Private Dropbox
Final Report
COSC480

Calum O'Hare
Supervisor: David Eyers

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 machines. It reads from configuration files to allow the user to specify what files should be synchronised; to which machines; and how often to sync. My program keeps track of the files being synchronised and the last time files were updated so that it knows when to sync with any other connected machines. My approach to this program was that different machines may require different replication strategies, this makes it unique from other file replication tools.

Contents

1 Introduction

Dropbox is a well known online file hosting tool which 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, efficiency (time taken for the files to become fully up to date), 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 too then 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 files[1].

There are other possible approaches to replicating files across multiple computers. For example you could use version control systems like Subversion, Mercurial, 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, 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 around $\approx$5MB and I like to have a relatively current backup of this collection.

---

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

My 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. Something which Dropbox will not provide but my system does.

(a) Personal Network
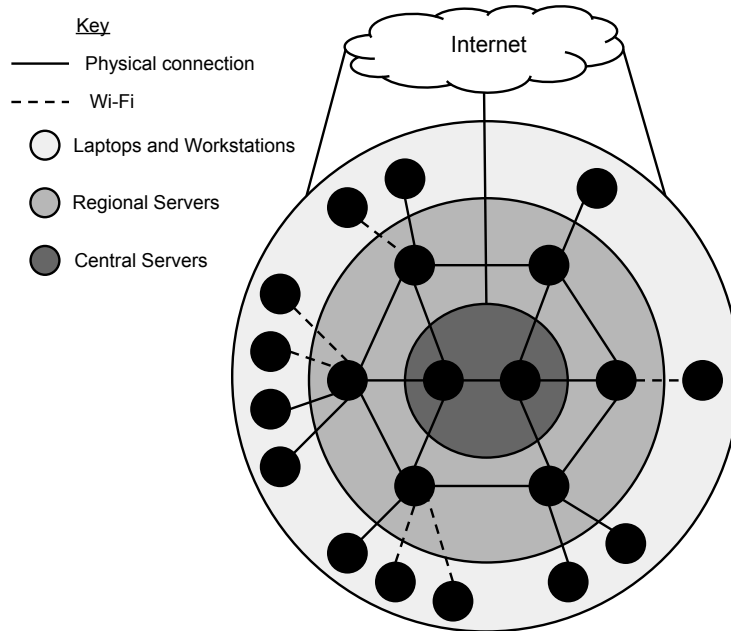
(b) Corporate Network

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 125 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 60). It does this by calling the `VboxManage`, command which

---

[2]www.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 Wireshark[4] 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.

(a) Line topology

(b) Circle topology

(c) Connected mesh topology

(d) More complex topology
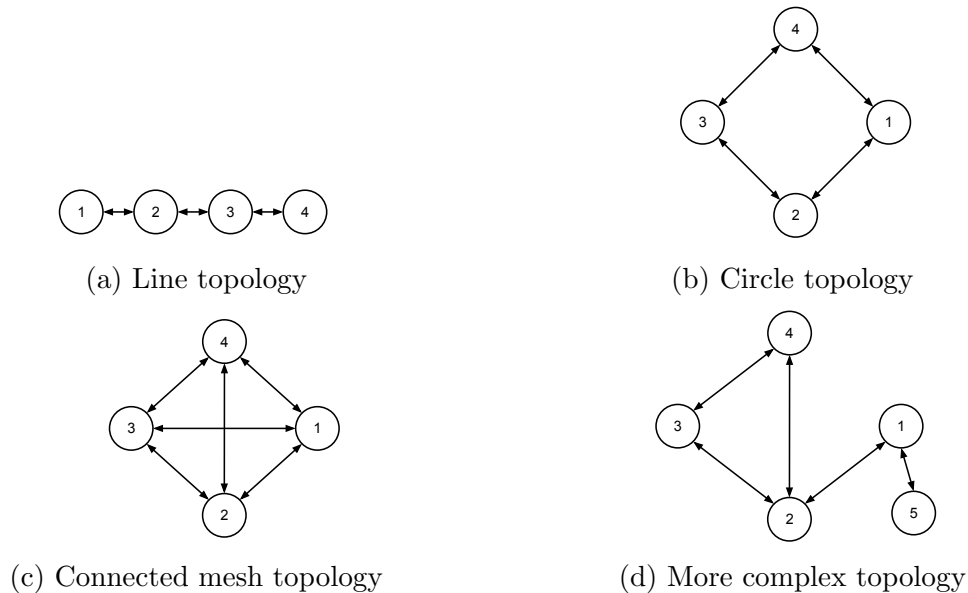
Figure 2: Simple network topologies

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

---

[4]http://www.wireshark.org/

other processes are using the network connection as you would expect in a real world application of my program. I found a tool called NetHogs[5]. 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 92). 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 38). 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 3.4 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   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.

3.3   Monitoring Directories

The application needs to monitor directories for changes (see section 3.4) 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 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 however 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. Inotify only uses one file descriptor and returns events to the listener as they occur [6] There is a Python module called Pyinotify[7] 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 X[8] 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 MacFSEvents[9].
  I also looked at using the `kqueue` [10] system call that is supported by MacOS X and FreeBSD. It notifies the user when a kernel event occurs. 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 scope of the project. I would have liked to look at it further but became too time consuming and not interesting from a research perspective.

3.4   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 3.5) I wanted to see how the freshness of different sub-nodes varied

---

[6]www.kernel.org/pub/linux/kernel/people/rml/inotify/README

[7]http://pyinotify.sourceforge.net/

[8]https://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/

[9]http://pypi.python.org/pypi/MacFSEvents/0.2.1

[10]http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man2/kqueue.2.html

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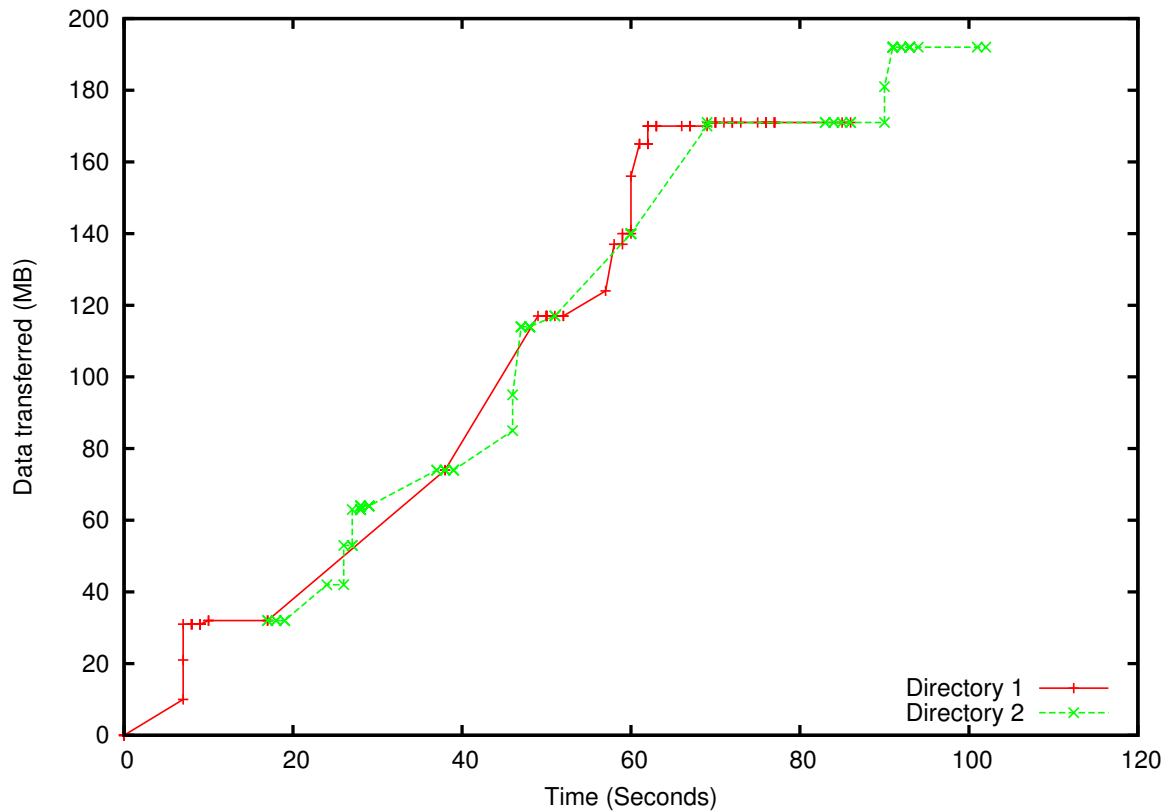
Figure 3: Line topology synchronisation using Unison. Two different directories are being synchronised with 10MB files are sent to each directory alternately every ten seconds.

3.5 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 filename. 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 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

After looking for cross-platform, open source, file synchronisation tools, I have found a tool called Unison[11] 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. 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 described above.

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

---

[11]http://www.cis.upenn.edu/~bcpierce/unison/
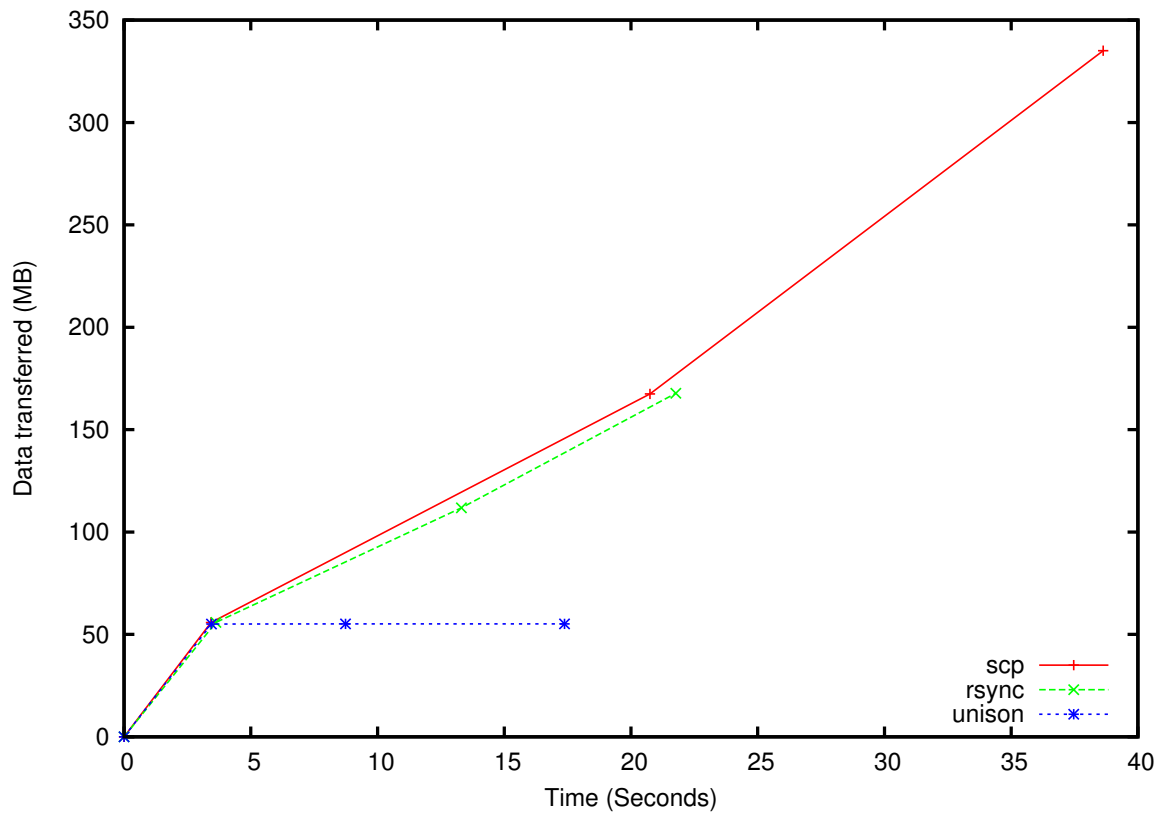
Figure 4: Comparison of SCP, Rsync, Unison. Three identical 50MB files with different names are being transfered between two nodes. Each point on the graph represents the start or end of a sync.

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 ≈30MB, 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.
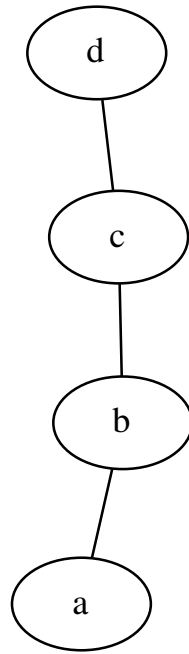
Figure 5: Generated graph of line topology

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 ≈30MB 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
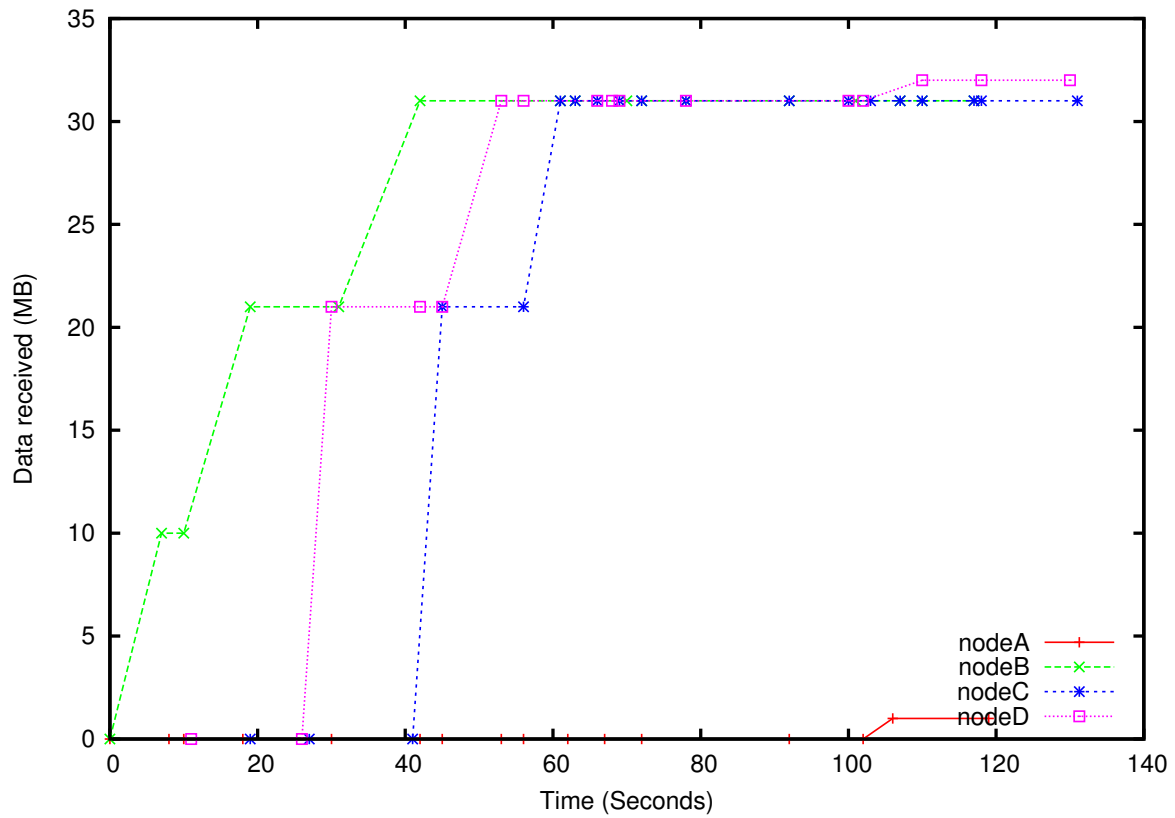
Figure 10: My program running using Unison on a circle topology. A 10MB file filled with random data is being sent every 10 seconds to node A for thirty seconds.
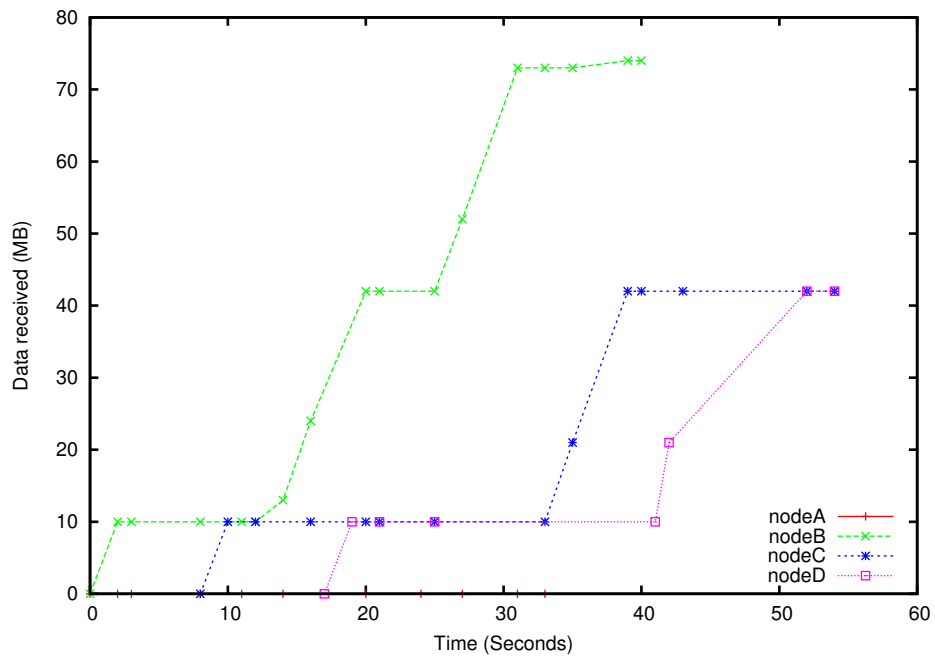
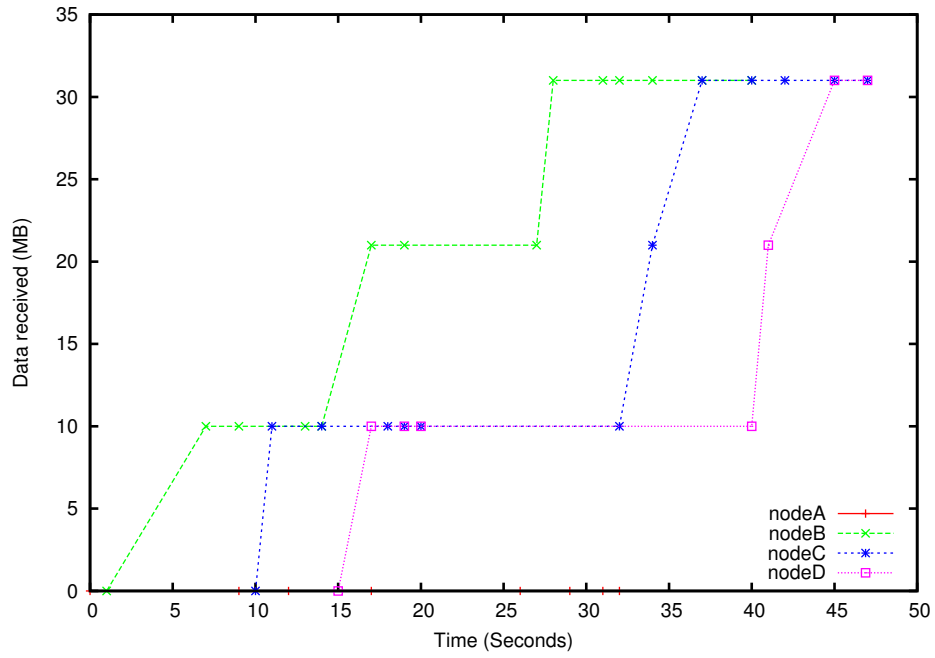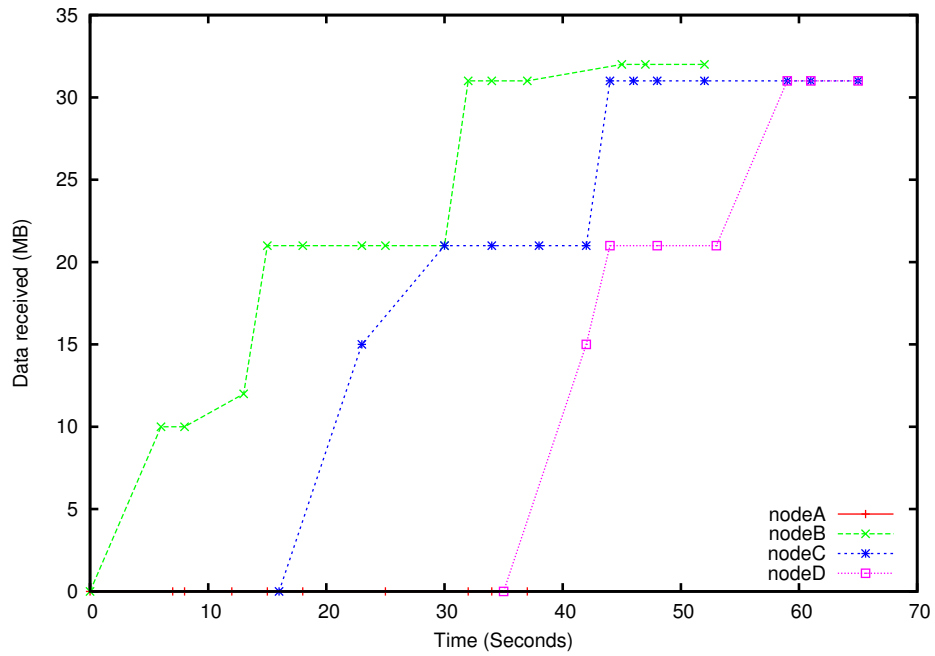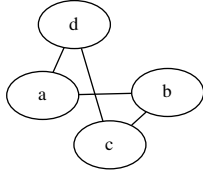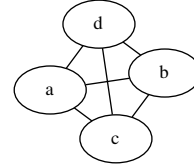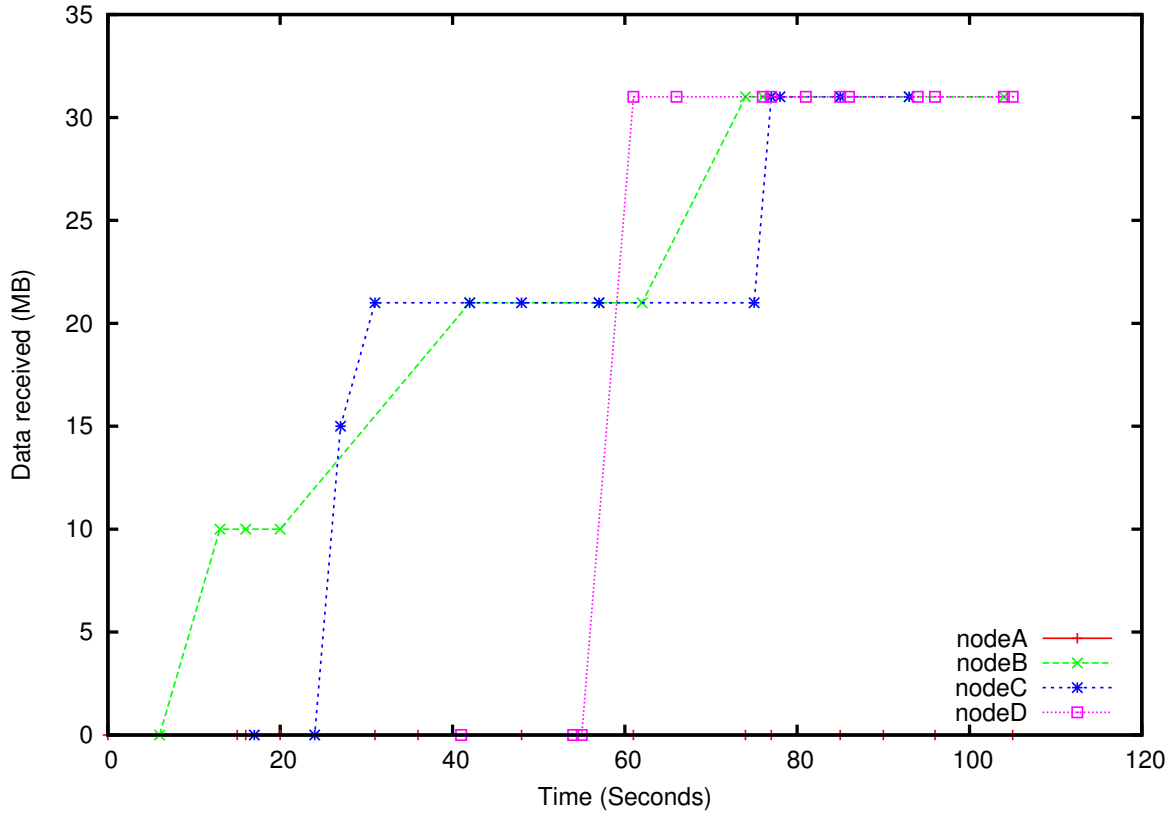Figure 11: My program running using Unison on a mesh topology. A 10MB file filled with random data is being sent every 10 seconds to node A for thirty seconds.

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 connected nodes. After testing my program on some simple topologies one problem became clear. The problem was that if the changes came from one of its neighbour nodes this would cause an infinite loop of two nodes trying to copy changes to each other. This was particularly a problem when using SCP to copy. When using Unison this was not as much of a problem because it could detect that no changes had occurred between the nodes and would stop synchronising after one check (which had minimal overhead).

Figure 12 shows three 10MB files being copied to node A in a line topology. The problem is that node B and node C continue to send data to each other even after every node has all of the files. NodeA receives a lot of data even though it was the source of the file changes.

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.

To fix this problem I used a control file. Each time a node synchronised with another
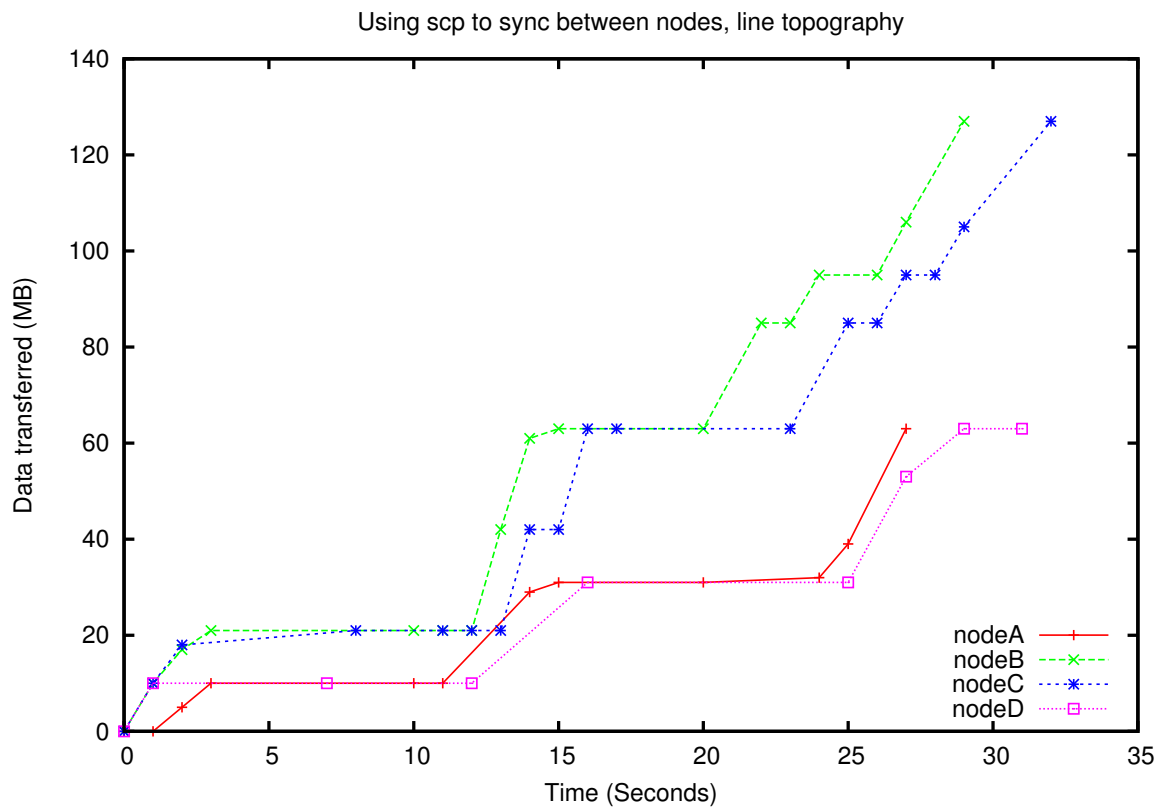
Figure 12: Line topology, using SCP, nodes copying data back and forth
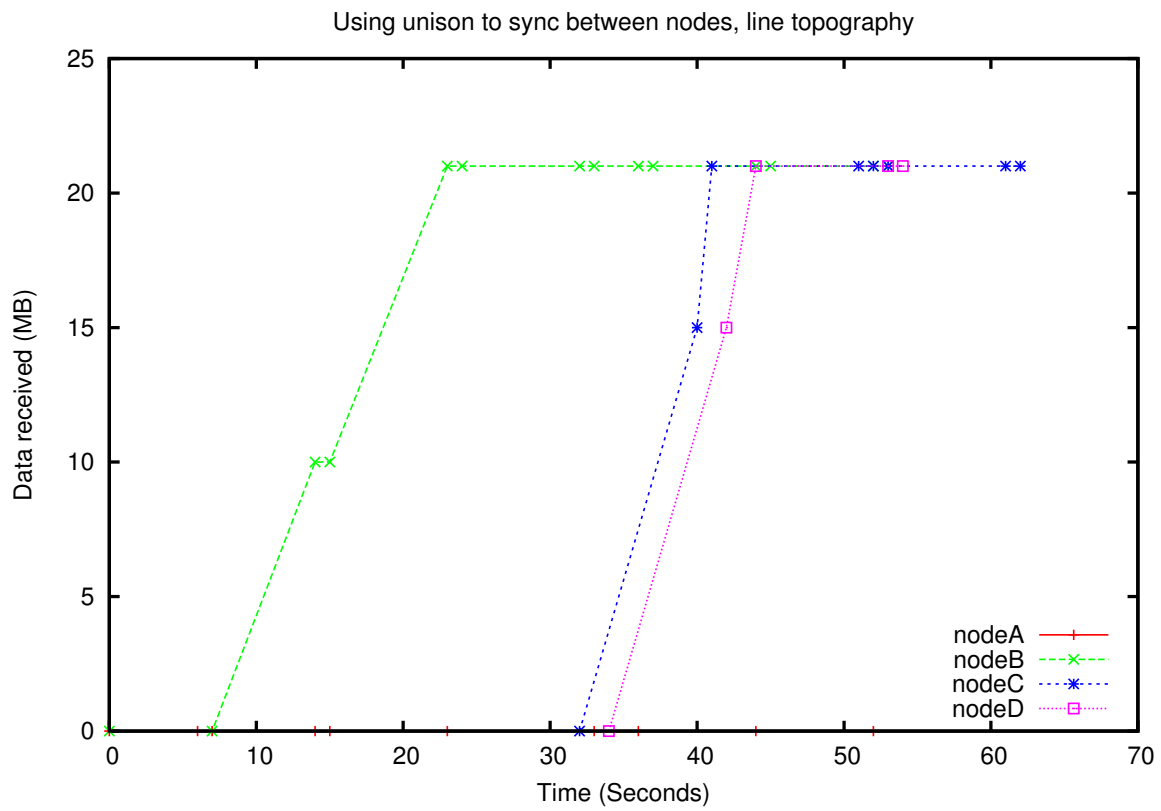
Figure 13: Line topology, using Unison, program continues to check for differences in files even after all nodes are the same. Each point represents an attempted sync.

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 a seemingly random fashion with some nodes receiving less data than others. 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 by the user 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 be set to more than the frequency that the files are changing but not too high as that slows down the system. 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.
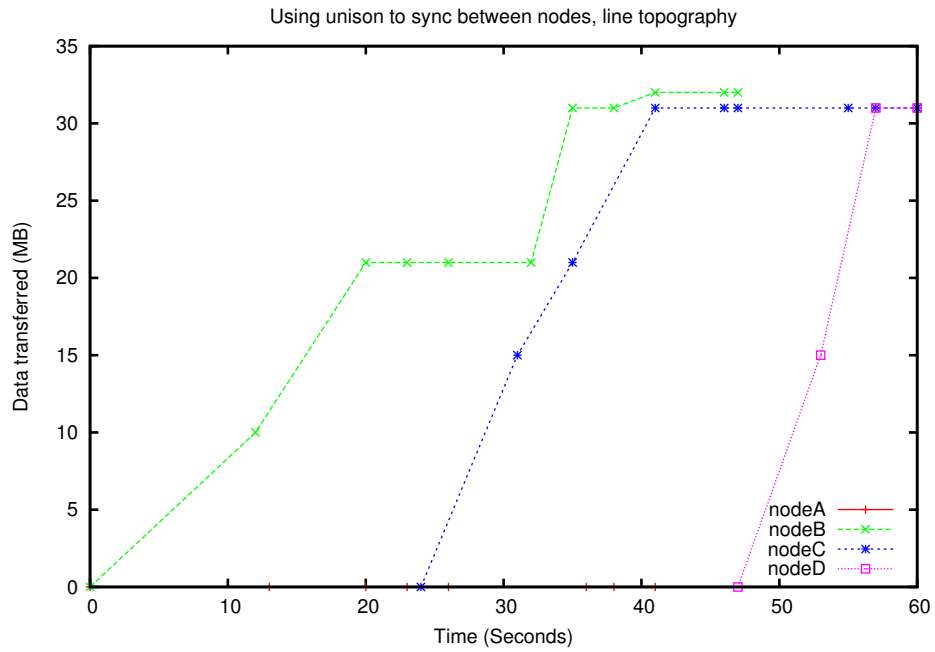
Figure 14: Five second delay between synchronising 10MB files over a line topology using Unison
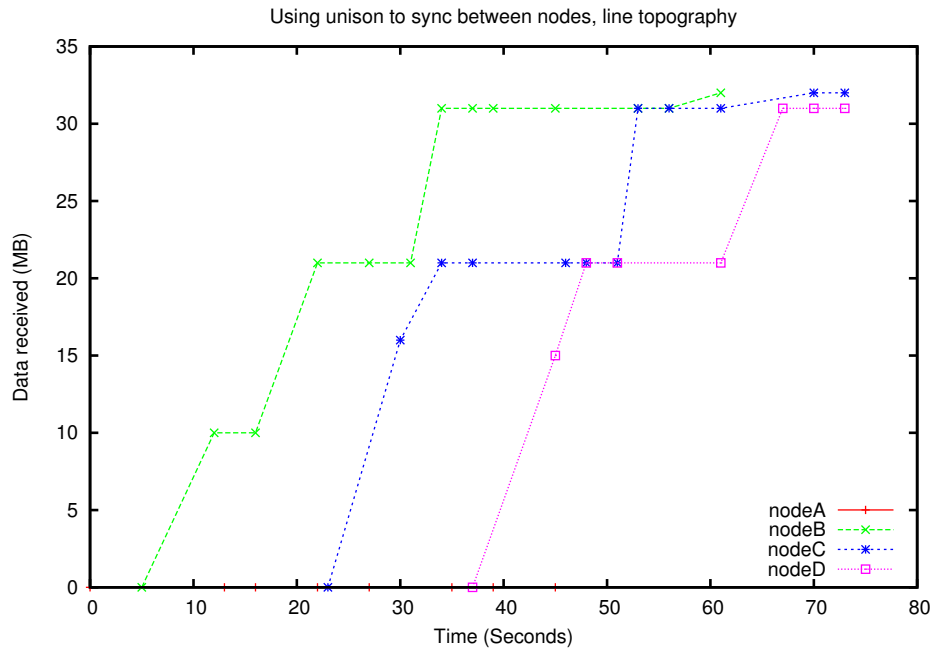
Figure 15: Ten second delay between synchronising 10MB files over a line topology using Unison
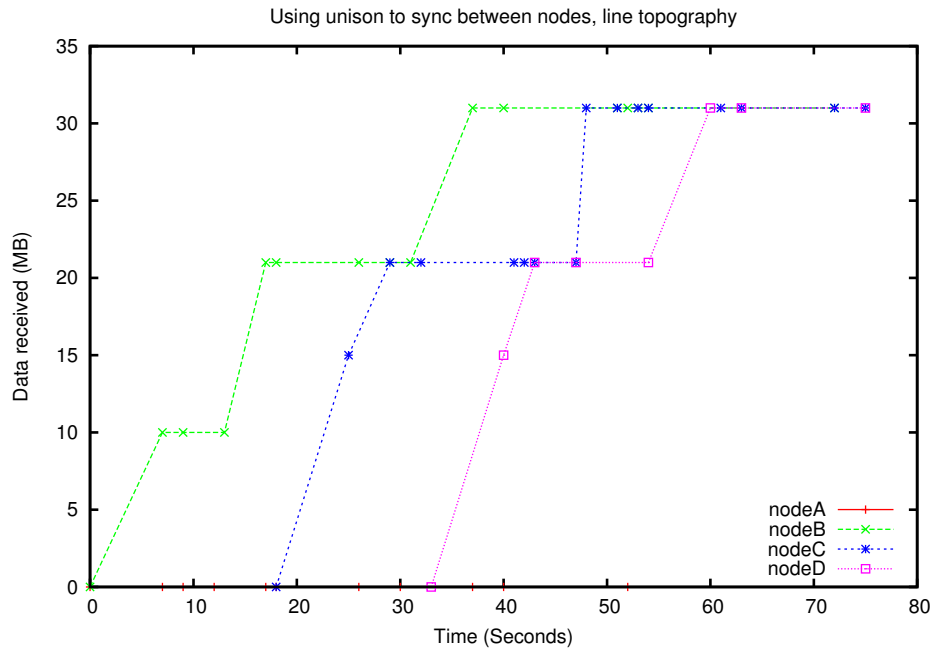
Figure 16: Twenty second delay between synchronising 10MB files over a line topology using Unison
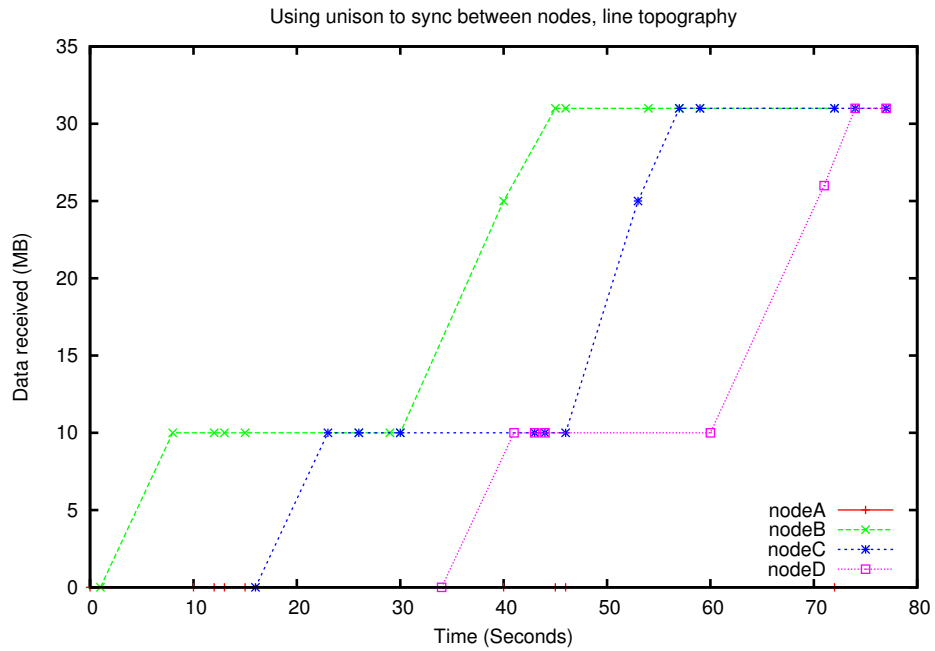
Figure 17: Thirty second delay between synchronising 10MB files over a line topology using Unison
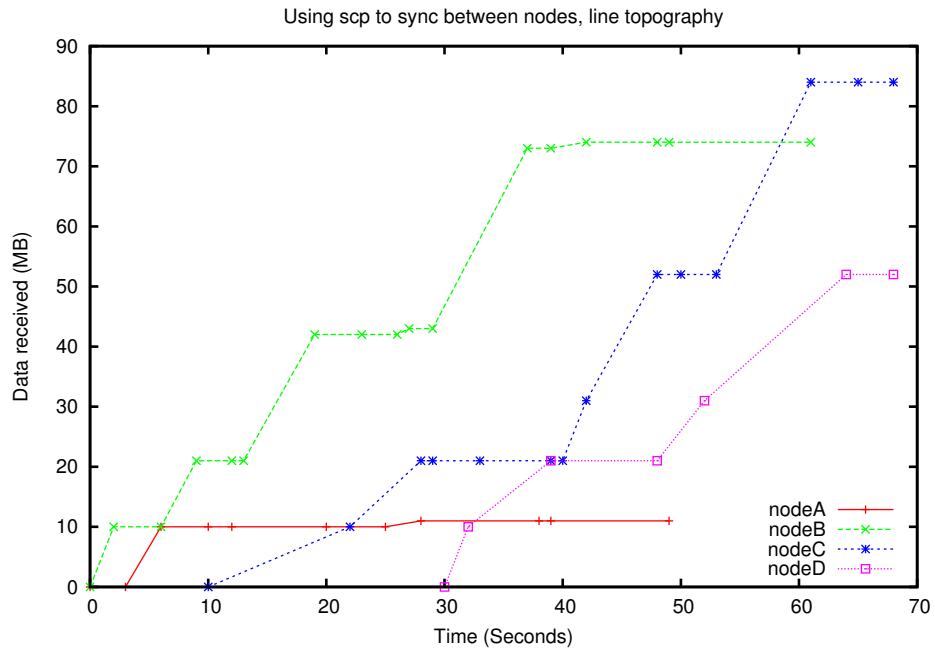
Figure 18: Five second delay between synchronising 10MB files over a line topology using SCP
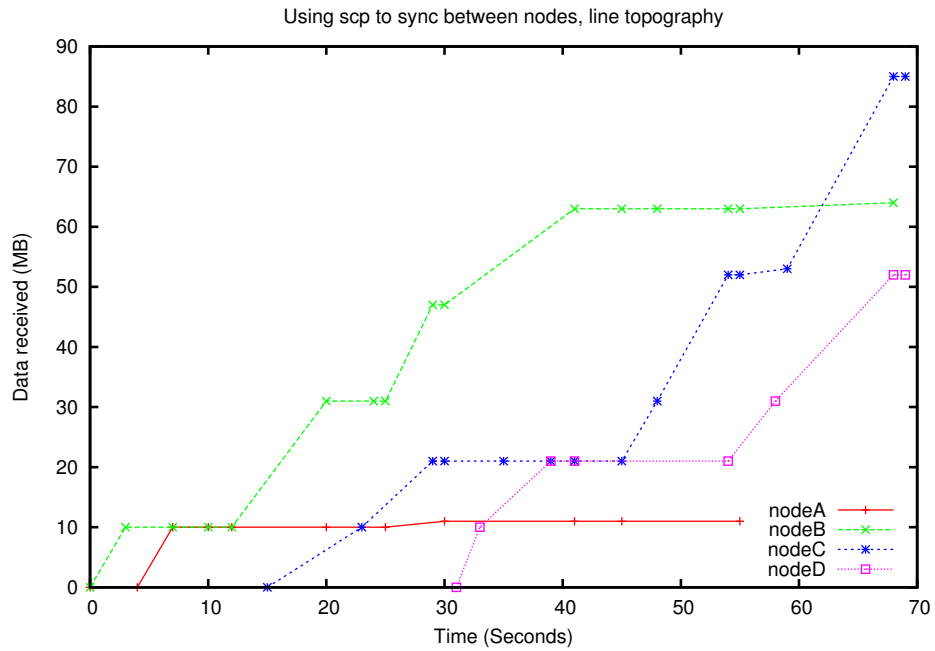
Figure 19: Ten second delay between synchronising 10MB files over a line topology using SCP

28
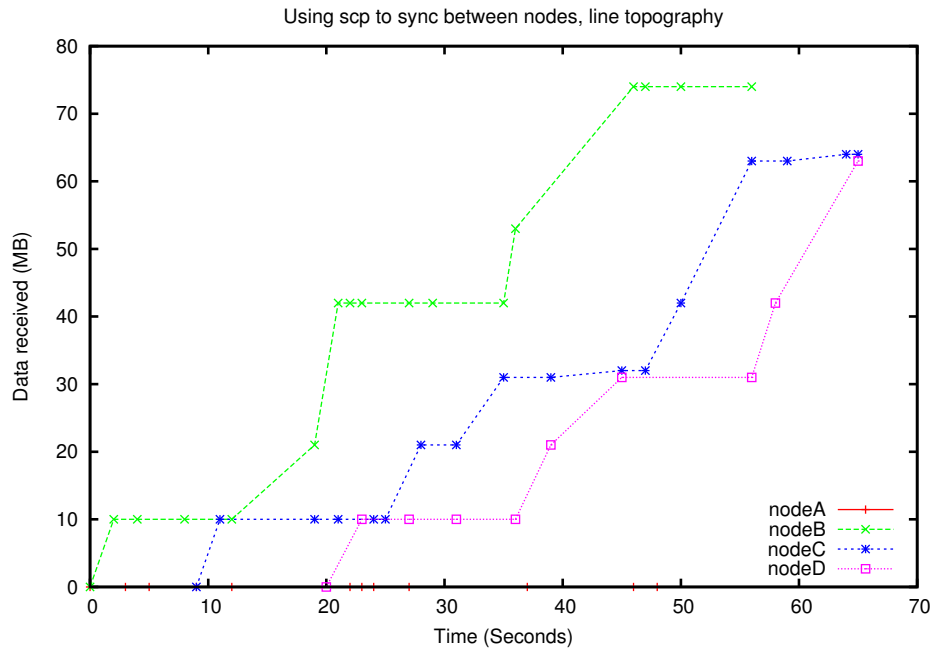
Figure 20: Eleven second delay between synchronising 10MB files over a line topology using SCP
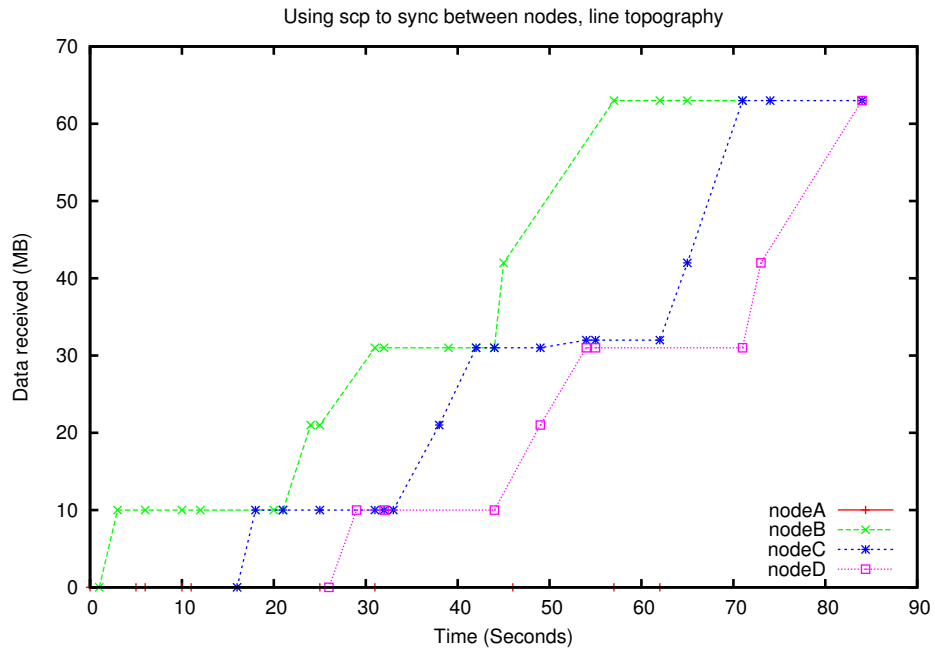
Figure 21: Twenty second delay between synchronising 10MB files over a line topology using SCP
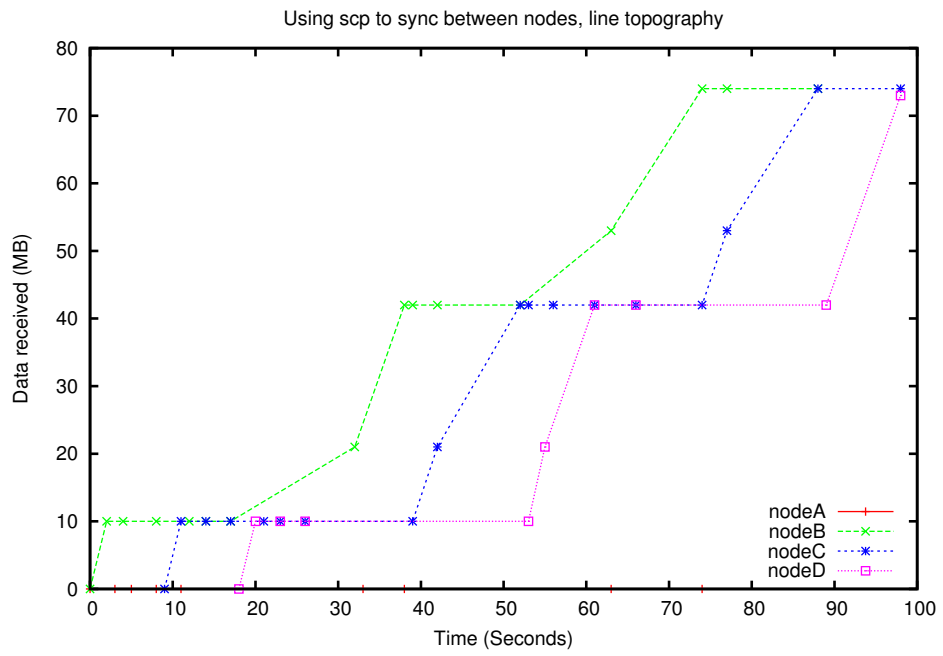
Figure 22: Thirty second delay between synchronising 10MB files over a line topology using SCP

4.5   Wi-Fi vs 3G

An important use case of my program was where one machine was connected by a 3G network or some other link that a user would not want to use often. 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. 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.
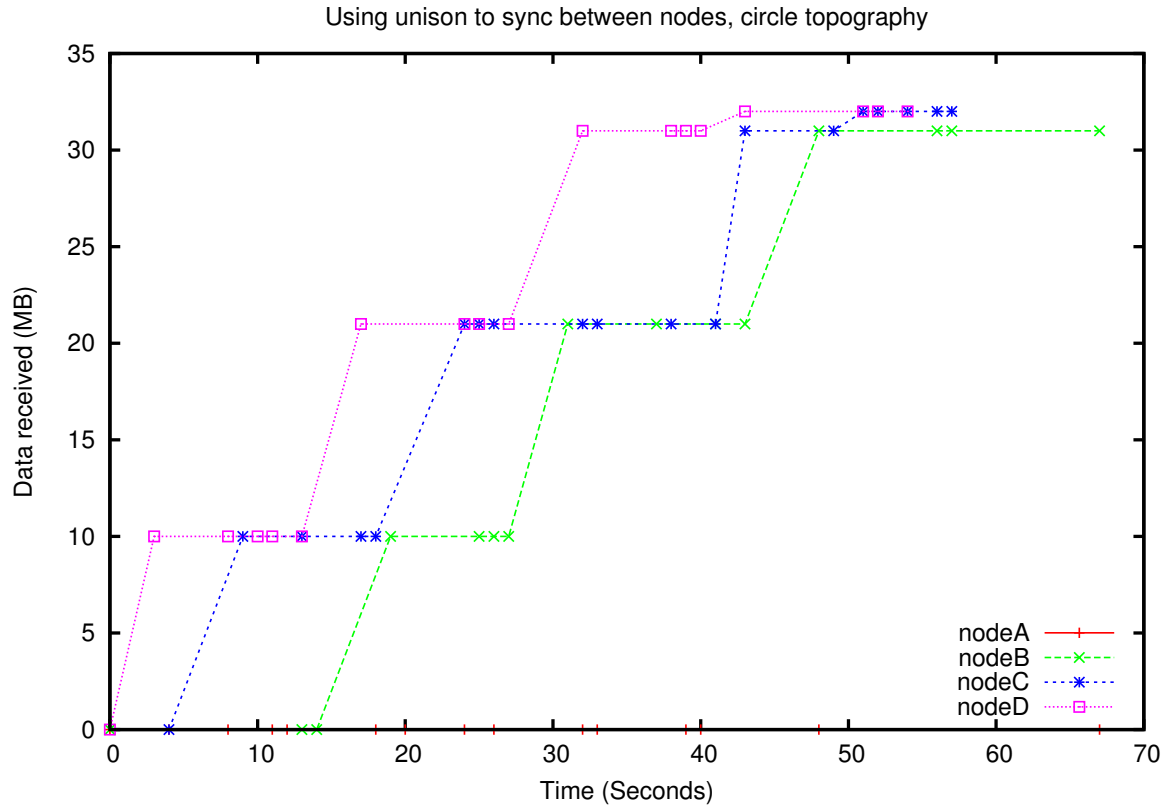
Figure 23: Node A is connected to node B by 3G, the other links are all Wi-Fi. Three 10MB files are sent to node A over ten second intervals. Node A has a longer delay on sending over 3G than Wi-Fi and node B is the last node to receive the changes.

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 what happens 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 would also have been interested in seeing how I could have improved the performance of the program with the data I am collecting.

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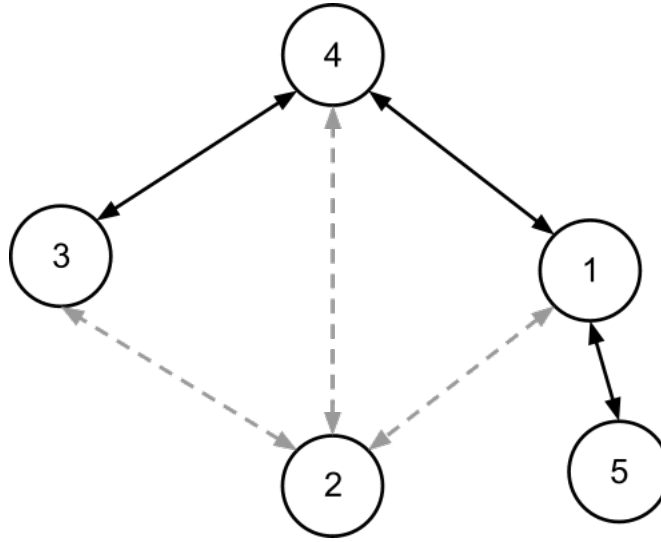
Figure 24: Node two intermittently has a connection to a subset of nodes 1, 3 and 4. The edges in grey are links that are not always present.

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 like to integrate these scripts my program to show an end user a graphical representation of the program running. Integrating my current graph scripts should be trivial. I could just shell out to them if I really had to. 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
2  import readnet
3
4  wm = pyinotify.WatchManager()
5  watchedfolders = {}
6  homepath = "/home/cal/Documents/Private-Sync/"
7  #homepath = "/Users/calum/Documents/Private-Sync/"
8
9  parser = argparse.ArgumentParser()
10 parser.add_argument("-c","--scp",action="store_true",help="
       Copy using scp")
11 parser.add_argument("-r","--rsync",action="store_true",help="
       Copy using rsync")
12 args = parser.parse_args()
13
14 class Tools():
15     def updateFolderInfo(self, wfolds):
16         f = open('./folders.dat','w')
17         for fold in wfolds:
18             f.write(fold + " ")
19             for i in range(0,len(wfolds[fold])-1):
20                 f.write(wfolds[fold][i] + " ")
21             f.write(wfolds[fold][len(wfolds[fold])-1] + "\n")
22         f.close()
23
24     def timeElapsed(self, dtstamp, diff):
25         if diff == "*":
26             print "Sync ASAP"
27             return
28         diff = int(diff)
29         FMT = '%Y-%m-%d %H:%M:%S.%f '
30         #FMT = '%Y-%m-%d %H:%M:%S '
31         tdelta = datetime.datetime.now() - datetime.datetime.
                 strptime(dtstamp, FMT)
32         print   tdelta.total_seconds()
33         timeDiff = tdelta.total_seconds()
34         if (timeDiff >= diff):
35             print "Time period reached"
```

```python
36          else :
37              print "Time_not_elapsed , _sleeping_for_" + str ( diff
                    − timeDiff + 1)
38              time . sleep ( int ( diff − timeDiff + 1))
39
40  class MyEventHandler ( pyinotify . ProcessEvent ):
41      def flipIP ( self , ip ):
42          octets = ip . split ( " . " )
43          if ( octets [ 3 ] == " 1 " ):
44              octets [ 3 ] = " 2 "
45          elif ( octets [ 3 ] == " 2 " ):
46              octets [ 3 ] = " 1 "
47          else :
48              octets [ 3 ] = " 1 "
49          return " . " . join ( octets )
50
51      #Get the last modified time of a file
52      def getModTime ( self , path ):
53          try :
54              return time . ctime ( os . path . getmtime ( path ) )
55          except Exception , e :
56              return time . ctime ( 0 )
57
58      #Deprecated − Check for IP not to copy too
59      def getStopInfo ( self ):
60          stopIP = [ "" , "" ]
61          try :
62              o = open ( " . / stop " , ' r ' )
63              stopIP = o . read ( ) . split ( )
64              o . close ( )
65          except IOError , e :
66              pass
67          return stopIP
68
69      def inStopFile ( self , ip , path ):
70          stopIPs = {}
71          stop = False
72          modTime = self . getModTime ( path )
73          while True :
74              tmpcount = 0
75              print " Files _found : _ " + str ( glob . glob ( " Stop −∗ " ) )
76              for files in glob . glob ( " Stop −∗ " ):
77                  #print "File: " + str(files)
```

```python
78                    if ".tmp" in files:
79                        tmpcount += 1
80                        time.sleep(5)
81                        break
82                    f = open(files,"r");
83                    for line in f:
84                        l = line.split()
85                        if self.exclusions(l[1]):
86                            print str(l[1]) + " was in ignore file
                                skipping"
87                        else:
88                            print "local " + str(path) + " modtime
                                : " + modTime
89                            print "Stop " + l[1] + " modtime: " +
                                str(l[2:])
90                            ts1 = time.strptime(modTime,"%a %b %d
                                %H:%M:%S %Y")
91                            ts2 = time.strptime(" ".join(l[2:]),"%
                                a %b %d %H:%M:%S %Y")
92                            print "local <= stop: " + str(ts1 <=
                                ts2)
93                            #if l[0] == ip and l[1] == path and
                                ts1 <= ts2:
94                            #If IP sending to has sent data more
                                recently don't send back
95                            if l[0] == ip and ts1 <= ts2:
96                                print "Stop = True, file: " + l[0]
97                                stop = True
98                            else:
99                                stopIPs[l[0]] = [l[1]," ".join(l
                                    [2:])]
100
101                    if stop:
102                        f.close()
103                        #f = open(files,"w")
104                        #for k in stopIPs.keys():
105                        #    f.write(k + " " + stopIPs[k][0] + " "
                                + stopIPs[k][1] + "\n")
106                        #f.close()
107                        #stopIPs.clear()
108                        return True
109
110                    f.close()
```

```python
111                    #stopIPs.clear()
112              if tmpcount == 0:
113                  break
114
115          return False
116
117      #Set flag on other server telling it not to immediately
              try and copy data here
118      def setStopFileUniq(self,ip,myIP,path,folder):
119          nodename = self.getNodeName()
120          #print "ssh",ip,"echo " + myIP + " " + path + " " +
                 self.getModTime(path) + " >> " + homepath + "Stop-"
                 + nodename + ".tmp;"
121          #subprocess.call(["ssh",ip,"echo " + myIP + " " + path
                 + " " + self.getModTime(path) + " >> " + homepath +
                 "Stop-" + nodename + ".tmp;"])
122          subprocess.call(["ssh",ip,"rm " + homepath + "Stop-" +
                 nodename + ".tmp;"])
123          for cpFile in glob.glob(folder + "/*"):
124              subprocess.call(["ssh",ip,"echo " + myIP + " " +
                     cpFile + " " + self.getModTime(cpFile) + " >> "
                     + homepath + "Stop-" + nodename + ".tmp;"])
125
126      #Sets the config files on the remote node
127      def beginCopy(self, ip):
128          nodename = self.getNodeName()
129          print "ssh",ip,"touch " + homepath + "Stop-" +
                 nodename + ".tmp; mv " + homepath + "Stop-" +
                 nodename + " " + homepath + "Stop-" + nodename + ".
                 tmp;"
130          subprocess.call(["ssh",ip,"touch " + homepath + "Stop-
                 " + nodename + ".tmp; mv " + homepath + "Stop-" +
                 nodename + " " + homepath + "Stop-" + nodename + ".
                 tmp;"])
131
132      #Moves the Stop files back into place
133      def endCopy(self, ip):
134          nodename = self.getNodeName()
135          print "ssh",ip,"mv " + homepath + "Stop-" + nodename +
                 ".tmp " + homepath + "Stop-" + nodename
136          subprocess.call(["ssh",ip,"mv " + homepath + "Stop-" +
                 nodename + ".tmp " + homepath + "Stop-" + nodename
                 ])
```

```
137
138     def setLastSync(self):
139         #print "echo \"" + str(datetime.datetime.now())+ "\" >
                " + homepath + "lastSync"
140         #subprocess.call(["echo", str(datetime.datetime.now())
                + " > " + homepath + "lastSync"])
141         f = open(homepath + "lastSync","w")
142         f.write(str(datetime.datetime.now()))
143         f.close()
144
145     def getLastSync(self):
146         f = open(homepath + "lastSync","r")
147         time = f.read()
148         f.close()
149         return time.rstrip()
150
151     def newerThanLast(self, fileName):
152         stop = False
153         print "Last sync time: " + str(self.getLastSync())
154         FMT = '%Y-%m-%d %H:%M:%S.%f'
155         #datetime.datetime.strptime(dtstamp, FMT)
156         ts2 = time.strptime(self.getLastSync(),FMT)
157         modTime = self.getModTime(fileName)
158         print "local " + str(fileName) + " modtime: " +
                modTime
159         ts1 = time.strptime(modTime,"%a %b %d %H:%M:%S %Y")
160         print "local <= stop: " + str(ts1 <= ts2)
161         if ts1 > ts2:
162             print "Newer file than last sync!"
163             stop = True
164         return stop
165
166
167     #Get node name from whoami file
168     def getNodeName(self):
169         w = open(homepath + "whoami","r")
170         nodename = w.read()
171         nodename = nodename[0].upper()
172         w.close()
173         return nodename
174
175     #Deprecated stop file
176     def setStopFile(self,ip,myIP,path):
```

```
177            subprocess.call(["ssh",ip,"echo " + myIP + " " + path
                   + "> " + homepath + "stop"])
178            print "ssh",ip,"echo " + myIP + "> " + homepath + "
                   stop"
179
180      def rmTree(self,path):
181            subprocess.call(["ssh",ip,"rm -r '" + path + "'"])
182            print "ssh",ip,"rm -r '" + path + "'"
183
184      #Exclude files matching patterns in the ignore file
185      def exclusions(self, path):
186            try:
187                  f = open("./ignore",'r')
188                  for line in f:
189                        if line.rstrip() in path:
190                              #print "Ignoring: " + path
191                              return True
192                  f.close()
193            except error, e:
194                  print e
195            return False
196
197      def initFileSync(self,event):
198            if self.exclusions(event.pathname):
199                  #print "Excluded returning"
200                  return
201            pathparts = event.pathname.split("/")
202            foldName = "/".join(pathparts[0:len(pathparts)-1])
203
204            print "Removing watch on: " + foldName
205            wm.rm_watch(wm.get_wd(foldName), rec=True)
206
207            self.setLastSync()
208            self.fileSync(event.pathname)
209
210            print "Putting watch back on: " + foldName
211            wm.add_watch(foldName.rstrip(),pyinotify.ALL_EVENTS,
                   rec=True, auto_add=True)
212
213            for i in range(0, len(watchedfolders[foldName]),4):
214            #      ip = watchedfolders[foldName][i]
215            #      myIP = readnet.getMyIP(ip)
216                  for f in glob.glob(foldName + "/*"):
```

41

```python
217                    if self.newerThanLast(f):
218                        print "init:_CONTINUE"
219                        self.setLastSync()
220                        self.fileSync(f)
221                    else:
222                        print "init:_STOP"
223
224        #Sync the files
225        def fileSync(self,pathname):
226            t = Tools()
227            if os.path.isdir(pathname):
228                print "Watching:_",pathname
229            for folder in watchedfolders.keys():
230                print "For_each_folder:_" + str(folder) + "_in_
                     watchedfolder_keys"
231                if folder in pathname:
232                    for i in range(0, len(watchedfolders[folder])
                         ,4):
233                        ip = watchedfolders[folder][i]
234                        path = watchedfolders[folder][i+1]
235                        waitTime = watchedfolders[folder][i+2]
236                        lastTime = watchedfolders[folder][i+3]
237                        print "Wait:_" + str(waitTime) + "_Last:_"
                             + str(lastTime)
238                        print "Current_ip_and_path:_" + ip + "_" +
                             path
239                        readnet.logIPtraffic(ip, pathname)
240                        myIP = readnet.getMyIP(ip)
241                        subprocess.call(["ssh",ip,"/usr/bin/python
                             _" + homepath + "readnet.py_-i_" + myIP
                             + "_-f_" + pathname])
242                        print "ssh",ip,"'/usr/bin/python_" +
                             homepath + "readnet.py_-i_" + myIP + "_-
                             f_" + pathname + "'"
243                        fparts = folder.split("/")
244                        fname = fparts[len(fparts)-1]
245                        #stopIP = self.getStopInfo()
246                        #print "STOP: " + stopIP[0] + " " + stopIP
                             [1]
247                        #if stopIP[0] == ip and stopIP[1] ==
                             pathname:
248                        if self.inStopFile(ip,pathname):
249                            print "STOPPED_to_" + ip + "_" + path
```

42

```python
250                         #os.remove("./stop");
251                     else:
252                         print "CONTINUE"
253                         t.timeElapsed(lastTime, waitTime)
254                         watchedfolders[folder][i+3] = str(
                                datetime.datetime.now())
255                         t.updateFolderInfo(watchedfolders)
256                         self.beginCopy(ip)
257                         self.beginCopy(myIP)
258                         if args.scp:
259                             #print "SCP: For cpFile in " +
                                    folder
260                             for cpFile in glob.glob(folder + "
                                    /*"):
261                                 #print "SCP GLOB:" + cpFile
262                                 print "scp","-rp",cpFile,ip +
                                        ":" + cpFile + ".tmp"
263                                 subprocess.call(["scp","-rp",
                                        cpFile,ip + ":" + cpFile + "
                                        .tmp"])
264                                 #subprocess.call(["ssh",ip,"
                                        yes y | find /tmp/" + fname
                                        + " -type f -exec cp -p {} "
                                        + path + fname + "/ \; rm /
                                        tmp/" + fname])
265                                 print "ssh",ip,"mv " + cpFile
                                        + ".tmp " + cpFile
266                                 subprocess.call(["ssh",ip,"mv
                                        " + cpFile + ".tmp " +
                                        cpFile])
267                                 print "END SCP GLOB"
268                         elif args.rsync:
269                             print "rsync","-rt",folder,ip + ":
                                    " + path
270                             subprocess.call(["rsync","-rt",
                                    folder,ip + ":" + path])
271                         else:
272                             time.sleep(5)
273                             print "unison","-batch","-
                                    confirmbigdel=false","-times",
                                    folder,"ssh://" + ip + "/" +
                                    path + fname
```

```
274                              subprocess.call(["unison","−batch"
                                     ,"−confirmbigdel=false","−times"
                                     ,folder ,"ssh://" + ip + "/" +
                                     path + fname])
275                          print "Set stop files uniq: " +
                                 pathname
276                          #Set stop file on foreign host
277                          self.setStopFileUniq(ip,myIP,pathname,
                                 folder)
278                          #Set stop file for myself to look at
279                          self.setStopFileUniq(myIP,myIP,
                                 pathname,folder)
280                          self.endCopy(ip)
281                          self.endCopy(myIP)
282                      subprocess.call(["ssh",ip,"/usr/bin/python
                             " + homepath + "readnet.py −i " + myIP
                             + " −f " + pathname])
283                      readnet.logIPtraffic(ip, pathname)
284
285      #Sync files − DEPRECATED
286      def oldfileSync(self,event):
287          t = Tools()
288          if os.path.isdir(event.pathname):
289              print "Watching: ",event.pathname
290          for folder in watchedfolders.keys():
291              print "For each folder: " + str(folder) + " in
                     watchedfolder keys"
292              if folder in event.pathname:
293                  for i in range(0, len(watchedfolders[folder])
                         ,4):
294                      ip = watchedfolders[folder][i]
295                      path = watchedfolders[folder][i+1]
296                      waitTime = watchedfolders[folder][i+2]
297                      lastTime = watchedfolders[folder][i+3]
298                      print "Wait: " + str(waitTime) + " Last: "
                             + str(lastTime)
299                      print "Current ip and path: " + ip + " " +
                             path
300                      readnet.logIPtraffic(ip, event.pathname)
301                      myIP = readnet.getMyIP(ip)
302                      subprocess.call(["ssh",ip,"/usr/bin/python
                             " + homepath + "readnet.py −i " + myIP
                             + " −f " + event.pathname])
```

44

```python
303                        print "ssh",ip," '/usr/bin/python " +
                              homepath + "readnet.py -i " + myIP + " -
                              f " + event.pathname + " '"
304                        fparts = folder.split("/")
305                        fname = fparts[len(fparts)-1]
306                        #stopIP = self.getStopInfo()
307                        #print "STOP: " + stopIP[0] + " " + stopIP
                              [1]
308                        #if stopIP[0] == ip and stopIP[1] == event
                              .pathname:
309                        if self.inStopFile(ip,event.pathname):
310                            print "STOPPED to " + ip + " " + path
311                            #os.remove("./stop");
312                        else:
313                            print "CONTINUE"
314                            t.timeElapsed(lastTime, waitTime)
315                            watchedfolders[folder][i+3] = str(
                                  datetime.datetime.now())
316                            t.updateFolderInfo(watchedfolders)
317                            self.beginCopy(ip)
318                            if args.scp:
319                                #print "SCP: For cpFile in " +
                                      folder
320                                for cpFile in glob.glob(folder + "
                                      /*"):
321                                    #print "SCP GLOB:" + cpFile
322                                    print "scp","-rp",cpFile,ip +
                                          ":" + cpFile + ".tmp"
323                                    subprocess.call(["scp","-rp",
                                          cpFile,ip + ":" + cpFile + "
                                          .tmp"])
324                                    #subprocess.call(["ssh",ip,"
                                          yes y | find /tmp/" + fname
                                          + " -type f -exec cp -p {} "
                                          + path + fname + "/ \; rm /
                                          tmp/" + fname])
325                                    print "ssh",ip,"mv " + cpFile
                                          + ".tmp " + cpFile
326                                    subprocess.call(["ssh",ip,"mv
                                          " + cpFile + ".tmp " +
                                          cpFile])
327                                    print "END SCP GLOB"
328                            elif args.rsync:
```

45

```python
329                                       print "rsync","-rt",folder,ip + ":
                                              " + path
330                                       subprocess.call(["rsync","-rt",
                                              folder,ip + ":" + path])
331                                   else:
332                                       time.sleep(5)
333                                       print "unison","-batch","-
                                              confirmbigdel=false","-times",
                                              folder,"ssh://" + ip + "/" +
                                              path + fname
334                                       subprocess.call(["unison","-batch"
                                              ,"-confirmbigdel=false","-times"
                                              ,folder,"ssh://" + ip + "/" +
                                              path + fname])
335                               print "Set stop files uniq: " + event.
                                      pathname
336                               #Set stop file on foreign host
337                               self.setStopFileUniq(ip,myIP,event.
                                      pathname,folder)
338                               #Set stop file for myself to look at
339                               self.setStopFileUniq(myIP,myIP,event.
                                      pathname,folder)
340                               self.endCopy(ip)
341                           subprocess.call(["ssh",ip,"/usr/bin/python
                                  " + homepath + "readnet.py -i " + myIP
                                  + " -f " + event.pathname])
342                           readnet.logIPtraffic(ip, event.pathname)
343
344       #def process_IN_CREATE(self, event):
345       #     print "Create:",event.pathname
346       def process_IN_DELETE(self, event):
347           print "Delete: ",event.pathname
348           #self.initFileSync(event)
349       def process_IN_CREATE(self, event):
350           print "CREATE: ",event.pathname
351           self.initFileSync(event)
352       def process_IN_MOVED_FROM(self, event):
353           print "Move from: ",event.pathname
354       #     self.initFileSync(event)
355       def process_IN_MODIFY(self, event):
356           #print "Modify: ",event.pathname
357           self.initFileSync(event)
358       def process_IN_MOVED_TO(self, event):
```

```
359             print "Move_to:_",event.pathname
360             self.initFileSync(event)
361
362
363   def main():
364       t = Tools()
365       f = open('./folderstowatch','r')
366
367       for folder in f:
368           if(folder[0] == '#'):
369               pass
370           else:
371               info = folder.split()
372               wm.add_watch(info[0].rstrip(),pyinotify.ALL_EVENTS
                      , rec=True, auto_add=True)
373               print "Watching:_", info[0].rstrip()
374               if info[0] not in watchedfolders.keys():
375                   watchedfolders[info[0].rstrip()] = []
376               watchedfolders[info[0].rstrip()].append(info[1])
377               watchedfolders[info[0].rstrip()].append(info[2])
378               watchedfolders[info[0].rstrip()].append(info[3])
379               watchedfolders[info[0].rstrip()].append(str(
                      datetime.datetime.now()))
380       f.close()
381
382       try:
383           f = open('./folders.dat','r')
384           for folder in f:
385               if(folder[0] == '#'):
386                   pass
387               else:
388                   info = folder.split()
389                   if info[0] in watchedfolders.keys():
390                       del watchedfolders[info[0].rstrip()]
391                       #wm.add_watch(info[0].rstrip(),pyinotify.
                          ALL_EVENTS, rec=True, auto_add=True)
392                       #print "Watching: ", info[0].rstrip()
393                       if info[0] not in watchedfolders.keys():
394                           watchedfolders[info[0].rstrip()] = []
395                       watchedfolders[info[0].rstrip()].append(
                          info[1])
396                       watchedfolders[info[0].rstrip()].append(
                          info[2])
```

47

```
397                          watchedfolders [ info [0]. rstrip () ]. append (
                                 info [3])
398                          watchedfolders [ info [0]. rstrip () ]. append (
                                 str ( datetime . datetime . now ()))
399                  else :
400                      print  "Removing:␣" + info [0]
401          f . close ()
402      except  IOError , e :
403          print  "Folders . dat␣does␣not␣exist ,␣skipping"
404
405      t . updateFolderInfo ( watchedfolders )
406
407      #print  watchedfolders
408      eh = MyEventHandler ()
409
410      notifier = pyinotify . Notifier (wm, eh )
411      notifier . loop ()
412
413  if __name__ == '__main__':
414      main ()
```

A.2   ReadNet.py

```
 1  import subprocess , datetime , socket , argparse
 2
 3  homepath = "/home/ cal /Documents/ Private−Sync/"
 4  #homepath = "/Users/calum/Documents/Private−Sync/"
 5
 6  parser = argparse . ArgumentParser ()
 7  parser . add_argument ( '−i ' , action="store" , dest='ip ' , help='IP␣
        address␣to␣record␣for ')
 8  parser . add_argument ( '−f ' , action="store" , dest='fold ' , help='
        Folder␣to␣record␣for ')
 9
10  interfacenames = []
11
12  w = open ( homepath + "whoami" , " r ")
13  nodename = w. read ()
14  nodename = nodename [ 0 ]
15  w. close ()
16
17  #Get my ip corresponding to the interface with ipaddr
18  def getMyIP ( ipaddr ):
```

```
19        route = subprocess.check_output("ip route get " + ipaddr,
            shell=True)
20        words = route.split()
21        interface = ""
22        for word in words:
23            if word.startswith("eth"):
24                interface = word
25                #print interface
26                break
27        ifconf = subprocess.check_output("ifconfig " + interface,
            shell=True)
28        words = ifconf.split()
29        now = False
30        for word in words:
31            if word == "inet":
32                now = True
33            elif now:
34                word = word.split(":")
35                #print word[1]
36                return word[1]
37
38    #Log interface coresponding to ipaddr
39    def logIPtraffic(ipaddr, folder):
40        route = subprocess.check_output("ip route get " + ipaddr,
            shell=True)
41        words = route.split()
42        interface = ""
43        for word in words:
44            if word.startswith("eth"):
45                interface = word
46                #print interface
47                break
48        writeIface(interface, folder)
49
50    #Write the upload/download data for a given interface and
        folder
51    def writeIface(iface, folder):
52        ifs = subprocess.check_output("ifconfig -s", shell=True)
53        ilines = ifs.split("\n")
54        for i in range(1, len(ilines)-1):
55            interfacenames.append(ilines[i].split()[0])
56        output = subprocess.check_output("ifconfig", shell=True)
57        splitput = output.split()
```

```python
58        interface = False
59        interfacename = ""
60        nex = ""
61        count = 0
62        upload = 0
63        download = 0
64        for split in splitput:
65            if split in interfacenames:
66                interface = True
67                interfacename = split
68                #print interfacename
69            if(nex != ""):
70                sp = split.split(":")
71                if(sp[0] == "bytes"):
72                    if(nex == "RX"):
73                        download = int(sp[1])
74                    else:
75                        upload = int(sp[1])
76                    nex = ""
77                    count += 1
78                    if(count == 2):
79                        interface = False
80                        if interfacename == iface:
81                            f = open(homepath + "log/" \
82                            + "node" + nodename.upper() + "-" \
83                            + iface + ".log",'a')
84                            f.write("#D " + folder + "\n")
85                            f.write(str(datetime.datetime.now()) +
                                    " " + interfacename + " download: "
                                    + str(download) + " upload: " + str
                                    (upload) + "\n")
86                            f.close()
87                        count = 0
88            elif(interface):
89                if(split == "RX" or split == "TX"):
90                    nex = split
91
92 #Log all interfaces
93 def main():
94     ifs = subprocess.check_output("ifconfig -s",shell=True)
95     ilines = ifs.split("\n")
96     for i in range(1,len(ilines)-1):
97         interfacenames.append(ilines[i].split()[0])
```

```python
 98        output = subprocess.check_output("ifconfig", shell=True)
 99        splitput = output.split()
100        interface = False
101        interfacename = ""
102        nex = ""
103        count = 0
104        upload = 0
105        download = 0
106        for split in splitput:
107            if split in interfacenames:
108                interface = True
109                interfacename = split
110                #print interfacename
111            if(nex != ""):
112                sp = split.split(":")
113                if(sp[0] == "bytes"):
114                    if(nex == "RX"):
115                        download = int(sp[1])
116                    else:
117                        upload = int(sp[1])
118                    nex = ""
119                    count += 1
120                    if(count == 2):
121                        interface = False
122                        f = open(homepath + "log/" \
123                        + str(socket.gethostname()) + "-" \
124                        + interfacename + ".log",'a')
125                        f.write(str(datetime.datetime.now()) + " "
                                + interfacename + " download: " + str(
                                download) + " upload: " + str(upload) +
                                "\n")
126                        f.close()
127                        count = 0
128            elif(interface):
129                if(split == "RX" or split == "TX"):
130                    nex = split
131
132  if __name__ == "__main__":
133      args = parser.parse_args()
134      if args.ip != None:
135          logIPtraffic(args.ip, args.fold)
136          #getMyIP(args.ip)
137      else:
```

```
138          pass
139          main()
```

A.3   onTheFly.sh

```
1  vm_name_arr=("Ubuntu-Pool" "Ubuntu-Silence" "Ubuntu-Black" "
       Ubuntu-Spheros" "Ubuntu-Wild")
2  vm_addr_arr=("192.168.0.28" "192.168.0.27" "192.168.0.30" "
       192.168.0.14")
3  #Wild = 19
4  intnetarr=("lion" "tiger" "cat" "dog" "fish" "kiwi" "swish" "
       boom" "roar")
5  #These should all be in one big dictionary apart from inet
       names
6  letterarr=("a" "b" "c" "d" "e" "f" "g" "h" "i" "j")
7  ifcountarr=(2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2)
8  ethcountarr=(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
9  incount=1
10 bigncount=2
11 littlencount=1
12 folderpath="/home/cal/Documents/t18"
13 folderpath2="/home/cal/Documents/t02"
14 homepath="/home/cal/Documents/Private-Sync/"
15 waitTime=5
16
17 function clear_ifaces() {
18     i=0
19     while [ "$i" -lt "${#vm_name_arr[@]}" ]; do
20         VBoxManage modifyvm ${vm_name_arr[$i]} --nic2 none
21         echo "VBoxManage modifyvm ${vm_name_arr[$i]} --nic2
             none"
22         VBoxManage modifyvm ${vm_name_arr[$i]} --nic3 none
23         echo "VBoxManage modifyvm ${vm_name_arr[$i]} --nic3
             none"
24         VBoxManage modifyvm ${vm_name_arr[$i]} --nic4 none
25         echo "VBoxManage modifyvm ${vm_name_arr[$i]} --nic4
             none"
26         let "i++"
27     done
28 }
29
30 function clear_watched_folders() {
31     i=0
```

```bash
32        while [ "$i" -lt "${#vm_addr_arr[@]}" ]; do
33            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"
34            let "i++"
35        done
36  }
37
38  function git_pull() {
39        i=0
40        while [ "$i" -lt "${#vm_addr_arr[@]}" ]; do
41            echo "ssh cal@${vm_addr_arr[$i]} \"cd /home/cal/
                  Documents/Private-Sync; git pull origin master\""
42            ssh cal@${vm_addr_arr[$i]} "cd /home/cal/Documents/
                  Private-Sync; git pull origin master"
43            let "i++"
44        done
45  }
46
47  function search_letters() {
48        index=0
49        while [ "$index" -lt "${#letterarr[@]}" ]; do
50            if [ "${letterarr[$index]}" = "$1" ]; then
51                echo $index
52                return
53            fi
54            let "index++"
55        done
56        echo "None"
57  }
58
59  function vbmMOD {
60        #Set the NIC to intnet on the VM and attach it to the
              given network
61        echo "VBoxManage modifyvm $1 --nic$3 intnet"
62        VBoxManage modifyvm $1 --nic$3 intnet
63        echo "VBoxManage modifyvm $1 --intnet$3 $2"
64        VBoxManage modifyvm $1 --intnet$3 $2
65  }
66
67  function gatherLogs {
```

```
68        index=0
69        while [ "$index" −lt "${#vm_addr_arr[@]}" ]; do
70            echo "scp cal@${vm_addr_arr[$index]}:/home/cal/
                  Documents/Private−Sync/log/* ../logs/"
71            scp cal@${vm_addr_arr[$index]}:/home/cal/Documents/
                  Private−Sync/log/* ../logs/
72            let "index++"
73        done
74  }
75
76  function clean {
77        index=0
78        while [ "$index" −lt "${#vm_addr_arr[@]}" ]; do
79            echo "ssh cal@${vm_addr_arr[$index]} \"rm ${homepath}
                  log/*; rm ${homepath}Stop−*; rm ${homepath}folders.
                  dat\""
80            ssh cal@${vm_addr_arr[$index]} "rm ${homepath}log/*;
                  rm ${homepath}Stop−*; rm ${homepath}folders.dat"
81            let "index++"
82        done
83  }
84
85  function cleanFold {
86        index=0
87        while [ "$index" −lt "${#vm_addr_arr[@]}" ]; do
88            echo "ssh cal@${vm_addr_arr[$index]} \"rm −rf ${
                  folderpath}/*;\""
89            ssh cal@${vm_addr_arr[$index]} "rm −rf ${folderpath
                  }/*;"
90            let "index++"
91        done
92  }
93
94  function sendKeys {
95        index=0
96        while [ "$index" −lt "${#vm_addr_arr[@]}" ]; do
97            #ssh cal@${vm_addr_arr[$index]} "rm /home/cal/.ssh/
                  authorized_keys"
98            for file in /Users/calum/.ssh/*.pub; do
99                #echo "$file"
100               echo "cat $file | ssh cal@${vm_addr_arr[$index]} \
                      "cat >> /home/cal/.ssh/authorized_keys\""
```

```
101              cat $file | ssh cal@${vm_addr_arr[$index]} "cat >>
                    /home/cal/.ssh/authorized_keys"
102           done
103           let "index++"
104       done
105       #for file in /Users/calum/.ssh/*.pub; do
106       #    echo "$file"
107       #    cat $file | ssh cal@192.168.0.17 "cat >> /home/cal/.
              ssh/testfile"
108       #    echo "cat $file | ssh cal@192.168.0.17 \"cat >> /home
              /cal/.ssh/testfile\""
109       #done
110   }
111
112   function ifconf {
113       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 '"
114       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
115   }
116
117   function ifconf2 {
118       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"
119       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
120   }
121
122   if [ $2 == "vm" ]; then
123       clear_ifaces
124
```

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

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

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