creates an update page.

Class GeneralPage(QtGui.QWidget): Upon __init__ call, this class creates a general configuration page. The configuration achievable here is more related to esthetics (backgrounds, colors, themes, etc…) than functionality.

class ConfigDialog(QtGui.QDialog):

- __init__(self, parent=None): Create a configuration dialog window.
- loadOptions(self): Loads the options from the configuration file into the local variables.
- changePage(self, current, previous): Handle a page change.
- createIcons(self): Create the icons for the different pages.
- hideEvent(self, event): Handle closing the configuration window.
- updateSettings(self): Update all pages with the current options.

Wizard(QtGui.QWizard): This class handles the creation of a first time configuration wizard.

Appendix C *gLoader.py*

- startGINI(myGINI, options): Start the instantiation process of GINI components. The order is important: First switches, then routers, and finally UMLs
- createVS(switches, switchDir): Creates the Swith config file and start the switches.
- createVWR(myGINI, options): Starts the Wireless access point.
- createVR(myGINI, options): create router config file, and start the router.
- createVM(myGINI, options): create UML config file, and start the UML. it creates one config file for each interface in the /tmp/ directory
- makeDir(dirName): create a directory if it doesn't exist
  Socket name is defined for the switch as *switchDir/switchName/SOCKET_NAME.ctl*
  and for the router as *routerDir/routerName/SOCKET_NAME_interfaceName.ctl*
- getSocketName(nwIf, name, myGINI, options): Get the socket name the interface is connecting to
- getVRIFOutLine(nwIf, socketName): convert the router network interface specs into a string.
- getVMIFOutLine(nwIf, socketName, name): convert the UML network interface specs into a string. Also checks whether somebody else using the same MAC address.
- destroyGINI(myGINI, options): Terminates the GINI instance by consecutively terminating the switches, routers, wireless access points, mobiles and UMLs.
- destroyVS(switches, switchDir): Stops the switch.
- destroyVWR(wrouters, routerDir): Stops the wrouter.
- destroyVR(routers, routerDir): Stops the router.
- destroyVM(umls, umlDir, mode): stops the UML instance.
- checkProcAlive(procName): grep the UML process .
- writeSrcFile(options): write the configuration in the setup file.
- deleteSrcFile():delete the setup file.
- checkAliveGini():check if any of the GINI components are already running.