# Alterations to GINI

## 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.

## 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.

## 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. As such, 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.

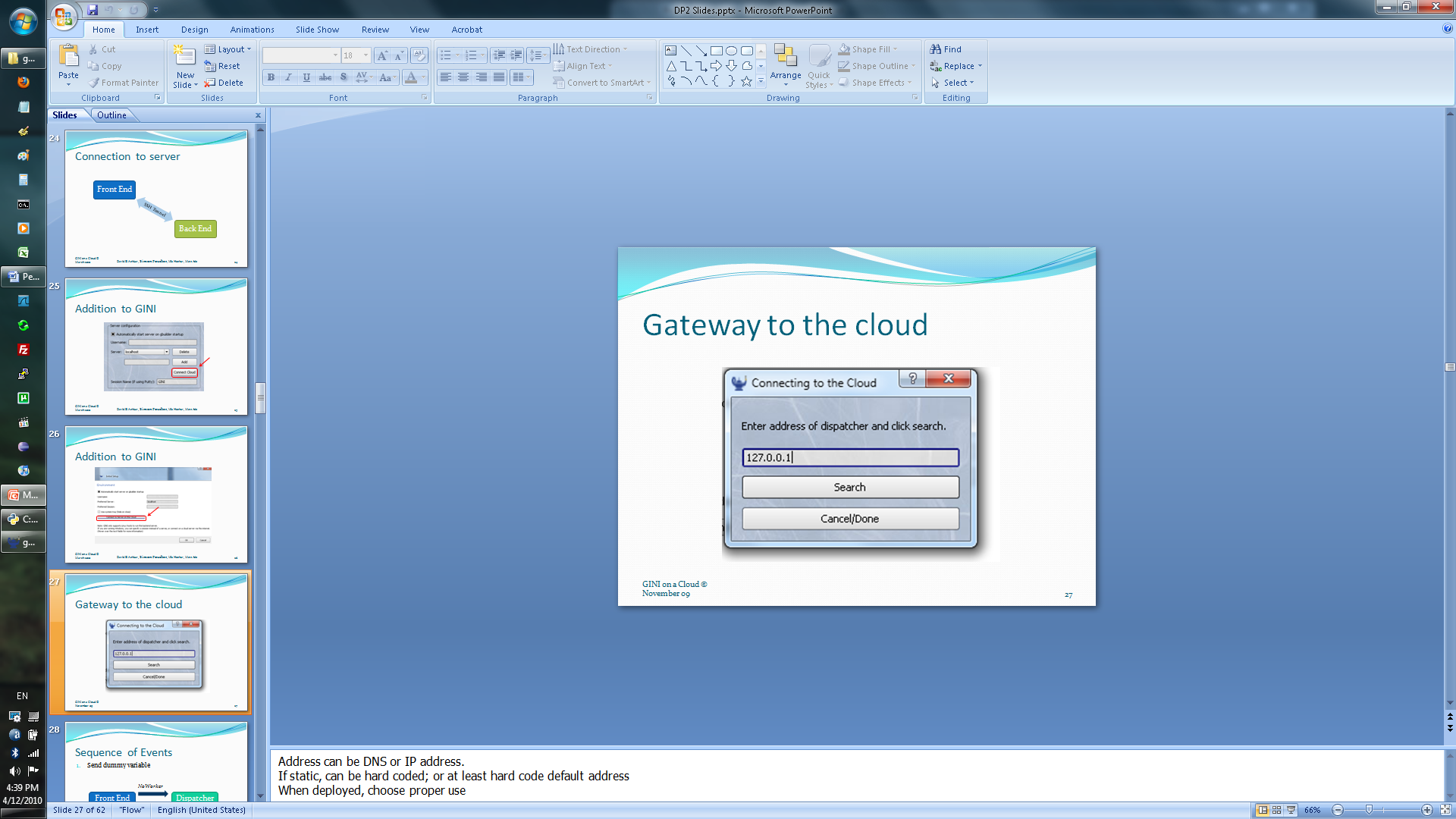
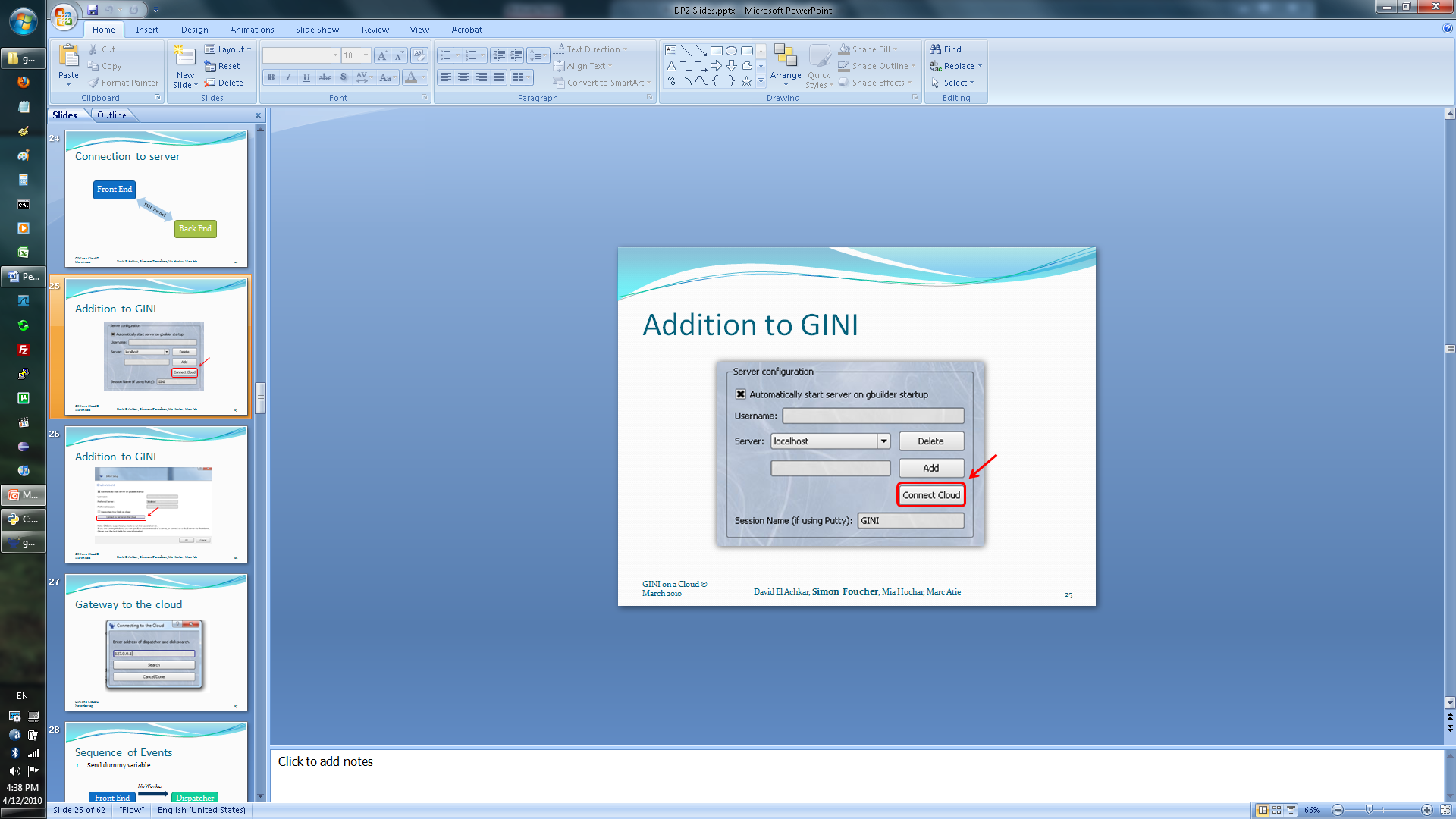
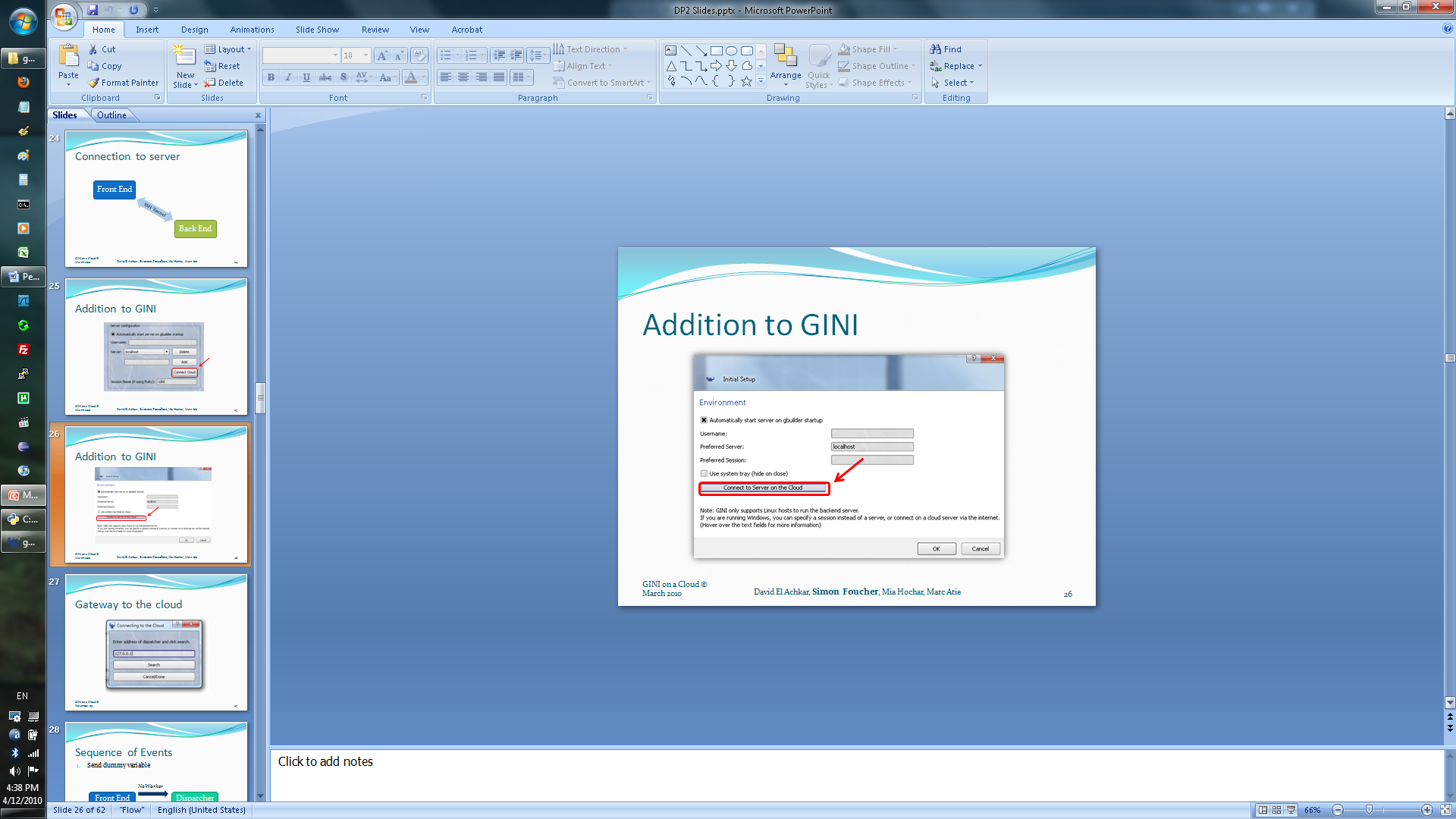


Figure xxxxxxxxxxxx: Access to the cloud component is provided from the initial setup Wizard (left), or from the server tab in the Config/Option dialogue (center). Pressing either of the buttons will generate a popup window with access to the cloud workforce.

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.

# 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 | DB Delay | Network Delay |
| 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. The Total delay was measured on the client machine, the DB delay on the Server and the network delay was extrapolated by subtracting the DB delay to the total delay. The bottom row is an average of the numbers above.

We can observe that the DB delay is extremely small compare dot the network delay and in the order of a millisecond. The database tested had only 6 entries, but considering the fact that the delay it causes is almost 20 times smaller than the best case network delay, we don’t anticipate that an increase in the size of the data store would cause a significant increase in the total delay.

The network delay is clearly the limiting factor in this system, and cause a latency of roughly 20 milliseconds. Since the tests were conducted within the same subnet, these results are best case results, and should be much larger when connecting via the internet. It is also worst to mention that the quantity of data being sent is in the order of less than 1kb, so the delays as measured are mostly composed of latency rather than travel time. As such, the geographical network topology will have much greater influence on these delays than the speed of the packets and it is doubtful that speeding up network bandwidth could have a significant impact in improving the performance of this system.