

Private Dropbox
Final Report
COSC480

Calum O'Hare
Supervisor: David Evers

Abstract

Private Dropbox is a file synchronisation tool designed to keep files in sync across multiple devices. It is decentralised, unlike many other file hosting services like Dropbox. My program is designed to run across any given network of computers. Configuration files allow the user to specify what files should be synchronised; to which machines; and how often to sync them. My program monitors the files being synchronised and the last time files were updated so that it knows when to sync with any other connected machines. Different machines may require different replication strategies, this makes it unique from other file replication tools.

Contents

1	Introduction	4
1.1	Project goals	4
1.2	Background	5
1.3	Example use case	5
2	Architecture and program development	8
2.1	Virtual machines and node networks	8
2.2	Reading network statistics	10
3	Program design	11
3.1	Python	11
3.2	Unison	11
3.3	User control	11
3.4	Monitoring directories	11
3.5	Dealing with sub-nodes	13
3.6	Unison and temporary files	14
4	Program evaluation	15
4.1	Point-to-Point synchronisation	15
4.2	Full graph replication	16
4.3	When to stop copying	21
4.4	How often to sync	23
4.5	Wi-Fi vs 3G	29
5	Future Work	30
5.1	Mobile nodes	30
5.2	Feedback	31
6	Conclusion	32
A	Program listings	33
A.1	WatchAndSync.py	33
A.2	ReadNet.py	45
A.3	ManipVMs.sh	49

1 Introduction

Dropbox is a well known online file hosting tool that will keep your data synchronised across multiple devices. I feel that there is a need for a private file synchronisation tool controlled by the end user rather than a corporation. In this section I will outline my project goals which explain how my program differs from Dropbox and others; give background information about the alternatives like Dropbox and what the problems with these solutions are; outline an example use case which shows why my program will be useful.

1.1 Project goals

The aim of this project was to develop a file synchronisation tool. Similar to Dropbox (and others) its main function should be to keep data synchronised between multiple devices. What makes it different however is it should:

- Support decentralised operation. It will not necessarily need to communicate with ‘the cloud’. The program should not require a centralised server. However it should be possible to configure the system to behave like a centralised system if the user wants to. The system should be flexible in this regard.
- Allow file synchronisation between multiple clients not just point-to-point between two clients. Synchronisation between two clients however is the basis for multiple client synchronisation. Clients may be running different operating systems. Clients may be connected to different networks, with different costs of access, including being disconnected from the Internet at times.
- Permit the user to choose what to replicate and how often to do it within different sets of files. Choosing what to replicate could be done based on file name, file types, file size *etc.* They system should allow for fine-grained user control for the majority of the program’s functionality.
- Show statistics about which files are being replicated, their efficiency (time taken for the files to become fully up to date), and cost (bandwidth, disk space used). These statistics could also possibly lead to a heuristic for when to synchronise a given file. For example if a file is updated and a node has many neighbours to potentially send the file to. Perhaps choosing the neighbour whose links has the lowest cost would be a good choice to send the data to first.
- Operate automatically, without the user having to initiate a file synchronisation themselves. The system’s autonomous operation should be influenced by the users choices on how often to sync and what files should be synced, this relates to the fine-grained controls mentioned above.

1.2 Background

There are already many services available that can synchronize your files between different devices. Dropbox, Google Drive, Microsoft SkyDrive, Apple iCloud are all examples of cloud-based solutions for distributing your files across your devices. The problems with these services is privacy and availability. Storing your data with a third party gives them access to your documents. If you are a commercial organisation with sensitive information this might be concerning. You could of course choose to encrypt your files. Encrypting your files adds two slow extra steps, encrypting them before you upload and decrypting files before you can use them. This is less than ideal. You also cannot guarantee that you will always be able to access your data, if the company that hosts your data goes bankrupt or decides to shutdown their service you could lose all of your data with little or no warning. For example Megaupload, a file hosting service, has recently been shut down by the United States Department of Justice for alleged copyright infringement. According to its founder, 100 million users lost access to 12 billion unique files¹.

There are other possible approaches to replicating files across multiple computers. For example you could use version control systems like Subversion and CVS. One problem with these is that they are centralised (they rely on a central server); should that server fail the replication will break. Not only that, they create a bottleneck at the server which can slow replication down. Cloud-based solutions are often centralised. Another problem is that even if they are decentralised like Git or Mercurial, they will not automatically push updates to other working sets. This could be accomplished with some Cron scripts or a post-commit hook to get Git to propagate data onwards. Git might have made a promising base to build my application on top of; the only real problem was the version control overhead that comes with it. Old revisions would take up space on the hard disk and require more data to be transmitted across network links. I decided that as a file synchronisation program, revision history was out of scope and that my program would deal with just keeping files in sync. Using Git would be an interesting extension to my program and could easily be integrated into my current system.

1.3 Example use case

Here is an example use case demonstrating why I find my program useful.

I like to keep all of the data on my laptop backed up to an external hard drive. The data on my computer that I wish to back up falls into three main categories: documents, music, and movies. Documents are mostly scripts and programs that I am writing for University or work projects. Documents also include reports for assessment. These documents change very frequently and are very important to me. Often these are small files (but not always). My music collection changes relatively infrequently, files are ≈ 5 MB and I like to have a relatively current backup of this collection. My

¹<http://computerworld.co.nz/news.nsf/news/kim-dotcom-wants-his-money-back>

movie collection contains fairly large files but I do not need it to be backed up very often as it does not change very much and I do not care if I loose some of these movies. Files that I work on at University would be very useful to have on my laptop at home. Files that I work on at work mostly stay at work but occasionally I might want to bring something home to work on. The other device I always have with me and may be on one of any given (Wi-Fi or 3G) network at a certain time is my smart phone. I would like to have photos taken on this backed up to either (or both) my laptop and external hard drive.

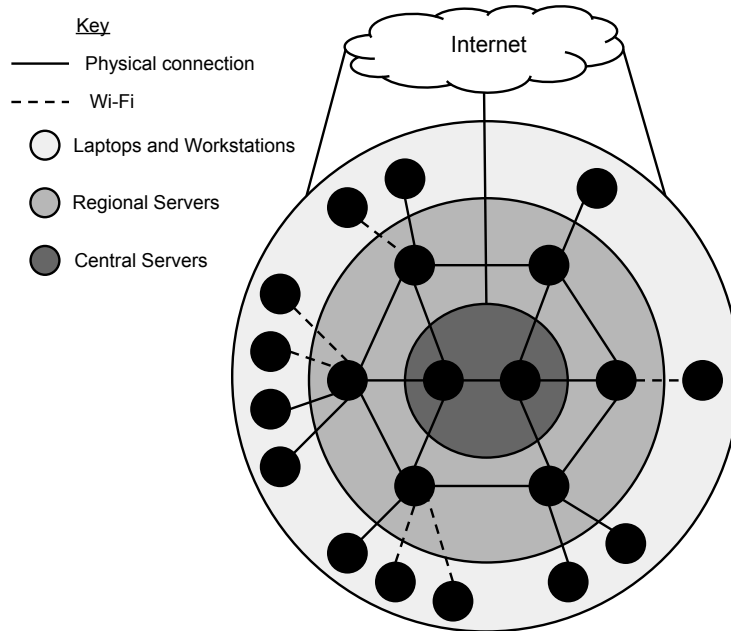
Some of the files that I move around are of a sensitive or personal nature and I would prefer not to store them with a third party vendor. I also have different synchronisation requirements for different types of data. For example my collection of large video files does not change that often and will chew up valuable network bandwidth whenever it has to transfer a new file. I like this to be replicated only occasionally as I do not use it that much. On the other hand my document collection which I use for work and coursework changes very often, is very important, and is fairly small. I would like this to be as up to date as possible.

Existing file synchronisation tools do not do enough for me. I do not have enough control over my data. I want to know which machines my files are going to and when. I want to feel confident that I will always be able to access my data even if the service closes down or my internet connection fails. My program is aimed at addressing these issues.

The personal network shown in Figure 1a is a graphical representation of my description above. Figure 1b shows a graph of a corporate network, this is another example use case. It will have many of the same basic needs as the personal graph. The coloured rings represent the need for different policies for different machines in a network. Unlike Dropbox this is an advantage my system provides.



(a) Personal Network



(b) Corporate Network

Figure 1: Example use cases

2 Architecture and program development

Before I could begin developing my program, I needed to have a test environment to run it in. Given that my program is designed to run across a network of computers I needed to be able to change links between machines and add or remove machines automatically for testing purposes. Doing this manipulation of the network manually would be too time consuming in the long run. In this section I will discuss how I control a virtual network of virtual machines for testing, how I manage the links between them and how I monitor these links to gather performance statistics regarding the execution of my program.

2.1 Virtual machines and node networks

For testing my program I needed to have a network of computers that could easily be linked together in different arrangements. I think of a network of machines as a graph, where each machine is a node of the graph and the links between machines are edges in the graph. I decided to use virtual machines for my network since it means I do not need to have a large number of physical machines. I can create new machines very easily, and manipulate the links between them.

I have used Oracle's VirtualBox software. I chose VirtualBox because of its straight forward command line interface. My program should be able to run across any network of nodes. For this reason I wanted the potential to be able to test a given arrangement of nodes. I decided that I needed to be able change network topologies easily without having to re-write my scripts.

I built some Bash scripts to run on top of a program called Graphviz.² Graphviz is open source software for generating graphical representations of graphs. I used Graphviz to generate graphs of all the topologies I worked with. This made it easy to keep track of what a topology looks like which was useful for debugging. It was also useful to display results alongside an image of what the topology looks like. Building my program on top of Graphviz meant that I could couple the production of the topology graph and the configuration of the virtual machines. I never wanted one without the other so this was very useful.

Graphviz takes input from scripts written in DOT language.³ DOT language is a simple graph description language. I have written a script to read in these DOT files and interpret the graph to set-up my virtual machines (see Appendix A.3 line 132 onwards). My script also calls a program called Neato (part of Graphviz) to generate graphical presentations of graphs. This means I only have to write one DOT file to get a graph of my network topology and set-up my virtual machines.

My Bash script enables the appropriate network adaptors on each virtual machine (see Appendix A.3 line 67). It does this by calling the `VboxManage`, command which

²www.graphviz.org

³<http://www.graphviz.org/doc/info/lang.html>

provides a command line interface to VirtualBox’s functionality and allows me to configure the virtual machines to meet my needs. Then it creates an internal network and attaches these adaptors to use that network. I chose to use an internal network as the link between any given machines because this way I could guarantee my program was the only program using the interface. This helped me monitor the network traffic generated by my program (see Section 2.2). I sniffed network traffic using packet sniffing tool Wireshark⁴ when my program was not running and also when my program was running to verify that no other programs were using the interfaces attached to internal networks.

I started initial tests by writing some simple network topology DOT scripts. The reason I chose to use simple topologies when testing my programs was because it is easier to visualise how my program runs from the data if the topologies are simple. The other reason I chose to do this is because more complex topologies can be thought of as being made up of many simple topologies. In this way I might be able to generate a model of how more complicated topologies might behave by looking at how the program runs in simpler topologies.

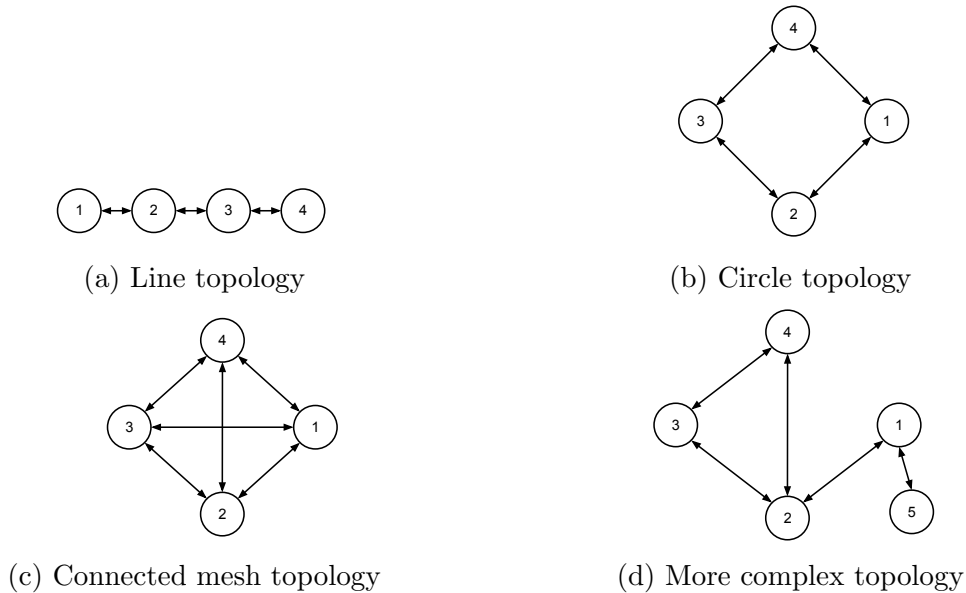


Figure 2: Simple network topologies

⁴<http://www.wireshark.org/>

2.2 Reading network statistics

In order to gather network traffic statistics I chose to monitor the upload and download data collected by the interfaces over time. Given that my program was the only program using these interfaces I did not have to differentiate between different processes using the network. I looked at the possibility of gathering statistics on a network where other processes are using the network connection as you would expect in a real world application of my program. For this I found a tool called NetHogs⁵. NetHogs runs on Linux and monitors the amount of data sent over the network by any given process. An interesting challenge I faced when I was trying to log network usage was what interface to log data for. Initially I logged the traffic data of all of the interfaces on the machine (see appendix A.2 line 100). The flaws in this approach became apparent very quickly; machines with an internet connection sometimes used large amounts of data which interfered with the results. It also did not work with machines that were on more than one internal network as they might be sending/receiving data on both of the networks at the same time. I overcame this problem by using a built-in Linux command `ip`. I always had an IP address to send data to, so all I needed to do was run `ip route get {IP address}` and examine the route information to see which interface was being used for communication to that IP address (see appendix A.2 line 46). Once I had the interface name I would then record that data before and after a sync occurred for that given interface. I would also do this on the machine I was sending data to. This would give me timestamps of when a sync was initiated and when it finished. Knowing the network traffic statistics before and after a sync also allowed me to determine how much data each machine sent or received during a transfer or over any given time period. I also decided to include the folder's name being synchronised in the log files. This was useful for differentiating which sub-node (See Section 3.5 for further details on sub-nodes) was being synchronised. It would be trivial to add more detailed information to my logs, such as the IP data is being sent to or received from. This is a feature I would like to add in the future: I discuss this possibility further in section 5.2.

⁵<http://nethogs.sourceforge.net/>

3 Program design

In this section I discuss the design decisions I have made when building my program. This section details why I chose to use Unison, Python and Inotify to build a file synchronisation tool. Some of my design decisions came from running test cases of the program and reacting to the results I obtained.

3.1 Python

I have chosen to use Python to implement my program. Python appealed to me because it supports many different platforms (Windows, Linux, Mac OS X). This is useful because it means I will encounter fewer compatibility problems when running my program across different operating systems in the future.

3.2 Unison

After looking for cross-platform, open source, file synchronisation tools, I have found a tool called Unison⁶ to be a promising starting base for this project. Unison is an open source file synchronisation tool. It supports efficient (*i.e.*, it attempts to only send changes between file versions) file synchronisation between two directories (including sub-folders) between two “roots” that may or may not be on the same machine. Unison calls the directories it is synchronising, roots. Section 4.1 looks at how efficient Unison is at copying files.

3.3 User control

One of the main goals of my project is to allow the user to have a large amount of control over how the program behaves. I currently have the program reading from configuration files that allow the user to specify which directories they want to watch and where those directories should be synchronised to.

I chose to replicate directories rather than files because keeping track of a large list of files may become unwieldy. Because I replicate directories recursively, I can replicate large amounts of data without a cluttered configuration file. Another reason I chose directories as my granularity was because it may be handy to have a directory full of symlinks pointing to other directories.

3.4 Monitoring directories

The application needs to monitor directories for changes (see section 3.5) so that it knows when to perform a sync. The reason I have chosen to do this is because synchronising a directory that has not been changed is a waste of time. I do not however want to be continually polling the watched directories to see if there have been any

⁶<http://www.cis.upenn.edu/~bcpierce/unison/>

changes made. This would be a significant waste of CPU time and the input/output time associated with checking the disk. Instead I have looked into ways of being notified of a change in the file system below the watched directory.

- Inotify

- Inotify is a kernel feature that has been included in the Linux kernel since version 2.6. It is used to watch directories for changes and notify listeners when a change occurs. Inotify is inode based and replaced Dnotify, an older system that provided the same functionality. Dnotify was inefficient, it opened up the file descriptors for each directory it was watching which meant the backing device could not be unmounted. It also had a poor user-space interface which used SIGIO. In contrast Inotify only uses one file descriptor and returns events to the listener as they occur ⁷ There is a Python module called Pyinotify⁸ that provides a Python interface to Inotify, which I have used in my program. Another reason I chose Inotify was because different kinds of changes triggered different Inotify events. So I can differentiate between a file being deleted, created or modified, *etc.*

- FSEvents

- FSEvents is an API in MacOS X⁹ It is similar to Inotify in that it provides a notification to other applications when a directory is changed however it does not inform you which file in the directory was changed. This does not matter for my application since Unison is smart enough not to copy unchanged files in a directory. There is a Python module for FSEvents called MacFSEvents¹⁰.

I also looked at using the `kqueue` ¹¹ system call that is supported by MacOS X and FreeBSD. It notifies the user when a kernel event occurs. However I decided against using `kqueue` as the high level approach of FSEvents suits my application's needs.

- ReadDirectoryChangesW

- Windows, like the other operating systems I have examined, provides a way of doing this too. There is a function called `ReadDirectoryChangesW`. There is a `FileSystemWatcher` Class in .NET version 4 and above. IronPython might prove to be a good choice for a Windows implementation as it is a version of Python integrated with the .NET framework. I have chosen only to implement my program on Linux because portability was not in the main

⁷www.kernel.org/pub/linux/kernel/people/rml/inotify/README

⁸<http://pyinotify.sourceforge.net/>

⁹https://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/

¹⁰<http://pypi.python.org/pypi/MacFSEvents/0.2.1>

¹¹<http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/kqueue.2.html>

scope of the project. I would have liked to look at it further but it became too time consuming and not interesting from a research perspective.

3.5 Dealing with sub-nodes

I chose to classify directories as ‘sub nodes’ of a graph. The reason I choose directories was because I thought it would be easier for a user to maintain the configuration file. A configuration file listing every file to be synchronised could become very cluttered. If a user wanted to synchronize certain files in a directory they could write a Unison configuration file with exclusions/inclusions in it or use my program’s built-in ignore list (see Section 3.6) I wanted to see how the freshness of different sub-nodes varied between nodes when the program was running so I tagged each sub-node when logging information about the programs run (see Section 2.2 for more details on logging). Figure 3 shows a network with two sub-nodes. I alternated between creating 10MB files in each sub-node. The data transferred for the second sub-node trails the first sub-node in time but a similar amount of data is transferred over time.

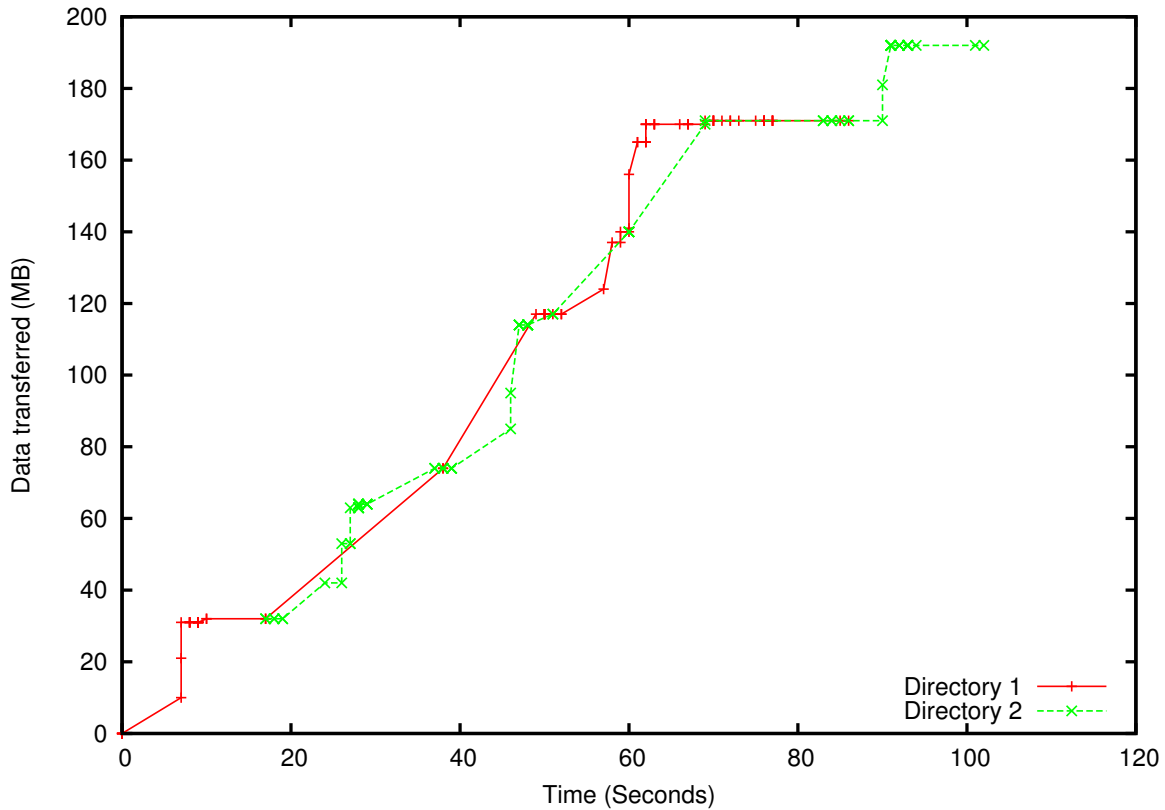


Figure 3: Line topology synchronisation using Unison. Two different directories are being synchronised with 10MB files are sent to each directory alternately every ten seconds.

3.6 Unison and temporary files

I noticed that when Unison ran it created temporary files in the directory and once these files had been fully copied it renamed them to their intended name. The problem with this was that my program was picking up these temporary files as they were created and trying to copy them to the next node, only to find that these files no longer existed. To get around this problem I decided to implement a filter on the files to be copied. The program filters out files that contain ".tmp" in the file name. Unison is not the only program that uses temporary files. I decided that this should be a user set preference given that users may want to filter out different files.

My program simply reads from a file with each file name pattern to exclude listed on a new line. It is easy to add to/remove from. As I said above I added .tmp to the file as a default. This could easily be extended to allow a user to omit certain files from the replication by adding all files in my program's ignore file to Unison's ignore list, or conversely by maintaining a white list of files to sync. This would allow for greater granularity when syncing nodes.

4 Program evaluation

During the development of my program I ran tests after major changes so that I would know if the program was behaving the way I expected or not. When problems arose such as machines continuously polling each other for changes. I made adjustments to my program to address these problems. In this section I discuss the results from testing my program and how my results influenced my program.

4.1 Point-to-Point synchronisation

I decided to run some tests using Unison and two machines running on the same network to determine whether Unison copies data efficiently.

I looked at three methods of file synchronisation across these two machines. Naïve copying using SCP; using Rsync, an application designed for efficiently copying files in one direction by looking at the differences in the files; and Unison as described in Section 3.2.

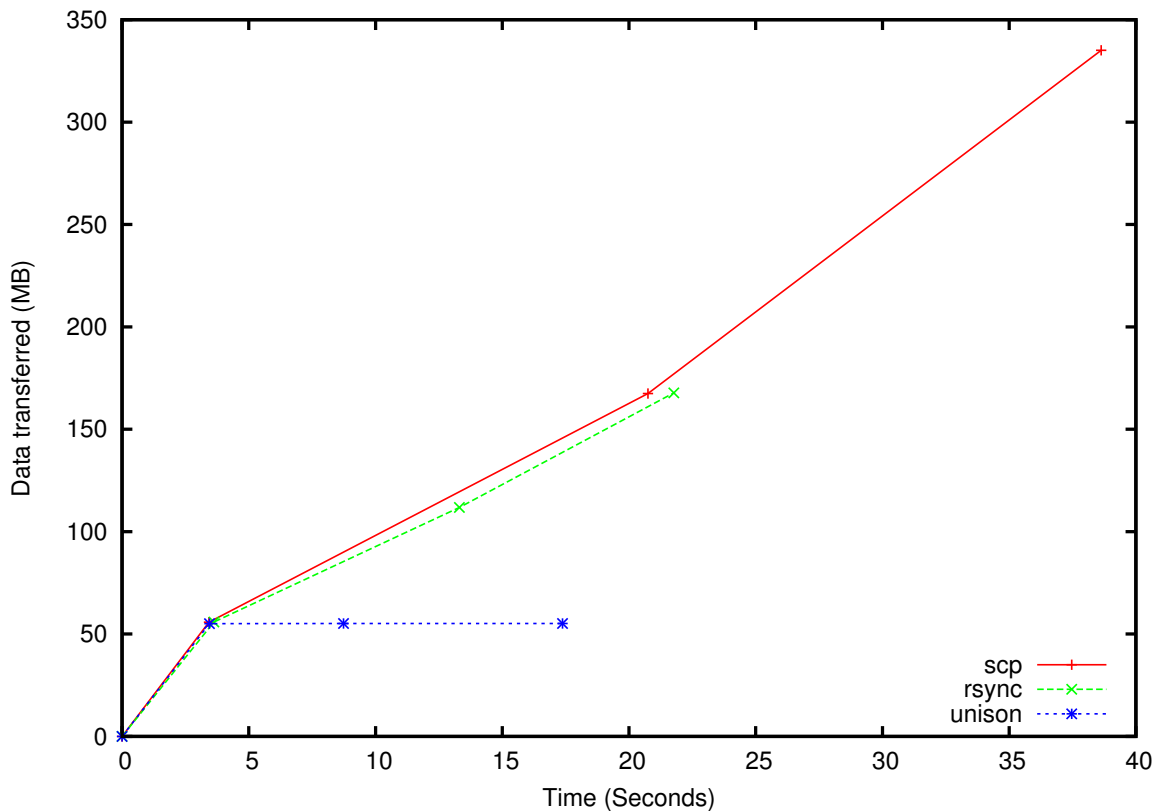


Figure 4: Comparison of SCP, Rsync, Unison. Three identical 50MB files with different names are being transferred between two nodes. Each point on the graph represents the start or end of a sync.

Figure 4 shows how Rsync and Unison performed significantly better than SCP (as

expected). After the initial file transfer, new files added to the directory resulted in much less data being transmitted over the network, which meant the node graph (see Section 2.1) became up to date much more quickly.

Figure 4 shows that SCP sent over 300MB of data to copy three 50MB files. This was because SCP is naïve; when it is told to copy a directory it will copy the entire directory without looking at the files inside. This means SCP copies the entire directory each time the directory is changed. Rsync and Unison were able to send less data because they work based on the differences between the files. SCP however will copy 50MB after file one is created, 100MB after the second file is added and finally 150MB when all three files are present for a total of 300MB.

Rsync copies the expected 150MB for three 50MB files. Figure 4 illustrates another advantage of Unison over Rsync. The graph shows three zero-filled binary files being copied from one node to the other, one file at a time. Unison recognised that even though the files were named differently they had the same content. Another advantage of Unison is that it handles replication in two directions without overwriting the files on one node when two nodes have conflicting changes.

Each of the three methods I trialled had some overhead associated with them. This overhead was due to the secure shell (SSH) tunnel between the machines that all three methods used. Unison and Rsync also incur some overhead when comparing the differences between the files in the directories. This is why the graph shows the three lines slightly above where you might expect them to be for the amount of data that was copied.

4.2 Full graph replication

I ran my program on a simple line topology (shown in Figure 5) of all three of my program's file synchronisation techniques. Figure 6 shows SCP performing poorly compared to Figure 7 (Rsync) and Figure 8 (Unison). Using Unison and Rsync each node only received ≈ 30 MB, given that 30MB of data was sent to node A this is a pleasing result. SCP on the other hand sent much more data than necessary as I have explored in Section 4.1.

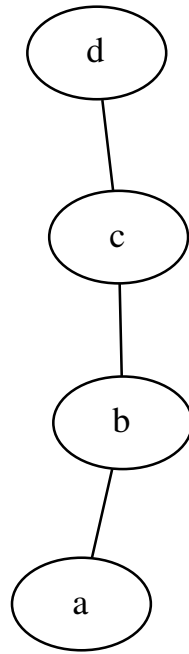


Figure 5: Generated graph of line topology

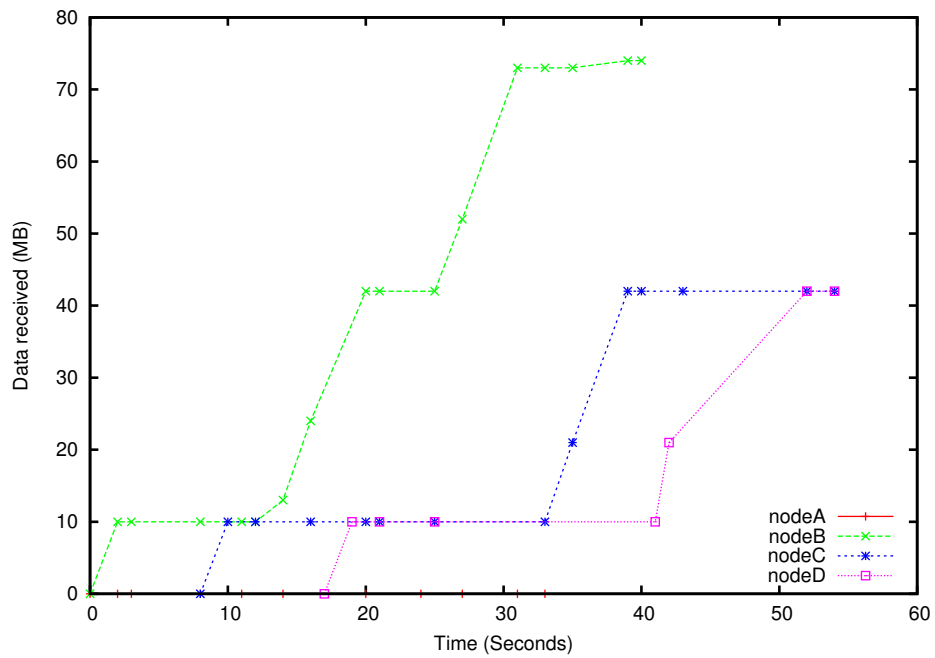


Figure 6: Program running using SCP. A 10MB file filled with random data is being sent every 10 seconds to node A for thirty seconds.

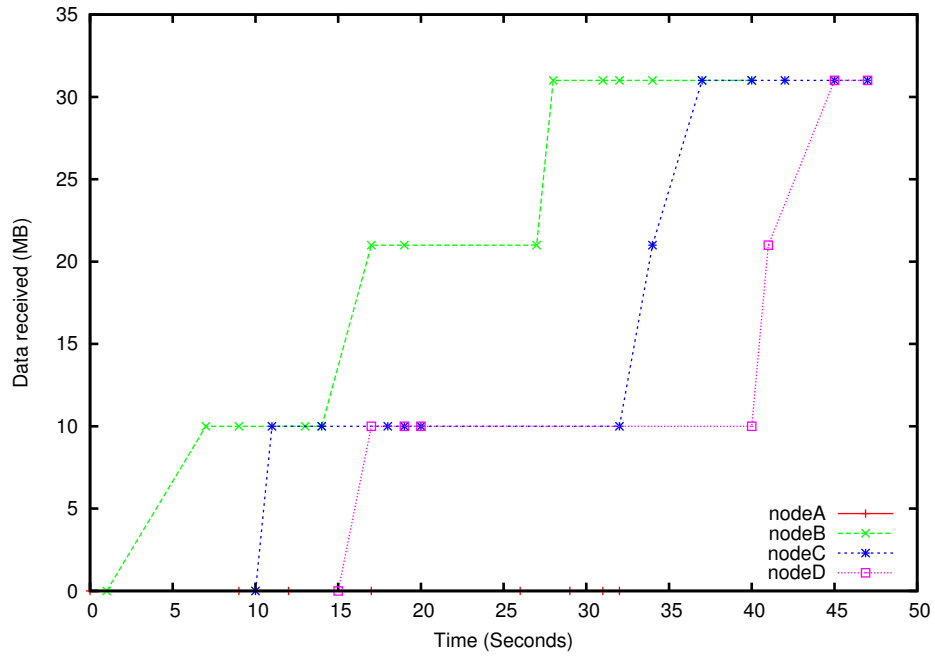


Figure 7: Program running using Rsync. A 10MB file filled with random data is being sent every 10 seconds to node A for thirty seconds.

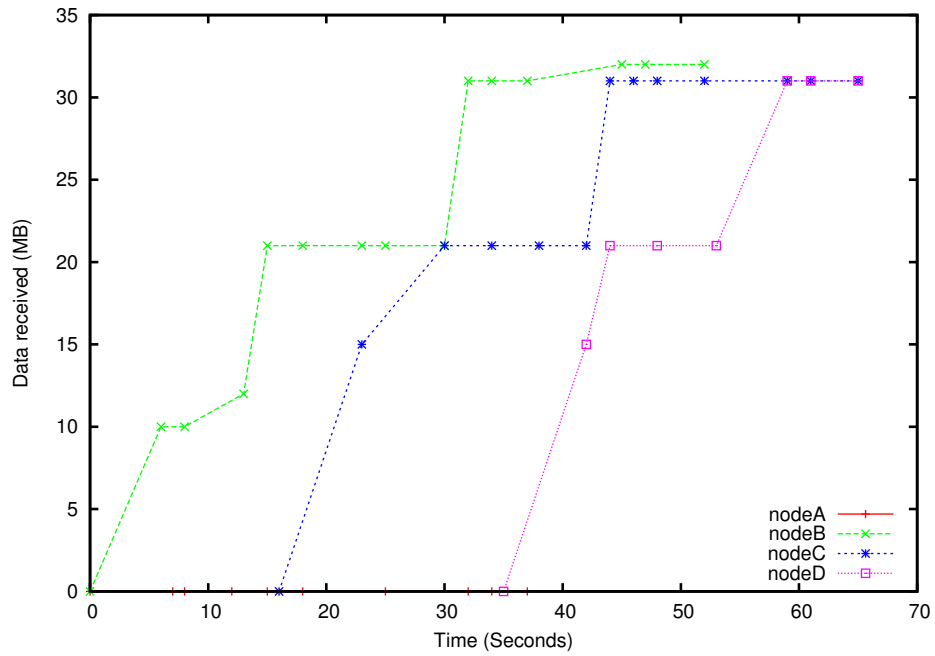
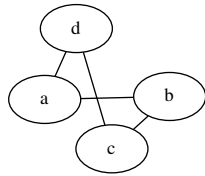
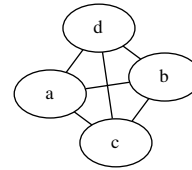


Figure 8: Program running using Unison. A 10MB file filled with random data is being sent every 10 seconds to node A for thirty seconds.

My program ran smoothly on a line topology so I set-up some new topologies as shown in Figure 9a and Figure 9b. These were the topology graphs generated by my set-up scripts (see Section 2.1) I designed them to be a circle (Figure 9a), although it does not look like a circle this is how Neato represented the topology, and a connected mesh (Figure 9b). My program ran as expected with the topologies that I tested it on. Figure 10 shows the results of the circle topology test case. Figure 11 contains the mesh topology results. These are both graphs of the program running using Unison. In both cases each node received $\approx 30\text{MB}$ of data which means each node received all of the changes created on node A and that my program successfully replicated data to all nodes in the network. You can see in Figure 11 that the changes propagated through the graph in a different order to Figure 10 this was to be expected given the more connected nature of the nodes in the mesh graph. With the completion of these tests I had achieved one of my main objectives; to have my program running over different network topologies.



(a) Generated graph of circle topology



(b) Generated graph of mesh topology

Figure 9: Topology graphs

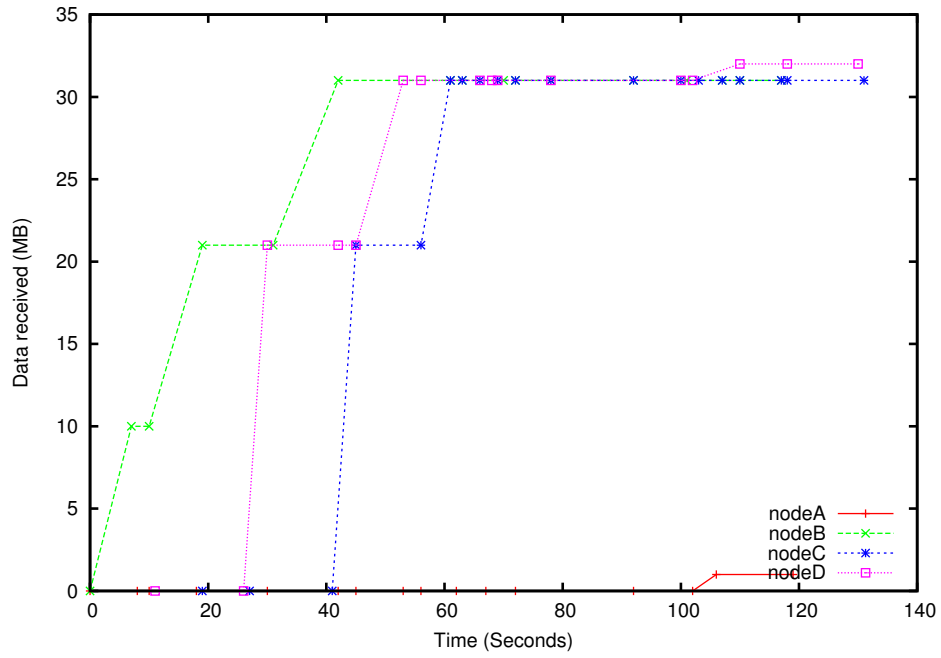


Figure 10: My program running using Unison on a circle topology. A 10MB file filled with random data is being sent every 10 seconds to node A for thirty seconds.

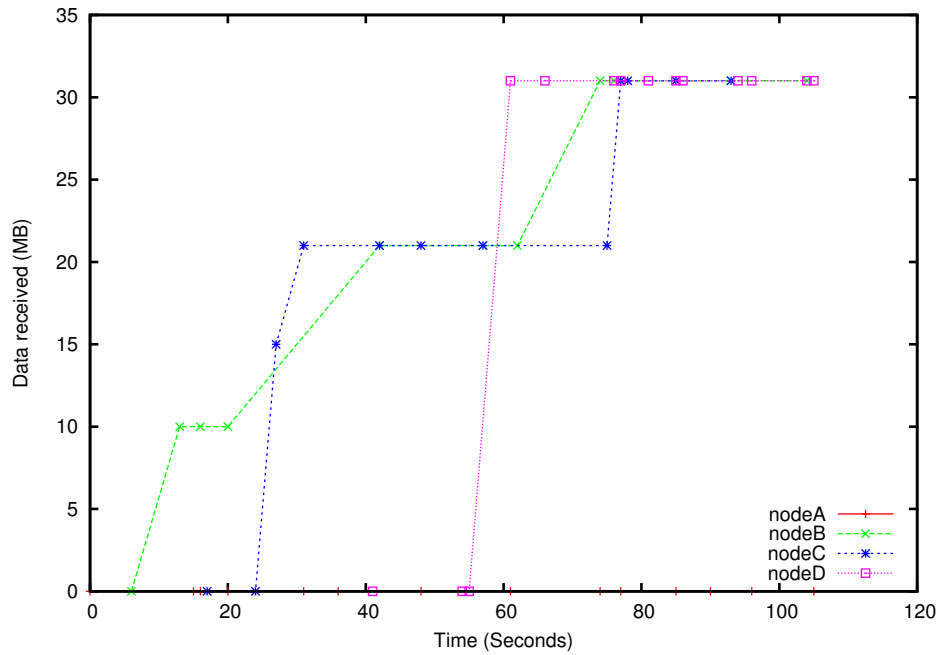


Figure 11: My program running using Unison on a mesh topology. A 10MB file filled with random data is being sent every 10 seconds to node A for thirty seconds.

4.3 When to stop copying

The way my program works is that each node notices when changes have occurred to a folder it is watching and when a change occurs, it copies these changes to other nodes specified in the configuration file. After testing my program on some simple topologies one problem became clear; if the changes came from one of its neighbour nodes the node would notice the change and send data back to its neighbour this would cause an infinite loop of two nodes trying to copy changes to each other. When using Unison this was not as much of a problem because Unison can detect that no changes had occurred between the nodes and would stop synchronising after one check (which had minimal overhead). However for SCP this was a big issue.

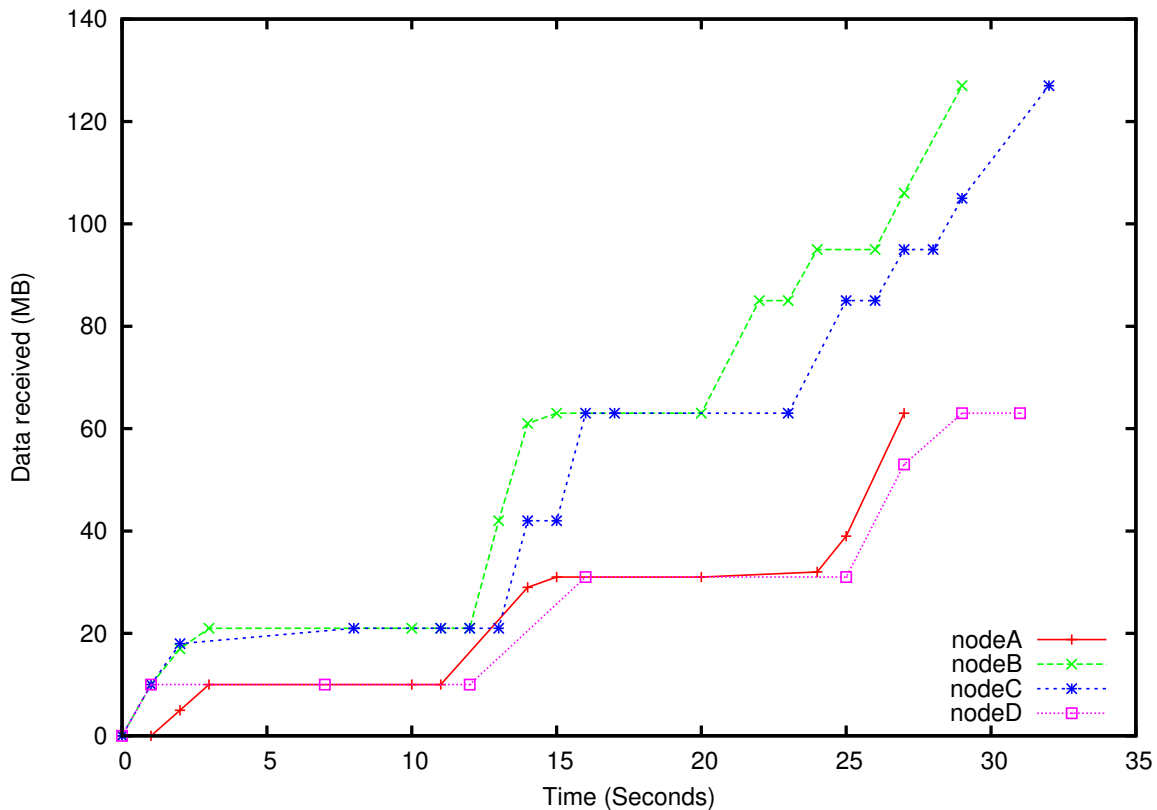


Figure 12: My program using SCP running on a line topology. 10MB files are being sent every ten seconds to node A. All of the nodes receive more data than they should because they are constantly propagating changes to their neighbours.

Figure 12 shows each node receives more data than it should. Node B and node C receive over 120MB worth of data even though there was only 30MB worth of changes and node A receives over 60MB even though it was the source of the file changes. My program would have run forever if I had not stopped it.

The data points in Figure 13 show that when using Unison, although no extra data was sent, Unison still had to make checks to see whether there were any changes or not.

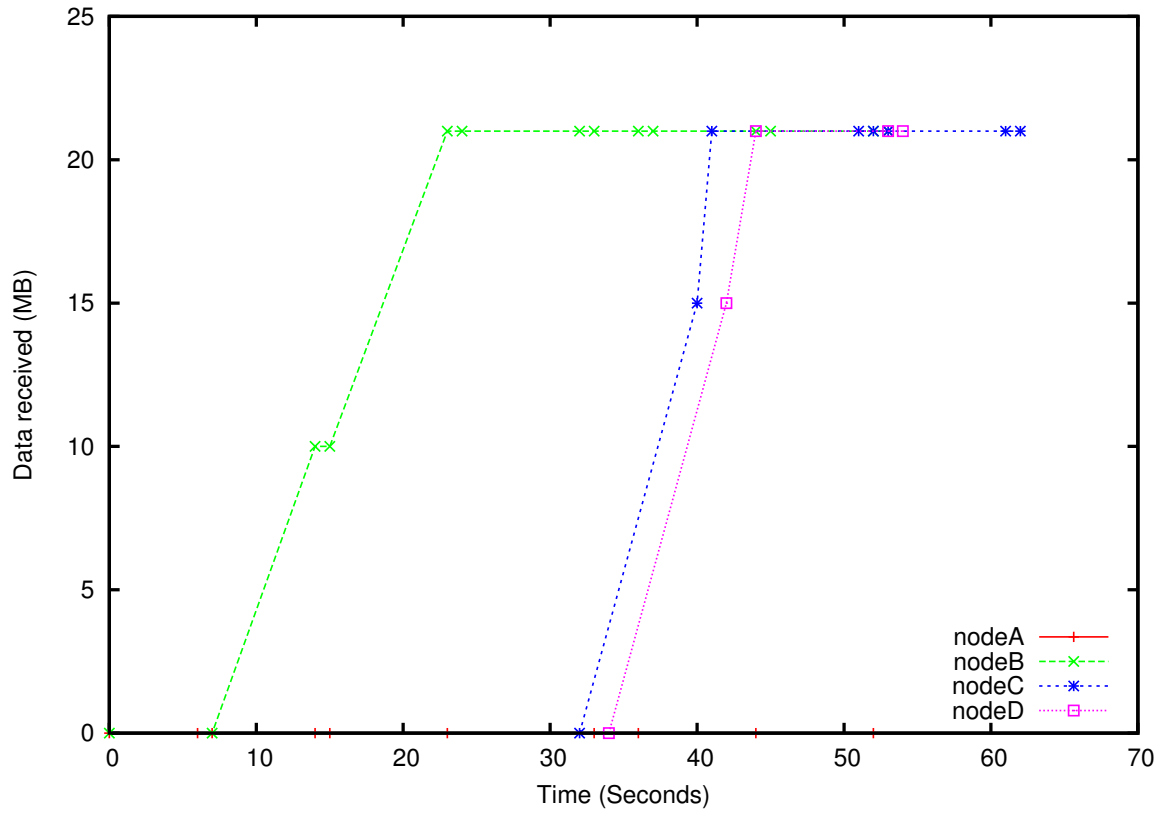


Figure 13: Line topology, using Unison, program continues to check for differences in files even after all nodes are the same. Each point represents an attempted sync.

To fix this problem I used a control file. Each time a node synchronised with another node it would write out a control file telling the other node what files had been copied, who sent them and what the modification times of the files were. In this way a node could check if it was about to synchronise a file back to the node it had just received the file from, or if local changes really had occurred to that file that were newer than a received file it should continue with its sync.

4.4 How often to sync

So how often should I sync once I notice a change? If lots of small changes are occurring frequently it might be more efficient to perform a synchronisation after several changes have occurred. Given that there is overhead with each synchronisation, fewer copies means less data sent over the network.

Figures 14, 15, 16, and 17 show my program running using Unison to sync 10MB over a network arranged in a line topology. The files were moved into the watched directory in ten second intervals. You can see that there is not much difference between the graphs except Figure 14 where the time taken for all of the nodes to become up to date is shorter than the others. In the case of Unison it is best just to sync as often as possible because Unison only sends file differences. The only saving when sending multiple changes at a time is on the SSH overhead and the overhead associated with Unison comparing the files. This overhead is negligible when the files are of any meaningful size.

Figures 18, 19, 21, and 22 show the same 10MB files being transferred every 10 seconds over the same line topology as previously mentioned except I used the SCP option on my program for these tests. Figure 21 shows that (20 second delay) the time taken for all nodes to get the changes was less than in Figure 22 (30 second delay) You can also notice that when the delay was set to less than or equal to the time the files were being created (10 seconds) that my program behaved in an erratic fashion with some nodes receiving less data than others. This could indicate an error in my program or that the data simply takes an odd way through the graph as it is being replicated as soon as possible. The data is still replicated in full to each node in the graph however. You can see that in Figure 20 that the graph became more uniform as soon as I increased the delay to 11 seconds.

These results reinforced my idea that the delay should be set through a configuration file since the appropriate delay will vary based on how often the files change. This is currently the case with my program, users can set a different (or the same) delay for each folder that they are watching. I also allow the user to use the ‘*’ wild card character to tell the program to sync as often as possible. Based on my findings I would recommend to any user that the delay time be greater than the average time between modification of the files but as short as possible as long delay times slow down replication proportional to the delay time. If the user is using Unison they should set the delay time to be ‘*’ (sync as often as possible) unless they are trying to synchronise very small files.

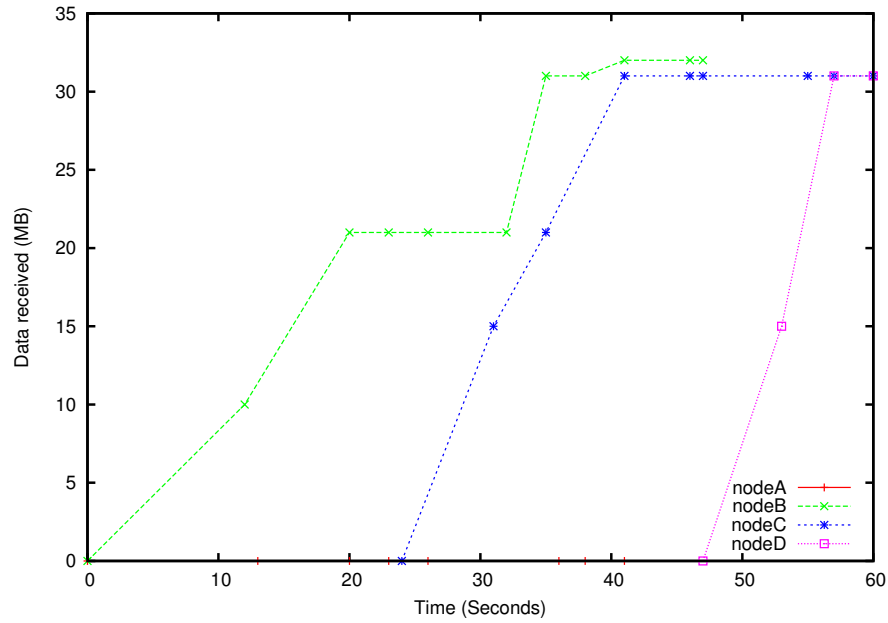


Figure 14: My program running on a line topology using Unison. The delay to synchronise time is set to five seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

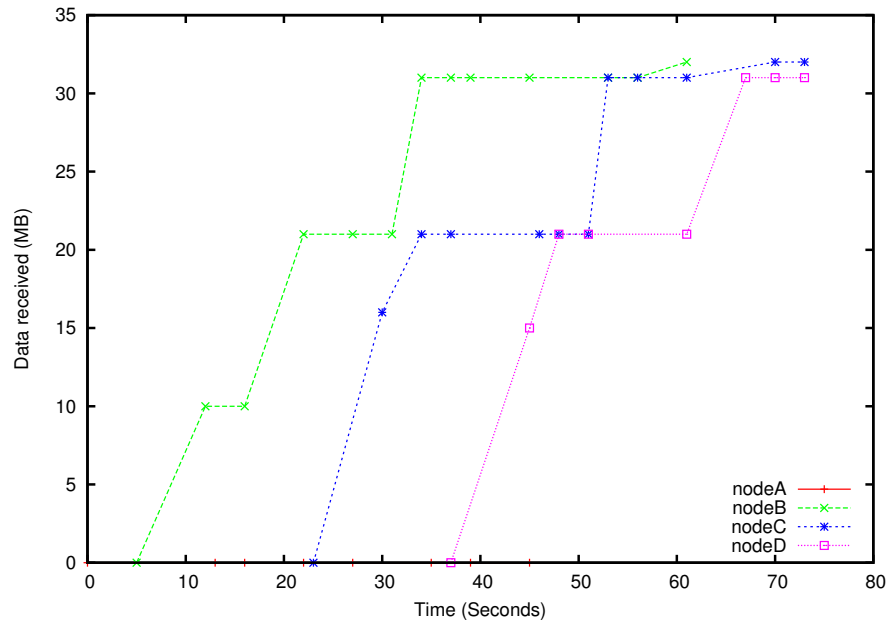


Figure 15: My program running on a line topology using Unison. The delay to synchronise time is set to ten seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

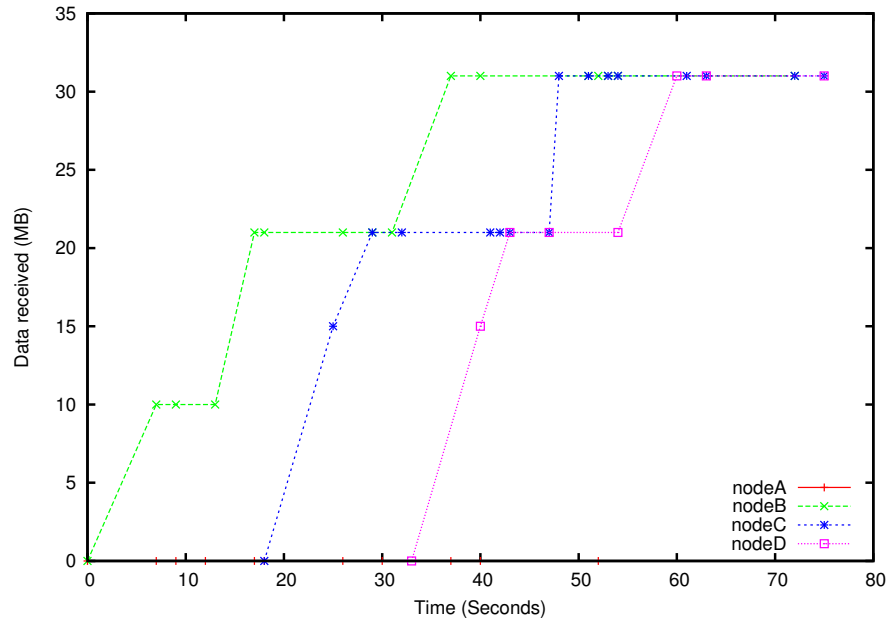


Figure 16: My program running on a line topology using Unison. The delay to synchronise time is set to twenty seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

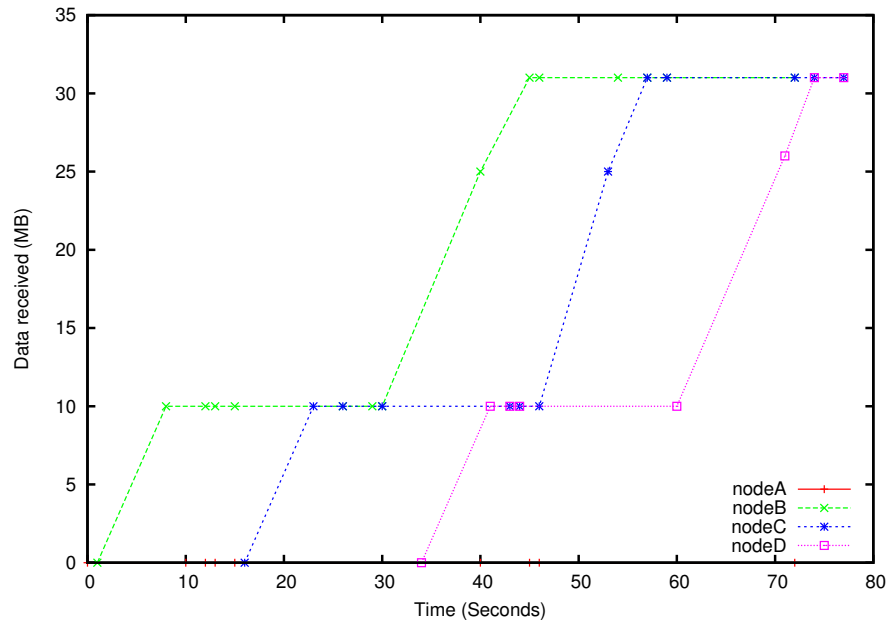


Figure 17: My program running on a line topology using Unison. The delay to synchronise time is set to thirty seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

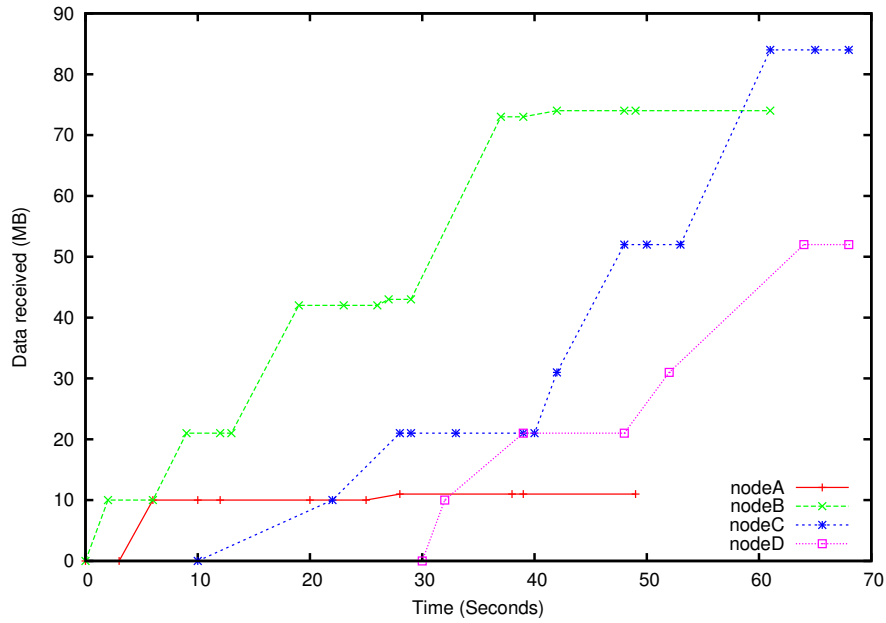


Figure 18: My program running on a line topology using SCP. The delay to synchronise time is set to five seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

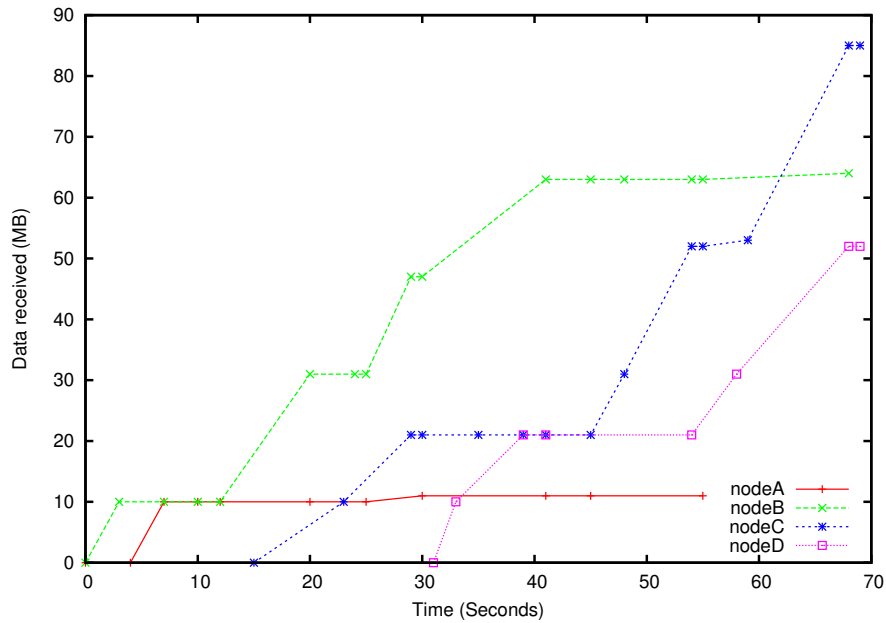


Figure 19: My program running on a line topology using SCP. The delay to synchronise time is set to ten seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

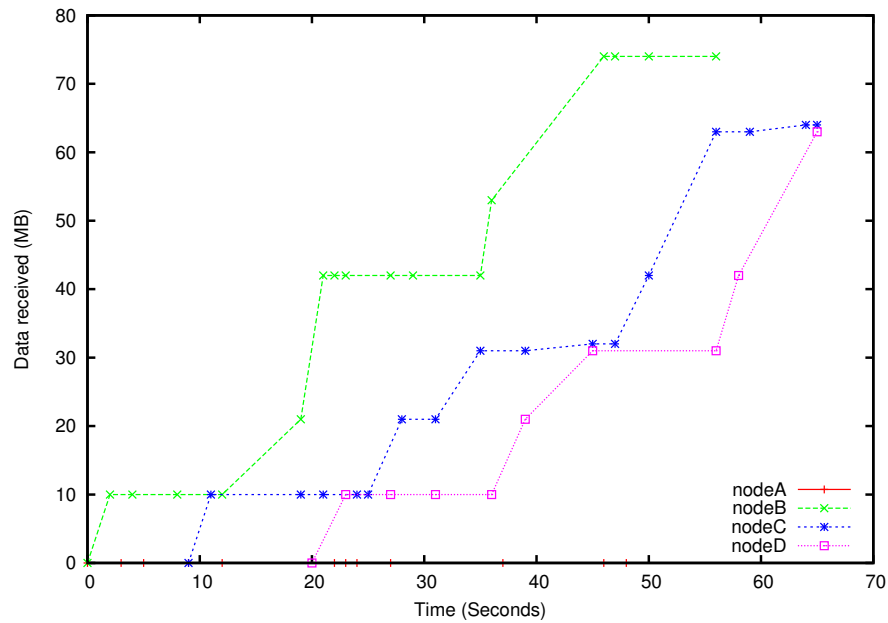


Figure 20: My program running on a line topology using SCP. The delay to synchronise time is set to eleven seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

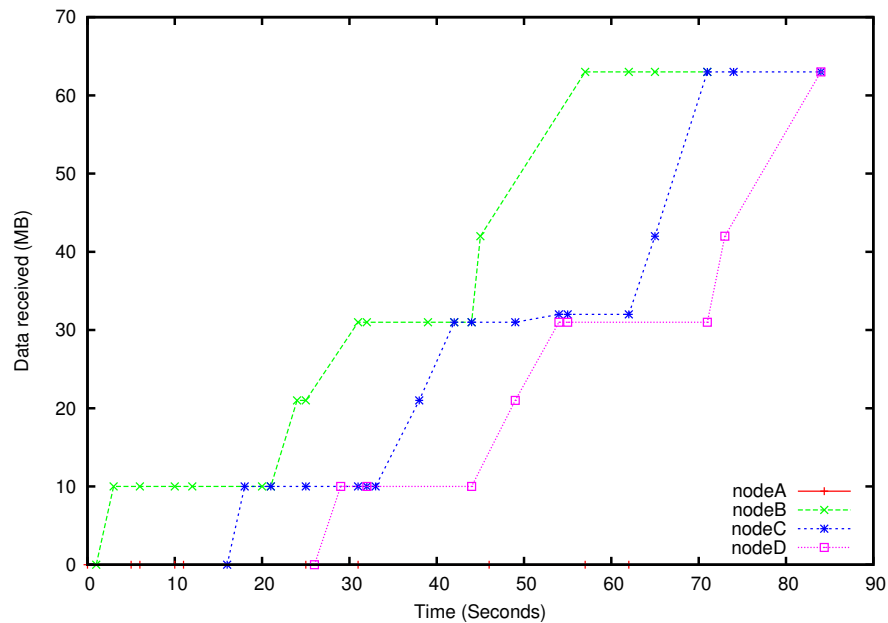


Figure 21: My program running on a line topology using SCP. The delay to synchronise time is set to twenty seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

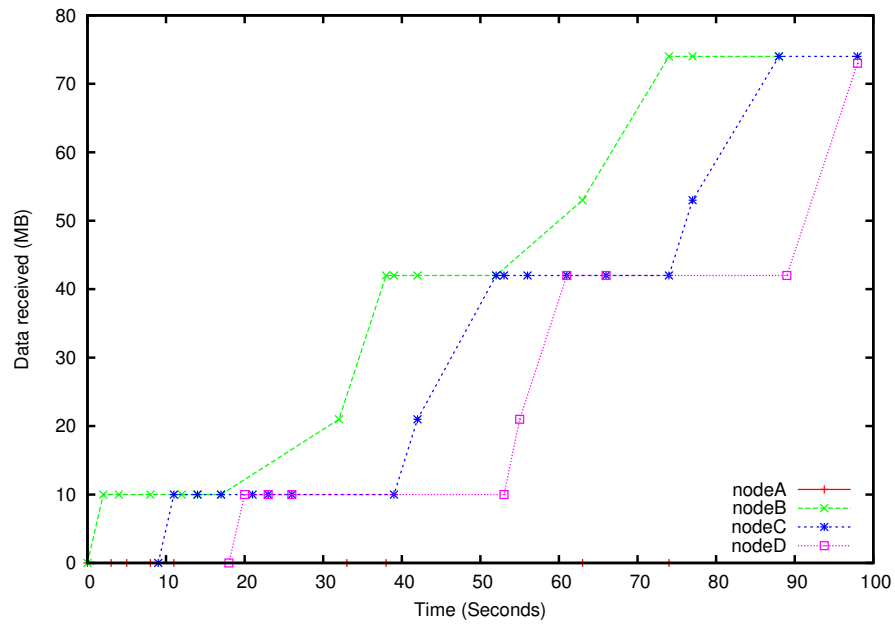


Figure 22: My program running on a line topology using SCP. The delay to synchronise time is set to thirty seconds. 10MB files are being sent every ten seconds to node A. The graph shows the amount of data received by each node.

4.5 Wi-Fi vs 3G

An important use case of my program was where one machine was connected by a link that a user would not want to use very often. For example a node might be connected by a 3G link which is slow and expensive to use. I wanted to make sure that my program could be set to prefer some links to others. For example to use Wi-Fi whenever possible and try not to use 3G. I designated the link between node A and node B as a 3G link in my circle topology (see Figure 9a in Section 4.2). Node A is linked directly to node B by a slow link and indirectly by fast links (Wi-Fi link to node D, node D links to node C and node C links to node B all designated as Wi-Fi links). Previously in Figure 10 (Section 4.2) node A sent data to node B and D which then propagated to node C. In Figure 23 node B receives data last only after nodes D and C have got the changes. This is because I set the delay time to 100 seconds between node A and node B as if it were a 3G link that we do not want to use very often. All the other nodes had delay times of 10 seconds on their links. My program spawns multiple threads for each link when a change has occurred in a watched directory (see Appendix A.1 line 327). The thread for the 3G link then sleeps for 100 seconds and when it wakes looks at the control files of the remote machine to see that it already has the changes node A was going to send so it aborts. Node B has already got the files because they travelled through the graph in the other direction and reached node B before A was ready to send.

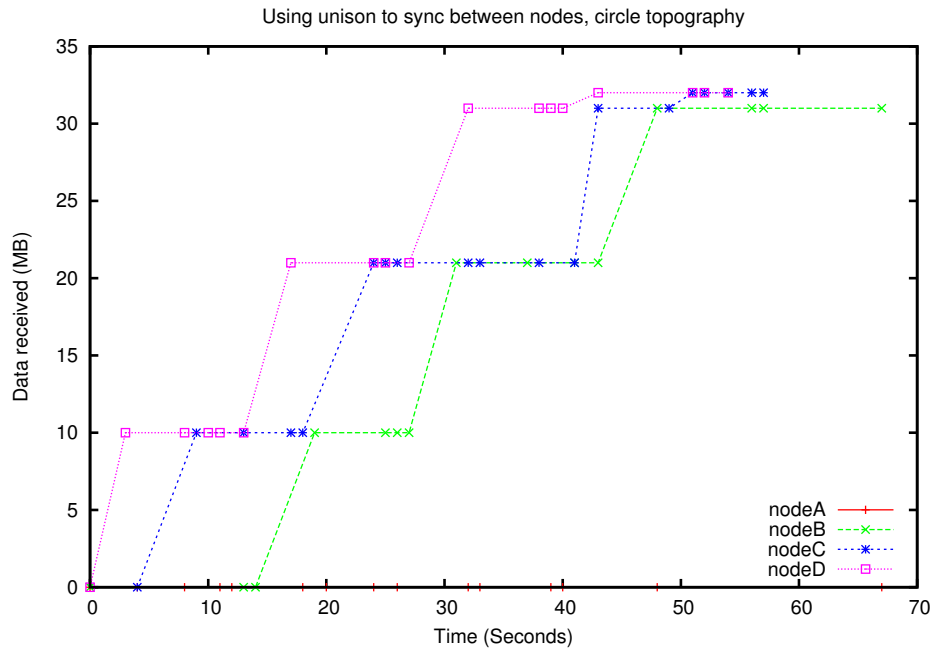


Figure 23: Node A is connected to node B by 3G, the other links are all Wi-Fi. Three 10MB files are sent to node A over ten second intervals. Node A has a longer delay on sending over 3G than Wi-Fi and node B is the last node to receive the changes.

5 Future Work

My project ended up requiring a lot of initial set-up time getting the testing infrastructure ready to produce different networks and graphs without requiring user input. As such this drew time away from other areas of research that I had originally planned to investigate. The main two areas that I would liked to have spent more time looking at were mobile nodes and user/system feedback of data. When nodes leave and join the graph *i.e.* when nodes leave a network and then join a new network, or rejoin the old one I call this a mobile node. I would also have been interested in seeing how I could have improved the performance of the program using the statistics the program gathers as it runs.

5.1 Mobile nodes

Connections between nodes in the graph (edges) may change over time. This could be because one of the nodes is a laptop and joins different networks at different times or because a network/machine is unreliable and is not up at a given point in time. I have represented edges that behave in this way as grey on the diagram below. I will refer to nodes with grey edges as ‘mobile’ nodes.

It would have been interesting to see how mobile nodes affect how up to date the nodes in the graph are. I predict that nodes that are not available for very long periods would lag behind others that are available. I wanted to see what would be the most effective way of getting these mobile nodes up to date quickly. Giving mobile nodes priority over normal nodes might be one solution. I expect that if a mobile node is a link between two parts of the network that these two parts will fall out of synchronisation when the node leaves the graph. I wanted to look at how nodes in the graph may rely less on unreliable edges. I would like my program to associate a cost with each edge and prioritise edges with lower costs when synchronising data. Another option would be to preference reliable edges and only fall back to using unreliable edges when necessary. Classifying edges could be done by cost or by how often an edge is available.

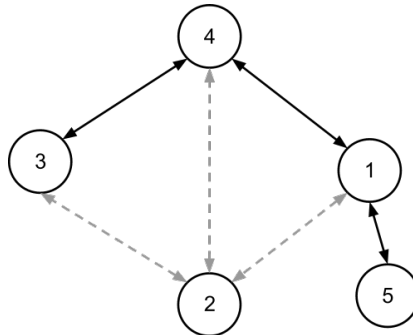


Figure 24: Node two intermittently has a connection to a subset of nodes 1, 3 and 4. The edges in grey are links that are not always present.

5.2 Feedback

My program currently keeps detailed logs of the system state. Logs are kept of the amount of data transferred between the machines; which files were updated and when; and the last time the data being watched was updated by another client (see Section 2.2 for more details on logging). I currently have scripts to generate graphs of the data used and the time taken for the graph to become up to date. I would have liked to integrate these scripts into my program to show an end user a graphical representation of the program running. Integrating my current graph scripts should be easy since these scripts are written in python and so is my main program. What would have been very interesting to look at would be to see if I could use data gathered from previous runs of the program to see if my program could estimate future performance based on changes made to the settings file (like how often to sync a given sub-node). I envisaged each of the clients sharing their data with each other to get a view of the network as a whole and then looking at nodes that were out of date as candidates for a shorter sync delay time *etc.* In hindsight I think that this would have been quite an undertaking, requiring significant time to develop and should not have been in the scope of this project.

6 Conclusion

In this dissertation I discussed the need for a private file synchronisation tool. I outlined the faults with current synchronisation tools such as Dropbox. My program aims to address these shortcomings. I detailed the implementation details of my program and the design decisions I made. I showed how I changed my design when I came across issues. I tested my program in as many different situations that I thought would be useful. I have shown that my program performs better than naïve copying, works in a variety of situations and behaves in general how I expect it too. I achieved my goals of creating a decentralised file synchronisation tool that will run across any network configuration. My program allows for fine-grained user control of settings such as which files should be replicated and how often they should be synchronised. My program generates logs of data transferred and the files being synchronised. This did not lead to detailed statistics about how the program behaved or heuristics on future behaviour as I intended. However I was able to generate graphs from my log files and this could be extended in a future implementation of my program. Once the user has written their configuration files the program operates autonomously as intended. My program has been a success and does its job of replicating files well.

A Program listings

A.1 WatchAndSync.py

```
1 import pyinotify, os, subprocess, argparse, socket, time, glob
   , datetime, thread
2 import readnet
3
4 #####
5 #                                     #
6 #   Main sync program               #
7 #   Calum O'Hare                    #
8 #   2012                            #
9 #                                     #
10 #####
11
12 wm = pyinotify.WatchManager()
13 watchedfolders = {}
14 homepath = "/home/cal/Documents/Private-Sync/"
15 #homepath = "/Users/calum/Documents/Private-Sync/"
16
17 parser = argparse.ArgumentParser()
18 parser.add_argument("-c", "--scp", action="store_true", help="
    Copy_using_scp")
19 parser.add_argument("-r", "--rsync", action="store_true", help="
    Copy_using_rsync")
20 args = parser.parse_args()
21
22 class Tools():
23     def updateFolderInfo(self, wfolders):
24         f = open('./folders.dat', 'w')
25         for fold in wfolders:
26             f.write(fold + "\n")
27             for i in range(0, len(wfolders[fold]) - 1):
28                 f.write(wfolders[fold][i] + "\n")
29             f.write(wfolders[fold][len(wfolders[fold]) - 1] + "\n")
30         f.close()
31
32     def timeElapsed(self, dtstamp, diff):
33         if diff == "*":
34             print "Sync ASAP"
35             return
36         diff = int(diff)
```

```

37 FMT = '%Y-%m-%d_%H:%M:%S.%f'
38 #FMT = '%Y-%m-%d %H:%M:%S'
39 tdelta = datetime.datetime.now() - datetime.datetime.
    strftime(dtstamp, FMT)
40 print tdelta.total_seconds()
41 timeDiff = tdelta.total_seconds()
42 if (timeDiff >= diff):
43     print "Time_period_reached"
44 else:
45     print "Time_not_elapsed, sleeping_for" + str(diff
        - timeDiff + 1)
46     time.sleep(int(diff - timeDiff + 1))
47
48 class MyEventHandler(pyinotify.ProcessEvent):
49     def flipIP(self, ip):
50         octets = ip.split(".")
51         if(octets[3] == "1"):
52             octets[3] = "2"
53         elif(octets[3] == "2"):
54             octets[3] = "1"
55         else:
56             octets[3] = "1"
57         return ".".join(octets)
58
59 #Get the last modified time of a file
60 def getModTime(self, path):
61     try:
62         return time.ctime(os.path.getmtime(path))
63     except Exception, e:
64         return time.ctime(0)
65
66 #Deprecated - Check for IP not to copy too
67 def getStopInfo(self):
68     stopIP = ["", ""]
69     try:
70         o = open("./stop", 'r')
71         stopIP = o.read().split()
72         o.close()
73     except IOError, e:
74         pass
75     return stopIP
76
77 def inStopFile(self, ip, path):

```

```

78     stopIPs = {}
79     stop = False
80     modTime = self.getModTime(path)
81     while True:
82         tmpcount = 0
83         print "Files found:_" + str(glob.glob("Stop-*"))
84         for files in glob.glob("Stop-*"):
85             #print "File: " + str(files)
86             if ".tmp" in files:
87                 tmpcount += 1
88                 time.sleep(5)
89                 break
90             f = open(files, "r");
91             for line in f:
92                 l = line.split()
93                 if self.exclusions(l[1]):
94                     print str(l[1]) + "_was_in_ignore_file_
                        _skipping"
95                 else:
96                     print "local_" + str(path) + "_modtime
                        :_" + modTime
97                     print "Stop_" + l[1] + "_modtime:_" +
                        str(l[2:])
98                     ts1 = time.strptime(modTime, "%a_%b_%d_
                        %H:%M:%S_%Y")
99                     ts2 = time.strptime(" ".join(l[2:]), "%
                        a_%b_%d_%H:%M:%S_%Y")
100                    print "local_<=stop:_" + str(ts1 <=
                        ts2)
101                    #if l[0] == ip and l[1] == path and
                        ts1 <= ts2:
102                        #If IP sending to has sent data more
                        recently don't send back
103                        if l[0] == ip and ts1 <= ts2:
104                            print "Stop_=True, _file:_" + l[0]
105                            stop = True
106                        else:
107                            stopIPs[l[0]] = [l[1], " ".join(l
                                [2:])]
108
109            if stop:
110                f.close()
111                #f = open(files, "w")

```

```

112         #for k in stopIPs.keys():
113         #    f.write(k + " " + stopIPs[k][0] + " "
114             + stopIPs[k][1] + "\n")
115         #f.close()
116         #stopIPs.clear()
117         return True
118
119     f.close()
120     #stopIPs.clear()
121     if tmpcount == 0:
122         break
123
124     return False
125
126     #Set flag on other server telling it not to immediately
127     try and copy data here
128     def setStopFileUniq(self, ip, myIP, path, folder):
129         nodename = self.getNodeName()
130         #print "ssh", ip, "echo " + myIP + " " + path + " " +
131             self.getModTime(path) + " >> " + hompath + "Stop-"
132             + nodename + ".tmp;"
133         #subprocess.call(["ssh", ip, "echo " + myIP + " " + path
134             + " " + self.getModTime(path) + " >> " + hompath +
135             "Stop-" + nodename + ".tmp;"])
136         subprocess.call(["ssh", ip, "rm_" + hompath + "Stop-" +
137             nodename + ".tmp;"])
138         for cpFile in glob.glob(folder + "/*"):
139             subprocess.call(["ssh", ip, "echo_" + myIP + "_" +
140                 cpFile + "_" + self.getModTime(cpFile) + " _>>_"
141                 + hompath + "Stop-" + nodename + ".tmp;"])
142
143     #Sets the config files on the remote node
144     def beginCopy(self, ip):
145         nodename = self.getNodeName()
146         print "ssh", ip, "touch_" + hompath + "Stop-" +
147             nodename + ".tmp; mv_" + hompath + "Stop-" +
148             nodename + "_" + hompath + "Stop-" + nodename + ".
149             tmp;"
150         subprocess.call(["ssh", ip, "touch_" + hompath + "Stop-"
151             + nodename + ".tmp; mv_" + hompath + "Stop-" +
152             nodename + "_" + hompath + "Stop-" + nodename + ".
153             tmp;"])

```

```

140 #Moves the Stop files back into place
141 def endCopy(self, ip):
142     nodename = self.getNodeName()
143     print "ssh",ip,"mv_" + hompath + "Stop-" + nodename +
        ".tmp_" + hompath + "Stop-" + nodename
144     subprocess.call(["ssh",ip,"mv_" + hompath + "Stop-" +
        nodename + ".tmp_" + hompath + "Stop-" + nodename
        ])
145
146 def setLastSync(self):
147     #print "echo \" + str(datetime.datetime.now())+ \"\ " >
        " + hompath + "lastSync"
148     #subprocess.call(["echo", str(datetime.datetime.now())
        + "> " + hompath + "lastSync"])
149     f = open(hompath + "lastSync","w")
150     f.write(str(datetime.datetime.now()))
151     f.close()
152
153 def getLastSync(self):
154     f = open(hompath + "lastSync","r")
155     time = f.read()
156     f.close()
157     return time.rstrip()
158
159 #Is the other node less fresh than this one?
160 def isnodefresh(self, ip):
161     print "scp","-rp",ip + ":" + hompath + "lastSync","/
        tmp/lastSync"
162     subprocess.call(["scp","-rp",ip + ":" + hompath + "
        lastSync","/tmp/lastSync"])
163
164     f = open("/tmp/lastSync","r")
165     remoteTime = f.read().rstrip()
166     f.close()
167
168     localTime = self.getLastSync()
169
170     stop = False
171     print "Local_sync_time:" + str(localTime)
172     print "Remote_sync_time:" + str(remoteTime)
173     FMT = '%Y-%m-%d_%H:%M:%S.%f'
174     #datetime.datetime.strptime(dtstamp, FMT)
175     ts1 = time.strptime(localTime,FMT)

```

```

176         #ts1 = time.strptime(remoteTime,"%a %b %d %H:%M:%S %Y
            ")
177         ts2 = time.strptime(remoteTime,FMT)
178         print "local_<=_stop:_" + str(ts1 <= ts2)
179         if ts1 <= ts2:
180             print "local_sync_<=_than_remote_sync!"
181             stop = True
182         return stop
183
184     #Is a given file newer than the last sync time?
185     def newerThanLast(self, fileName):
186         stop = False
187         print "Last_sync_time:_" + str(self.getLastSync())
188         FMT = '%Y-%m-%d_%H:%M:%S.%f'
189         #datetime.datetime.strptime(dtstamp, FMT)
190         ts2 = time.strptime(self.getLastSync(),FMT)
191         modTime = self.getModTime(fileName)
192         print "local_" + str(fileName) + "_modtime:_" +
            modTime
193         ts1 = time.strptime(modTime,"%a_%b_%d_%H:%M:%S_%Y")
194         print "local_<=_stop:_" + str(ts1 <= ts2)
195         if ts1 > ts2:
196             print "Newer_file_than_last_sync!"
197             stop = True
198         return stop
199
200
201     #Get node name from whoami file
202     def getNodeName(self):
203         w = open(homepath + "whoami","r")
204         nodename = w.read()
205         nodename = nodename[0].upper()
206         w.close()
207         return nodename
208
209     #Deprecated stop file
210     def setStopFile(self,ip,myIP,path):
211         subprocess.call(["ssh",ip,"echo_" + myIP + "_" + path
            + ">_" + homepath + "stop"])
212         print "ssh",ip,"echo_" + myIP + ">_" + homepath + "
            stop"
213
214     def rmTree(self, path):

```

```

215         subprocess.call(["ssh",ip,"rm-r_" + path + ""'])
216     print "ssh",ip,"rm-r_" + path + ""'
217
218     #Exclude files matching patterns in the ignore file
219     def exclusions(self, path):
220         try:
221             f = open("./ignore",'r')
222             for line in f:
223                 if line.rstrip() in path:
224                     #print "Ignoring: " + path
225                     return True
226             f.close()
227         except error, e:
228             print e
229         return False
230
231     def initFileSync(self, event):
232         if self.exclusions(event.pathname):
233             #print "Excluded returning"
234             return
235         pathparts = event.pathname.split("/")
236         foldName = "/" .join(pathparts[0:len(pathparts)-1])
237
238         print "Removing_watch_on:_" + foldName
239         wm.rm_watch(wm.get_wd(foldName), rec=True)
240
241         self.setLastSync()
242         self.fileSync(event.pathname)
243
244         print "Putting_watch_back_on:_" + foldName
245         wm.add_watch(foldName.rstrip(),pyinotify.ALL_EVENTS,
246                     rec=True, auto_add=True)
247
248         for i in range(0, len(watchedfolders[foldName]),4):
249             # ip = watchedfolders[foldName][i]
250             # myIP = readnet.getMyIP(ip)
251             for f in glob.glob(foldName + "/*"):
252                 if self.newerThanLast(f):
253                     print "init:_CONTINUE"
254                     self.setLastSync()
255                     self.fileSync(f)
256                 else:
257                     print "init:_STOP"

```

```

257
258 #Send the data
259 def copyData(self ,pathname , folder , t , i):
260     ip = watchedfolders[folder][i]
261     path = watchedfolders[folder][i+1]
262     waitTime = watchedfolders[folder][i+2]
263     lastTime = watchedfolders[folder][i+3]
264     print "Wait:_ " + str(waitTime) + "_Last:_ " + str(
        lastTime)
265     print "Current_ip_and_path:_ " + ip + "_ " + path
266     readnet.logIPtraffic(ip , pathname)
267     myIP = readnet.getMyIP(ip)
268     subprocess.call(["ssh",ip,"/usr/bin/python_" +
        homopath + "readnet.py_i_" + myIP + "_f_" +
        pathname])
269     print "ssh",ip,"'/usr/bin/python_" + homopath + "
        readnet.py_i_" + myIP + "_f_" + pathname + "'
270     fparts = folder.split("/")
271     fname = fparts[len(fparts)-1]
272     #stopIP = self.getStopInfo()
273     #print "STOP: " + stopIP[0] + " " + stopIP[1]
274     #if stopIP[0] == ip and stopIP[1] == pathname:
275     if self.inStopFile(ip ,pathname):
276         print "STOPPED_to_" + ip + "_ " + path
277         #os.remove("./stop");
278     else:
279         print "CONTINUE"
280         t.timeElapsed(lastTime , waitTime)
281
282     #if self.isnodefresh(ip):
283     #     print "Remote host has changes, stoping..."
284     #     return
285
286     watchedfolders[folder][i+3] = str(datetime.
        datetime.now())
287     t.updateFolderInfo(watchedfolders)
288     self.beginCopy(ip)
289     self.beginCopy(myIP)
290     if args.scp:
291         #print "SCP: For cpFile in " + folder
292         for cpFile in glob.glob(folder + "/*"):
293             #print "SCP GLOB:" + cpFile

```



```

294         print "scp", "-rp", cpFile, ip + ":" + cpFile
           + ".tmp"
295         subprocess.call(["scp", "-rp", cpFile, ip + ":"
           + cpFile + ".tmp"])
296         #subprocess.call(["ssh", ip, "yes y | find /
           tmp/" + fname + "-type f -exec cp -p {
           " + path + fname + "/ \; rm /tmp/" +
           fname])
297         print "ssh", ip, "mv_" + cpFile + ".tmp_" +
           cpFile
298         subprocess.call(["ssh", ip, "mv_" + cpFile +
           ".tmp_" + cpFile])
299         print "END_SCP_GLOB"
300     elif args.rsync:
301         print "rsync", "-rt", folder, ip + ":" + path
302         subprocess.call(["rsync", "-rt", folder, ip + ":"
           + path])
303     else:
304         #time.sleep(5)
305         print "unison", "-batch", "-confirmbigdel=false"
           , "-times", folder, "ssh://" + ip + "/" + path
           + fname
306         subprocess.call(["unison", "-batch", "-confirmbigdel=false", "-times", folder, "ssh://"
           + ip + "/" + path + fname])
307         print "Set_stop_files_uniq:" + pathname
308         #Set stop file on foreign host
309         self.setStopFileUniq(ip, myIP, pathname, folder)
310         #Set stop file for myself to look at
311         self.setStopFileUniq(myIP, myIP, pathname, folder)
312         self.endCopy(ip)
313         self.endCopy(myIP)
314         subprocess.call(["ssh", ip, "/usr/bin/python_" +
           homopath + "readnet.py_-i_" + myIP + "_-f_" +
           pathname])
315         readnet.logIPtraffic(ip, pathname)
316
317     #Sync the files
318     def fileSync(self, pathname):
319         t = Tools()
320         if os.path.isdir(pathname):
321             print "Watching:", pathname
322         for folder in watchedfolders.keys():

```

```

323         print "For_each_folder:_" + str(folder) + "_in_"
           watchedfolder_keys"
324     if folder in pathname:
325         for i in range(0, len(watchedfolders[folder])
           ,4):
326             #self.copyData(pathname, folder, t, i)
327             #Start new thread
328             thread.start_new_thread(self.copyData, (
           pathname, folder, t, i))
329
330     #Handler functions
331     def process_IN_DELETE(self, event):
332         print "Delete:_" ,event.pathname
333         #self.initFileSync(event)
334     def process_IN_CREATE(self, event):
335         print "CREATE:_" ,event.pathname
336         #thread.start_new_thread(self.initFileSync, (event,))
337         self.initFileSync(event)
338     def process_IN_MOVED_FROM(self, event):
339         print "Move_from:_" ,event.pathname
340         #self.initFileSync(event)
341     def process_IN_MODIFY(self, event):
342         print "Modify:_" ,event.pathname
343         self.initFileSync(event)
344         #thread.start_new_thread(self.initFileSync, (event,))
345     def process_IN_MOVED_TO(self, event):
346         print "Move_to:_" ,event.pathname
347         self.initFileSync(event)
348         #thread.start_new_thread(self.initFileSync, (event,))
349
350
351     def main():
352         t = Tools()
353         f = open('./folderstowatch', 'r')
354
355         for folder in f:
356             if(folder[0] == '#'):
357                 pass
358             else:
359                 info = folder.split()
360                 wm.add_watch(info[0].rstrip(), pyinotify.ALLEVENTS
           , rec=True, auto_add=True)
361                 print "Watching:_" , info[0].rstrip()

```

```

362         if info[0] not in watchedfolders.keys():
363             watchedfolders[info[0].rstrip()] = []
364             watchedfolders[info[0].rstrip()].append(info[1])
365             watchedfolders[info[0].rstrip()].append(info[2])
366             watchedfolders[info[0].rstrip()].append(info[3])
367             watchedfolders[info[0].rstrip()].append(str(
                 datetime.datetime.now()))
368     f.close()
369
370     try:
371         f = open('./folders.dat', 'r')
372         for folder in f:
373             if(folder[0] == '#'):
374                 pass
375             else:
376                 info = folder.split()
377                 if info[0] in watchedfolders.keys():
378                     del watchedfolders[info[0].rstrip()]
379                     #wm.add_watch(info[0].rstrip(), pyinotify.
                        ALL_EVENTS, rec=True, auto_add=True)
380                     #print "Watching: ", info[0].rstrip()
381                     if info[0] not in watchedfolders.keys():
382                         watchedfolders[info[0].rstrip()] = []
383                         watchedfolders[info[0].rstrip()].append(
                             info[1])
384                         watchedfolders[info[0].rstrip()].append(
                             info[2])
385                         watchedfolders[info[0].rstrip()].append(
                             info[3])
386                         watchedfolders[info[0].rstrip()].append(
                             str(datetime.datetime.now()))
387                 else:
388                     print "Removing:_" + info[0]
389             f.close()
390     except IOError, e:
391         print "Folders.dat_does_not_exist, _skipping"
392
393     t.updateFolderInfo(watchedfolders)
394
395     #print watchedfolders
396     eh = MyEventHandler()
397
398     notifier = pyinotify.Notifier(wm, eh)

```

```
399     notifier.loop()
400
401 if __name__ == '__main__':
402     main()
```

A.2 ReadNet.py

```
1 import subprocess, datetime, socket, argparse
2
3 #####
4 #                                     #
5 #   Read Network Data               #
6 #   Calum O'Hare                   #
7 #   2012                           #
8 #                                     #
9 #####
10
11 homopath = "/home/cal/Documents/Private-Sync/"
12 #homopath = "/Users/calum/Documents/Private-Sync/"
13
14 parser = argparse.ArgumentParser()
15 parser.add_argument('-i', action="store", dest='ip', help='IP_
    address_to_record_for')
16 parser.add_argument('-f', action="store", dest='fold', help='
    Folder_to_record_for')
17
18 interfacenames = []
19
20 w = open(homopath + "whoami", "r")
21 nodename = w.read()
22 nodename = nodename[0]
23 w.close()
24
25 #Get my ip corresponding to the interface with ipaddr
26 def getMyIP(ipaddr):
27     route = subprocess.check_output("ip_route_get_" + ipaddr,
        shell=True)
28     words = route.split()
29     interface = ""
30     for word in words:
31         if word.startswith("eth"):
32             interface = word
33             #print interface
34             break
35     ifconf = subprocess.check_output("ifconfig_" + interface,
        shell=True)
36     words = ifconf.split()
37     now = False
```

```

38     for word in words:
39         if word == "inet":
40             now = True
41         elif now:
42             word = word.split(":")
43             #print word[1]
44             return word[1]
45
46 #Log interface coresponding to ipaddr
47 def logIPtraffic(ipaddr, folder):
48     route = subprocess.check_output("ip route get " + ipaddr,
49                                     shell=True)
49     words = route.split()
50     interface = ""
51     for word in words:
52         if word.startswith("eth"):
53             interface = word
54             #print interface
55             break
56     writeIface(interface, folder)
57
58 #Write the upload/download data for a given interface and
folder
59 def writeIface(iface, folder):
60     ifs = subprocess.check_output("ifconfig -s", shell=True)
61     ilines = ifs.split("\n")
62     for i in range(1, len(ilines)-1):
63         interfacenames.append(ilines[i].split()[0])
64     output = subprocess.check_output("ifconfig", shell=True)
65     splitput = output.split()
66     interface = False
67     interfacename = ""
68     nex = ""
69     count = 0
70     upload = 0
71     download = 0
72     for split in splitput:
73         if split in interfacenames:
74             interface = True
75             interfacename = split
76             #print interfacename
77         if (nex != ""):
78             sp = split.split(":")

```

```

79         if(sp[0] == "bytes"):
80             if(nex == "RX"):
81                 download = int(sp[1])
82             else:
83                 upload = int(sp[1])
84             nex = ""
85             count += 1
86             if(count == 2):
87                 interface = False
88                 if interfacename == iface:
89                     f = open(homepath + "log/" \
90                         + "node" + nodename.upper() + "-" \
91                         + iface + ".log", 'a')
92                     f.write("#D_" + folder + "\n")
93                     f.write(str(datetime.datetime.now()) +
94                         "_" + interfacename + "_download:" +
95                         str(download) + "_upload:" + str
96                         (upload) + "\n")
97                     f.close()
98                 count = 0
99             elif(interface):
100                 if(split == "RX" or split == "TX"):
101                     nex = split
102
103 #Log all interfaces
104 def main():
105     ifs = subprocess.check_output("ifconfig -s", shell=True)
106     ilines = ifs.split("\n")
107     for i in range(1, len(ilines)-1):
108         interfacenames.append(ilines[i].split()[0])
109     output = subprocess.check_output("ifconfig", shell=True)
110     splitput = output.split()
111     interface = False
112     interfacename = ""
113     nex = ""
114     count = 0
115     upload = 0
116     download = 0
117     for split in splitput:
118         if split in interfacenames:
119             interface = True
120             interfacename = split
121             #print interfacename

```

```

119     if(nex != ""):
120         sp = split.split(":")
121         if(sp[0] == "bytes"):
122             if(nex == "RX"):
123                 download = int(sp[1])
124             else:
125                 upload = int(sp[1])
126         nex = ""
127         count += 1
128         if(count == 2):
129             interface = False
130             f = open(homepath + "log/" \
131 + str(socket.gethostname()) + "-" \
132 + interfacename + ".log", 'a')
133             f.write(str(datetime.datetime.now()) + "_"
134 + interfacename + "_download:" + str(
135 + download) + "_upload:" + str(upload) +
136 + "\n")
137             f.close()
138             count = 0
139         elif(interface):
140             if(split == "RX" or split == "TX"):
141                 nex = split
142
143 if __name__ == "__main__":
144     args = parser.parse_args()
145     if args.ip != None:
146         logIPtraffic(args.ip, args.fold)
147         #getMyIP(args.ip)
148     else:
149         pass
150     main()

```


A.3 ManipVMs.sh

```
1 #####
2 #                                     #
3 #      Manipulate VM's              #
4 #      Calum O'Hare                 #
5 #      2012                         #
6 #                                     #
7 #####
8 vm_name_arr=("Ubuntu-Pool" "Ubuntu-Silence" "Ubuntu-Black" "
      Ubuntu-Spheros" "Ubuntu-Wild")
9 vm_addr_arr=("192.168.0.28" "192.168.0.27" "192.168.0.30" "
      192.168.0.14")
10 #Wild = 19
11 intnetarr=("lion" "tiger" "cat" "dog" "fish" "kiwi" "swish" "
      boom" "roar")
12 #These should all be in one big dictionary apart from inet
      names
13 letterarr=("a" "b" "c" "d" "e" "f" "g" "h" "i" "j")
14 ifcountarr=(2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2)
15 ethcountarr=(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
16 incount=1
17 bigncount=2
18 littlencount=1
19 folderpath="/home/cal/Documents/t26"
20 folderpath2="/home/cal/Documents/t02"
21 homepath="/home/cal/Documents/Private-Sync/"
22 waitTime=10
23
24 function clear_ifaces() {
25     i=0
26     while [ "$i" -lt "${#vm_name_arr[@]}" ]; do
27         VBoxManage modifyvm ${vm_name_arr[$i]} --nic2 none
28         echo "VBoxManage modifyvm_${vm_name_arr[$i]} --nic2_
            none"
29         VBoxManage modifyvm ${vm_name_arr[$i]} --nic3 none
30         echo "VBoxManage modifyvm_${vm_name_arr[$i]} --nic3_
            none"
31         VBoxManage modifyvm ${vm_name_arr[$i]} --nic4 none
32         echo "VBoxManage modifyvm_${vm_name_arr[$i]} --nic4_
            none"
33         let "i++"
34     done
```

```

35 }
36
37 function clear_watched_folders() {
38     i=0
39     while [ "$i" -lt "${#vm_addr_arr[@]}" ]; do
40         ssh cal@${vm_addr_arr[$i]} "echo \">#Local folder path
            to watch, host to copy to, remote dir to copy to,
            min time between syncs\#" > /home/cal/Documents/
            Private-Sync/folderstowatch; echo ${letterarr[$i]} >
            /home/cal/Documents/Private-Sync/whoami"
41         let "i++"
42     done
43 }
44
45 function git_pull() {
46     i=0
47     while [ "$i" -lt "${#vm_addr_arr[@]}" ]; do
48         echo "ssh cal@${vm_addr_arr[$i]} \#"cd /home/cal/
            Documents/Private-Sync; git pull origin master\#"
49         ssh cal@${vm_addr_arr[$i]} "cd /home/cal/Documents/
            Private-Sync; git pull origin master"
50         let "i++"
51     done
52 }
53
54 function search_letters() {
55     index=0
56     while [ "$index" -lt "${#letterarr[@]}" ]; do
57         if [ "${letterarr[$index]}" = "$1" ]; then
58             echo $index
59             return
60         fi
61         let "index++"
62     done
63     echo "None"
64 }
65
66 function vbmMOD {
67     #Set the NIC to intnet on the VM and attach it to the
        given network
68     echo "VBoxManage modifyvm $1 --nic$3 intnet"
69     VBoxManage modifyvm $1 --nic$3 intnet
70     echo "VBoxManage modifyvm $1 --intnet$3 $2"

```

```

71     VBoxManage modifyvm $1 --intnet$3 $2
72 }
73
74 function gatherLogs {
75     index=0
76     while [ "$index" -lt "${#vm_addr_arr[@]}" ]; do
77         echo "scp cal@${vm_addr_arr[$index]}:/home/cal/
78             Documents/Private-Sync/log/*../logs/"
79         scp cal@${vm_addr_arr[$index]}:/home/cal/Documents/
80             Private-Sync/log/*../logs/
81         let "index++"
82     done
83 }
84
85 function clean {
86     index=0
87     while [ "$index" -lt "${#vm_addr_arr[@]}" ]; do
88         echo "ssh cal@${vm_addr_arr[$index]} \"rm ${homepath}
89             log/*; rm ${homepath}Stop-*; rm ${homepath}folders.
90             dat\""
91         ssh cal@${vm_addr_arr[$index]} "rm ${homepath}log/*;
92             rm ${homepath}Stop-*; rm ${homepath}folders.dat"
93         let "index++"
94     done
95 }
96
97 function cleanFold {
98     index=0
99     while [ "$index" -lt "${#vm_addr_arr[@]}" ]; do
100         echo "ssh cal@${vm_addr_arr[$index]} \"rm -rf ${
101             folderpath}/*;\"
102         ssh cal@${vm_addr_arr[$index]} "rm -rf ${folderpath
103             }/*;"
104         let "index++"
105     done
106 }
107
108 function sendKeys {
109     index=0
110     while [ "$index" -lt "${#vm_addr_arr[@]}" ]; do
111         #ssh cal@${vm_addr_arr[$index]} "rm /home/cal/.ssh/
112             authorized_keys"
113         for file in /Users/calum/.ssh/*.pub; do

```

```

106         #echo "$file"
107         echo "cat _$file _| _ssh _cal@${vm_addr_arr[$index]} _\
        "cat >> /home/cal/.ssh/authorized_keys\"
108         cat $file | ssh cal@${vm_addr_arr[$index]} "cat _>>
        _/home/cal/.ssh/authorized_keys"
109     done
110     let "index++"
111 done
112 #for file in /Users/calum/.ssh/*.pub; do
113 #     echo "$file"
114 #     cat $file | ssh cal@192.168.0.17 "cat >> /home/cal/.ssh/testfile"
115 #     echo "cat $file | ssh cal@192.168.0.17 \"cat >> /home/cal/.ssh/testfile\""
116 #done
117 }
118
119 function ifconf {
120     echo "ssh _cal@$1 _'sudo _/sbin/ifconfig _eth$2 _192.168.$3.$4 _
        netmask _255.255.255.0 _up; _echo _\"$folderpath 192.168.$3.
        $5 /home/cal/Documents/ $waitTime\" _>> _/home/cal/
        Documents/Private-Sync/folderstowatch'"
121     ssh cal@$1 "sudo _/sbin/ifconfig _eth$2 _192.168.$3.$4 _
        netmask _255.255.255.0 _up; _echo _\"$folderpath 192.168.$3.
        $5 /home/cal/Documents/ $waitTime\" _>> _/home/cal/
        Documents/Private-Sync/folderstowatch" < /dev/null
122 }
123
124 function ifconf2 {
125     echo "ssh _cal@$1 _\"sudo /sbin/ifconfig eth$2 192.168.$3.$4
        netmask 255.255.255.0 up; echo \" $folderpath _192.168.$3
        . $5 _/home/cal/Documents/ _*\" >> /home/cal/Documents/
        Private-Sync/folderstowatch; echo \" $folderpath2 _
        192.168.$3. $5 _/home/cal/Documents/ _*\" >> /home/cal/
        Documents/Private-Sync/folderstowatch\" _< _/dev/null"
126     ssh cal@$1 "sudo _/sbin/ifconfig _eth$2 _192.168.$3.$4 _
        netmask _255.255.255.0 _up; _echo _\"$folderpath 192.168.$3.
        $5 /home/cal/Documents/ *\" _>> _/home/cal/Documents/
        Private-Sync/folderstowatch; _echo _\"$folderpath2
        192.168.$3. $5 /home/cal/Documents/ *\" _>> _/home/cal/
        Documents/Private-Sync/folderstowatch" < /dev/null
127 }
128

```

```

129 if [ $2 == "vm" ]; then
130     clear_ifaces
131
132     #Read in the DOT script
133     while read line
134     do
135         first=$(echo "$line" | awk '{print $1}')
136         last=$(echo "$line" | awk '{print $(NF)}' | sed 's
            /[:]//g')
137         #echo "$first and $last"
138         index=$(search_letters $first)
139         if [ "$index" = "None" ]; then
140             #echo "None"
141             :
142         else
143             vmMOD ${vm_name_arr[$index]} ${intnetarr[$incount
                ]} ${ifcountarr[$index]}
144             #echo "in: $index"
145             (( ifcountarr[$index]++ ))
146             index=$(search_letters $last)
147             vmMOD ${vm_name_arr[$index]} ${intnetarr[$incount
                ]} ${ifcountarr[$index]}
148             #echo "in: $index"
149             (( ifcountarr[$index]++ ))
150             incount=incount+1
151         fi
152     done <graphs/$1
153 elif [ $2 == "if" ]; then
154     clear_watched_folders
155
156     while read line
157     do
158         first=$(echo "$line" | awk '{print $1}')
159         last=$(echo "$line" | awk '{print $(NF)}' | sed 's
            /[:]//g')
160         echo "$first_and_$last"
161         index=$(search_letters $first)
162         if [ "$index" = "None" ]; then
163             #echo "None"
164             :
165         else
166             ifconf ${vm_addr_arr[$index]} ${ethcountarr[$index
                ]} $bigncount $littlencount $(( $littlencount+1

```

```

    ))
167     #echo "in: $index"
168     (( ethcountarr[$index]++ ))
169     (( littlencount++ ))
170     index=$(search_letters $last)
171     ifconf ${vm_addr_arr[$index]} ${ethcountarr[$index]
        } $bigncount $littlencount $(( $littlencount-1
        ))
172     #echo "in: $index"
173     (( ethcountarr[$index]++ ))
174     incount=$incount+1
175     (( bigncount++ ))
176     (( littlencount — ))
177     fi
178     done <graphs/$1
179 elif [ $2 == "if2" ]; then
180     clear_watched_folders
181
182     while read line
183     do
184         first=$(echo "$line" | awk '{print $1}')
185         last=$(echo "$line" | awk '{print $(NF)}' | sed 's
            /[:]//g')
186         echo "$first_and_$last"
187         index=$(search_letters $first)
188         if [ "$index" = "None" ]; then
189             #echo "None"
190             :
191         else
192             ifconf2 ${vm_addr_arr[$index]} ${ethcountarr[
                $index]} $bigncount $littlencount $((
                $littlencount+1 ))
193             #echo "in: $index"
194             (( ethcountarr[$index]++ ))
195             (( littlencount++ ))
196             index=$(search_letters $last)
197             ifconf2 ${vm_addr_arr[$index]} ${ethcountarr[
                $index]} $bigncount $littlencount $((
                $littlencount-1 ))
198             #echo "in: $index"
199             (( ethcountarr[$index]++ ))
200             incount=$incount+1
201             (( bigncount++ ))

```

```

202             (( littlencount — ))
203         fi
204     done <graphs/$1
205 elif [ $2 == "key" ]; then
206     sendKeys
207 elif [ $2 == "gather" ]; then
208     gatherLogs
209 elif [ $2 == "clean" ]; then
210     clean
211 elif [ $2 == "pull" ]; then
212     git_pull
213 elif [ $2 == "clean-fold" ]; then
214     cleanFold
215 elif [ $2 == "help" ]; then
216     echo "vm—————setup vm networking"
217     echo "if—————setup network addresses etc for each
vm"
218     echo "if2—————setup network addresses etc for each
vm for two folders"
219     echo "gather————gather the logs in"
220     echo "clean————clean out the logs/config files"
221     echo "clean-fold——clean out the files folder"
222     echo "pull————pull the latest code from the
repository to each vm"
223     echo "help————display this help message"
224 else
225     echo "Oops try again"
226 fi
227
228 neato -Teps graphs/$1 > graphs/$1-graph.eps

```