UNIVERSITY OF HERTFORDSHIRE
Faculty of Engineering and Information Sciences

7COM0177 Computer Science MSc Project (Online)

Final Report
January 2014

Improved Open Source Backup: Incorporating inline deduplication and sparse indexing solutions

G. P. E. Keeling

Abstract

This paper investigates whether incorporating inline deduplication techniques can improve open source backup offerings, and whether 'sparse indexing' is an effective solution to the disk deduplication bottleneck.

Problems with an existing open source cross-platform network backup solution were identified, along with how it compares to other open source backup solutions, and a literature search and review of relevant algorithms and techniques was performed.

A new backup engine was designed and implemented, using Agile methodology and ideas garnered from the research.

A test procedure was produced and executed, and the results empirically show that the new software is either superior or comparable to its 'competition' in terms of speed, storage space, storage nodes, and network utilisation.

These improvements came at the cost of increased memory utilisation, which was then partially mitigated by implementing 'sparse indexing'.

There are some suggestions for further improvement and extra work to be done before an initial release.

The paper ends with some conclusions and an evaluation of the project as a whole.

Acknowledgements

I would like to thank all the users of the original 'burp', all around the world.

Contents **Page**

1. Introduction

This chapter begins by defining the problem domain. It goes on to formulate the research questions within the problem domain, and then finishes by outlining the aims and objectives of the project.

The paper then proceeds systematically through the following chapters, describing the methodology utilised, the design and development of the new software, a test plan and results, a second round of development and testing, and finally the conclusions and evaluation.

1.1. The problem domain

A typical scenario is a small to medium sized business (though the solution should not rule out enterprise users) where an administrator needs to back up from and restore to a mix of heterogenous machines. They want to back up to cheap disk based storage, rather than using expensive tape machines. They may want to back up to a remote location over a slow link.

I am the designer and author of an open source cross-platform network backup program. It is called 'burp' (Keeling, 2011), and aims to provide a solution to this problem. The experience of developing and supporting this software enables me to produce a list of what I consider to be the most important features necessary for such a solution:

- Backs up Unix/Linux and Windows computers.
- Backs up and restore files, directories, symlinks, hardlinks, fifos, nodes, permissions, timestamps, xattrs, acls and other meta data.
- Can do 'image' backups and restores.
- Can restore individual files.
- Space used on the server should be kept to a minimum.
- Time taken to back up should be kept to a minimum.
- Time taken to restore is less important, but should be reasonable.
- Network usage should be kept to a minimum.
- Network communications must be secure.
- Scheduling of backup times.
- Success and failure notifications via email can be configured.
- Multiple retention periods can be configured.
- An interrupted backup can be resumed.

The original burp supports most of this already. But there are flaws, or weaknesses. 'Appendix A - How the original burp works' describes how the original burp works in more detail. For the purposes of this introduction, the important points are that:
a) It has a client/server design, where the backups are stored on the central server.
b) It uses librsync (Pool, 2004) in order to save network traffic and on the amount of space that is used by each backup.
c) It can do deduplication 'after the fact', once all the data has been transferred, and then only at the granularity of whole files.

1.2. How the rsync algorithm works

This is a very brief summary of how the rsync algorithm works. For the full details, please see the original paper (Tridgell et al, 1996).

The rsync algorithm provides a mechanism for updating a file on one machine (the original file) to be identical to a file on another machine (the new file). The original file is split into fixed sized chunks. A pair of checksums is generated for each chunk. One, the 'weak' checksum, is very fast to calculate. The other, the 'strong' checksum takes longer to calculate, but two chunks of different content are highly unlikely to generate the same value. These checksums are sent to the other machine. The other machine now generates a 'delta' to be applied to the original file in order to make it match the new file. It does this by reading the new file byte-by-byte, generating the weak checksum as it goes. If it finds a chunk whose weak checksum matches one of the signatures that it received, it also generates a strong checksum for that chunk. If that also matches, the original file is thought to contain an indentical chunk.
By this means, a delta is generated that contains all the data not in the original file, and details on how to apply them in order to reconstruct the new file.

This algorithm was made, by the authors of the paper, into an open source program named rsync. It became very popular and has evolved over time. Its main purpose is to efficiently produce mirrors of the contents of file systems.
It is also used as the back end of various backup systems, such as rsnapshot (Rosenquist et al, 2003).

1.3. librsync

The librsync library was intended to provide a simple way for other programs to include the rsync algorithm. It is important to note that the original rsync project does not use librsync.

However, various backup programs, such as burp and rdiff-backup (Escoto et al, 2001), do use it.

Since the start of the original burp project, I have noticed that backing up large files can take a very long time when librsync is involved. So, I decided that this could be a good avenue of investigation with which to start this computer science project.

1.4. The problems with librsync

The original rsync continues to be maintained at the time of writing, whereas librsync has not been updated since 2004.
This means that improvements to rsync have not made it into librsync.

My initial investigation into the large file librsync problem involved timing the application of deltas to files, using both rsync and librsync, and comparing the results.

I was surprised when I discovered that librsync is actually faster than rsync for reasonably large files of a few gigabytes. On further investigation, I found that this was because librsync uses MD4 (Rivest, 1990) for its strong checksums, whereas rsync "now uses MD5 checksums instead of MD4" (*rsync-3.0.0-NEWS*, 2008). MD5 (Rivest, 1992) is stronger cryptographically, at the cost of a little speed.

Upon searching, I did not find a documented reason for rsync to have made this change. But this does show that the original rsync is continuing to evolve whilst librsync is not.

Increasing the size of the test files enabled me to find the point where the librsync speed slowed down dramatically, whilst the rsync speed did not.

I tracked the problem down to the relevant place in the librsync code. It turns out that, once a weak checksum has been looked up in the in-memory hash table, there is an attached list of strong checksums to search. This final search is done linearly.

After I made this discovery, I discovered that somebody else had found the same thing and had come up with a patch for it.

```
"When files being rsynced are hundreds of Gbytes size collisions in hash
table kill librsync.
So linear collision resolution has been replaced with log n collision
resolution based on binary search.
Size of hash table is 65536 buckets. So when files size is
(block_size * 65536 * t) then linear collision resolution is t / (log t)
slower than binary search resolution. If block size is 2048 bytes then for
1TB speed up is 630 times. for 100GB - 80 times." (Denisov, 2012)
```

At this point, I tried the patch and ran the tests again and found that it did help.

However, I then recalled that a lot depended on the fixed block size that you gave librsync. When writing the original burp, I assumed that the block size should vary depending on the size of the file.
But now, investigating the original rsync, I found that this is not what rsync itself does. It does a similar calculation, but then limits it to a maximum value.

So, by this point, I was being faced with reimplementing complicated logic from rsync into burp and rewriting chunks of librsync. I was effectively considering writing my own version of librsync within burp. And even if I did that, there would still be limitations due to hash table size limits and the design of the original burp still would not allow inline deduplication of data.

1.5. Variable length chunks

I continued with my academic research. I soon found information about variable length content chunking, as opposed to librsync's fixed length blocks.
The blocks are stored on the server, and the file to be backed up on the client side is read through and split into variable length chunks. This is the opposite way round to rsync splitting the server file, not the client, into fixed chunks.

```
"In backup applications, single files are backup images that are made up
of large numbers of component files. These files are rarely entirely
identical even when they are successive backups of the same file system.
A single addition, deletion, or change of any component file can easily
shift the remaining image content. Even if no other file has changed, the
```

shift would cause each fixed sized segment to be different than it was last time, containing some bytes from one neighbor and giving up some bytes to its other neighbor. The approach of partitioning the data into variable length segments based on content allows a segment to grow or shrink as needed so the remaining segments can be identical to previously stored segments.
Even for storing individual files, variable length segments have an advantage. Many files are very similar to, but not identical to other versions of the same file. Variable length segments can accommodate these differences and maximize the number of identical segments."
(Zhu, B. et al, 2008)

This same paper talks about methods for improving the speed of hash table index look ups in order to avoid disk bottlenecks when there is not enough RAM to hold it in memory.

1.6. Sparse indexing

This led me to another paper that proposes a different approach, named 'sparse indexing'. Put simply, the client reads the variable length chunks from its files to back up and groups them into 'segments'. Instead of sending the checksums of all the blocks to the server, it chooses a few 'hook' checksums and sends those first instead.
The server has already stored data from previous backups, including 'manifests', which consist of instructions on how to rebuild files from the blocks. These manifests also have 'hooks' chosen from them (these are the sparse indexes). The server will use the incoming 'hooks' to match against the sparse indexes in order to choose which manifest's blocks to deduplicate against. So, only a few manifests are chosen at a time, since loading from disk is costly. These manifests are called 'champions'.
Since not all the blocks are loaded into memory, the deduplication is not perfect. But, due to locality of data, the results will still remain good, as the experimental results of the paper show.

"To solve the chunk-lookup disk bottleneck problem, we rely on chunk locality: the tendency for chunks in backup data streams to reoccur together. That is, if the last time we encountered chunk A, it was surrounded by chunks B, C, and D, then the next time we encounter A (even in a different backup) it is likely that we will also encounter B, C, or D nearby. This differs from traditional notions of locality because occurrences of A may be separated by very long intervals (e.g., terabytes). A derived property we take advantage of is that if two pieces of backup streams share any chunks, they are likely to share many chunks." (Lillibridge, M. et al, 2009)

1.7. Putting it all together

It occurred to me that the sparse indexing paper makes no mention of the kind of efficiency savings that programs like bacula (Sibbald, 2000) or burp make when they are asked to back up a file on which the modification time has not changed since the previous backup.
It also occurred to me that I have not yet heard of an open source backup program that brings all of these techniques together, along with existing beneficial features (such as the usage of the Windows backup API, notifications, scheduling and so on). Such a program would surely be useful to many people around the world.
Some additional research was performed at this point to see if I was correct in this. I was unable to find any such software already in existence.

1.8. The research questions

At this point, having given an outline of the concepts involved, I can state the research questions:

- Can incorporating inline deduplication techniques improve open source backup offerings and produce an empirically superior product?
- Is sparse indexing an effective solution to the disk deduplication bottleneck?

1.9. Aims and objectives

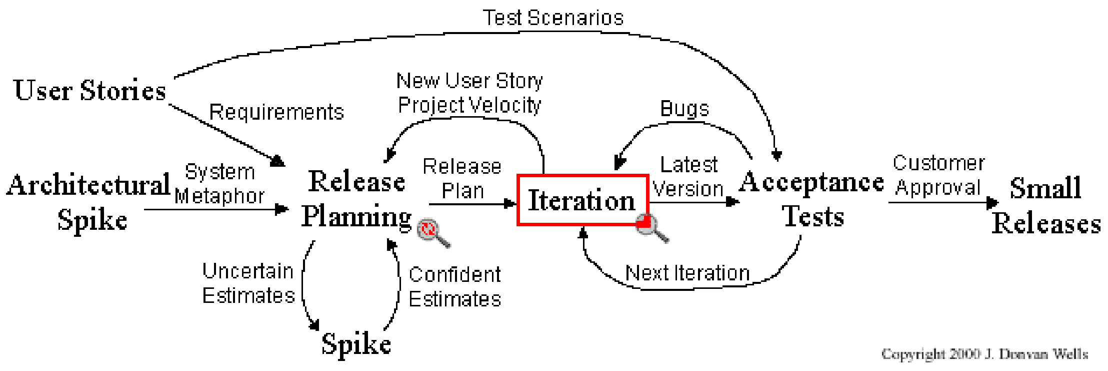
I can also now state that the main aim of the project is to improve upon the existing 'burp' software to produce the basis of a new open source cross platform network backup software that is demonstrably superior to others in the same field.
To achieve this, I have produced a list of project objectives. They are divided into 'core' and 'advanced' categories, as was required for the Extended Project Proposal.

- *Core:* Complete a search and review of popular existing open source network backup solutions, explaining the areas in which current backup offerings are flawed.
- *Core:* Complete a literature search and review of relevant algorithms and techniques that will be needed to implement a new software engine.
- *Core:* Design and develop the new software.
- *Advanced:* By conducting suitable tests and analysis, prove that the new software is superior to other offerings.
- *Advanced:* Demonstrate that sparse indexing is an effective solution to the disk deduplication bottleneck.
- *Core:* Complete the final project report.

2. Software development methodology

I generally followed the Agile method of XP ('Extreme Programming'), having previously found that it worked reasonably well in a previous computer science module.

Wells (2009) provides the following diagram, which gives a general outline of the XP process.



The coding was done on a Debian Linux platform, in C, although C++ was considered. A major factor influencing this decision was my more extensive experience with C, meaning that I would be able to utilise the limited time more effectively, rather than dealing with problems arising out of C++ inexperience. However, a considerable effort was made to make sure that my C structures followed object oriented principles. Another factor in the decision to use C was that most of the original burp code was written in C, albeit with the occassional C++ module for interfacing with Windows APIs.

Most of the testing during development was done on the Debian Linux platform, with occassional testing using a Windows client cross-compiled with a mingw-w64 toolchain. This enabled quick progress whilst ensuring that nothing major was broken on the Windows platform.

During the implementation of the second iteration in particular, once it was clear that essential functionality would be complete before the project deadline, a fair chunk of time was spent cleaning up existing burp code. Code redundancy was reduced and old code started to be shifted towards a more object-oriented approach. The structure of the source tree was altered significantly in order to logically separate server and client code. This was all done for the purpose of reducing 'technical debt', which is very desirable from an Agile viewpoint.

2.1. Initial design decisions

I now needed to make some design decisions on exactly which algorithms to use, and the changes that needed to be made to the design of burp to support these new proposals. However, I did feel that it was important to design the software in such a way that I could easily switch between different choices of algorithms at some point in the future. It is not actually greatly important to the design and implementation of the software that I choose the best algorithms from the start.

I chose MD5 for the strong checksum. This was because I was familiar with it (and MD4) via experience with rsync, librsync, and the original burp. The original burp uses MD5 in order to checksum whole files. It is a suitable, low risk choice for strong chunk checksums in the new software. An MD5 checksum also fits into 128 bits, which saves on memory compared to something like the SHA family (Eastlake, 2001) - which use a minimum of 160 bits - but still provides good uniqueness. That is, two different chunks of data are vastly unlikely to have matching MD5 checksums. For a match, the blocks also need to have matching weak checksums, which further reduces the chances of collisions. At some future point, experimentation with other algorithms may be performed though.

Lillibridge, M. et al, used their own Two-Threshold Two-Divisor chunking algorithm (Eshghi, 2005), and assert that it "produces variable-sized chunks with smaller size variation than other chunking algorithms, leading to superior deduplication".
However, I chose 'rabin fingerprinting' (Rabin, 1981) for this task. It gives a suitable weak checksum and the implementation that I produced seems suitably fast, as will be seen in the test results. Again, at some future point, experimentation with other algorithms may be performed.

3. Proof of concept programs

In order to ensure a solidly functional rabin algorithm and disk storage format before embarking on the complex asymetric network I/O that would be required for the first iteration of the software, I designed and implemented some proof of concept programs. I class this as 'release planning' in the XP process - my estimates were uncertain until I went through a 'spike' and experimented with rabin and disk storage. This made estimates more confident.

The idea for the prototype was to have programs that could use simple I/O so that data streams could simply be piped between them, or written to or read from files.

3.1. Design

Due to lack of space in the main project report, the design documents that I wrote for the proof of concept programs are reproduced in Appendix B.
The implementation of these programs was completed in the middle of June 2013.

3.2. Rabin fingerprinting

A reproduction of the C function containing the rabin algorithm that was produced for the proof of concept programs can be found in Appendix C. I judge this as being the most important outcome from that exercise, although it was modified slightly during later iterations in order to support slightly different usage required in the final software where asyncronous I/O had to be taken into account.

Note that the full code of the prototype can be found in the extra materials submitted with this report.

4. The first iteration

The plan for the first iteration was to replace the core of the original burp with an inline deduplicating engine that uses variable length chunks. In the prototype, I had already produced the basic parts of this engine, so the problem became how to merge this into burp.

4.1. Design

Due to lack of space in the main report, the design documents that I wrote for the first iteration are reproduced in Appendix D.
The implementation of the first iteration was completed on the 2nd of September 2013. I experienced a lot of trouble getting the asymmetric I/O and stream multiplexing to work correctly, but since the 2nd of September, it has been solid.

5. Comparative testing

Once the first iteration was basically working, I designed a procedure with which to test the various backup solutions.

The testing was done using a Linux server and client, because this greatly simplifies setup and the measurement of resource utilisation. The machines were connected to each other using a 100Mb/s switch.

Server

CPU:	Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz (double core)
RAM	4GB:
OS	Linux version 3.2.0-3-amd64 (Debian 3.2.21-3)
Disk 1:	ATA WDC WD400BB-00JH 05.0 PQ (40GB - for the OS)
Disk 2:	ATA ST3400620A 3.AA PQ (400GB - for the storage)

Client

CPU:	Intel(R) Atom(TM) CPU D510 @ 1.66GHz (quad core)
RAM:	4GB
OS:	Linux version 3.2.0-4-amd64 (Debian 3.2.46-1+deb7u1)
Disk 1:	ATA SAMSUNG HD501LJ CR10 PQ: 0 ANSI: 5 (500GB)

There were two test sequences for each backup software. In both cases, files were initially copied into a directory on the client computer for the purposes of being backed up.
a) Many small files - I downloaded 59 different linux kernel source packages from http://kernel.org/ and unpacked them.
This resulted in 1535717 files and directories, and 20GB (20001048kb) of data, which is an average of about 13kb per file.
b) One large file - I used a 22GB VirtualBox VDI image file of a Windows 7 machine. I took one copy of this, started and stopped the Windows 7 virtual machine then took another copy of the file, which was now changed.

Each sequence had the following steps, each of which is targetting potential weaknesses of backup software. For example, updating the timestamp of a large file could cause the whole file to be copied across the network even though none of the data has changed.

1. Perform a backup.
2. Perform a backup without changing anything.
3. Perform a backup after changing some of the data.
   For the small files, I randomly scrambled the files in one of the kernel directories.
   For the large file, I used the rebooted VDI image.
4. Perform a backup after updating the timestamp on some of the files.
   For the small files, I updated all of the timestamps in one of the kernel directories without changing the data.
   For the large file, I updated its timestamp without changing its data.
5. Perform a backup after renaming some of the files.
   For the small files, I created a new directory and moved half of the kernel sources into it.
6. Perform a backup after deleting some of the files.
   For the small files, I deleted half of them.
   For the large file, I truncated it to 11GB.
7. Restore all the files from each of the six backups.

These measurements were taken for each backup or restore:

- The time taken.
- The cumulative disk space used after each backupe
- The cumulative number of file system nodes used by the backup.
- Bytes sent over the network from the server.
- Bytes sent over the network from the client.
- Maximum memory usage on the server.
- Maximum memory usage on the client.

I would also have liked to measure the CPU utilisation, but I was not able to find a satisfactory practical way to do this for each piece of software.

To get the time taken and the memory usage statistics, I used the GNU 'time' program. To get the disk space statistics, I used the 'du' command. To get the number of file system nodes, I used the 'find' command, piped to 'wc'. To get the network statistics, I was able to use the linux firewall, 'iptables', to count the bytes going to and from particular TCP ports.

Immediately before each test, I would reset the firewall counters and flush the disk cache on both server and client.

Since each test sequence might take a lot of time, scripts were written to automate the testing process so that they could be run without intervention. The scripts had to be customised for each backup software under test. These scripts are included with the software as part of the submitted materials for this project.

I ensured that each software was configured with the same features for each test; there would be no compression on the network or in the storage, and there would be encryption on the network but not in the storage. For those software that had no native network support, this meant running the software over secure shell (ssh).
During the initial testing, I discovered that burp's network library was automatically compressing on the network, leading to incorrect results. I made a one line patch to the versions of burp under test in order to turn off the network compression. This will become a configurable option in future versions. I redid the initial testing of both versions of burp for this reason.

These are the candidate software to be initially tested. For more verbose information on each of these, see the Bibliography and Appendix F.
burp-1.3.36 was the latest version of the original burp at the time the testing was started.
burp-2.0.0 is the first iteration of the new software.

Software	**Good cross platform support**	**Native network support**	**Good attribute support**	**Good imaging support**	**Notifications and scheduling**	**Retention periods**	**Resume on interrupt**	**Hard link farm**	**Inline deduplication**
amanda 3.3.1	No	No	Yes	No	Yes *	Yes	No	No	No
backshift 1.20	No	No	Yes	No	No *	Yes	Yes	No	Yes
backuppc 3.2.1	No	No	Yes	No	Yes	Yes	Yes	Yes	No
bacula 5.2.13	Yes	Yes	Yes	No	Yes	Yes	No	No	No
bup-0.25	No	No	No	Yes	No *	No	Yes	No	Yes
obnam 1.1	No	No	Yes	Yes	No *	Yes	Yes	No	Yes
rdiff-backup 1.2.8	No	No	Yes	No	No *	No *	No	Yes	No
rsync 3.0.9 --link-dest	No	Yes	Yes	No	No *	No *	Yes	Yes	No
tar 1.26	No	No	Yes	No	No *	No *	No	No	No
urbackup-1.2.4	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
burp 1.3.36	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No
burp 2.0.0/1	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Yes

* Possible with management via external tools, such as cron.

'Good cross platform support' means that the software is able to back up and restore a set of files on Unix/Linux/Mac/Windows computers, and is able to use the Windows backup API.
'Good imaging support' means that the software is able to perform image backups and restores efficiently.
'Hard link farm' means that the software saves its data in individual files, one for each file that is backed up, and may hard link unchanged versions together. This can be beneficial on small backup systems, and you can copy the files for restore using standard file system tools. However, with large backup sets, they become unwieldy due to file system overhead.

5.1. Conclusions from feature comparison

When comparing the feature lists prior to doing any testing, it seems that the original burp is already one of the best choices in the field, and some of its deficiencies are addressed by the newly developed software.

For example, the only other software with good cross platform support is bacula. None of the solutions offering inline deduplication except the newly developed software manage to have good cross platform support.
I expected urbackup (Raiber, 2011) to be a good contender, but it turns out that it doesn't work well on Linux, as described in the next section.

Many of the technically interesting offerings, such as bup-0.25 (Pennarun, 2010), lack features that help with simplifying administration of a central server. A few of them, for example backshift (Stromberg, 2012) and obnam (Wirzenius, 2007), are not really network based solutions and require remote filesystems to be externally mounted so that they appear to the software as local filesystems. These suffer appropriately in the testing that follows.

You may have noticed that the newly developed software has lost the ability to delete old backups. This will be addressed in a future iteration beyond the scope of this project, but the planned concept is explained in a following chapter about further iterations.

6. Initial test results

6.1. Software that did not complete the tests

backshift 1.20: I had to 'disqualify' this software, because its first backup took over 43 hours to complete. Most other software took less than two hours for the first backup, so there was not much point in continuing to test backshift. The raw figures for its first backup are included in Appendix G.
It seems that backshift was suffering from the disk deduplication bottleneck problem, since it looked like it was using one file system node for each chunk that it had seen, named in a directory structure after some kind of checksum. Due to the way that it can only back up over the network on a network mounted share, it was doing file system look ups for each incoming checksum over the network. This must badly exacerbate the disk bottleneck problem.

obnam-1.1: This software also had problems with the first backup that meant there was not much point in continuing to test it.
It did not finish after eight hours, had taken up more than 50GB of space on the server, and that space was rising. This was suprising as there were only 22GB of small files to back up.

urbackup 1.2.4: Although this software boasts an impressive feature list, I was not able to complete the tests with it.
The main problem was that restoring more than one file at a time was impossible, and even that had to be done with point-and-click via its web interface. That made restoring a few million files impractical.
The online manual states "the client software currently runs only on Windows while the server software runs on both Linux and Windows". However, at the time of running the tests, I did find source for a Linux client, and its mode of operation was quite interesting and unique - the server broadcasts on the local network, and backs up any clients that respond. If this software gains better Linux support in the future, I think it will be one to watch.

6.2. Software that did complete the tests

The rest of the software completed the tests. I took the data and created a series of simple graphs. The intention is to make it easy for the reader to interpret the results. I have also included the raw data from the tests in Appendix G.

6.3. Areas in which the new software did well

In this section, I present the graphs representing the areas in which the software developed in the first iteration did well. After that, I present the areas in which the new software did not do so well.

In the graphs, I coloured the line representing the original burp red, the line representing the new software green, and everything else grey. This is so that it easy to see the change in performance over the original burp whilst still enabling comparisons against other software.

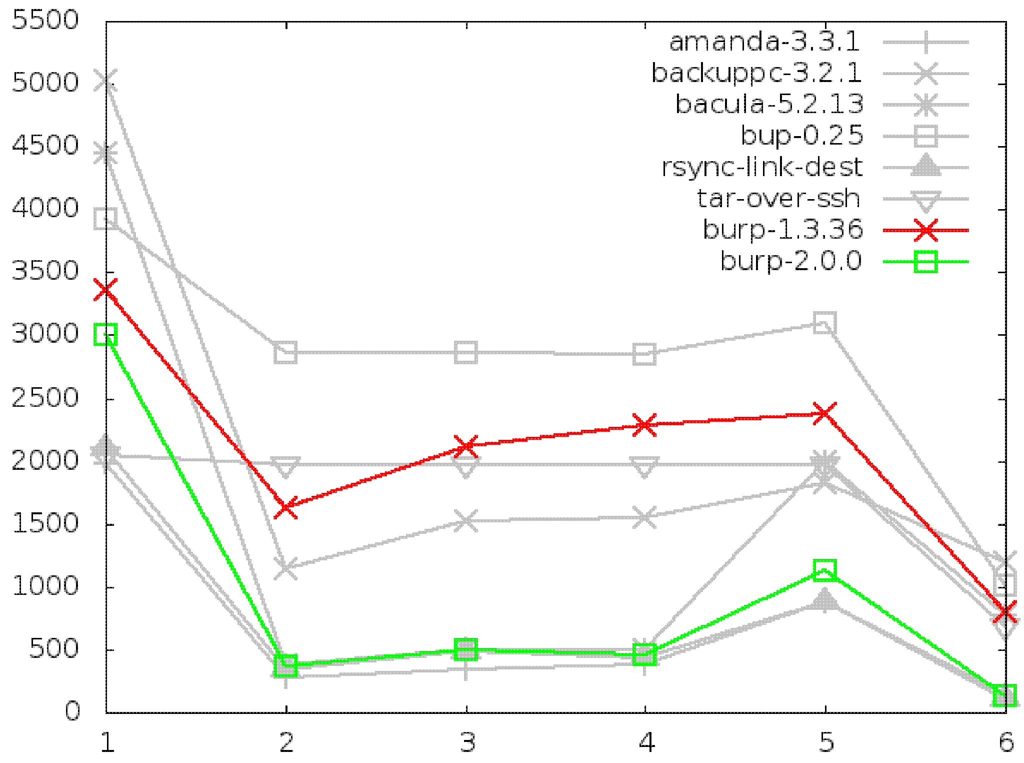
6.3.1. Time taken to back up

The time taken to back up was considerably improved over the original burp, particularly in circumstances where lots of data changed between backups. It was mostly faster than, or comparable to, other software in both small and large file test sequences. The slight exception was the first backup that the software makes, although even that is within an acceptable time frame.
I believe that this is because the software needs to checksum all the data that it finds on its first run, and does not gain any advantage from time stamp comparison of a previous backup. Software like tar (GNU, 1999) sends the data directly without calculating checksums, and will probably always win on a fast network in terms of the speed of the first backup. However, it will probably lose in terms of disk space used. Note also that, because amanda (da Silva et al, 1991) uses a special mode of tar to do its backups, its line closely follows that of tar and does well in this test.

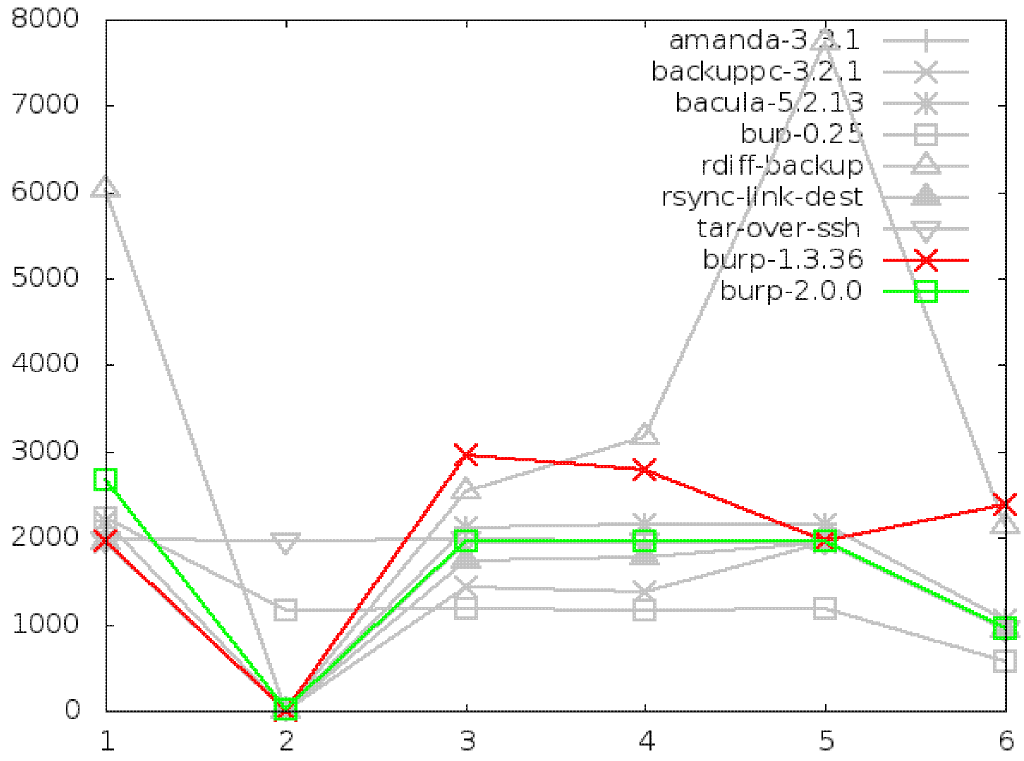
The software that was generally fastest at backing up the large file was bup-0.25. However, it is notable that it didn't perform well when that file had not been touched between backups (the second test in the sequence). This is because although bup does inline deduplication, it always reads in all the data, and doesn't do a check on file time stamps. This may be of concern if your data set contains large files that don't change very often - for example, iso images or movie files. It is notable too that it was generally slower than most software in the small files test.

In the first of the following graphs, I have excluded rdiff-backup because its fifth and sixth backups took about 11 hours and 12 and a half hours respectively. Since all the other software took less than 2 hours, it was making the rest of the graph hard to read.

Time taken to back up small files, in seconds



Time taken to back up large file, in seconds



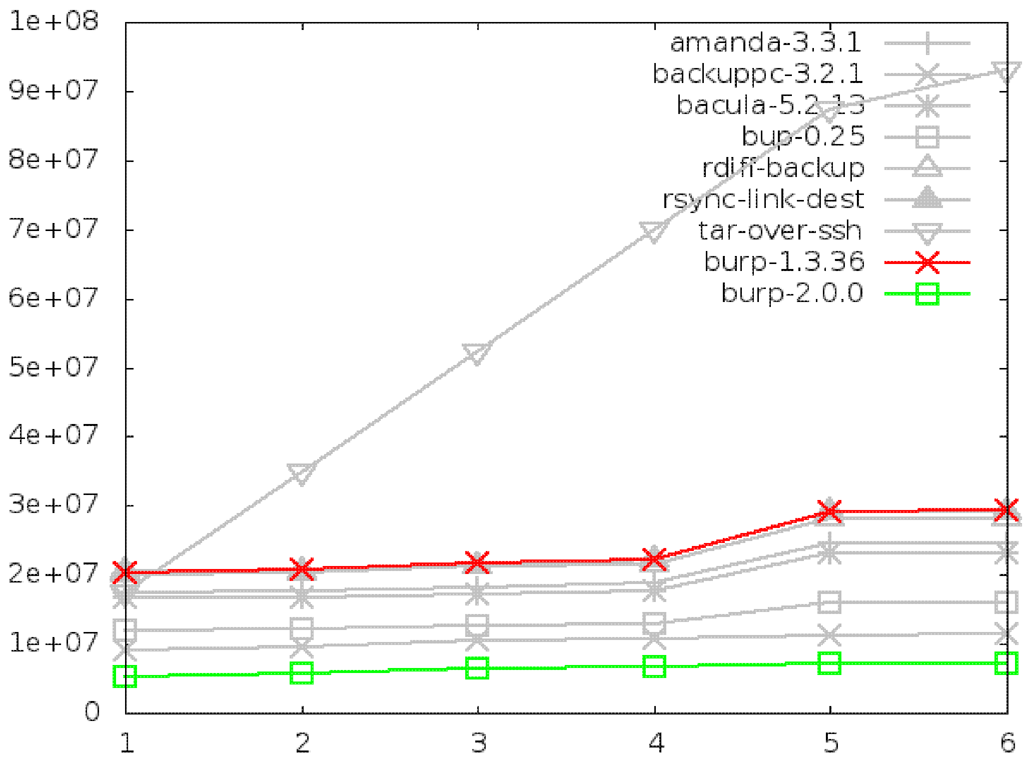
6.3.2. Disk space utilisation after backing up small files, in kbytes

The disk space utilised on the server after backing up small files was an area in which the new software performed considerably better than the competition. For both small and large file tests, it took up around 60% less space than the best of its rivals.

The difference between the original burp and the new burp by the sixth test in this graph is around 23GB.

I have to admit that I was expecting bup-0.25 to do a bit better than it actually did in this test. It is very gratifying that the new software used about half the space that bup did, because bup was the only remaining contender with an inline deduplication feature.
Backuppc also performed impressively well, falling in the middle between bup and burp-2.0.0. The file-level deduplication that it uses appears very effective on the data set.

One more notable thing about this graph is that it clearly shows the disadvantage of backing up all the files every time - tar uses about 93GB of space in the end.



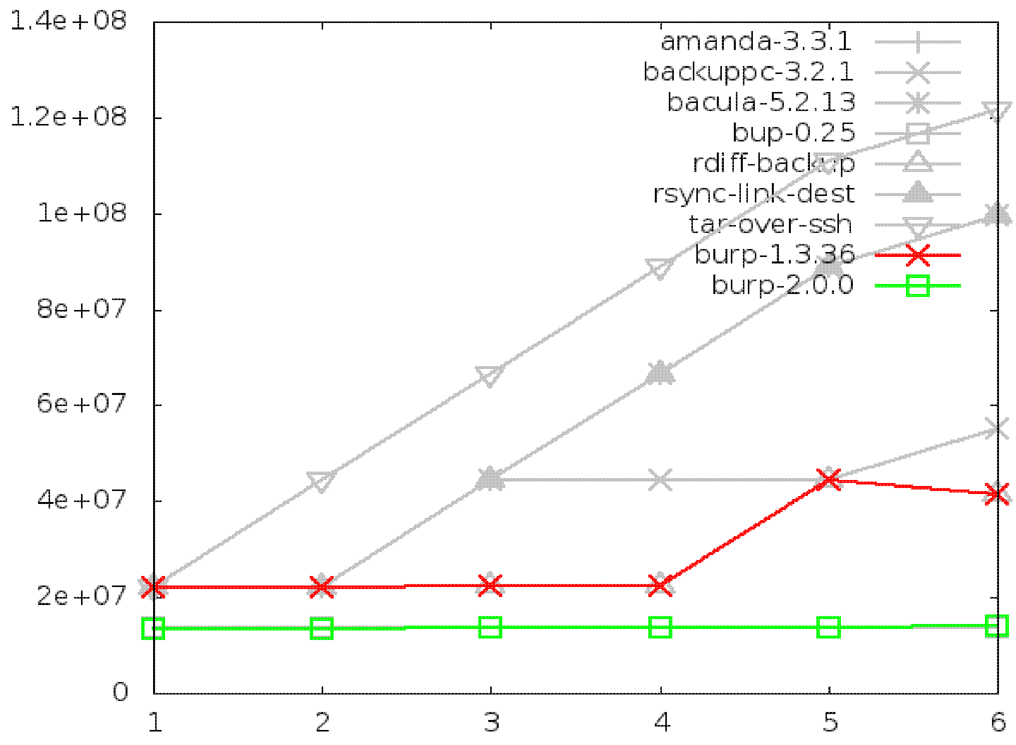
6.3.3. Disk space utilisation after backing up large file, in kbytes

The disk space utilised on the server after backing up the large file was another area in which the new software performed better than the other contenders - that is, except for bup-0.25. This is not completely obvious on the graph because the two lines are on top of one another. In fact, they are so close that burp-2.0.0 uses less space for the first three backups, and then bup-0.25 uses less for the last three. After the sixth backup, there is only 181MB between them. Note that bup-0.25 actually stores some files on the client machine as well as the server, which is something that no other software does.

This graph clearly demonstrates the weaknesses of software that has to store changed files as complete lumps. Again, tar does particularly badly, but this time so do amanda, backuppc (Barratt, 2001), bacula and rsync, all lying on almost exactly the same line not far beneath tar.

Once more, backuppc's file-level deduplication performs surprisingly effectively as it identifies the file with the same content at step four (timestamp update without changing file contents) and step five (rename).

Finally for this test, burp and rdiff-backup perform identically. This is because they are both saving old backups as reverse differences with librsync.

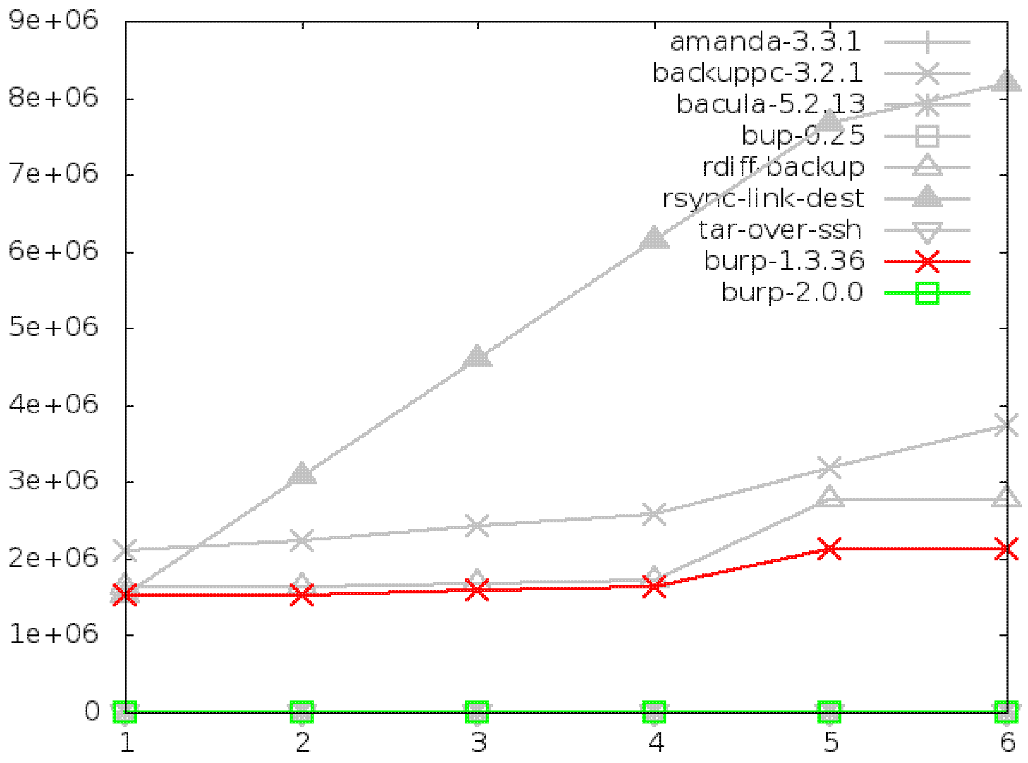


6.3.4. Number of file system entries after backing up small files

For the small files, the new software used fewer file system nodes than the 'link farm' solutions by several orders of magnitude, which is to be expected. It was comparable to the non-'link farm' solutions. This is because it can pack the chunks from many small files into far fewer data files.

Creating the largest 'link farm' by far was rsync, which is to be expected because in '--link-dest' mode, it creates a mirror of the source system for every backup. You may have noticed rsync performing consistently well in the speed results, but this is really where it falls down as a versioned backup system, because the number of file system nodes becomes unmanageable. Here, there are eight million nodes for six backups.

I am somewhat surprised to see the original burp do better than the other 'link farm' solutions. On reflection, it must be because it has a mechanism whereby older backups do not keep a node referencing a file that is identical in a newer backup.

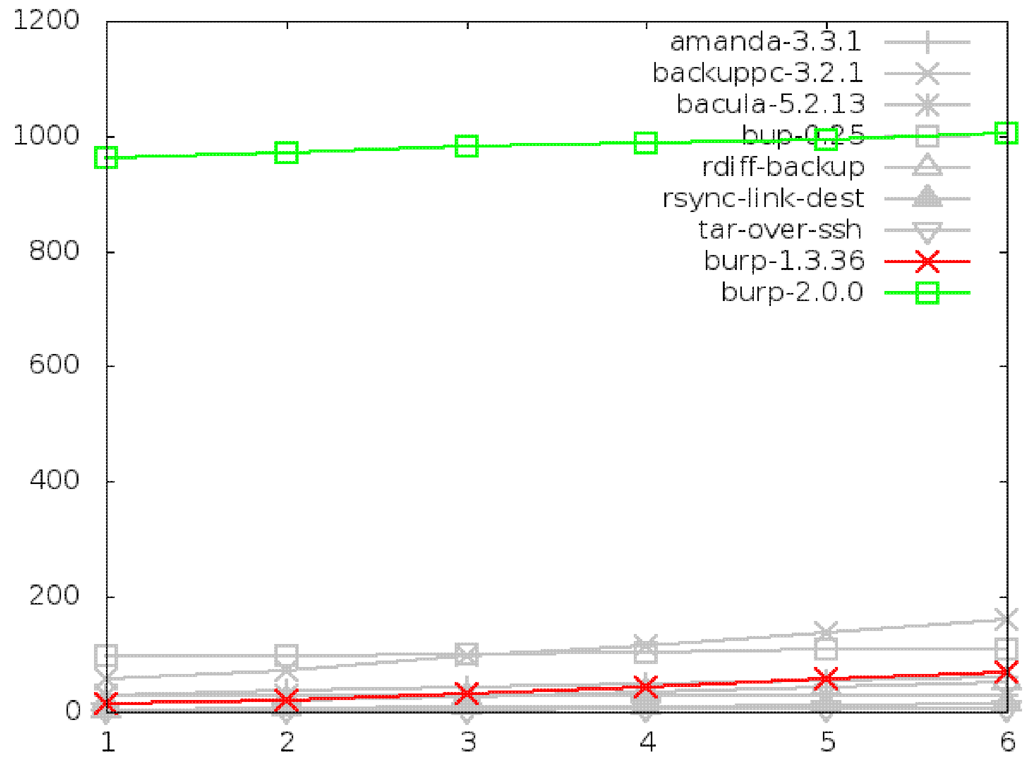


6.3.5. Number of file system entries after backing up large file

For large files, it is notable that the new burp creates more nodes than all the other solutions.

This is because chunks making up the file need to be split over several data files, whereas the 'link farm' solutions will only use one node. Also, it appears that the inline deduplication of bup-0.25 packs its chunks into fewer, larger data files than burp-2.0.0.

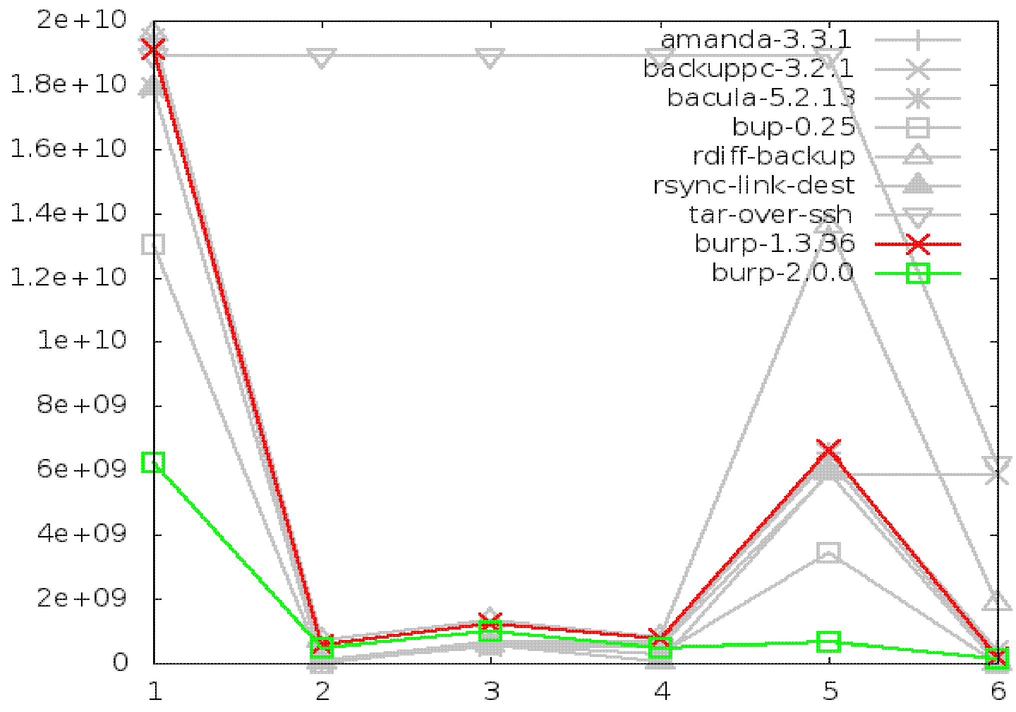
At first glance, this doesn't look like a good result for burp-2.0.0. However, in reality, it is more than satisfactory, because it still only used a manageable figure of around 1000 nodes for the large file tests anyway.
Further, this figure will not vary very much if identical data were spread across multiple files and then backed up - whereas software using a node for each file would increase linearly with the number of files.



6.3.6. Network utilisation when backing up small files, in bytes

When backing up small files, the new software performed better overall than other solutions in terms of network utilisation during backup. This is because the way that it deduplicates means that it only needs to send chunks that the server has not previously seen.
If you look closely (or check the raw data), you can see that amanda, backuppc, bacula, bup, and rsync all beat it at steps two, three, and four. I am not entirely sure why this might be. Perhaps it is because burp sends a complete scan of the file names and statistics for every backup, and maybe the other software have a more efficient way of doing the same thing. This deserves investigation at some future time.
However, this is more than made up for by the massive differences between the new software and the nearest rival at steps one and five, where it uses less than half the bandwidth of bup-0.25.

Also worth pointing out with this graph is the strange behaviour of backuppc on its last backup, where I would have expected the figure to drop, as all the other backups did. At first I thought this was a transcription error, but the figure in the raw data looks plausible, and different from the figure at step five. I cannot explain this, and it may be worth testing backuppc again to see if this behaviour is repeated.

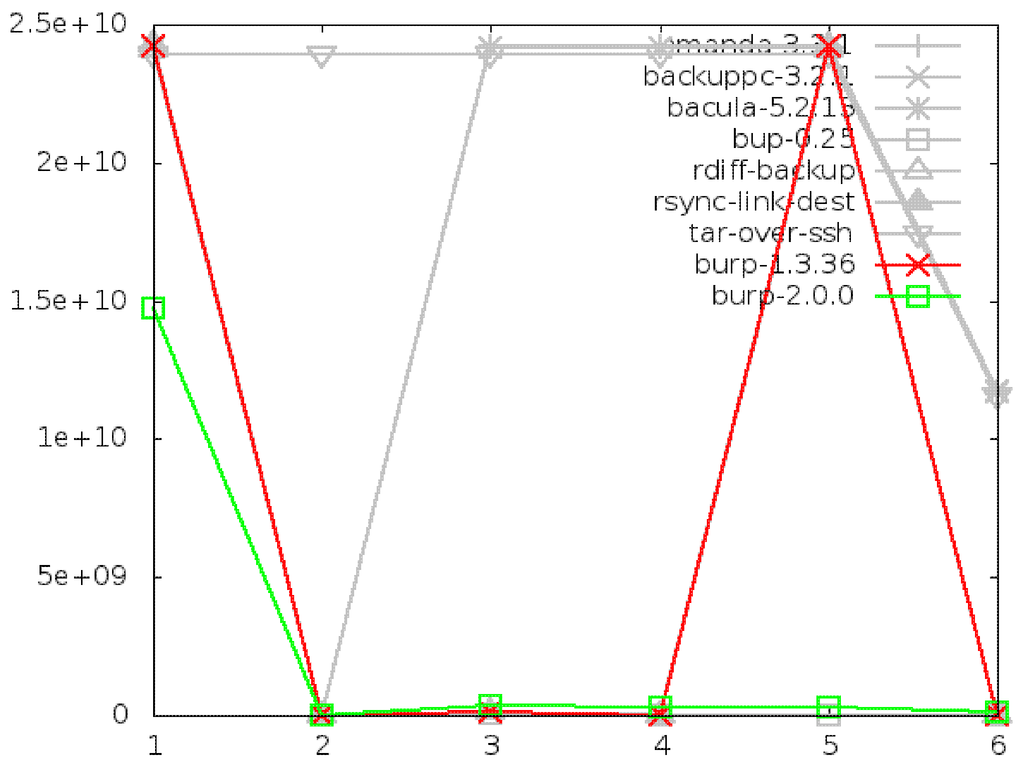


6.3.7. Network utilisation when backing up large file, in bytes

Once more, the new software generally performs better than the competitors in terms of network utilisation when backing up large files, with the exception of bup-0.25, which is so close it overlaps with burp-2.0.0's line.

What jumps out at me from this graph is how badly all the software except the inline deduplicators (bup-0.25 and burp-2.0.0) do when the file to back up is renamed (step five) - this causes them to send 25GB across the network, as opposed to burp-2.0.0's 290MB, and bup's very impressive 63KB.

In fact, I have to concede that, if you look at the raw figures, bup clearly performs the best in this test, with burp-2.0.0 a few hundred megabytes behind. It should be noted though that there is a massive void between burp-2.0.0 and the next nearest competitor.



6.3.8. Network utilisation when restoring small files

We find that the new software was comparable to the other solutions in terms of network utilisation during restore, and all the solutions except amanda followed a nearly identical path.

Tar did the best in the category, although the difference was minor. I believe that burp's performance here could be considerably improved by sending the blocks and instructions on how to put them together instead of a simple data stream. There is more on this idea in the 'future iterations' section of this report.

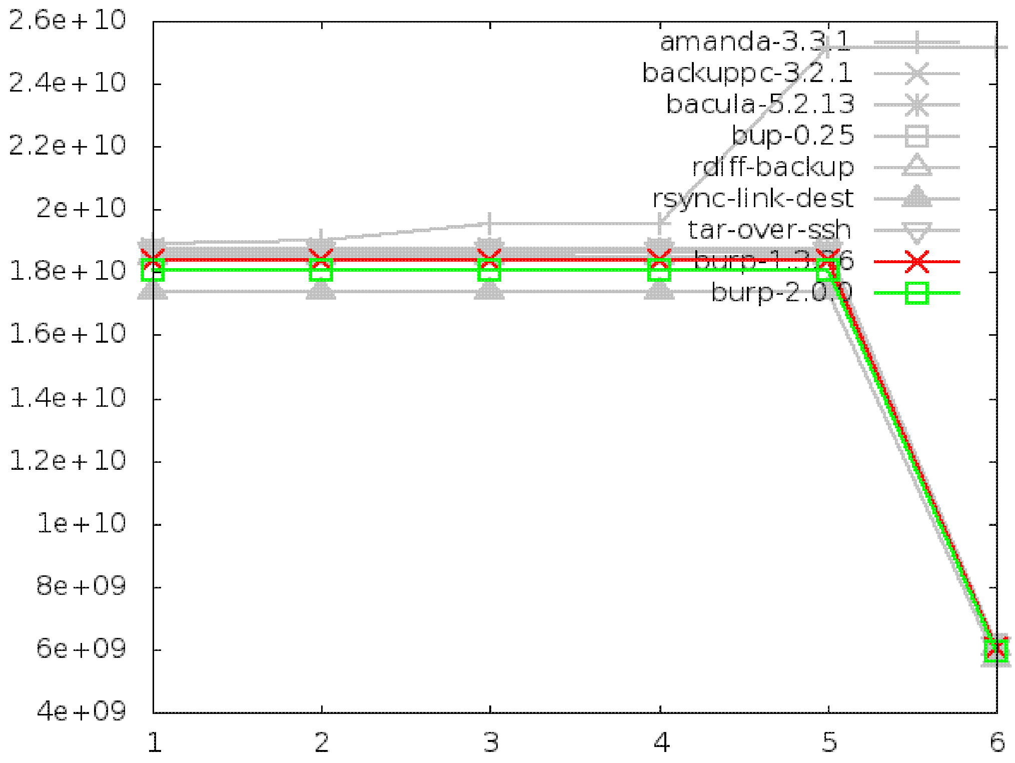
The other notable remark to make about this test was the strange behaviour of amanda on the later restores, where it suddenly leaves the path set by all of the other software and the network usage shoots up.

Amanda has an odd tape-based mentality, and on disk, it creates files that it treats like tapes, with one backup per tape (file).
When you ask it to restore all files from incremental backup 5, it will go through each previous 'tape' in turn, and restore everything that doesn't have an identical name in a subsequent backup. If a file was deleted in a subsequent backup, it will delete the file that it just restored.

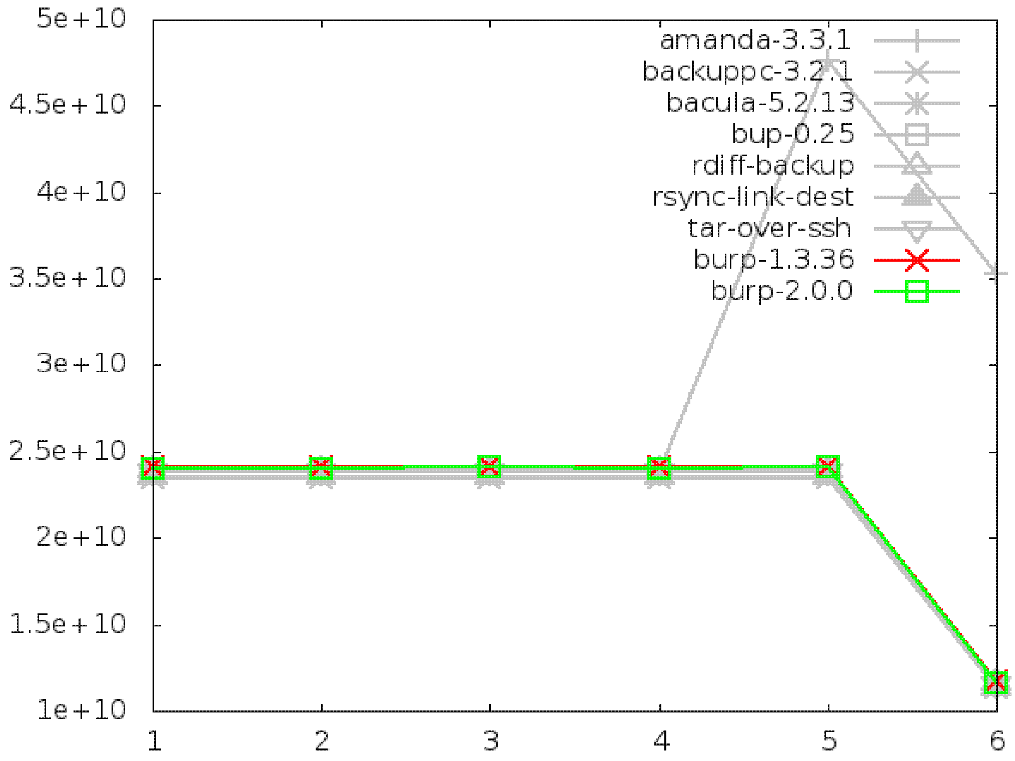
In the case of the large file test, this means that nothing odd happens until the file is renamed in step 5. When restoring that, amanda first restores the original file name, then deletes it and restores the renamed file. Since it restored two large files, the network utilisation is high.

This is clearly very inefficient behaviour when you have the ability to seek to any point in a disk much faster than you can with tapes.

Network utilisation when restoring small files, in bytes



Network utilisation when restoring large file, in bytes



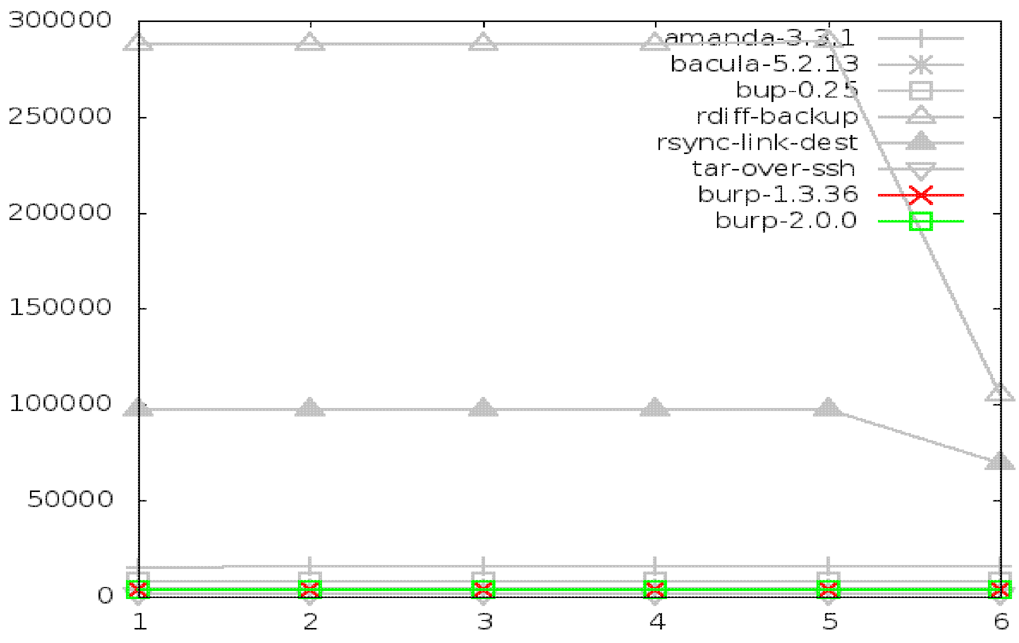
6.3.9. Maximum memory usage of client when restoring small files

Before analysing the data in the graphs in this category, it should be noted that, for the software that uses ssh for its network transport mechanism, the memory usage of ssh itself was not captured. At the very least, this would add a few thousand KB to their figures. The software that this should be considered for are amanda, backuppc, bup, rdiff-backup, rsync and tar. All of them except bacula and burp, which have their own network transport mechanism.
With this in mind, it should be clear from the following graphs that both the original and new versions of burp perform the best in terms of memory usage on the client side during restores. The client restore mechanism didn't change between the two burp versions, but the server side restore did. This is demonstrated in the next section.

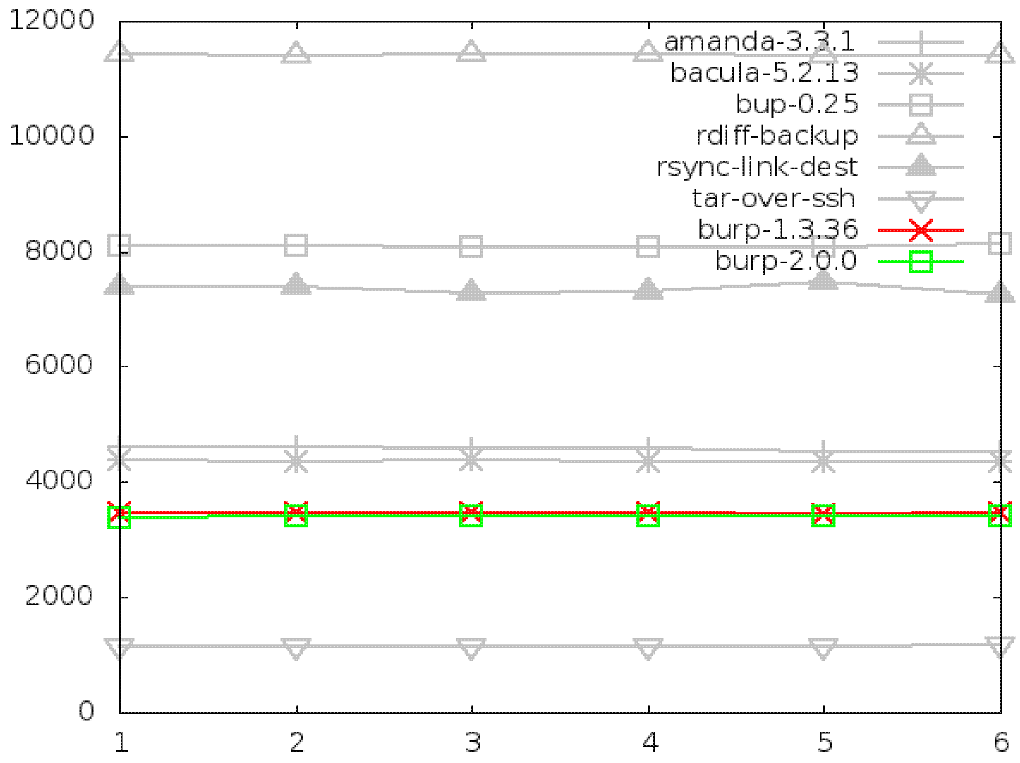
Note also that I was unable to measure the client memory usage of backuppc during the restore, so it doesn't appear in the graphs. However, the result would have been equivalent to tar over ssh, since that is the mechanism that backuppc uses to restore files on Linux.

Coming out badly in the small file category were rsync and, in particular, rdiff-backup which did three times worse than rsync. For the large file category, there wasn't actually much material difference between the contenders.

Maximum memory usage of client when restoring small files, in kbytes



Maximum memory usage of client when restoring large file, in kbytes



6.4. Areas in which the new software did not do so well

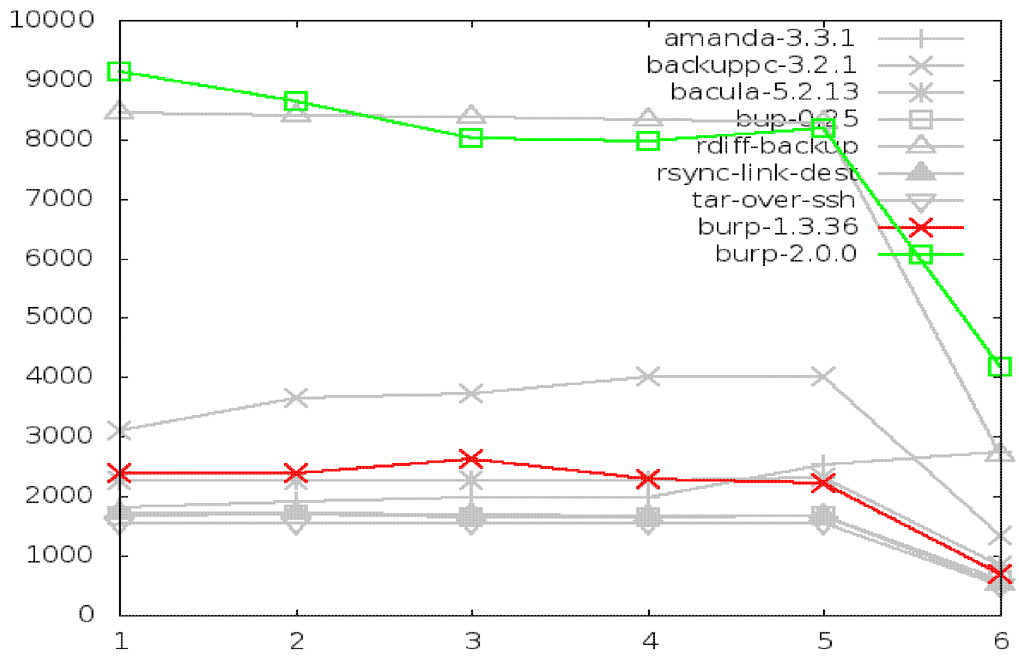
6.4.1. Time taken to restore

Firstly, the time taken to restore files takes twice as long as the original burp, and at least four times as long as solutions like tar and rsync that are not having to reassemble files from disparate chunks in a selection of storage files.

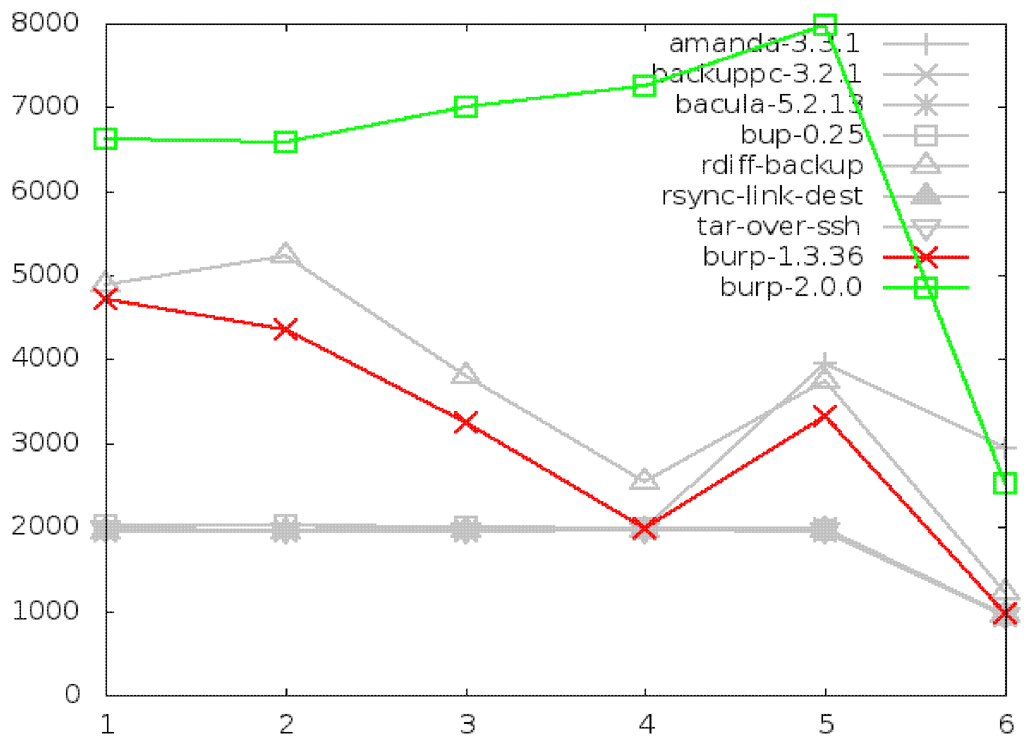
This is to be somewhat expected. It could be argued that, since a restore happens far less often than a back up, that this state of affairs is acceptable - time regularly saved during backing up is likely to exceed the time spent waiting longer for an occasional restore. Nonetheless, I will attempt to improve the restore times in later iterations because it is understood that people want to recover quickly in a disatrous situation.

A result that I found unexpected here was the impressive speed of bup-0.25, which is in the same region as tar and rsync. Since bup, like burp-2.0.0, has to reassemble chunks from its storage, I was expecting it to be slow. So how does it manage this? A probable reason for this kind of performance can be found in the later section about server memory usage.

Time taken to restore small files, in seconds



Time taken to restore large file, in seconds



6.4.2. Maximum memory usage of server when restoring

This is an area in which bup-0.25 does spectacularly poorly, using up all the memory available on the server. This is how it manages to restore so quickly, because it must be loading all of the chunks it needs to send into memory. By that, I mean all of the data and probably a full index of all the checksums as well. This means that it can look up each chunk to send very rapidly, making it nearly as fast as software that only needs to send the data without doing any lookups.

The memory usage of bup-0.25 (or more specifically, the 'git' backend that it uses to do the heavy work) is so bad that it makes burp-2.0.0 look reasonable by comparison, when it really isn't. If bup were not being tested, burp-2.0.0 would be the worst performer by a large margin, using 1.5GB consistently.

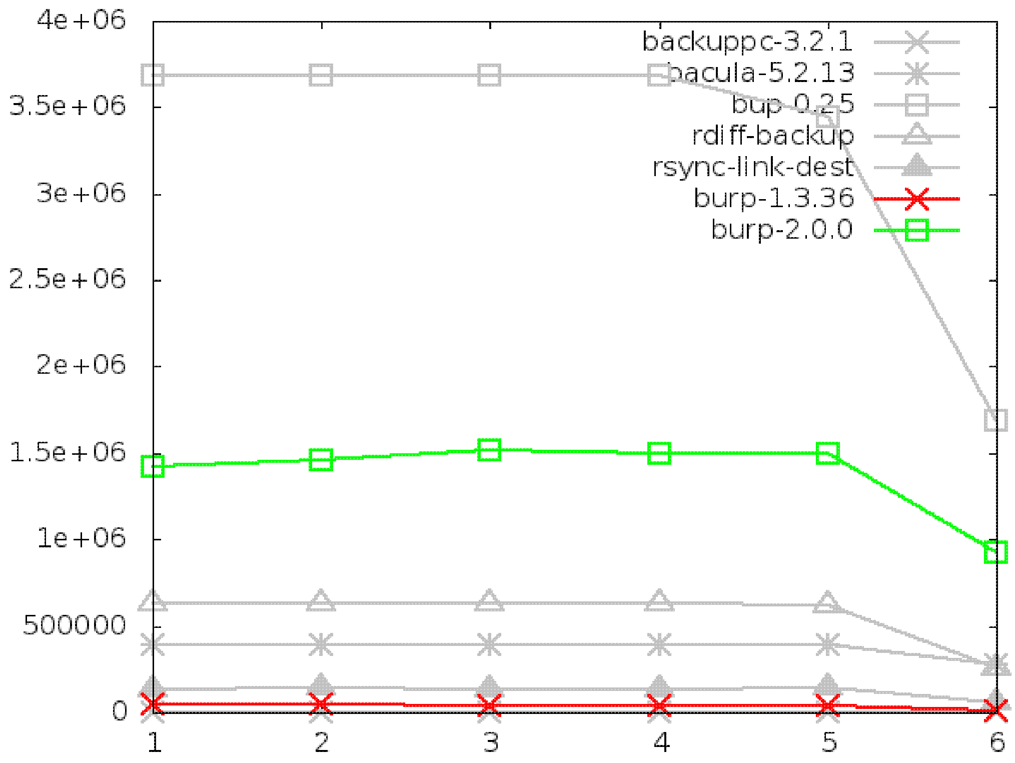
This is something that I will attempt to improve in the second iteration, by improving the storage format and lookup algorithms so that the server holds less data in memory at any one time.

Tar is missing from the graphs because I was unable to capture the server ssh memory usage. Although, as it was the only server process running, it would have used minimal memory anyway.

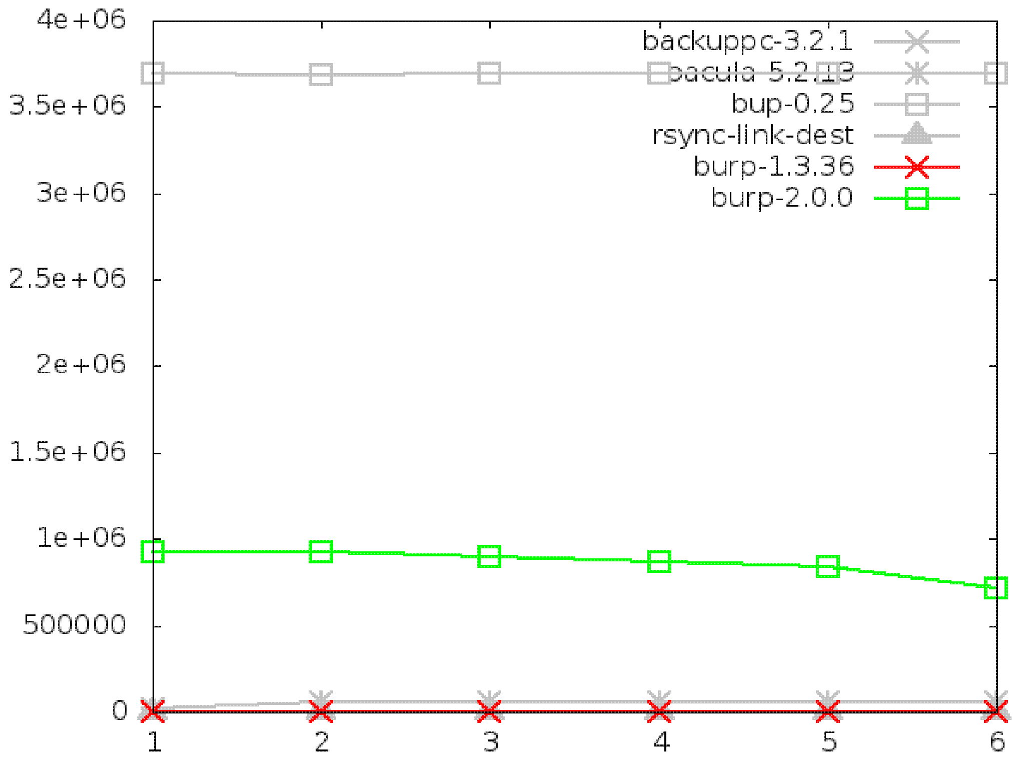
Amanda is also missing from the graphs. Due to the complexity of the multiple child processes involved in an amanda restore, I was not able to capture server memory figures for it. I would estimate, based on knowledge of how it works, that its memory usage would be minimal.

And rdiff-backup is also missing from the graphs. I was unfortunately unable to capture figures for it when restoring large files, despite several repeated attempts.

Maximum memory usage of server when restoring small files, in kbytes

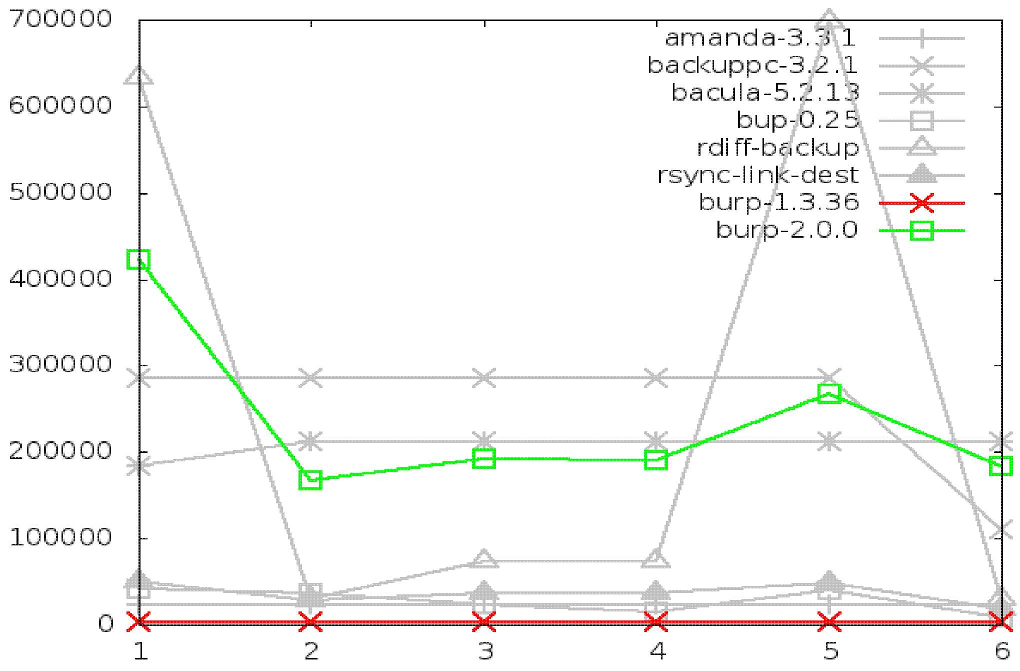


Maximum memory usage of server when restoring large file, in kbytes



6.4.3. Maximum memory usage of server when backing up small files, in kbytes

The memory usage of the new software when backing up small files is an area in which the new software, although not the worst performer, has scope for improvement. This is because it is loading all the chunk checksums that it has ever seen, and their locations, into memory in order to perform the inline deduplication. The second iteration of the new software will attempt to mitigate this by implementing sparse indexing. This will mean that only a small percentage of checksums and their locations will be loaded into memory.
It is notable that the original burp does very well in this test. It generally holds a minimal amount of data in memory, and when reading files, it tends to fill a fixed sized buffer and process it straight away, rather than reading the whole of the file into memory. Other solutions doing well were amanda, bup-0.25, and rsync.
Some might observe that bacula is not doing well on any of the server memory tests. It has a severe handicap in this area, because it relies on a mysql database process, which was measured as part of its tests.
There are large peaks on the first step and the fifth step (rename) for rdiff-backup, showing that it uses far more server memory when there are new files to process.
Tar is missing from the graphs because I was unable to capture the server ssh memory usage. Although as it was the only server process running, it would have used minimal memory anyway.



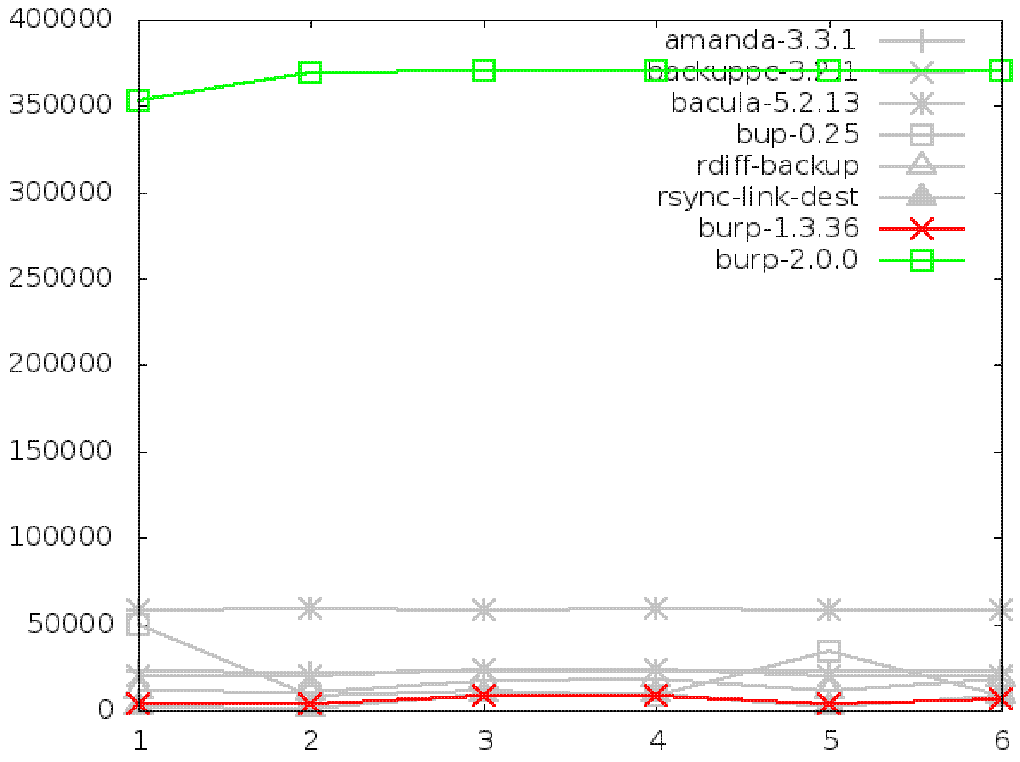
6.4.4. Maximum memory usage of server when backing up large file, in kbytes

This is clearly an area in which burp-2.0.0 does spectacularly poorly, using seven times as much memory as bacula, the next worst out of the contenders.

Again, this is because it is holding a full index of all the checksums in its memory, and this is something that will be addressed in the second iteration by implementing sparse indexing.

All the other software, except bacula, performed in a similar range to each other. There are obvious peaks for bup-0.25 when there are new file names to process, but it never gets worse than bacula. The original burp does very well.

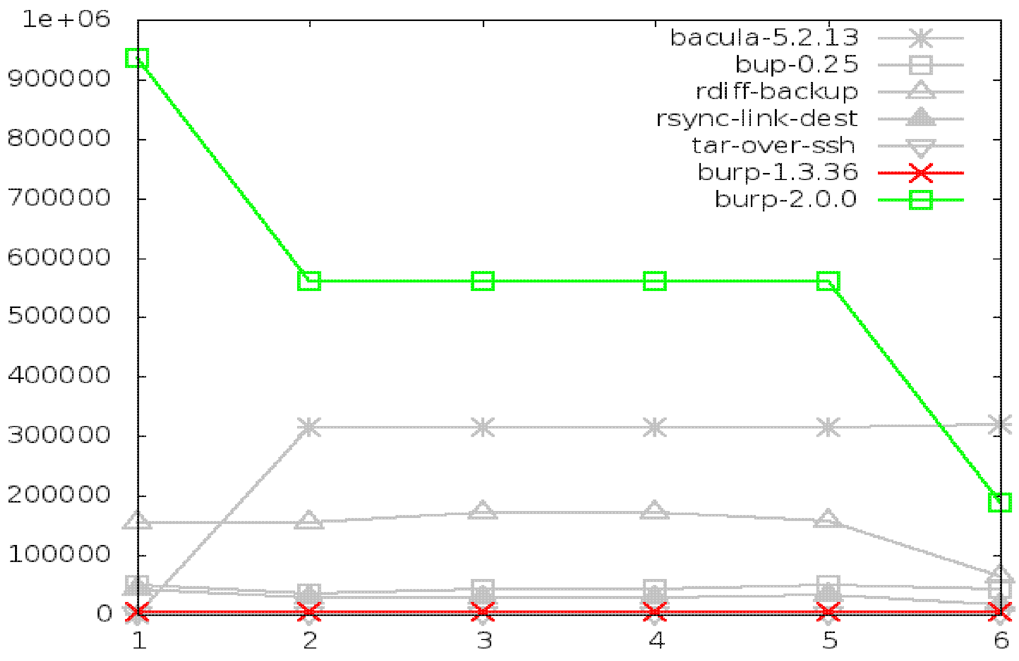
Tar is missing from the graphs because I was unable to capture the server ssh memory usage. Although as it was the only server process running, it would have used minimal memory anyway.



6.4.5. Maximum memory usage of client when backing up small files, in kbytes

As with the server memory usage, this is an area in which the new software performs badly. It has high memory utilisation because, for each file that the server asks for, the client has to load all the chunk data and their checksums into memory. It keeps them in memory until the server asks for them, or indicates that it doesn't need them. There is a built in limit to this already - the client will keep a maximum of 20000 blocks in memory at once. When it reaches this limit, it will not read in any more until it is able to remove at least one of those blocks from memory.
Perhaps of some concern is the peak in memory usage on the first backup of each sequence, when I was expecting the line on the graph to be flat. I am currently unable to explain this - it may indicate some kind of memory leak. I will address this again later in the report.

After its first backup, bacula does poorly. I believe that this may be because the server sends the client a list of all the file names that it has seen before in order to deduplicate on time stamps, and the client holds them in memory.
Again, all the other software perform within a similar range to each other.
Missing from the graph are amanda and backuppc, for which I was unable to get figures. However, since they both use tar over ssh in order to retrieve files from clients, it is fair to say that they would have performed similarly or even identically to the results for tar.

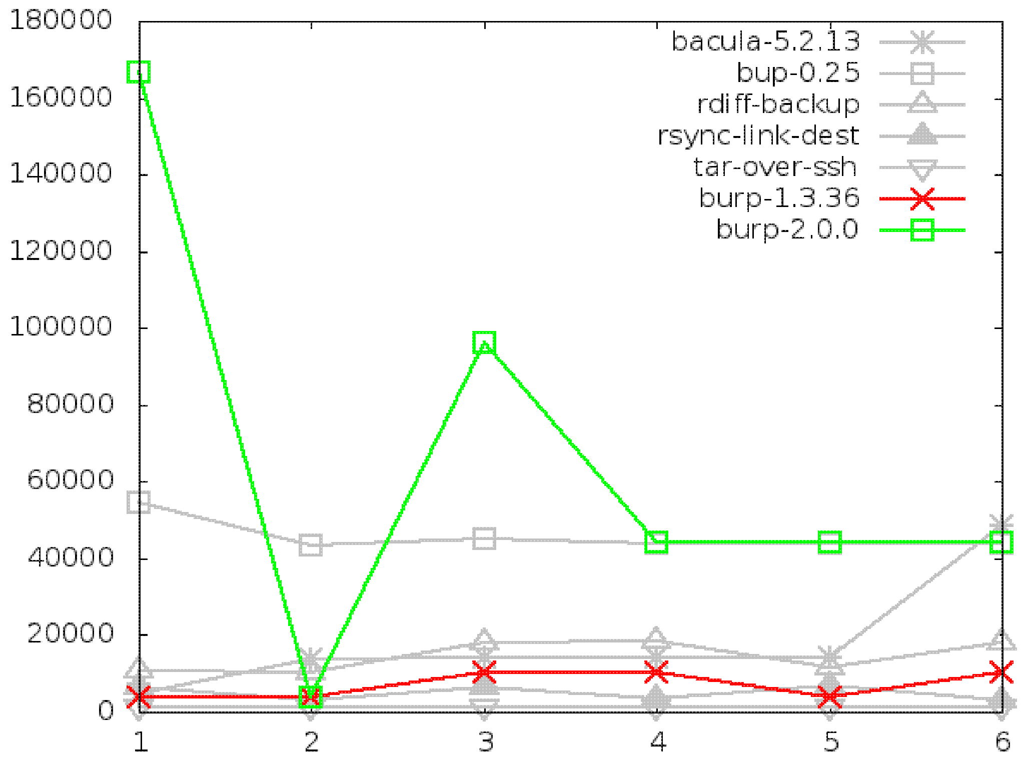


6.4.6. Maximum memory usage of client when backing up large file, in kbytes

Finally, the new software performed badly at steps 1 and 3 (altered file contents). At steps 4, 5 and 6, it performed similarly to the second worst software in this area, bup-0.25. It did beat bup-0.25 at step 2, where bup's lack of file level time stamp checking meant it had to read the whole file in again.

The rest of the software performed within a similar range to each other, with tar being the best in this field. Bacula showed a strange peak at the last step (file trunaction), taking its memory usage above that of bup-0.25 and burp-2.0.0. I don't have a sensible explanation for this.

Missing from the graph are amanda and backuppc, for which I was unable to get figures. However, since they both use tar over ssh in order to retrieve files from clients, it is fair to say that they would have performed similarly or even identically to the results for tar.



7. The second iteration

Having a set of test results, the second iteration could now be produced in order to attempt to address the weaknesses of the first iteration.

As already mentioned, the plan for the second iteration was to now implement sparse indexing in order to eleviate the memory usage of the server.

There were no plans at this point to improve the restore times, or the client memory usage, other than to continue to improve the efficiency of the code in general. These factors will be addressed again in the 'conclusion and evaluation' section of this report, where future plans will be explained.

The documents that were originally written for the second iteration were not quite right, and needed to be modified due to some slight misconceptions from earlier designs.

Again, due to space limitations in the main report, the designs for the second iteration are reproduced in Appendix E.

The implementation of the second iteration was completed at the end of November 2013.

8. Final test results

Upon implementing the design changes in the previous section, the new software was retested and graphs produced that demonstrate the changes in performance.

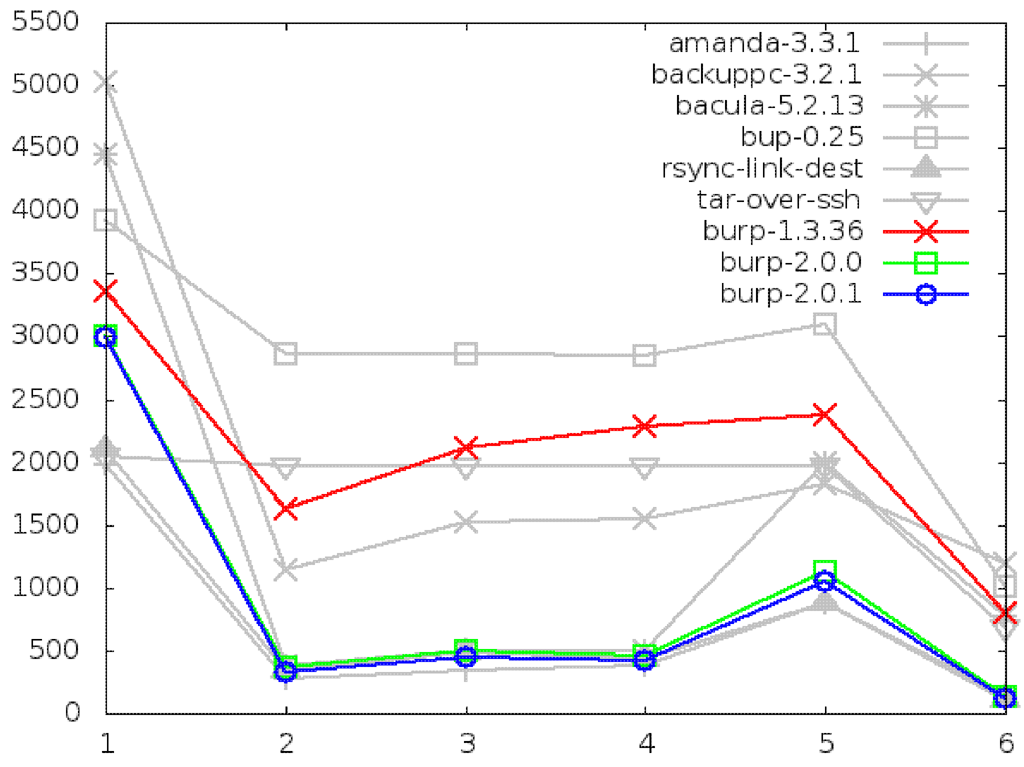
The graphs are the same as in the original testing, except that a new blue line for 'burp-2.0.1', the new software, has been added.

Firstly, I present the results for which the new software previously did well. This will demonstrate that, for those, that good level of performance remained unchanged after the second iteration. These graphs generally need no commentary because that has already been done adequately in the chapter on the initial results.

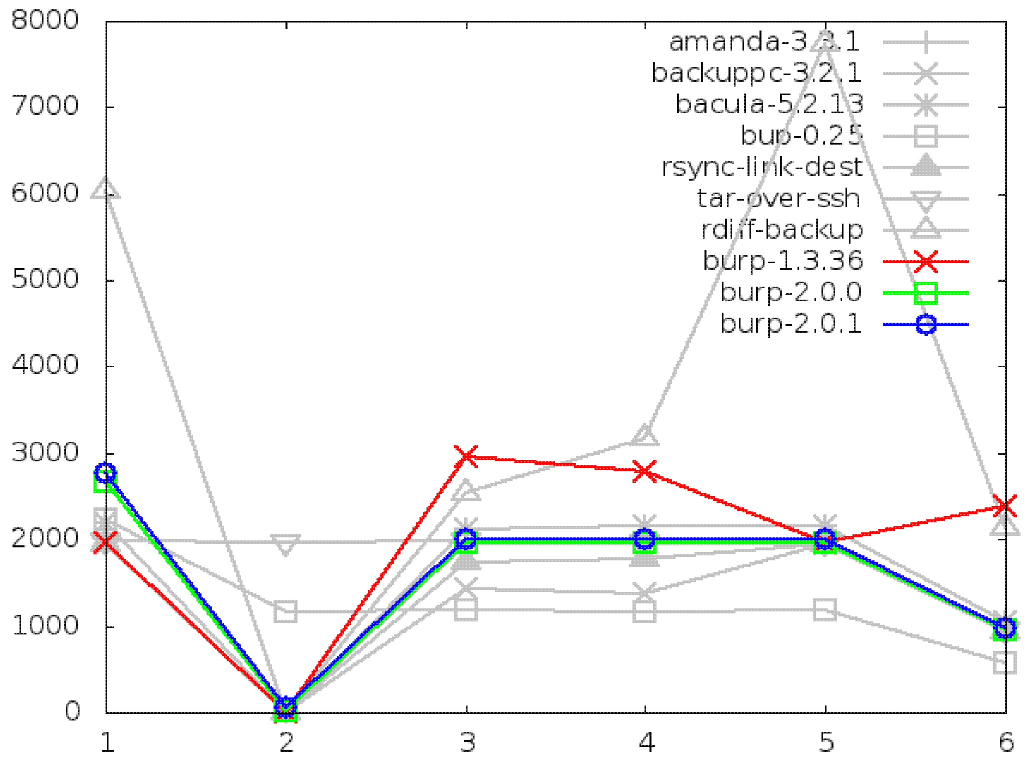
After that, I present the results for which the new software previously did poorly and look for whether the second iteration made any significant change, either good or bad.

8.1. Areas in which the second iteration continued to do well

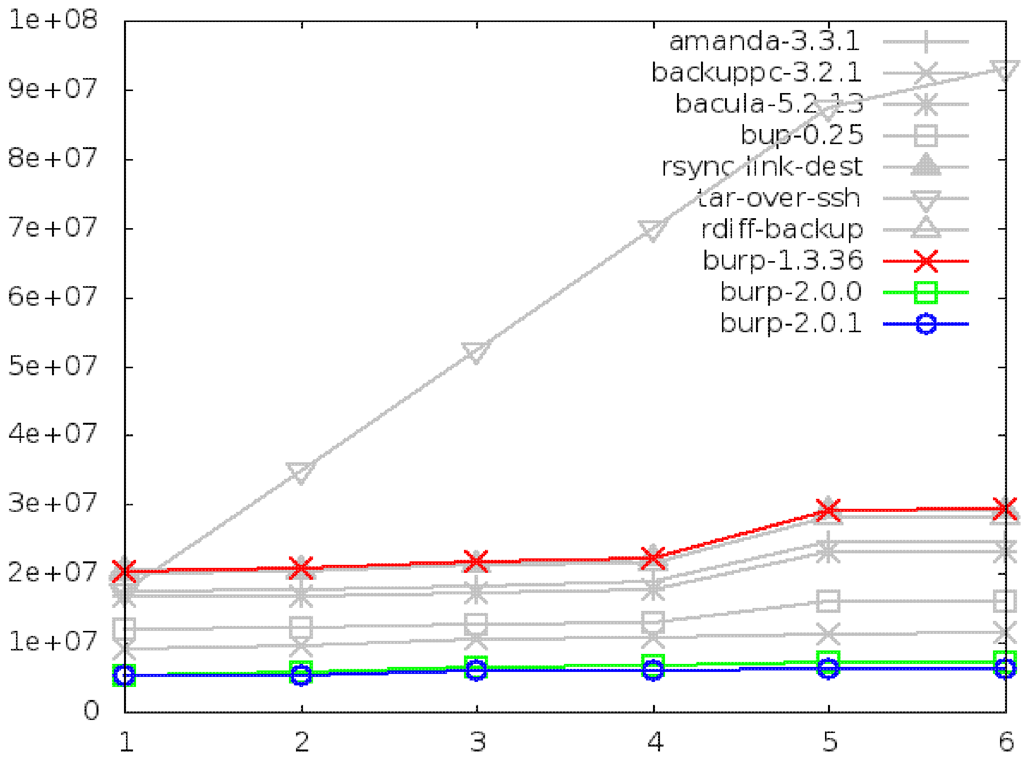
Time taken to back up small files, in seconds



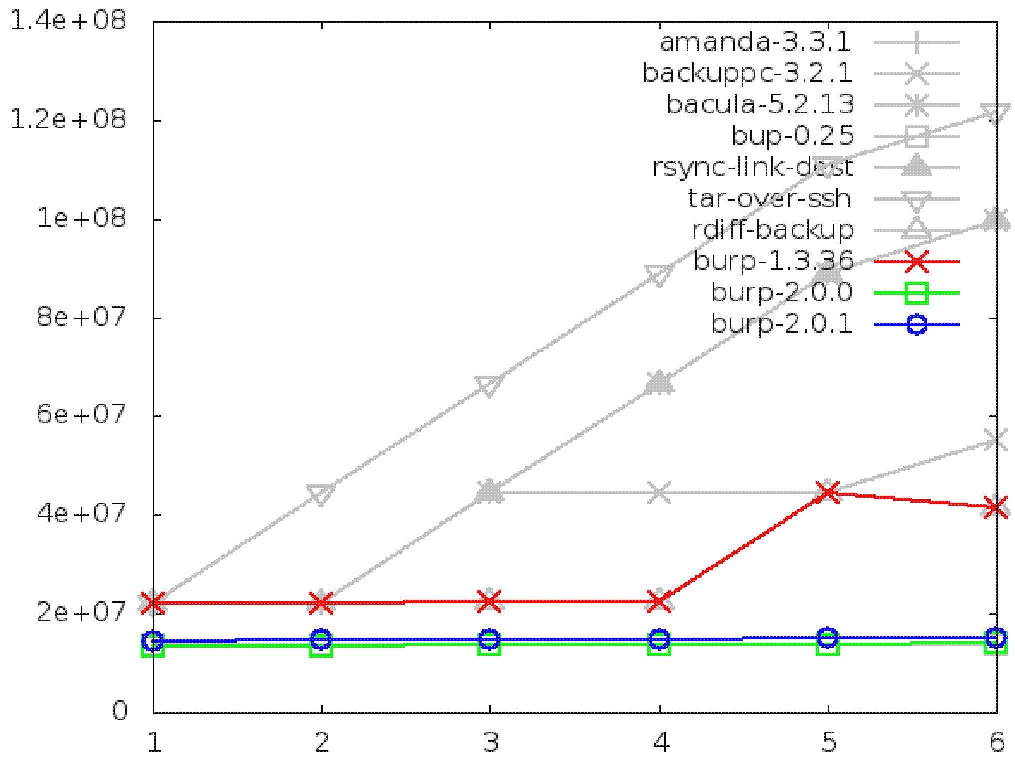
Time taken to back up large file, in seconds



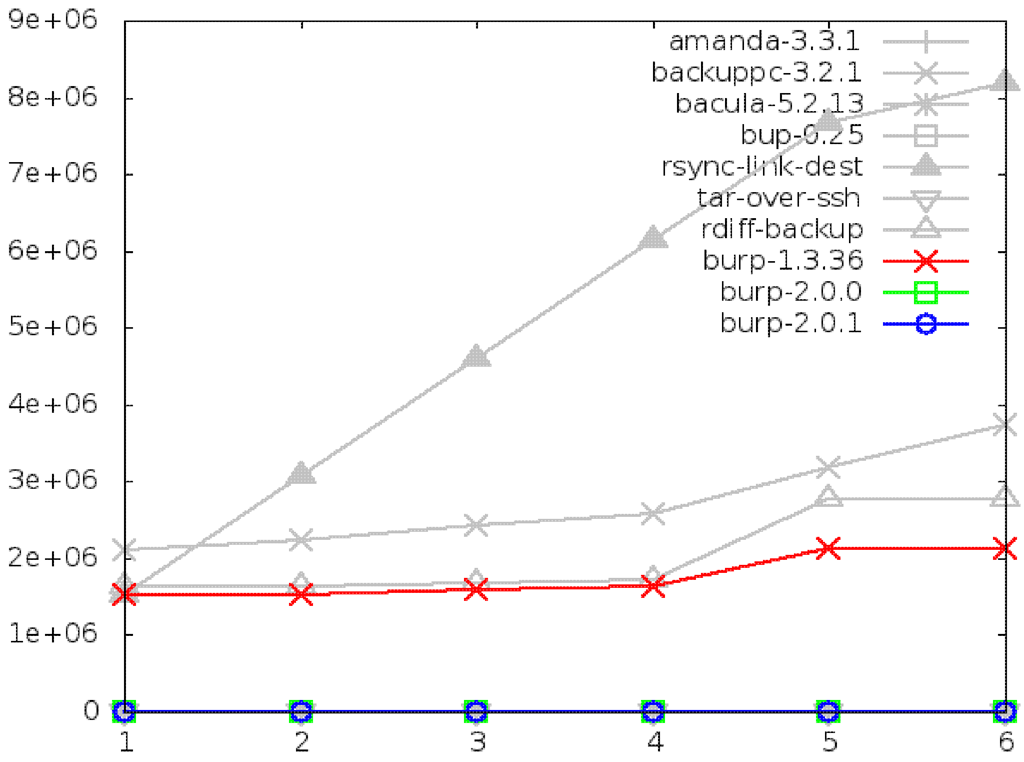
Disk space utilisation after backing up small files, in kbytes



Disk space utilisation after backing up large file, in kbytes



Number of file system entries after backing up small files

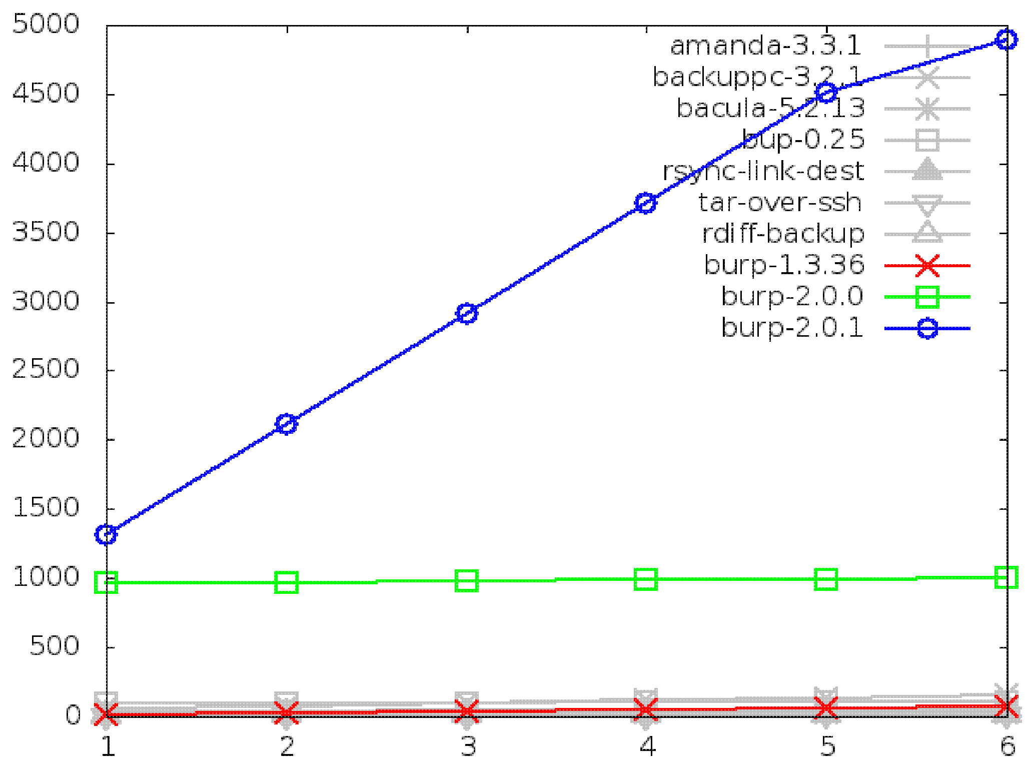


Number of file system entries after backing up large file

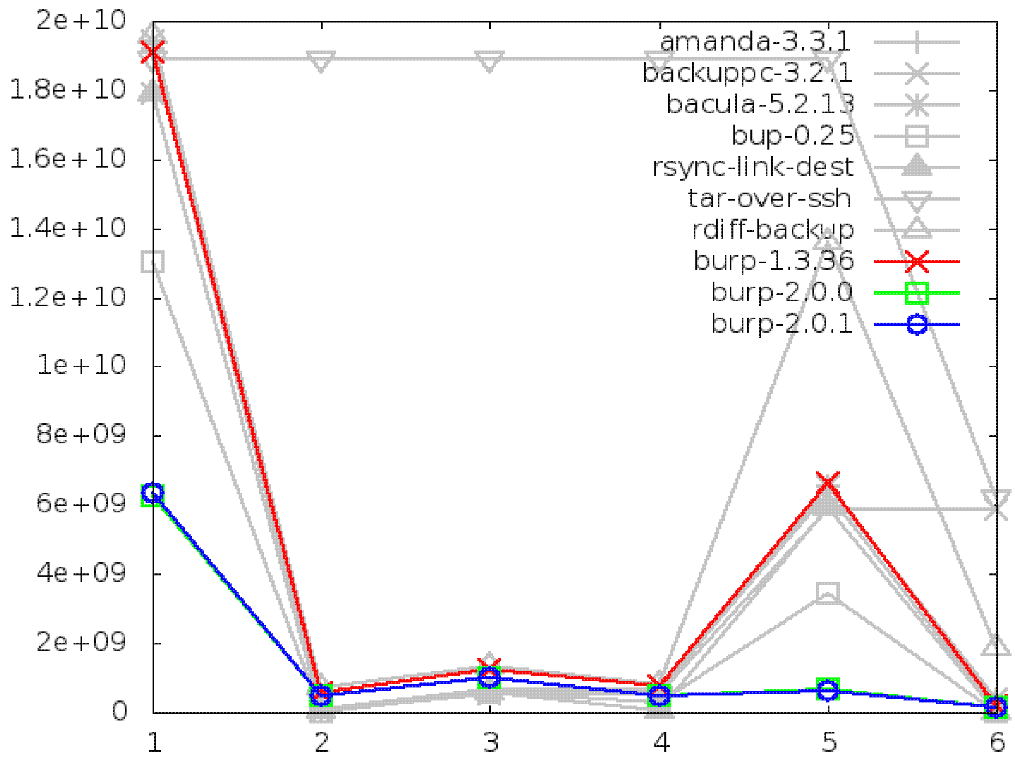
The second iteration clearly affected the results here, with each backup adding a little fewer than 1000 new file system nodes.

The reason for this is that the design of the second iteration split up the manifest files (containing information on how to reconstruct each backed up file out of blocks) into multiple smaller pieces. This was done in order to be able to have a sparse index for each of the smaller manifest pieces, rather than a full index on every block that has ever been seen.

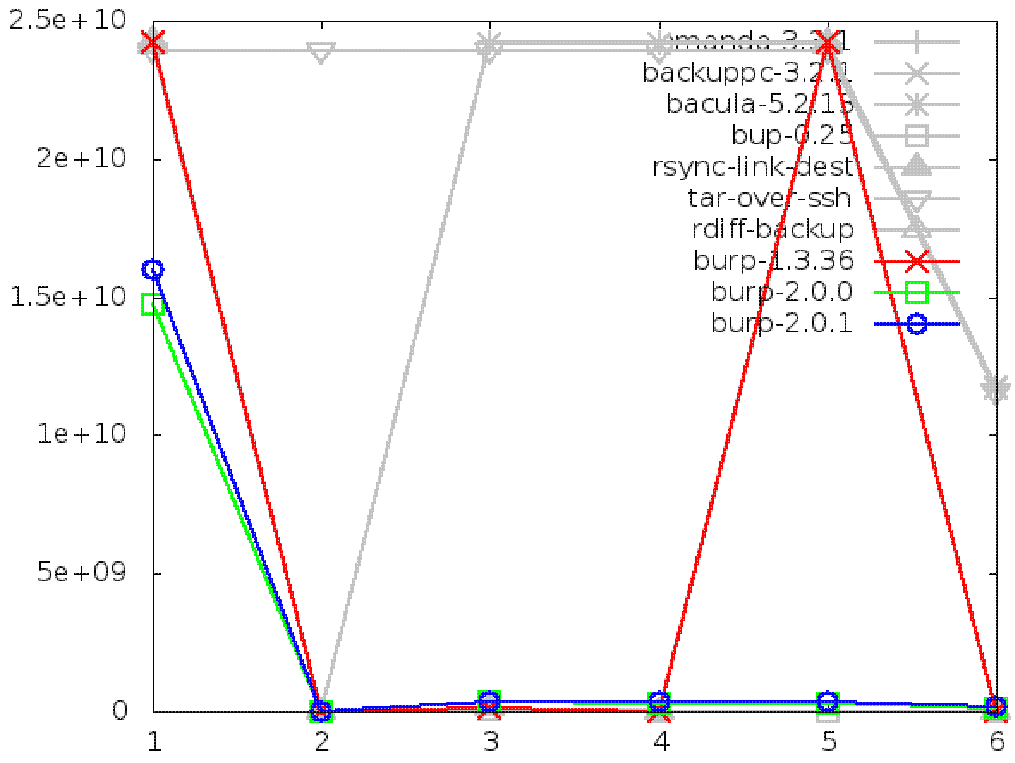
Although the change in the graph looks dramatic, there are still less than 5000 file system nodes in use by the last backup. As explained previously, this figure will not vary very much if identical data were spread across multiple files and then backed up - whereas software using a node for each file would increase linearly with the number of files (note the test results for file system nodes when backing up many small files). Consequently, I am happy to maintain that this is still a good result for the new software.



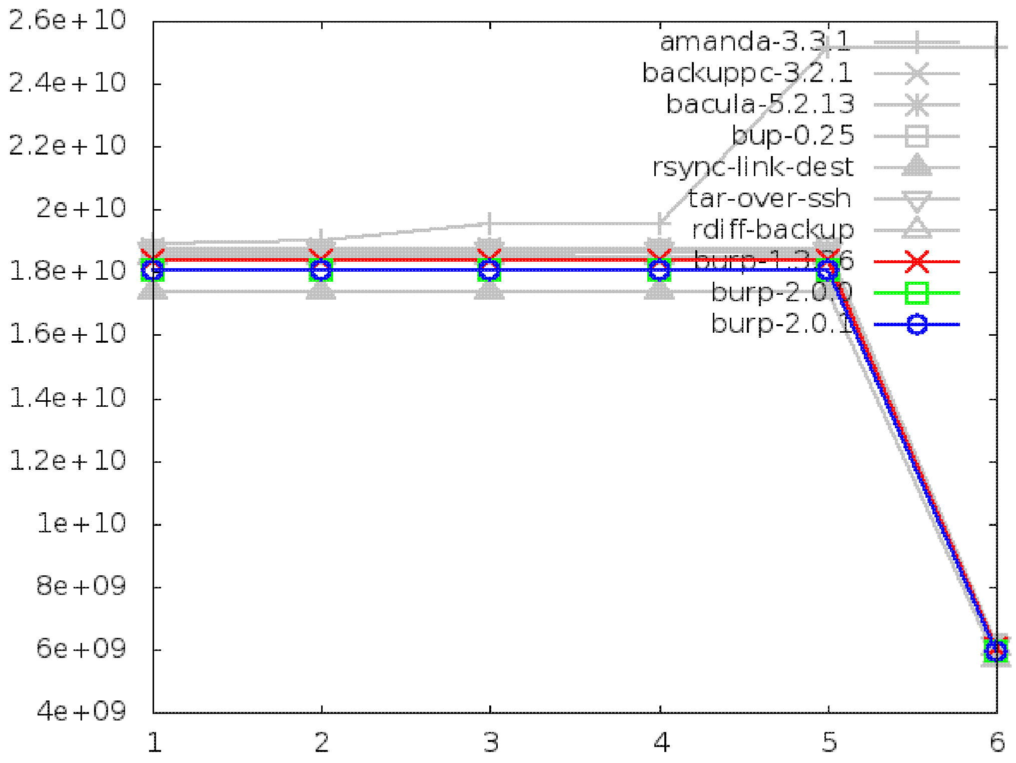
Network utilisation when backing up small files, in bytes



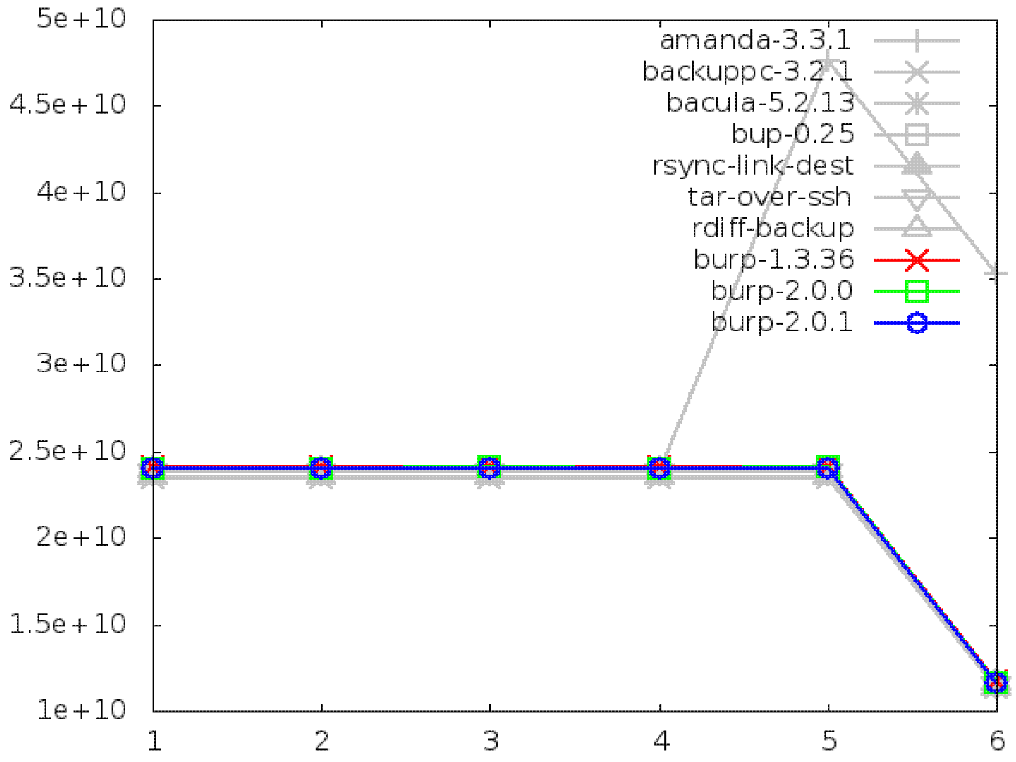
Network utilisation when backing up large file, in bytes



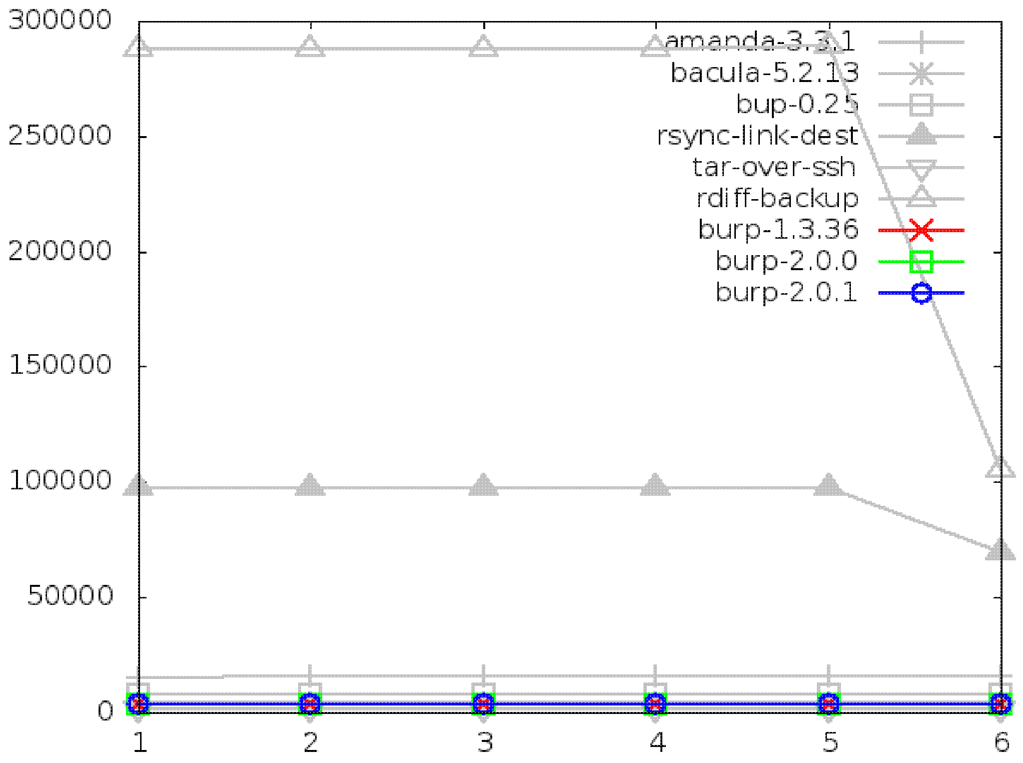
Network utilisation when restoring small files, in bytes



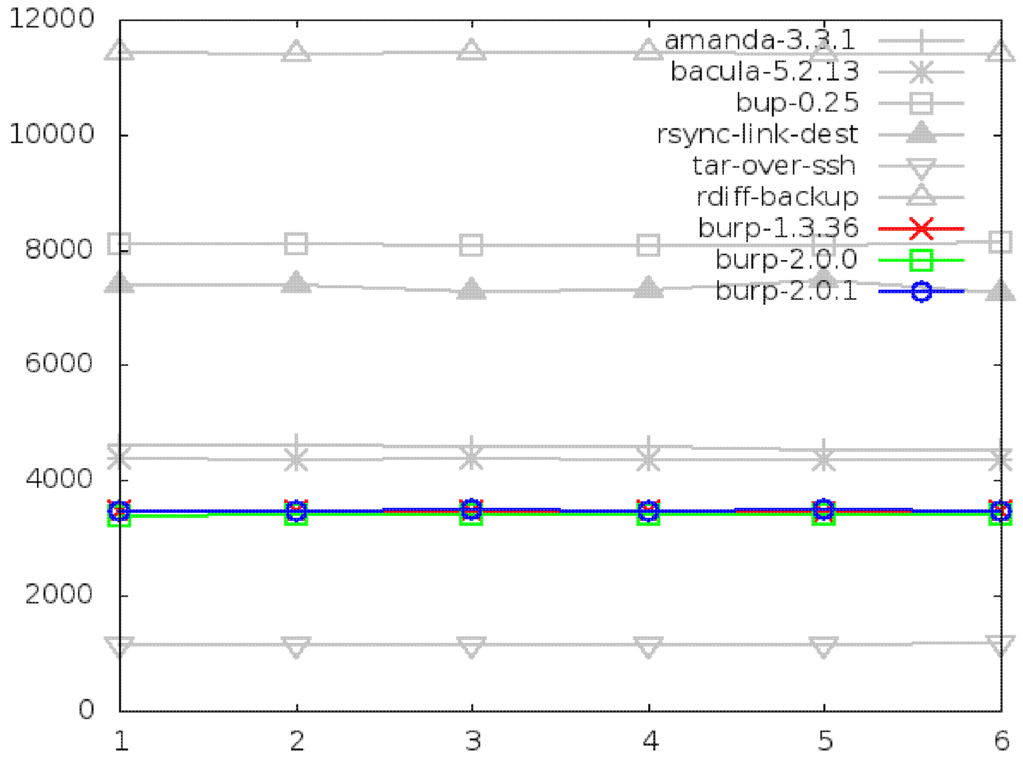
Network utilisation when restoring large file, in bytes



Maximum memory usage of client when restoring small files, in kbytes



Maximum memory usage of client when restoring large file, in kbytes



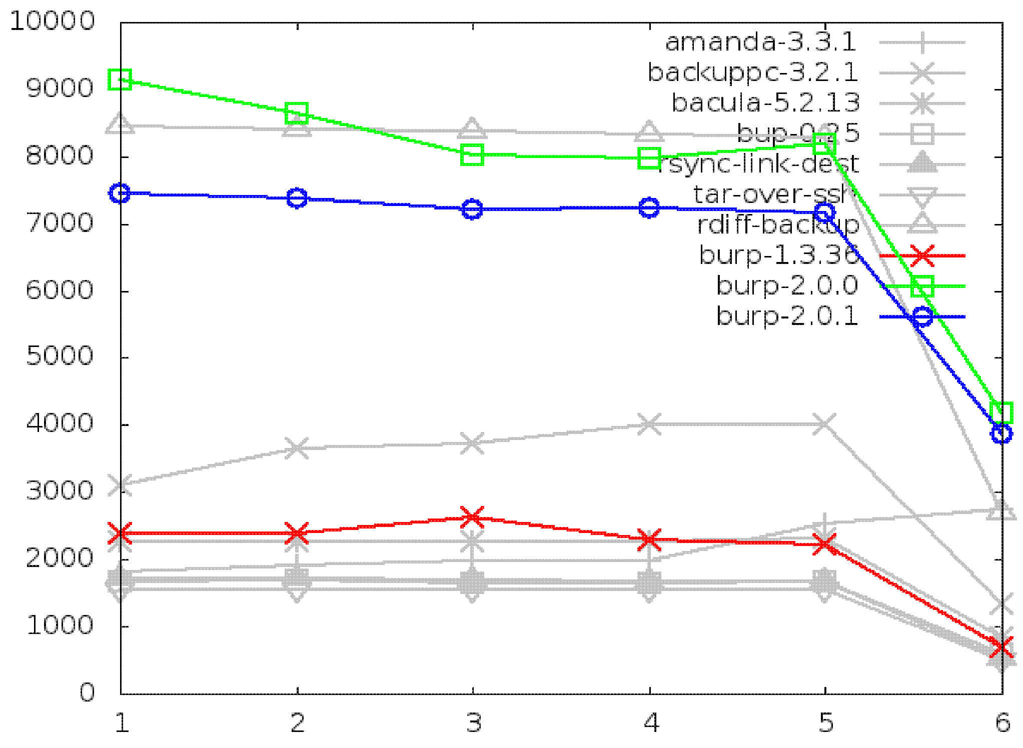
8.2. Areas in which the second iteration showed improvement

8.2.1. Time taken to restore small files, in seconds

The time taken to restore small files was improved slightly. I cannot give any particular reason for this, other than perhaps improving code efficiency between the two versions. Note that the benefit of sparse indexing only relates to backing up, not restores.
On reflection, there was some work focused on factoring out functions related to reading and writing manifest files into a class-like structure. This, at least, improved ease of maintenance.

Another reason for the improvement could possibly have been due to the changes in the structure of the data storage, getting rid of the signature files and holding them only in the manifests. I didn't expect this to improve the speed when I did it, because the first iteration didn't look at the signature files on restore anyway. However, maybe there was an efficiency improvement there that I overlooked.

Regardless, although the test showed improvement in this area, the new software is still not doing well when compared to the other solutions.



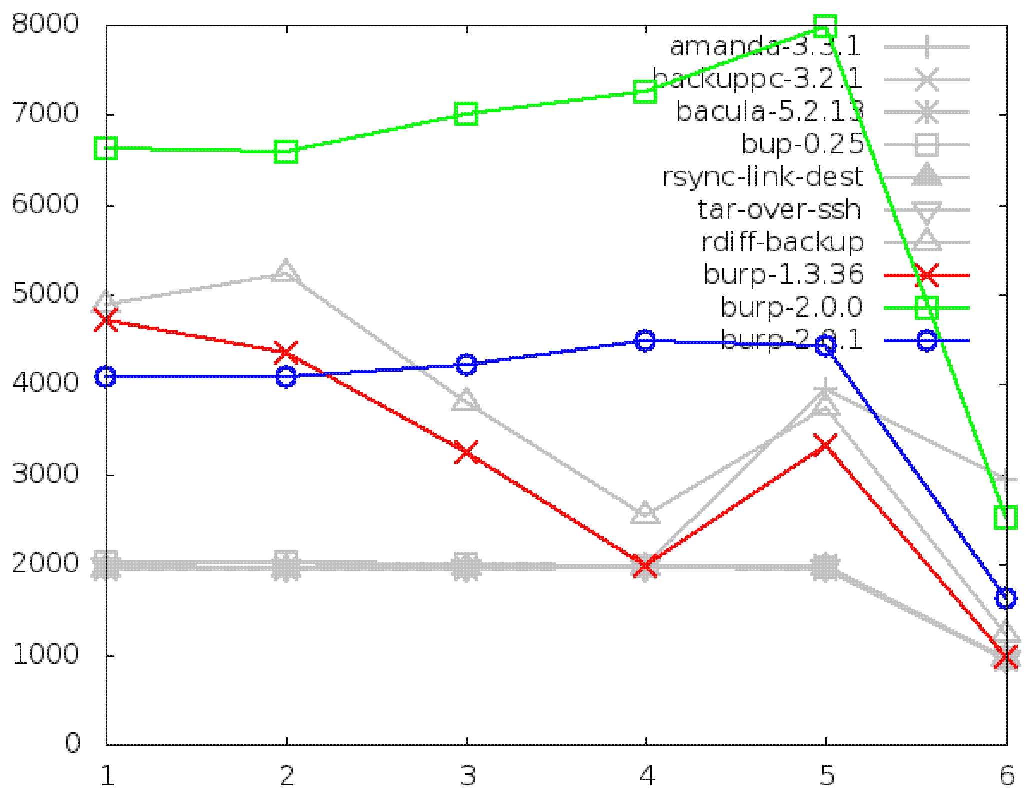
8.2.2. Time taken to restore large file, in seconds

The time take to restore the large file had a much more significant improvement than that for the small files.

My thoughts on the reasons for this obviously echo the speculation in the previous section to do with improved code efficiency and the changing storage structure.

However, although there was a large improvement, the new software is still not doing particularly well in this category.

The following chapter on further iterations proposes a possible method of improving the restore times, and describes my initial implementation of that method.



8.2.3. Maximum memory usage of server when restoring

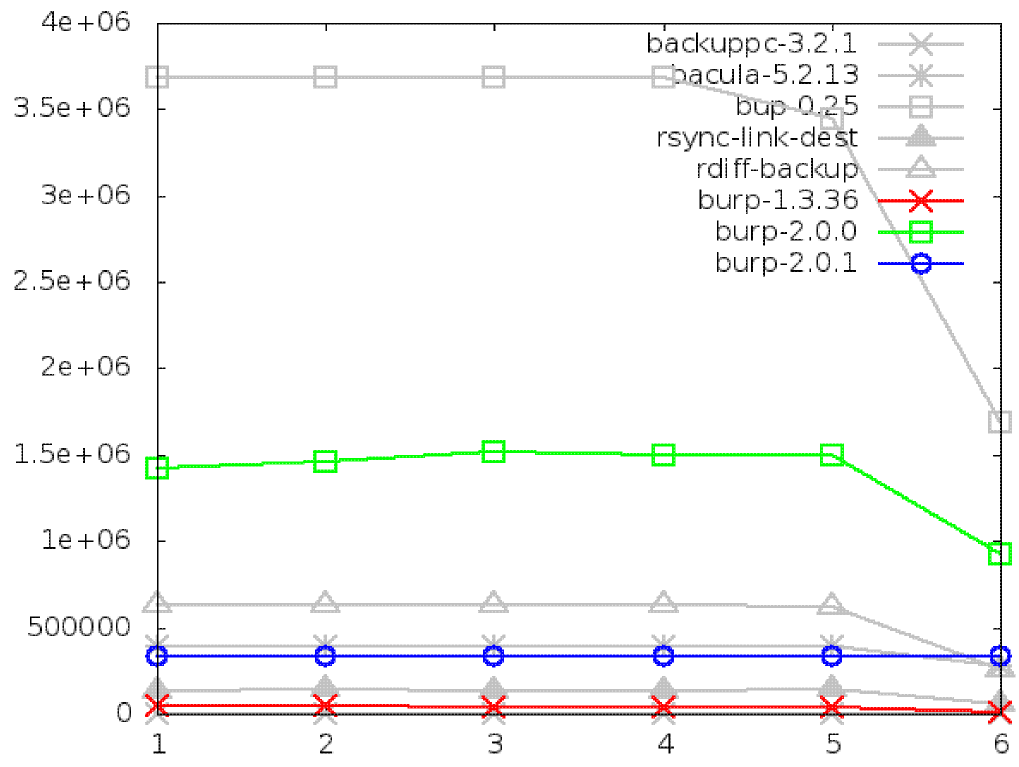
There is significant improvement in the second iteration for both large and small file restores in terms of server memory usage.

The small file memory was reduced to a third of the original result, and the large was halved. This is excellent.

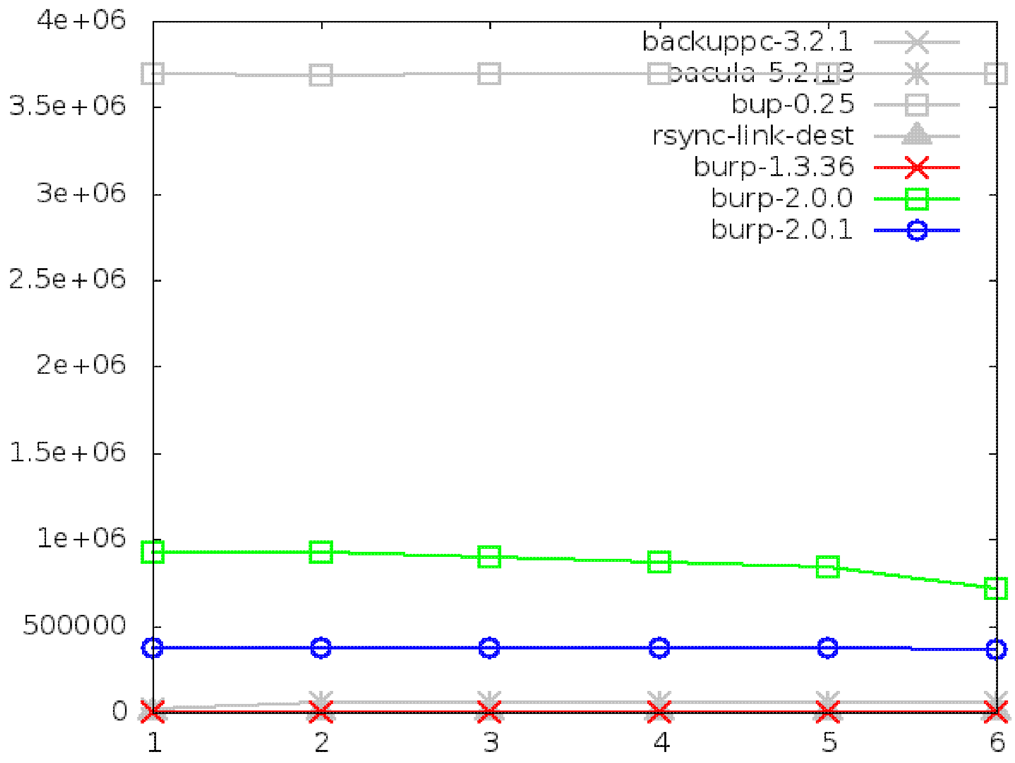
Both sets of figures are now comparable with each other. This seems to suggest that the original iteration had some sort of memory problem relating to client file names, because there really shouldn't be much difference between the memory usage for restoring lots of small files and restoring a large file - the server is just sending a stream of blocks being read from a similar number of manifest files.

The memory for small files has now dropped below bacula's figures. I don't think that this can be improved further without altering the method for restore, as explained in the following chapter about further iterations.

Maximum memory usage of server when restoring small files, in kbytes



Maximum memory usage of server when restoring large file, in kbytes



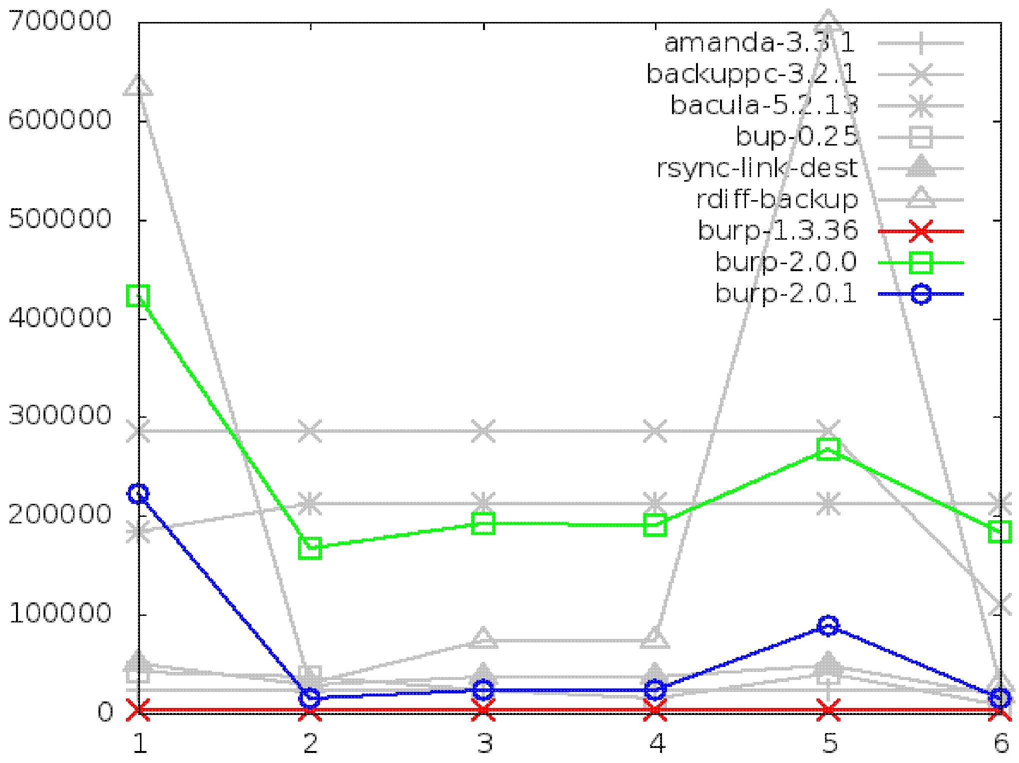
8.2.4. Maximum memory usage of server when backing up

This is the area in which I had hoped to see improvement due to the addition of the sparse indexing feature. Rather than loading a full index into memory, the server will load a much smaller sparse index, then only fully load the manifest pieces that have a high chance of matching the next incoming blocks.

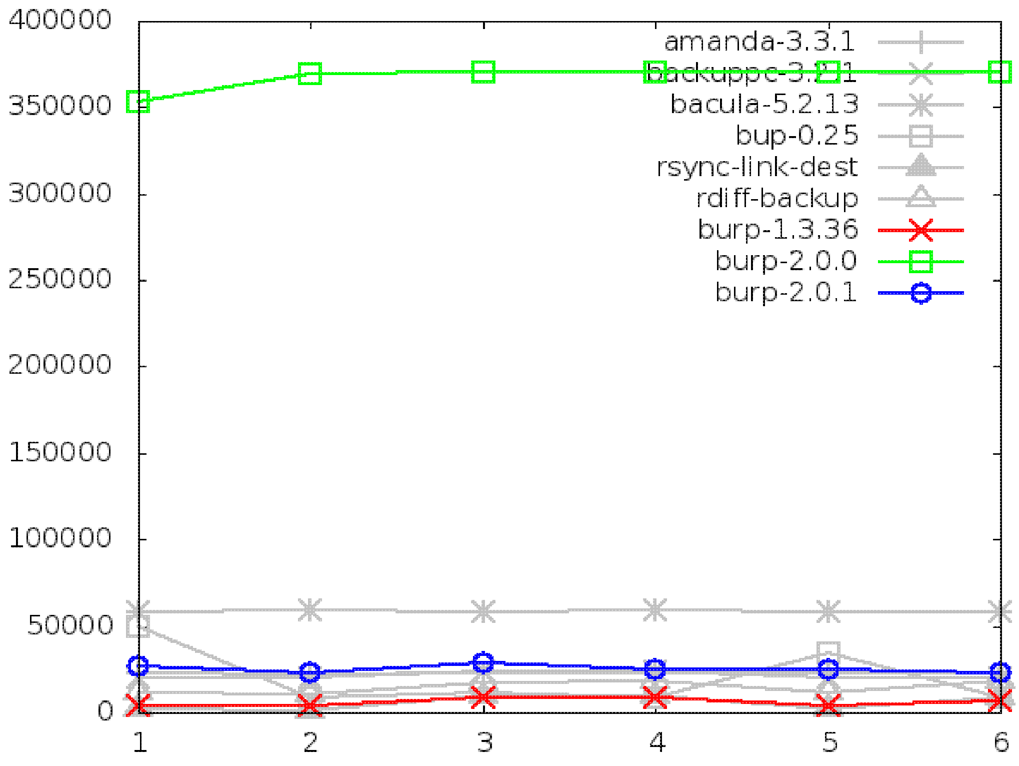
And indeed, in both small and large file restores, there is significant improvement. And this time, it brings the new software to a comparable level with the better group of its competitors. This is excellent.

The only problem that I have with this is that there are peaks in the small backup at steps one and five (renames). I believe that this is something to do with the way that the server receives the file system scan and deals with new file names. I have a potential solution to this that I will describe in a later chapter.

Maximum memory usage of server when backing up small files, in kbytes



Maximum memory usage of server when backing up large file, in kbytes

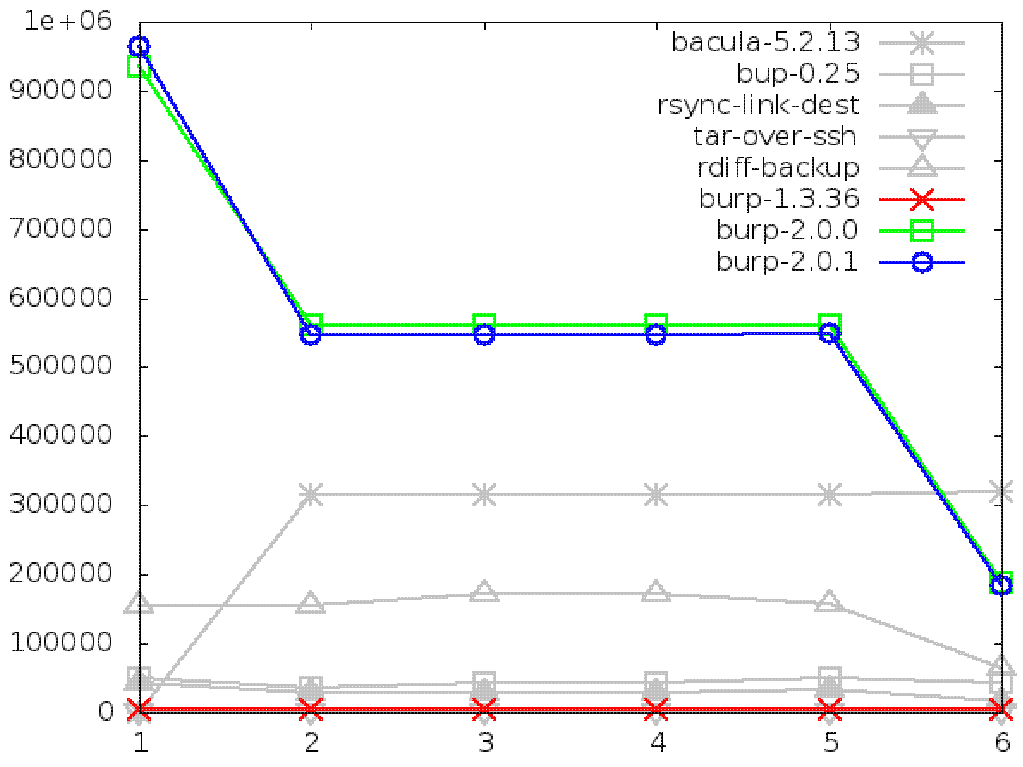


Important note: the graph above is showing that the memory usage of the server has been limited by the implementation of sparse indexing. In conjuction with the results showing that the disk space use changed negligibly between iterations and that the time taken actually decreased, this indicates that sparse indexing is an effective alternative to keeping a full index in memory, and hence also an effective solution to the disk deduplication bottleneck.

8.3. Areas in which the second iteration performed badly

8.3.1. Maximum memory usage of client when backing up small files, in kbytes

The results for this test did not change to any significant degree. This is to be expected, since no changes were specifically made in the second iteration to improve this.



8.3.2. Maximum memory usage of client when backing up large file, in kbytes

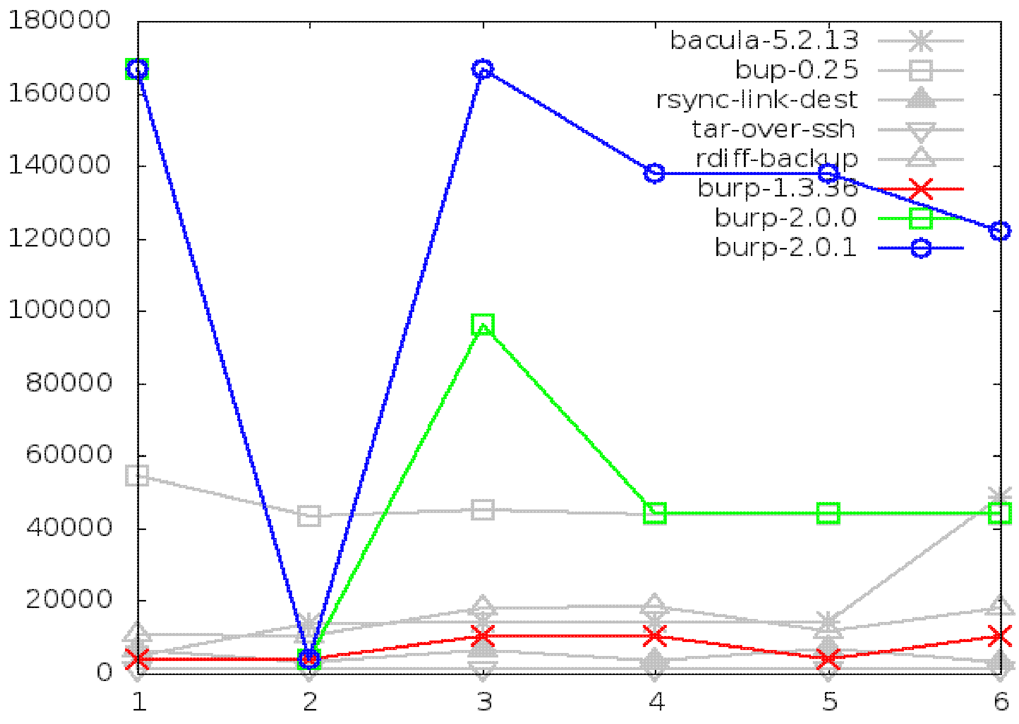
Finally, we come to the worse result of all. When backing up large files, the situation actually became worse after the second iteration. I was expecting the results to remain similar to the previous iteration.
However, on reflection, there is a reasonable explanation for this behaviour.

When backing up, the client reads data into memory, splitting it into blocks and checksumming them. It then needs to keep the blocks in memory until the server has told it either to send or to drop them.

The code limits the client to keeping 20000 blocks in memory at once. The blocks are an average of 8KB each, according to the parameters input into the rabin algorithm. 20000 x 8KB = 160000KB, which matches the peaks we see in the graph.
My theory is that, for the first iteration, the server was fast enough at processing incoming blocks that the client only ever loaded the maximum amount of blocks at some point during the first backup. And, after the second iteration, the server was slowed down slightly by having to load the right pieces of manifest based on the sparse index, giving the client a chance to fill up its memory with more blocks.

If this is really the case, a future possibility is to experiment with reducing the number of blocks that the client will keep in memory at any one time. Without changing anything else, this could be as low as 4096 blocks - the number that the server is currently counting as a segment to deduplicate. This should bring the maximum memory usage down to 4096 x 8KB = 32768KB, which is a better figure than bup-0.25's average in the large file tests, and would also be one of the better results in the small file tests.

However, reducing the buffer that low is likely to impact other results, such as backup speed, because the client will be waiting on the server more often.
Consequently, more experimentation is required in order to find a satisfactory balance.



9. Further iterations

Since completing and testing the second iteration, the software has been further improved in various ways through further development.

An attempt has been made to improve the restore time and restore network utilisation.
The idea was to make a list of all the data files containing the required chunks on the server side, then send them to the client, which would store them in a 'spooling' area. The server would then send the sequence of chunks to be read from the data files to reconstruct the original files. In this way, the reconstruction of the original files moves from the server side to the client side, and blocks are only transferred once, albeit with potentially unneeded ones.

It is possible that, in some situations, this is less efficient. For example, when a small file containing a single chunk is required by the user. So, I implemented a simple mechanism that first estimates the data that needs to be sent by both restore methods. If the 'spooling' method needs to send less than some percentage (I arbitrarily chose 90%) of the total data of the 'streaming' method, the 'spooling' method is chosen.

However, the burden is then on the client to then reconstruct the original files from the data files. And, assuming the client has one disk, it will probably try to doing many reads and writes simultaneously. On my test setup, this meant that the client disk write speed was slower and hence the restores became slower. Regardless, the network utilisation was massively improved, benefiting from the deduplication of the chunks being transferred. It was using around a third of the network bandwidth that the other solutions used - around 12GB less.

9.1. Additional work required prior to release

Before the software is properly released to the public, I have a list of items that I would like to address. These are reproduced in Appendix H. There are some important items to do with locking write access to data files, for example, so that multiple clients backing up simultaneously do not interfere with each other.

A design element that I believe I got wrong in the completed iterations was to do the client file system scan in parallel with the backing up of data. Firstly, this makes the backup code more complicated. Secondly, it means that the original burp's progress counter system no longer works. When the file system scan is done as a separate stage at the start, you know how much data is left to back up and can provide time estimates. Thirdly, both client and server have to keep the scan data in memory, which affects the memory statistics. If the software reverted to the original burp's file system scan, the memory usage for the scan would be minimal.

Although I am very pleased with the new software, I am aware that there are existing burp users who may prefer the original. For example, those that run the server on a 'plug computer' containing minimal resources such as memory, or those that like the ability to copy files direct from storage (as opposed to using the client software).
Since there is a lot of shared code between the old and new versions, I plan to implement a mode where you can choose to run it and get the old behaviour. This would also help existing users to transition. You may have clients using the old behaviour, and slowly add new style clients, for example.
Implementing this requires merging the unique parts of the original burp into the new code.

The last one of these items that I would like to note is the ability to delete data files and backup directories that are no longer required. This is often problematic for software that does deduplication, because it is hard to know if all the saved blocks in a data file are no longer referenced. For example, the disk space that bup-0.25 uses only ever grows.

My idea for this is to maintain summary files in a similar way to that in which the sparse indexes are maintained.

Whenever a new backup is done for a client, a summary file would be generated that lists the data files that the backup used. Then another summary file is updated that encompasses all of the client's individual backup summaries. One of these second summaries is kept for each client.
When a client backup directory is deleted according to the retention periods set, a new summary file for that client is generated from the individual backup summaries.
On comparing the new and old summary file, if it is found that a data file is no longer referenced, the other top level client summaries are checked. If the reference is not found, then the data file can be deleted and disk space is recovered.
Since the sparse index maintenance code is already doing a very similar job, these summary files can be produced at the same time with the same code. Therefore, I believe the overhead for this will be minimal.

I estimate that the work given in Appendix H to take around two months to complete.

10. Conclusion

The main aim of the project was to improve upon the existing 'burp' software to produce the basis of a new open source cross platform network backup software that is demonstrably superior to others in the same field.

As explained in a previous chapter, research on the software available showed that the original burp (and hence the new software), was one of the strongest in terms of supported features.

The new software was then shown under test to be either superior or comparable to its 'competition' in terms of backup speed, storage space, storage nodes, and network utilisation.

It only performed satisfactorily in terms of restore speed, although there are ideas for improving this in the future.

Upon the implementation of sparse indexing, server memory usage was either satisfactory or comparable to the best of the other solutions.

Client memory usage is a potential issue, with the new software performing worse than other software. However, experiments to reduce the size of the client block buffer will be performed for future iterations, with the expectation that the client memory usage can be brought down to a level consistent with other software. This expectation was shown using simple calculations based on average block size and the existing test results.

When taking into consideration both the test results and the feature comparison, I have no doubt that the new software will be a better choice than the other options currently available.

It should be noted that the new software is not quite ready for release at the time of writing. Appendix H reproduces a list of the remaining work to be done over the next few months before its first official release.

10.1. Answering the research questions

The first research question asked whether incorporating inline deduplication techniques can improve open source backup offerings and produce an empirically superior product.
This paper has demonstrated that it can.

The second research question asked whether sparse indexing is an effective solution to the disk deduplication bottleneck.
It was noted that another software solution, backshift, does lookups on a full index located on disk. And it took longer than two days to complete its first backup.
The design of the new software never actually referenced the full index on disk while chunks of data were incoming. The first iteration loaded the full index into memory at the start, then did the lookups in RAM. This was shown in the high results for server memory usage during backing up.
Upon implementing sparse indexing, it was shown that server memory usage of the new software when backing up was effectively limited without notably reducing the effectiveness of the deduplication.
Therefore, I am confident in claiming that sparse indexing is an effective solution to the disk deduplication bottleneck.

11. Evaluation of the project as a whole

In order to produce the new software, I initially produced a list of objectives. During the span of the time allocated to this project, I was able to achieve all of the objectives, as follows.

Core: Complete a search and review of popular existing open source network backup solutions, explaining the areas in which current backup offerings are flawed.
This was done early on in the project, and the results can be read in 'Appendix F - Open source competition'.

Core: Complete a literature search and review of relevant algorithms and techniques that will be needed to implement a new software engine.
This was also done early on in the project, and the information learnt was used in the design of the new software engine.

Core: Design and develop the new software.
The design was fairly quick, but the actual development took a significant amount of time, including proof of concept programs and iterations of the final software. The iteration used for the test results was completed with around a month of time remaining before the project deadline.

Advanced: By conducting suitable tests and analysis, prove that the new software is superior to other offerings.
The new software was tested and shown to be either comparable or superior to other solutions in most areas, with the exception of client memory usage. This will be addressed before the initial release of the new software.

Advanced: Demonstrate that sparse indexing is an effective solution to the disk deduplication bottleneck.
Another deduplicating solution (backshift) taking longer than two days to complete one backup demonstrated the bottleneck problem.
Analysis of the new software's results for server memory usage, disk space and the time taken for backing up showed that sparse indexing was an effective solution.

Core: Complete the final project report.
The final project report was completed before the deadline.

Overall, I was pleased with the way that the project went. I put a lot of my spare time into this, and with a little more work the result has the potential to be useful to many people.

The two hardest parts of the project were implementing the multiple streams of asynchronous I/O, and the testing of the various software.

The former would have been made slightly simpler if I had left the file system scan part as a separate phase before backing up the data, which would then only require two streams in each direction instead of three.

Testing the various backup solutions was hard because of the amount of time involved with it. I would very often run a test sequence over a few days, and find at the end that I would have to run it again for some reason. For example, network compression may have been turned on and so the results could not be compared fairly against other solutions that were not doing network compression.

Fortunately, as mentioned in the Intermediate Project Report, I began the testing ahead of schedule and was therefore able to resolve these issues and complete the testing on time.

The decision to continue to code in C was the correct one, as I think I would not have finished the iterations in time had I tried to move to C++. However, the object-orientated style of coding that I am moving towards seems effective enough.

In the end, I find it surprising that the ideas utilised during this project have not already been combined in open source software. Perhaps a possible explanation for this is that the hard work that is involved with the development of this kind of software causes the authors to wish to sell the results. I do not intend to do that, but I can see a potential route to remuneration via providing support for the software instead. That is, if it really *does* turn out to be useful to other people.
It would be fantastic to work on this software full time.

Bibliography

This is a list of references that I have consulted in relation to work in the area.

Barratt, C. (2001). *BackupPC*. Available at: http://backuppc.sourceforge.net/. [Accessed 28th September 2013]

Beck, K; et al. (2001). *Manifesto for Agile Software Development*. Available at: http://agilemanifesto.org/. [Accessed 12th February 2013]

Broder, A. (1993). *Some applications of Rabin's fingerprinting method*. Sequences II. Springer New York. 143-152.

Denisov, V. (2012). *Performance issue resolution for large files*. Available at: http://sourceforge.net/p/librsync/patches/11/. [Accessed 26th May 2013]

Eastlake, D. (2001). *US Secure Hash Algorithm (SHA1)*. Network Working Group Request for Comments: 3174.

Escoto, B; et al. (2001). *rdiff-backup*. Available at: http://rdiff-backup.nongnu.org/. [Accessed 28th September 2013]

Eshghi, K. (2005). *A Framework for Analyzing and Improving Content-Based Chunking Algorithms*. HP Laboratories Palo Alto.

Germain, S. (2012) *Revolutions dans le monde de la sauvegarde de donnees*. Available at: http://linuxfr.org/news/r-evolutions-dans-le-monde-de-la-sauvegarde-de-donnees/. [Accessed 26th May 2013]

GNU (1999) *GNU Tar* Available at: http://www.gnu.org/software/tar/ [Accessed 26th May 2013]

Keeling, G. (2011). *Burp - BackUp and Restore Program*. Available at: http://burp.grke.net/. [Accessed 26th May 2013]

Langford, J. (2001). *Multiround Rsync*.

Lillibridge, M; et al. (2009). *Sparse Indexing: Large Scale, Inline Deduplication Using*

Sampling and Locality. Proceedings of the 7th conference on File and storage technologies.

Meyer, D. T; et al. (2012). *A Study of Practical Deduplication*. ACM Transactions on Storage (TOS), 7(4), 14.

Pennarun, A. (2010). *bup*. Available at: https://github.com/apenwarr/bup/. [Accessed 28th September 2013]

Pool, M; et al. (2004). *librsync*. Available at: http://librsync.sourceforge.net/. [Accessed 26th May 2013]

Raiber, M. (2011). *urbackup*. Available at: http://urbackup.sourceforge.net/. [Accessed 28th September 2013]

Rabin, M. (1981). *Fingerprinting by Random Polynomials*. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University.

Rivest, R. (1990). *The MD4 Message-Digest Algorithm*. Network Working Group Request for Comments: 1186.

Rivest, R. (1992). *The MD5 Message-Digest Algorithm*. Network Working Group Request for Comments: 1321.

Rosenquist, N; et al. (2003). *rsnapshot*. Available at: http://www.rsnapshot.org/. [Accessed 14th September 2013]

Royce, W. (1970) *Managing the Development of Large Software Systems*. Proceedings of IEEE WESCON 26 (August): 1-9.

rsync-3.0.0-NEWS (2008). Available at: http://rsync.samba.org/ftp/rsync/src/rsync-3.0.0-NEWS [Accessed 26th May 2013]

Schwaber, K. (2004). *Agile Project Management with Scrum*. Microsoft Press.

Shore, J. & Warden S. (2007) *The Art of Agile Development* Cambridge: O'Reilly.

Sibbald, K. (2000). *Bacula*. Available at: http://bacula.org/. [Accessed 26th May 2013] da Silva, J; et al. (1991). *Amanda*. Available at: http://amanda.org/. [Accessed 28th September 2013]

Stromberg, D. (2012). *backshift*. Available at: http://stromberg.dnsalias.org/~strombrg/backshift/. [Accessed 28th September 2013]

Tridgell, A. & Mackerras, P. (1996). *The rsync algorithm*. The Australian National University.

Tridgell, A. (1999). *Efficient Algorithms for Sorting and Synchronization*. Australian National University.

Wells, D. (2009) *Extreme Programming: A Gentle Introduction*. Available at: http://www.extremeprogramming.org/. [Accessed 10th February 2013]

Whitten, J. & Bentley L., (2007). *Systems Analysis and Design Methods* 7th edition. McGraw-Hill Irwin.

Wirzenius, L. (2007). *obnam*. Available at: http://liw.fi/obnam/. [Accessed 28th September 2013]

Zhu B; et al. (2008). *Avoiding the Disk Bottleneck in the Data Domain Deduplication File System*. Proceedings of the 6th USENIX Conference on File and Storage Technologies. Vol. 18.

Appendix A - How the original 'burp' works

Burp has a client/server design. The machines to be backed up are the clients, and the machine that stores the backups is the server.

The clients are usually set up to regularly contact the server at short intervals of, for example, 20 minutes.
The server decides whether or not it is time for the client to be backed up, based on configuration options that give it timebands and minimum time elapsed since the last successful backup.

This has several advantages over a system like bacula, where a central server contacts the clients on a schedule. For example:

- Firewalls tend to allow outgoing connections far more frequently than incoming connections. Having the clients connect to the server rather than the other way round means that only access to the single server port is required. This is a huge administrative advantage.
- Server logic that deals with retrying after failed backups becomes massively simplified. The client will connect again in a few minutes time, and the decision is simply over whether that client has a recent enough successful backup, and whether it is still in timeband. Conversely, having the server initiate connections on a schedule means that it also needs complicated logic for scheduling retries. Indeed, I found this impossible to configure successfully in bacula.

Network connections use the OpenSSL library to secure communications. There is an automatic certificate signing and exchange mechanism to provide confidence and certainty that subsequent communications are with the same peer.

When a client initiates a connection, the server forks a child process to deal with it. A nice feature is that the configuration for the client on the server is read in at this point, meaning that most configuration changes do not need the administrator to reload or restart the server.

A.1. Backup sequence

There are four phases to the backup sequence.

In the first phase, the client scans its file system (based upon user configuration to include or exclude files) and sends the results to the server. The scan includes path names and meta information, such as file system entry type (regular file, directory, soft link, etc), permissions, time stamps, and so on. Retrieving this information is fast because no files are actually opened at this point - it is just file system data.
The server records all this information in a file called the 'manifest'.

In the second phase, the server reads through the manifest from the immediately previous backup and the new manifest. Because the paths in the manifests are always in alphabetical order, only a single pass is required.

If it finds files that do not exist in the previous manifest, it asks the client to transfer the whole file.
If it finds files in both manifests where the modification times are the same, it makes a note that that file has not changed.
If it finds files in both manifests where the modification times are different, it initiates a transfer using librsync. It reads the previous version of the file in order to generate 'signatures'. It sends the signatures to the client, and the client uses them to transfer only blocks that have changed. This results in a 'delta' file appearing on the server. There is more on librsync and its limitations, or problems, below.

When the second phase is complete, the client has no more work to do, and disconnects from the server.
However, the server child process still has more to do.

In the third phase, the server combines the notes that it made of what was and was not transferred, and builds a final manifest for the backup. This is a very quick phase, usually taking only a few seconds.

In the fourth phase, the server goes through the final manifest and the new files and deltas on disk, along with the unchanged files from the previous backup, and finalises the new backup. It needs to apply the deltas to the previous files to generate new files, and it makes hard linked entries to represent the unchanged files.

Optionally, it can then generate reverse deltas to take you from the latest versions of files to the previous versions, and deletes the original file from the previous backup. This can save significant disk space, but could add significant time to this phase, as well as adding to the time needed for restore.

A.2. Storage directories

To do a restore from any backup in a sequence, the server will only ever need to access that backup, and possibly subsequent backups. It will never need to access previous backups in the sequence.
This means that it is always 'safe' to delete the oldest backup in the sequence. This is not the case for backup solutions where you need access to older backups to recreate newer backups (bacula, for example). For such solutions, a satisfactory retention policy is highly problematic.

So, the resultant burp backup directories contain a manifest file, and a heap of (possibly very many) files and (optionally) deltas.

Since the files are stored in a directory structure that reflects the original client, a convenient way to restore a file is just to copy it directly out of the storage directory, rather than using the burp client.

A client's sequence of backups is always self-contained. That is, no information external to them is needed for them to be complete. Additionally, if you are not generating reverse deltas, each individual backup is self-contained. This is an advantage over backup solutions like bacula that use a separate 'catalogue' to store paths, meta data and storage locations. Such solutions add a significant administrative headache and many opportunities to end up with useless backups. You must have provision to back up your catalogue, for example.

A disadvantage to the burp storage structure is that there may be millions of files in a single backup, each needing its own file system entry. This will slow down the fourth phase of a backup significantly. Especially if you have set up your backup server to use Network Attached Storage (NAS), where each 'stat' system call goes over the network.

A.3. Deduplication

Burp's deduplication mechanism operates 'after the fact', as it were. It has a separate program that can be run on a cron job, which scans the storage file system looking for duplicate files. When it finds one, it replaces it with a hard link.
Obviously, this takes significant time when there are millions of files to scan. It also does not deduplicate chunks within files. The deduplication program is not run on a default install of burp.

It would be far preferable to have an 'inline' deduplication mechanism, where the deduplication happens to chunks during the usual backup process. A backup solution that does this is 'bup'.

Appendix B - Design of the proof of concept programs

In order to ensure a solidly functional rabin algorithm and disk storage format before embarking on the complex asymetric network i/o that will be required for the first iteration of the software, I designed and implemented some proof of concept programs.

The idea was to have programs that could use simple i/o so that data streams could simply be piped between them, or written to or read from files.

So, there are four programs: 'client', 'server', 'tneilc' and 'revres', and they operate in pairs.

```
files on disk      backup stream   backup stream           manifest file
----------> client ------->        ----------> server ---> signature files
                                                           data files

files on disk      restore stream  restore stream          manifest file
<---------- tneilc <-------        <---------- revres <--- signature files
                                                           data files
```

'client': Given a list of files, it reads them from the disk, splits them into variable length chunks, generates signatures, and outputs a backup stream (either to a file, or on standard output). There are options available for tweaking the rabin fingerprinting parameters.

```
Usage: client [options] <input file> ...

 -o output file
 -w sliding window size (between 17 and 63)
 -a average block size
 -m minimum block size
 -x maximum block size
```

'server': Reads a backup stream on standard input or from a file, and converts the file names, signatures and data to a storage format on disk. The storage format consists of three parts:
a) Data files, containing actual chunks of data.
b) Signature files, containing the signatures for the data files.
c) A manifest, containing the paths of the original files, and a list of the signatures that represent the data making up the original file.
The server will read any existing signatures on disk into memory at the start of execution.
If the server has seen a chunk in the backup stream already, it will not save it twice. It will

instead save the signature reference to the manifest and discard the duplicate data.

```
Usage: server [options] <input file> ...

 -d <directory> directory for output
```

'revres': The reverse of 'server'. It reads a manifest file and converts the data it references into a restore stream, which can be output to a file, or to standard output.

```
Usage: revres [options] <input file> ...

 -d base directory
 -o output file
```

'tneilc': The reverse of 'client'. It reads a restore stream, and converts it back into the files that were originally backed up.

```
Usage: tneilc [options] <input file> ...

 -d <directory> directory for output
```

Examples of the expected usage of these programs:

```
To back up two files, named 'a' and 'b':
client README | server -d /tmp/storage
To restore them:
revres -d /tmp/storage/ /tmp/storage/man/0 | tneilc -d /tmp/restore
```

B.1. Backup stream

In the original burp, each peace of information sent over the wire starts with a single byte that indicates the type of information. The next four bytes after that are a hexidecimal number representing the length of the rest of the line. This means that I can have a maximum of 65535 bytes in a line. I have found that this schema works well, and will continue to use it for this project.
For the proof of concept program, I will use:
'f' to mean 'file name'.
's' to mean 'signature'.
'a' to mean 'data' (I reserve 'd' to mean 'directory' later on).

So, when backing up a small file called 'testfile', the backup stream might look something like the following. For ease of reading, I have split it onto separate lines.

```
f0008testfile
s00300000006E18F503EA810FFB91AF5E8C4874BA3F8CE6DF96A9
a0014This is my testfile.
```

The signature is made up of the rabin fingerprint (16 characters) and the md5sum (32 characters) of the data in the file ('This is my testfile').
There may be many signature/data pairs per file backed up.

B.2. Restore stream

This is the same as the backup stream, except that there are no 's' (signature) lines required.

B.3. Storage format

There are three parts to the storage format.
The manifest (instructions on how to recreate a set of files), the signatures, and the actual data.

In the prototype, there will be three top level diretories in the storage: 'man', 'dat' and 'sig'.
Each backup will have a single manifest. The first manifest will be '0', and subsequent ones will increment that name by one each time. The 'dat' and 'sig' directories will each contain a path and file named like this, where each component of the path and name is a hexidecimal number:
'0000/0000/0000'
The signature file references the data file with the same path and name.
When enough data is written to this file (the number of chunks to be determined via some experimentation), it is closed and a new one is opened with an incremented file name:
'0000/0000/0001'

When the file part gets to 'FFFF', a new subdirectory '0000/0001' will be created, and the file within it will start from '0000' again.
Since popular file systems, like ext3, only support 32000 subdirectories, The subdirectory increments will be limited to a maximum of 30000.

So, there can be 30000*30000*65535=58981500000000 data files, which should be enough! Most file systems will run out of inodes before the number of storage data files run out.

The contents of the manifest file will look like the following, where the format of each line is following the same outline as described for the backup stream:

```
f0009testfile2
s00130000/0000/0000/0000
s00130000/0000/0000/0001
s00130000/0000/0000/0000
s00130000/0000/0000/0001
s00130000/0000/0000/0002
f0009testfile3
s00130000/0000/0000/0000
```

The signature lines reference locations in the data and signature files, the first three components being the file itself, and the last being the index of the signature/data.

The signature files will look like this:

```
s0030F45FE5EFF25857E918C552AEFA3CFD39A8ECB1F4E8D09196
s0030A423F6E2AF45C796EA35BD8C9ABC4D328133349EB80DC52A
s00301C9472B37A8DDC2106A9EBC217CF44AAC33A4AB50641A3A5
s0030933AEF79A6ED384202DA19419BC5F72908596D9D2E8B7C8F
s0030C1BEF23979ADDAADACDD8BEDF8BA67A91090777FEBD69266
```

And so on, for however many signatures there are in the signature/data file.

These signatures are loaded by the server before it reads the backup stream for the client, and uses them for deduplication.

The data files will contain the chunks for each of the signatures, each preceeded by 'a<four characters of hex>'.
Note that the server does not have to read the signatures in order to retrieve the data for a restore, because the references in the manifest also point directly to the correct location in the data files.
Also, this format allows for client compression and encryption of chunks on the client side at some future date. This will not be done for this project, but I am ensuring that the decisions that I make now do not preclude that.

Appendix C - Rabin fingerprinting algorithm in C

This is a reproduction of the C function containing the rabin algorithm that was produced for the proof of concept programs. I judge this as being the most important outcome from that exercise, although it was modified slightly during later iterations in order to support slightly different usage required in the final software where asyncronous i/o had to be taken into account. *Note that the full code of the prototype can be found in the extra materials submitted with this report.*

```
static int blk_read(struct rconf *rconf, FILE *ofp, uint64_t multiplier,
        char *buf, char *buf_end, struct win *win,
        struct blk **blkbuf, int *b)
{
        char c;
        char *cp;
        struct blk *blk;

        for(cp=buf; cp!=buf_end; cp++)
        {
                blk=blkbuf[*b];
                c=*cp;

                blk->fingerprint = (blk->fingerprint * rconf->prime) + c;
                win->checksum    = (win->checksum    * rconf->prime) + c
                                 - (win->data[win->pos] * multiplier);
                win->data[win->pos] = c;
                win->pos++;
                win->total_bytes++;
                blk->data[blk->length++] = c;

                if(win->pos == rconf->win) win->pos=0;

                if( blk->length >= rconf->blk_min
                 && (blk->length == rconf->blk_max
                  || (win->checksum % rconf->blk_avg) == rconf->prime))
                {
                        (*b)++;
                        if(*b<SIG_MAX) continue;
                        if(blks_output(rconf, ofp, blkbuf, b)) return -1;
                }
        }
        return 0;
}
```

Appendix D - Design of the first iteration

D.1. Storage format

Each client will have its own client storage directory on the server. This will contain a separate directory for each backup made, plus a single directory containing all the signature and data files.

The signature and data file format will be the same as in the proof of concept programs (see Appendix B).

Each individual backup directory will contain a 'manifest' file, which are the equivalents of the manifest files in the prototype. They contain the instructions on how to reconstruct files from the data.
They will contain more information than in the prototype.
For example, they will contain lines representing various file system entry types, such as directories, hard links, soft links, and so on.
Preceding each of these will be a line containing meta data for individual file system entry being backed up (time stamps, permissions, etc). This allows for the very significant cost saving measure of deciding whether to back up data in a file name that has been saved before, based on modification time.

The following is an example of a section of a manifest file.
'r' lines are base64-encoded meta information for the following entry.
'd' is a directory
'f' is a regular file
'S' is a reference to a location in a signature/data file.
'l' is a soft link, which come in pairs. The first line is the location of the file system entry. The second line is the target of the soft link.
Of course, there will be more file system entry types than this.

```
r003CA gB QAAF EH9 C Po Po A BAA BAA I BSB5nr BR8cCx BR8cCx A A J
d005D/home/graham/burp-cross-tools/mingw-w64-i686/libexec/gcc/i686-w64-
 mingw32/4.7.3/install-tools
r003Cn7 gB QBYV IHt B Po Po A x8 BAA I BSByPY BR8cCw BR8cCw A A J
f0067/home/graham/burp-cross-tools/mingw-w64-i686/libexec/gcc/i686-w64-
 mingw32/4.7.3/install-tools/mkheaders
S00130000/0000/000B/0C4F
r003Cn8 gB QBYg IHt B Po Po A 3S BAA I BSByPY BR8cCx BR8cCx A A J
```

```
f006B/home/graham/burp-cross-tools/mingw-w64-i686/libexec/gcc/i686-w64-
 mingw32/4.7.3/install-tools/mkinstalldirs
S00130000/0000/000B/0C50
r003AA gB QBYY KH/ B Po Po A W BAA A BSByNP BR8cCw BR8cCw A A J
l0060/home/graham/burp-cross-tools/mingw-w64-i686/libexec/gcc/i686-w64-
 mingw32/4.7.3/liblto_plugin.so
l0016liblto_plugin.so.0.0.0
```

It should be possible to process manifest files quickly, for various purposes, such as listing all the files in a directory, merging two manifests together, or finding the differences between two manifests. To facilitate this, the file system entries in the manifest will be ordered alphabetically (with case sensitive 'strcmp()' at each path component). The following simple list demonstrates that ordering:

```
/
/abc/def/ghi
/abc/def/ghi/aaa
/abc/def/ghi/mmm/nnn/ooo
/abc/def/ghi/zzz
/abc/def/zzz
/abc/dff/zzz
/zzz
```

This means that, when given a pair of paths, you will know if one is 'less than', 'equal to', or 'greater than' another path. The comparison, therefore of two manifests, will always be linear. That is, it is only necessary to make forward progress through any manifest.
A consequence of this is that, during backup, the initial client scan of the meta data for the file system entries to be backed up should also proceed in the same order.

D.2. Network data streams for backup

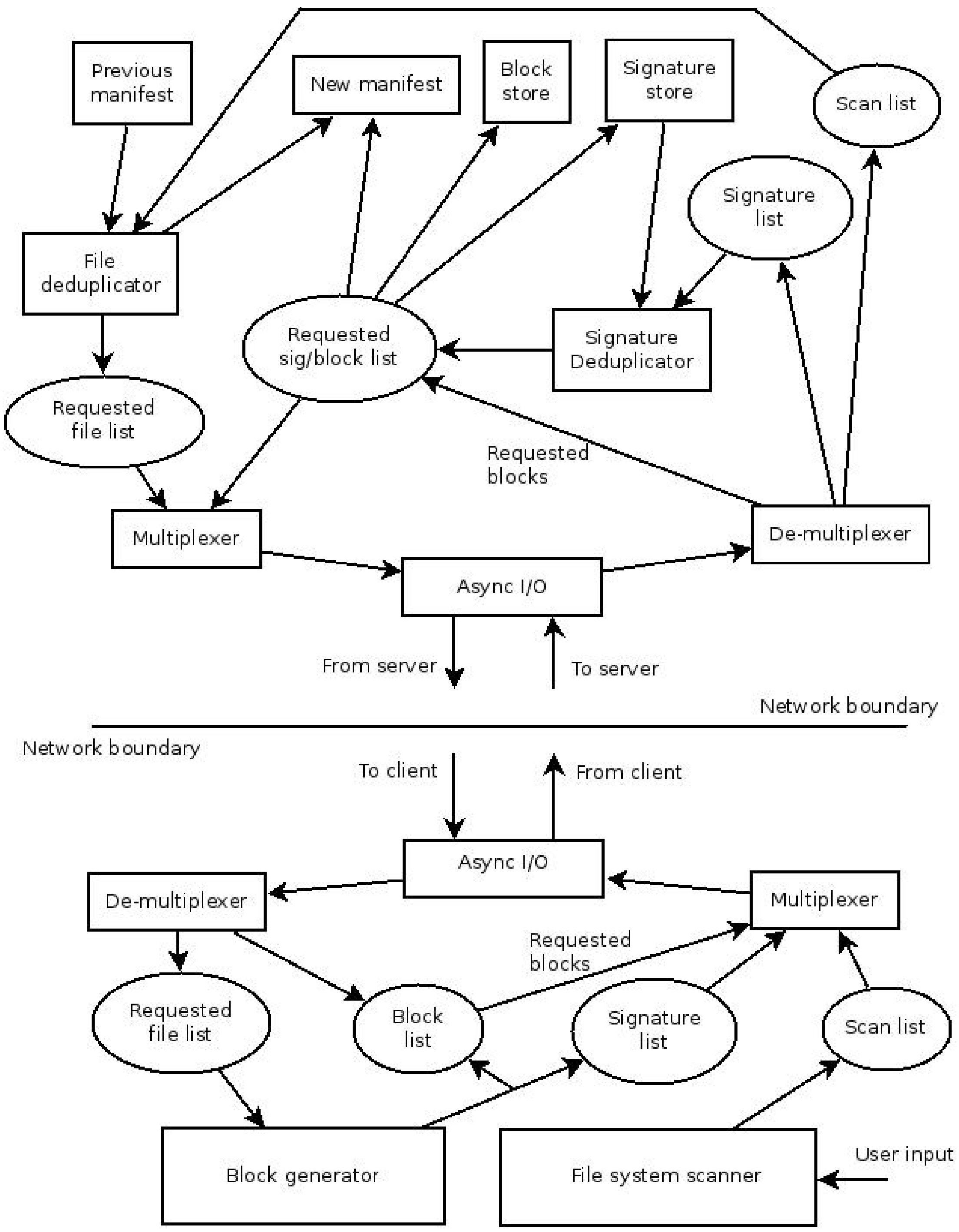
For this project, this will be changed. In order to maximise throughput, the client will send the scan information, and as soon as the server starts receiving it, it will start requesting data from the files that it needs. The client will then open those files, compute the signatures of the blocks that make up the contents of those files, and send the signatures to the server.
As the server receives the signatures, it will attempt to find them in the existing data store. If it fails, it requests the blocks it needs from the client, which will then send them. The server will save the blocks and note in the manifest the locations of the blocks that are needed to reproduce the file.
So, there will be five main components to the network communications in a backup. Three

from client to server, and two from server to client.

- Client to server stream 1 - file scan data.
  This will generally be 'r' (meta data), followed by '<entry type>' (file system entry type') with its path. Repeat until scan finished.
  r...d...r...f...r...f...
  (this is a representation of how each component of the stream will look, showing only the leading byte)
- Server to client stream 1 - file requests.
  This will be '<entry type>' with its path. Repeat until no more files need to be requested.
  The server keeps track of the files that it requested. It knows that the next set of signatures from the client will be for the next file in the list of requested files.
  f...f...f...f...f...f...
- Client to server stream 2 - signatures.
  This will be 'R' (meta data), followed by one or more 'S' (signature) entries. The meta data contains an index number unique to each signature/block in this backup.
  Repeat until no more signatures need to be sent.
  R...S...S...R...S...S...
- Server to client stream 2 - data requests.
  This will be 'D' (data request), which contains the index number for the block required.
  Repeat until no more data needs to be requested.
  D...D...D...D...
- Client to server stream 3 - data.
  This will be 'B' (block data), which contains the actual data requested. Repeat until no more data needs to be sent.
  B...B...B...B...

Each of these streams will be multiplexed over the same TCP connection, and demultiplexed again on the other side. The TCP connection should never be left waiting with no traffic passing over it, in order to maximise throughput (simultaneously in both directions) and minimise time spent.
Streams later in the above sequence have priority over those earlier in the sequence.
So, for example, while the client has an outstanding signature or block to send, it will not be sending scan information, (although it could still be gathering scan information from disk).

I have produced a high level diagram of the intended backup data flow around the client and server, showing the major components and their inputs and outputs.



D.3. Scanning the client file system

In the original burp, the client would perform an initial scan of the files to be backed up, sending the meta data to the server. Then based on the scan, once finished, and the contents of the previous manifest, it would tell the client which files that it needed the contents of and the client would then send the data.

Due to the multiplexing nature of the new system, the file scan component needs to be reworked. Instead of doing the whole scan from beginning to end and then returning (after having sent all the data across the network), there needs to be an interface that, when called, returns the next file system entry in the scan. The calling routine can then decide what to do with the data, or keep it for use later.

D.4. Pseudo code for main client backup process

```
while backup not finished {
        // multiplexer
        if nothing in write buffer {
                try to fill write buffer from requested block list
                if nothing in write buffer {
                        try to fill write buffer from signature list
                        if nothing in write buffer {
                                try to fill write buffer from scan list
                        }
                }
        }

        run async i/o (this will attempt to empty the write buffer into
          the async i/o buffer, as well as attempting to empty some of
          the async i/o buffer onto the network, and attempting to fill
          the read buffer with any incoming data)

        if something in read buffer {
                // demultiplexer
                a) if incoming file request, add to requested file list, or
                b) if incoming block request, mark block as requested
        }

        try to add to scan list by scanning the file system
        try to generate more signatures and blocks from requested files
          by opening the requested files and reading them
}
```

D.5. Pseudo code for main server backup process

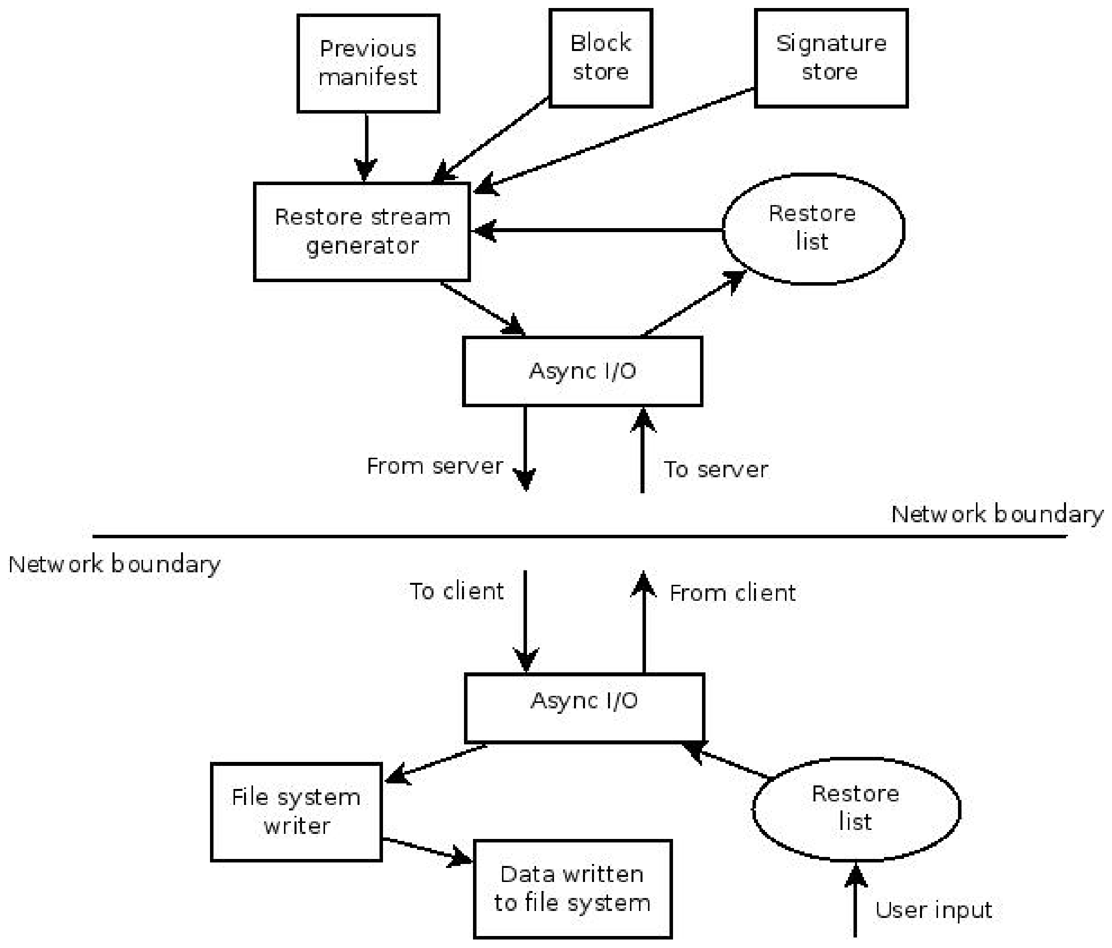
```
while backup not finished
{
        // multiplexer
        if nothing in write buffer
        {
                try to fill write buffer from signatures from client
                if nothing in write buffer
                {
                        try to fill write buffer from scan list from
                        client
                }
        }

        run async i/o (this will attempt to empty the write buffer
          into the async i/o buffer, as well as attempting to empty some
          of the async i/o buffer onto the network, and attempting to
          fill the read buffer with any incoming data)

        if something in read buffer
        {
                // demultiplexer
                a) if incoming scan, add to scan list from client,
                   deduplicating via modification time comparison if
                   possible
                or
                b) if incoming signature, add to signature list from
                  client, deduplicating via signatures from signature
                  store if possible
                or
                c) if incoming block, add incoming block and signature
                  to data store
        }
}
```

D.6. Network data streams for restore

Initially, the data streams for restore will be the same as in the original burp. That is, the server will send a meta data line, then the type of file system entry to create, and then zero or more lines containing the content to restore.
In time, the new software can do similar deduplication in the reverse direction. For example,

chunks of the data store could be replicated onto the client, and then instructions for how to use it to reconstruct the original data could be sent, thereby meaning that each chunk would need to cross the network once. For now though, I am aiming for reasonable restore times, so this kind of reverse deduplication is beyond the scope of the project.

I have produced a high level diagram of the intended restore data flow around the client and server, showing the major components and their inputs and outputs.



Appendix E - Design of the second iteration

The documents that were originally written for the second iteration were not quite right, and needed to be modified due to some slight misconceptions from earlier designs.

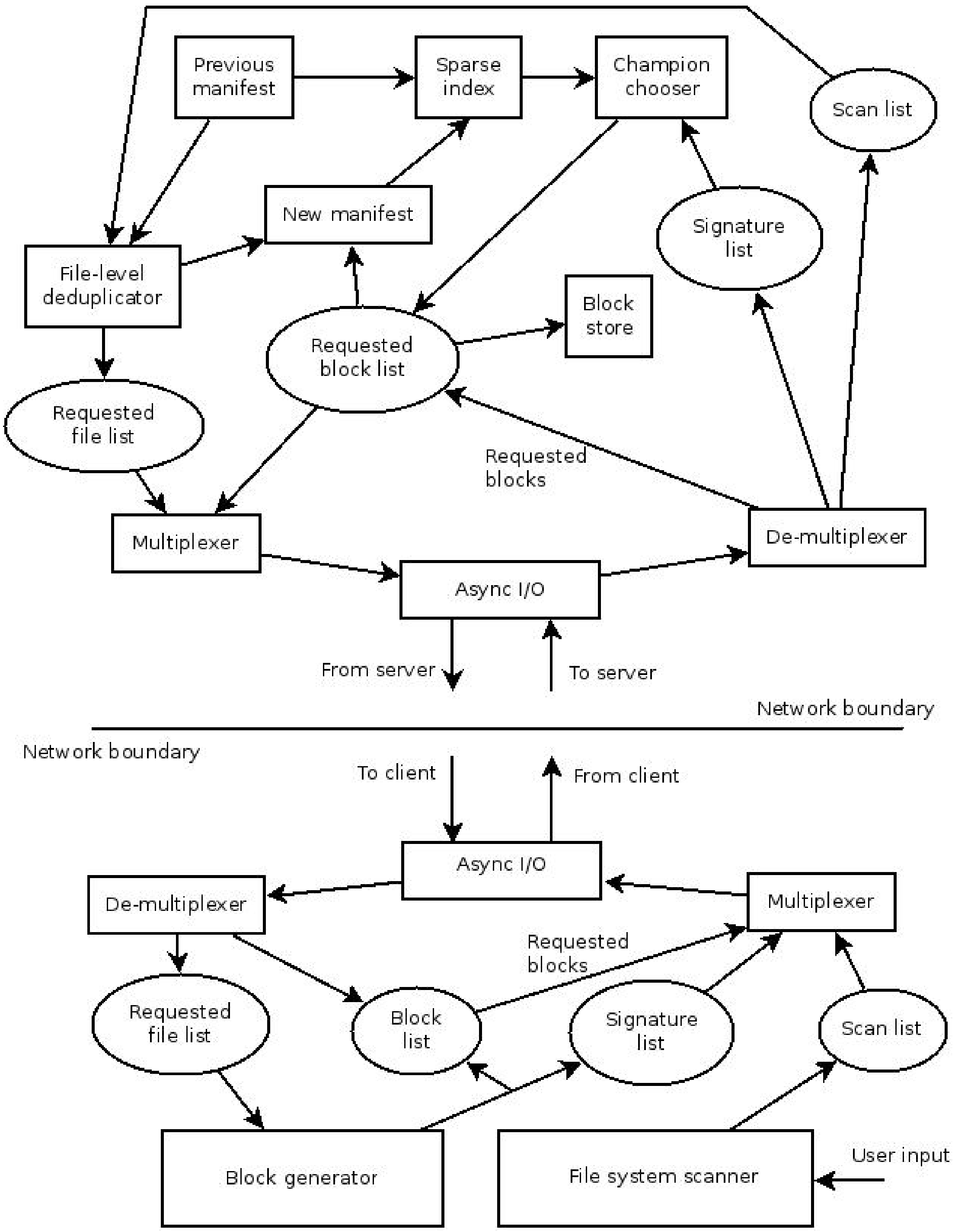
After the first iteration, I realised that the storage format was not quite right. There is a server 'champion chooser' module now required for the second iteration, which selects lists of signatures to duplicate on, given a collection of hooks from the client. To select the signatures to load, it has a 'sparse index' of hooks in memory.
I had imagined that it could load the signatures from the signature store in the previous design, which would map directly to the blocks in the block store.
However, this was not right, because after multiple backups, the signatures in the signature store start to lose the property of locality as deduplication takes effect, since only new blocks get added to the data store.

Instead, the champion chooser needs to use sparse indexes on the manifest of each backup. That is, the actual sequence of blocks as they appeared in each backup, rather than the compacted data in the block store.

This also means that there is no longer any need for the 'signature store' in the previous design, since nothing looks at it any more.

So, there follows a slightly modified version of the original design for the second iteration.

First, an updated high level diagram of the intended backup data flow around the client and server, showing the major components and their inputs and outputs.



E.1. Network data streams

The data streams remained the same as in the first iteration for both backup and restore.

E.2. Scanning the client file system

This also remained the same. In fact, there were no design changes at all for the client.

E.3. Storage format

In order to allow deduplication across multiple different clients, the directory structure for backup storage now looks like this:

```
<group>/data/
<group>/data/sparse
<group>/data/0000/0000/0000
<group>/data/0000/0000/0001 (and so on)
<group>/clients/<name>/<backup number>/manifest/
<group>/clients/<name>/<backup number>/manifest/sparse
<group>/clients/<name>/<backup number>/manifest/00000000
<group>/clients/<name>/<backup number>/manifest/00000001 (and so on)
```

Clients can be configured to have different deduplication groups, and can thereby share saved blocks. When opening files in the 'data' directory for writing, the server now needs to get a lock before doing so, since more than one child process could be trying to open the same file.

Due to the need for the 'champion chooser' module to index the manifests from each backup, the manifest files are now broken down into multiple files, containing a maximum of 4096 signatures each. They still have the same internal format as described in the first iteration.

At the end of each backup, a sorted sparse index for each of the manifest components is generated, and the 'sparse' file in the manifest directory is updated to contain all of them. This file is then merged into the 'sparse' file in the 'data' directory for the group. If there are entries with identical hooks, the more recent manifest component is chosen.
The following is an example of the format of the 'sparse' files. The paths are always relative to whatever the root directory is. This ensures that backup group directories can be copied to other locations at anytime without causing path problems.

```
M0042global/clients/black/0000005 2013-10-22 16:44:54/manifest/0000022F
F0010F00008A155DF4307
F0010F00258BF988BFC28
...
M0042global/clients/white/0000003 2013-10-10 00:04:22/manifest/00000198
F0010F000182980D81EEC
F0010F000227AB578BE4C
...
```

The 1 character plus 4 character syntax at the beginning of line follows the strategy described previously.'M' means 'path to manifest component', and 'F' means 'fingerprint'. Duplicate fingerprints ('hooks') within an index are removed. The rabin fingerprints chosen for the hooks are those that start with 'F'. This generally gives between 200 and 300 hooks for each manifest component. Since there are 4096 signatures in each manifest component, this reduces the memory requirement for the index to about 1/16.
At some future point, this can be halved if the software is modified to pack the fingerprints into 4 bits per character. Additionally, fewer hooks could be chosen per manifest component for more memory reduction. However, just choosing those that begin with 'F' is enough for the current project.

E.4. Champion chooser

The champion chooser module takes the sparse index (generated from previous manifests) as an input, plus a sequence of incoming hooks from the client. The server program waits until enough signatures (currently 4096, or no more left to receive) have been received before running the main champion chooser code.

The champion chooser needs to choose the manifest component that contains the largest number of hooks in the sequence from the client. When it has done so, it needs to repeat the procedure to choose another, but *exclude hooks from already chosen manifest components*. Then repeat again, X number of times, or until all the hooks have been chosen. Crucially, all this needs to be fast.

In memory, the fingerprint hooks are converted to uint64 types and are used as the indexes for 'sparse' structs making up entries in a hash table. Each struct contains an array of pointers to candidate manifest component entries. When a hook is added from a candidate, a new 'sparse' struct is added if one didn't already exist for the hook, and a pointer to the relevant candidate is added to array.

The candidate manifest components are structures containing the path to the component, and a uint16 pointer for its score - the content of the pointer is contained in a separate array, which means that the scores array can be very quickly reset without needing to iterate over all the candidates.

Each incoming hook structure from the client has a pointer to a uint8 type called 'found', which indicates whether a candidate has been chosen that contains this hook. Again, this points to an entry in a separate array, so that resetting these values is just a matter of setting the contents of the array to zero, rather than having to iterate over all the incoming components.

When it is time to choose the set of champions, the incoming 'found' array is reset.
Then, while not enough champions manifest components have been found, the scores are reset, and the incoming hooks are iterated over. If an incoming hook's 'found' field is set, it is skipped. Note that none of the 'found' fields have yet been set for hooks from the immediately previous iteration.
If a hook is found in the hash table, the scores of all the candidates that contain that hook are incremented in turn. A pointer to the highest scoring candidate is maintained as the scores are incremented.
If a candidate pointer is arrived at that is the same as the immediate previously found manifest, the incoming hook's 'found' field is set, and any incremented scores that were set on its candidates are reset. This excludes previously accounted for hooks from the score, and since the 'found' field is now set, subsequent passes discount them faster.

After iterating over the incoming hooks, the highest scoring candidate manifest component is chosen to be loaded into memory via its path component. As this contains all the signatures from that candidate and not just the hooks, the server now has a reasonable set of signatures with which to deduplicate incoming blocks.
If not enough champions have been found, the incoming hooks are iterated over again.

As this requires only one file to be open and read for each candidate chosen, the problem of the disk deduplication bottleneck has been significantly reduced.

Finally, an important feature to note is that the server needs to continually add manifest components *from the current backup in progress* as candidates. Otherwise, the first backup would potentially store far more data than necessary, as it has nothing to deduplicate against. Subsequent backups would be able to deduplicate against previous ones, but not themselves.

Appendix F - Open source competition

Before any design, coding, or testing was done, I researched the open source network backup solutions available. The following are the results of that initial research.

burp-1.3.36 (Keeling, 2011):
Has a simple client/server architecture.
Uses librsync in order to save network traffic and to save on the amount of space that is used by each backup. It also uses VSS (Volume Shadow Copy Service) to make snapshots when backing up Windows computers. Operates at a file-level granularity.
Version 1.3.36 was the latest version at the time of writing.

Advantages:

- Backup clients can be centrally managed on a Unix-based server.
- Good cross platform support for clients. Supports Unix-based systems, including Macs. Supports the native Windows Backup API with VSS snapshots. This ensures a consistent file system image and also ensures that all files are available to be read. Backup solutions that do not use the API will not be able to open files that other applications already have open.
- Backs up file differences via delta differencing with librsync.
- Supports files, directories, symlinks, hardlinks, fifos, nodes, permissions and timestamps.
- Supports Linux and FreeBSD acls and xattrs.
- Supports Windows permissions, file attributes, and so on, via VSS.
- Supports Windows EFS files.
- Storage and network compression using zlib.
- Ability to continue interrupted backups.
- Network communications encrypted with SSL.
- Automatic SSL certificate authority and client certificate signing.
- Client side file encryption - (note: this turns off delta differencing).
- Scheduling.
- Email backup success/failure notifications.
- Pre/post backup/restore client scripts.
- Storage data deduplication.
- Automatic client upgrade.
- Due to the design of the server, most configuration changes do not need a server restart in order to become effective.
- Simple retention periods (e.g, keep 1 backup per day for 7 days, 1 backup per week for 4

weeks, 1 backup per 4 weeks for a year).
- MD5 Verification of saved data.

Disadvantages:
- Since each file is stored as a separate file system entry on the server, Backups containing many files can end up using a lot of file system inodes. This has consequences on the amount of time it takes to complete subsequent backups, because many system calls are required, for example, to hard link unchanged files into place in the latest backup. Or to delete old backups, because each inode has to be unlinked.
- As previously described, there are limitations of the librsync mechanism related to large files that change.
- Identical files across multiple clients will be stored multiple times. This can be dealt with by post-event file deduplication, but this is not optimal and with large numbers of files, takes a significant amount of time.
- Identical blocks will be stored multiple times across all files and all clients.
- If configured to use reverse librsync deltas to save storage space, restore times for large files can become long, because each delta has to be applied and the large file regenerated for each change in the sequence.

amanda (da Silva et al, 1991):
Server contacts each client to perform a backup at a scheduled time. Has a native Windows client. On Unix-like systems, it uses native tools to make the backup, like tar.

Advantages over burp-1.3.36:
- Has a long history, and therefore should be expected to be stable.
- No hard link farm.

Disadvantages over burp-1.3.36:
- Like bacula, it is clearly focused on tape storage and pretends that a file on the disk is actually a tape, which leads to inefficiencies that don't need to exist when your medium has random access.
- Hard to configure.

backshift (Stromberg, 2010):
Uses variable-length blocks to perform inline deduplication. Stores the chunks on disk in a directory structure named after the checksums it creates. It appears to use the file system as its full chunk index.

Appears to be designed with local backups in mind, rather than networked.

Advantages over burp-1.3.36:

- Inline deduplication.

Disadvantages over burp-1.3.36:

- No Windows API support. Open files cannot be backed up.
- No central management, or scheduling.
- Poor network support. Cannot run over the network with ssh. Instead, it needs to mount a network share (smbfs, sshfs, etc) and read the data from that. Suspect that all the data is transferred every time.
  On investigation,
  http://stromberg.dnsalias.org/~strombrg/backshift/documentation/for-all/backing-up.html, says "writing to a remote filesystem is faster than reading from one - if you have the choice", so pushing probably means that not all the data is transferred every time. I will push when I test it.
- Running on Windows requires installation of cygwin and python, or for the Windows filesystem to be mounted on the server via a network filesystem, which means that the backup has to be pulled.
- Will suffer from disk deduplication bottleneck issues due to the full index created on the disk.

backuppc (Barratt, 2001):
Uses no client side software, instead backs up clients using network shares and tar or rsync on the server.
Has a 'pool based' deduplication mechanism to deduplicate across multiple clients. This apparently operates at a file level granularity, not block level, and is therefore not suitable for backups of disk images. The 'pool' appears to operate as a hard link farm.

Advantages over burp-1.3.36:

- Claims to be 'enterprise grade'.

Disadvantages over burp-1.3.36:

- No Windows API support. Open files cannot be backed up.

bacula (Sibbald, 2010):
The original burp was based on bacula-5.0.3, so it contains many of the best features of bacula, whilst solving many of its problems.
Instead of being of a client/server architecture, bacula has four main components - the director, the file daemon, the storage daemon, and the catalog.

Advantages over burp-1.3.36:

- Since bacula tries to emulate a tape drive when it saves to disk, it stores the data in a tar-like format containing the data for many files. This means that there are no issues with management of large hard link or mirror farms. However, it does have other issues related to the way it tries to view these storage files as tapes.
- Claims to be 'enterprise grade'.

Disadvantages over burp-1.3.36:

- Complexity to configure - each of the main components has their own set of configuration files, for example.
- Although it avoids problems with hard link farms, it still works badly with disk storage - Bacula's mentality is very highly geared towards tape usage and therefore it works poorly with disks.
- Stores the catalog separately to the backups - This causes a massive maintenance headache. For example, you now have to think about backups of your catalog. Additionally, changes to your configuration files might not take effect because some of the previous configuration gets written to the catalog, and then it is not easy to make the changes take effect. Furthermore, you end up needing to be a mysql or postgres database expert.
- Backs up the whole file even if only a few bytes in it have changed.
- Relies far too heavily on clock accuracy - Bacula goes very badly wrong if your computer's clock somehow gets skewed. In fact, it relies so heavily on the clock and timestamps that it does not actually track which backup another was based on.
- Laptop backups are difficult to schedule.
- Cannot resume an interrupted backup.
- Retention configuration - my experience taught me that it is just impossible to configure a sensible retention policy for bacula. The reasons why are too long to go into here, but my post to the bacula email user list on the subject can be found at: http://adsm.org//lists/html/Bacula-users/2011-01/msg00308.html
- No Windows EFS support - EFS files are silently ignored.
- Has a commercial edition, into which most new features go.

bup (Pennarun, 2010):
Uses the versioning control system 'git' as its backend. Reads data directly on standard input, splits it into chunks using a rolling checksum, and packs it directly into git packfiles. Before writing a chunk, it first deduplicates using previously written packfiles. Can be run securely over the network with ssh.

Advantages over burp-1.3.36:
- Inline deduplication.
- Efficient at backing up large files, such as huge disk images.
- No need to apply deltas when restoring old versions of files, as each chunk is retrieved directly from storage when needed.

Disadvantages over burp-1.3.36:
- Immature meta data support.
- No Windows API support. Open files cannot be backed up.
- Running on Windows requires installation of cygwin.
- Cannot prune away old backups.
- No central management, or scheduling.
- Performing a backup stores some data on the client.

obnam (Wirzenius, 2007):
Uses ssh to transfer data. Seems to be able to do inline deduplication.

Advantages over burp-1.3.36:
- Inline deduplication.

Disadvantages over burp-1.3.36:
- No Windows API support. Open files cannot be backed up.
- Running on Windows requires installation of cygwin.
- No central management.

rdiff-backup (Escoto et al, 2001):
Backs up one directory to another, possibly over a network.
Like burp, it uses librsync. It creates a mirror, and reverse diffs are created so that previous versions of files can be restored.

Disadvantages over burp-1.3.36:
- No Windows API support. Open files cannot be backed up.
- Running on Windows requires installation of cygwin.
- No central management, or scheduling.

rsync (Tridgell et al, 1996) --link-dest wrappers:
The rsync --link-dest functionality is used as the back end of various backup scripts, such as rsnapshot.
A set of files is backed up as a mirror. In subsequent backups, files that have not changed are hard linked to the entries in the previous backup in order to save disk space. Or, in other words, you have a hard link farm.

Disadvantages over burp-1.3.36:
- No Windows API support. Open files cannot be backed up.
- Running on Windows requires installation of cygwin.
- No central management, or scheduling.

tar over ssh (GNU, 1999):
The name 'tar' is derived from 'tape archive'. is a utility that combines files and meta data into a single stream. Running it over ssh ('ssecure shell') means that the stream can be sent across the network securely. I am including this as a backup option because it gives me some sort of scientific control. Each backup that this method makes will be self contained, not relying on any other backup, so the network utilisation and storage will be consistent for each run.

Advantages over burp-1.3.6:
- These tools are ubiquitous over Unix-like operating systems.

Disadvantages over burp-1.3.6:
- No Windows API support. Open files cannot be backed up.
- Windows does not come with open source programs like tar, or ssh. A linux-like environment needs to be installed with cygwin in which to run them.
- No central management, or scheduling.
- All the data is transferred each time.
- Massive redundancy of stored data. Will take up a lot of disk space. *Note: After performing the tests, I discovered that tar has the ability to do incremental backups, and amanda uses this capability. Therefore, amanda's backup results can be seen as analagous to the results of tar's incremental backups had I run tests on tar in that mode.*

urbackup (Raiber, 2011):
Client/server backup system. File and image backups are made while the system is running without interrupting current processes. Also continuously watches directories that you want backed up, in order to quickly find differences to previous backups. Has a native Windows client.

Advantages over burp-1.3.36:
- Image backups of Windows (but not Unix-style systems)
- Has an interesting method of broadcasting on the LAN in order to find clients to back up.

Disadvantages over burp-1.3.36:
- Windows or posix ACLs, alternate data streams or permissions are not backed up during a file backup.
- Poor linux support - the online manual states "the client software currently runs only on Windows while the server software runs on both Linux and Windows". However, at the time of writing, I did find source for a Linux client.
- Has an underdeveloped command line interface. It is impossible to restore from the command line, or to trigger a backup. Files can only be restored one at a time from its web interface. Consequently, I am not able to test this software properly.

Appendix G - Raw test data

Tests a1-a6 are backups of small files.
Tests b1-b6 are restores of small files.
Tests c1-c6 are backups of small files.
Tests d1-d6 are restores of small files.

amanda-3.3.1
This software uses either tar (in incremental mode) over ssh, or 'dump' to retrieve files from the client. For the tests, I configured it to use tar.
I did not capture memory statistics for the use of tar and ssh, and due to the complexity of the restore procedure, could not get server memory statistics for the restore.
Each backup was performed as an incremental (amanda automatically promoted the first of each sequence to a full).

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	32:56	-	23248	19183345724	17510752	30
a2	04:31	-	23344	107900600	17606856	36
a3	05:41	-	23328	656049880	18203648	42
a4	06:15	-	23360	655611900	18800440	48
a5	14:45	-	23252	6360937308	24597208	54
a6	02:22	-	23468	36993628	24630116	60
b1	30:19	15348	-	18903206284	-	-
b2	31:51	15464	-	19009897796	-	-
b3	33:08	15524	-	19544418832	-	-
b4	33:12	15532	-	19545361840	-	-
b5	41:58	15548	-	25166198412	-	-
b6	45:47	15544	-	25179850864	-	-
c1	32:36	-	23260	24220077632	22190460	30
c2	00:03	-	22184	39692	22190564	36
c3	32:31	-	23252	24227710432	44380932	42
c4	32:28	-	23272	24224534548	66571300	48
c5	32:30	-	23244	24229732244	88761668	54
c6	15:44	-	23260	11725042740	99503956	60
d1	33:30	4608	-	23807491248	-	-
d2	33:37	4620	-	23804837640	-	-
d3	32:56	4580	-	23806787244	-	-
d4	33:00	4584	-	23818501728	-	-
d5	1:05:58	4512	-	47611538272	-	-
d6	49:04	4512	-	35333166188	-	-

backshift-1.20
The server storage directory was mounted on the client using sshfs and backshift was run on the client side, saving to the mounted directory. Several hours into the second backup, where no files had been touched in the backup set, testing backshift was abandoned.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	43:37:50	125132	-	6709214436	6314132	1290187

backuppc-3.2.1
This software uses tar over ssh to retrieve files from clients.
I did not capture memory statistics for the use of tar and ssh.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	1:23:49	-	284984	17901726148	9100204	2106563
a2	19:08	-	284972	28261484	9511692	2226526
a3	25:23	-	284968	542277068	10456412	2423805
a4	25:43	-	284968	542110772	10835368	2579155
a5	30:25	-	285008	5872127792	11253144	3188940
a6	19:52	-	110464	5854417192	11431256	3728321
b1	51:47	-	1172	18580045616	-	-
b2	1:00:56	-	1168	18616293176	-	-
b3	1:02:02	-	1168	18608841452	-	-
b4	1:06:50	-	1172	18627536544	-	-
b5	1:06:38	-	1168	18611113920	-	-
b6	22:21	-	1160	6088763864	-	-
c1	32:28	-	19644	24239202344	22190452	58
c2	00:04	-	19648	8680	22190484	71
c3	23:49	-	23704	53683896	44380820	96
c4	22:46	-	23704	53678308	44380864	114
c5	32:21	-	19620	24240577464	44380928	137
c6	15:47	-	19656	11734617416	55123184	161
d1	32:55	-	1144	23500251316	-	-
d2	32:54	-	1140	23499752736	-	-
d3	32:56	-	1144	23505114172	-	-
d4	32:56	-	1140	23504421352	-	-
d5	33:05	-	1148	23497633972	-	-
d6	16:03	-	1152	11377405112	-	-

bacula-5.2.13
This software has multiple components - the database, the director, the storage daemon and the file daemon. MySQL was used as the database software, and ran on the server along with the director and storage daemon. The file daemon runs on the client side. Memory statistics were recorded for all of these. MySQL disk space and nodes were also included in the results.
Each backup was performed as an incremental (bacula automatically promoted the first of each sequence to a full).

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	1:14:13	1188	183516	19415316808	16613252	28
a2	06:24	315816	212648	7339	16613252	28
a3	08:22	315712	212592	558876468	17090484	28
a4	08:21	315820	212852	558831313	17567712	28
a5	33:18	315932	212464	6558268748	23074912	28
a6	12:51	318896	212900	403522651	23214996	28
b1	37:49	4364	394420	18761519812	-	-
b2	37:53	4364	394404	18761497347	-	-
b3	37:41	4364	394348	18761753952	-	-
b4	37:50	4360	394444	18760842419	-	-
b5	38:41	4368	394380	18762252423	-	-
b6	14:06	4372	279184	6143147781	-	-
c1	36:36	4820	58416	24148768088	22207036	28
c2	00:01	13648	58980	7337	22207036	28
c3	35:27	14168	58416	24150990612	44413760	28
c4	36:00	14376	58792	24154976470	66620488	29
c5	36:06	14380	58496	24151605070	88827212	29
c6	17:27	48420	58224	11689696758	99577364	29
d1	32:28	4372	22583	24035061412	-	-
d2	32:28	4360	58076	24033754963	-	-
d3	32:27	4376	58328	24034781702	-	-
d4	32:45	4360	58516	24094745919	-	-
d5	32:29	4360	57904	24034420735	-	-
d6	15:44	4356	57952	11634940513	-	-

bup-0.25
For the tests, bup used ssh for the network transport. Memory usage
for ssh was not recorded. The bup 'split' command was used for backing up,
and 'join' for restore.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	1:05:25	50396	41564	13028027920	12024828	92
a2	47:36	36924	36976	22788	12054464	97
a3	47:41	43160	23764	552641840	12541864	98
a4	47:28	42076	15656	301338788	12818772	101
a5	51:40	50160	40564	3441960928	15993348	113
a6	16:46	43636	8828	123336	15993444	116
b1	27:49	8084	3687524	18721975020	-	-
b2	28:12	8080	3689344	18727867332	-	-
b3	27:31	8072	3688592	18720814964	-	-
b4	27:15	8072	3689556	18725122480	-	-
b5	27:49	8076	3446688	18731302336	-	-
b6	09:57	8120	1689772	6136944368	-	-
c1	37:06	54728	49860	14760055588	13673276	97
c2	19:30	43400	8214	32492	13647632	98
c3	19:38	45132	11000	44790792	13688684	101
c4	19:34	43888	8132	64096	13688720	104
c5	19:35	43892	34472	63836	13715140	109
c6	09:26	43872	8124	80428	13689000	110
d1	33:40	8116	3694364	23825024780	-	-
d2	33:53	8116	3687716	23821504284	-	-
d3	33:28	8084	3695360	23811360164	-	-
d4	32:58	8084	3696864	23815830112	-	-
d5	32:56	8084	3697192	23823484740	-	-
d6	16:00	8120	3696736	11533677068	-	-

obnam-1.1
The server storage directory was mounted on the client using sshfs and
obnam was run on the client side, saving to the mounted directory.
Testing on obnam was abandoned after it hadn't finished after
8 hours and had taken up more than 50GB of space on the server,
when there were only 22GB of files to bakup up on the client.

rdiff-backup

For the tests, rdiff-backup used ssh for the network transport. Memory usage for ssh was not recorded. I was unable to record server memory statistics for the server when restoring large files.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	2:08:09	155932	632632	19549959636	20492104	1624474
a2	25:17	154156	29548	732119660	20579220	1624480
a3	1:05:54	172512	73508	1330676312	21232008	1669655
a4	1:04:33	172576	73516	826853828	21492456	1714830
a5	10:56:29	156352	699136	13588674368	28216628	2763315
a6	12:28:14	65212	29532	1856630968	28095444	2763324
b1	2:20:46	287712	630348	18493237092	-	-
b2	2:19:55	287832	630416	18493619200	-	-
b3	2:19:32	287840	630028	18494316560	-	-
b4	2:18:52	287960	630416	18494140436	-	-
b5	2:18:07	289328	623580	18494732116	-	-
b6	44:49	105024	255252	6056832020	-	-
c1	1:40:30	10908	11256	24084094872	22190296	13
c2	00:05	10260	10396	68516	22190304	19
c3	42:21	17984	17292	104168884	22272648	27
c4	52:52	18364	18600	1999400	22272684	35
c5	2:09:00	11404	11092	24086698904	44462964	44
c6	35:39	17996	17996	2029536	41372716	52
d1	1:21:19	11416	-	23953765636	-	-
d2	1:27:13	11412	-	23955454164	-	-
d3	1:03:15	11416	-	23953752660	-	-
d4	42:23	11416	-	23954690536	-	-
d5	1:02:19	11412	-	23957680960	-	-
d6	20:29	11412	-	11596528360	-	-

rsync-link-dest

For the tests, rsync used ssh for the network transport. Memory usage for ssh was not recorded.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	34:57	42224	49964	17921288120	20001104	1535719
a2	05:38	28316	28412	43588912	20360088	3071436
a3	08:11	28292	37028	560651704	21285484	4607153
a4	07:12	28580	37224	54572948	22210876	6142870
a5	14:36	34208	47856	5899248188	29008644	7678588
a6	01:51	16708	17624	14553352	29131180	8187702
b1	28:37	97540	138424	17396435028	-	-
b2	28:29	97544	138532	17407095180	-	-
b3	28:05	97540	136904	17394005376	-	-
b4	27:54	97544	135868	17392775068	-	-
b5	27:42	97544	138756	17391105544	-	-
b6	08:46	69624	61852	5686955940	-	-
c1	32:33	6588	1488	24254325112	22190276	4
c2	00:02	3188	1248	7316	22190280	6
c3	28:58	6312	9140	96024684	44380552	8
c4	29:41	3320	9144	2229896	66570824	10
c5	32:26	6852	1488	24242023980	88761096	12
c6	15:33	3216	9068	1887460	99503292	14
d1	32:31	7380	1256	23513700756	-	-
d2	32:33	7376	1252	23515115572	-	-
d3	32:40	7280	1244	23511657216	-	-
d4	32:46	7316	1252	23529028584	-	-
d5	32:54	7476	1248	23520135780	-	-
d6	15:54	7256	1048	11383879512	-	-

tar-over-ssh

Memory usage settings for ssh were not recorded. For the server, that meant no memory usage results, since that was the only process running on the server.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	34:14	1416	-	18906459540	17457040	2
a2	32:46	1420	-	18901056092	34914076	3
a3	32:55	1416	-	18902683324	52371112	4
a4	32:53	1416	-	18901313052	69828148	5
a5	32:49	1436	-	18903018656	87286192	6
a6	11:27	1396	-	6192213332	93003248	7
b1	25:47	1120	-	18716011072	-	-
b2	25:49	1116	-	18710756096	-	-
b3	25:50	1124	-	18710245600	-	-
b4	25:52	1124	-	18714191624	-	-
b5	25:53	1128	-	18713813356	-	-
b6	08:26	1118	-	6127883736	-	-
c1	33:05	1204	-	23944470936	22190280	2
c2	32:40	1208	-	23944195580	44380556	3
c3	33:04	1208	-	23943847812	66570832	4
c4	32:47	1216	-	23943428308	88761108	5
c5	32:45	1216	-	23944522232	110951384	6
c6	15:54	1216	-	11591519516	121693580	7
d1	33:03	1152	-	23814698300	-	-
d2	32:44	1148	-	23813943948	-	-
d3	32:48	1152	-	23806047744	-	-
d4	32:47	1148	-	23805303488	-	-
d5	32:46	1156	-	23806356912	-	-
d6	15:55	1160	-	11526917756	-	-

urbackup-1.2.4

Note: Testing on urbackup was abandoned when it was discovered that the only way to restore files was one at a time via its web interface. This made restoring millions of files impractical.

burp-1.3.36
A one line patch was applied to burp-1.3.36 in order to turn off network compression. Another patch was applied to prevent it compressing individual files on restore.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	56:06	5136	3712	19073221246	20381968	1535686
a2	27:13	5124	3752	559650895	20762924	1535691
a3	35:12	5140	3844	1232999529	21720168	1580870
a4	38:03	5144	3828	764511451	22281984	1626049
a5	39:35	5128	3712	6630073883	29226320	2135147
a6	13:21	5124	3752	193122037	29355744	2135158
b1	39:58	3452	49216	18408345630	-	-
b2	39:51	3448	49980	18408178034	-	-
b3	43:34	3452	41896	18408416587	-	-
b4	38:01	3452	35288	18403160479	-	-
b5	37:05	3456	34568	18405181477	-	-
b6	11:26	3448	13908	6025288820	-	-
c1	32:56	3916	3560	24209453432	22190320	15
c2	0:04	3912	3652	10605	22190340	20
c3	49:24	10124	8536	125113914	22272284	32
c4	46:25	10124	8644	22968510	22272352	44
c5	32:56	3912	3560	24209415813	44462664	56
c6	39:41	10124	6788	22835167	41393692	68
d1	1:18:28	3456	3424	24168034544	-	-
d2	1:12:39	3452	3424	24167416085	-	-
d3	53:58	3456	3428	24167803329	-	-
d4	33:13	3452	3424	24167751901	-	-
d5	55:25	3448	3428	24167297229	-	-
d6	16:08	3452	3428	11699804162	-	-

burp-2.0.0
A one line patch was applied to burp-1.3.36 in order to turn off network compression.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	50:12	936276	422436	6261782638	5321000	459
a2	6:01	560492	167004	463561084	5647588	466
a3	8:22	560604	191472	993873340	6447976	521
a4	7:38	560488	191036	479991676	6774508	528
a5	18:47	561020	267716	646705357	7102040	535
a6	2:17	188924	183604	154855021	7211576	542
b1	2:32:14	3392	1419164	18072607863	-	-
b2	2:24:02	3396	1460392	18072342507	-	-
b3	2:13:37	3392	1514896	18071492855	-	-
b4	2:12:54	3396	1502624	18071360723	-	-
b5	2:16:30	3396	1499876	18072840238	-	-
b6	1:09:27	3396	925132	5919571611	-	-
c1	44:27	166552	352840	14740348487	13442384	963
c2	00:22	3752	369148	10712	13529832	970
c3	32:45	96172	370104	333668235	13652820	981
c4	32:38	44292	370268	294625537	13740276	988
c5	32:42	44292	370012	294212673	13827728	995
c6	15:50	44284	370036	143341975	13870216	1004
d1	1:50:26	3392	929832	24079869152	-	-
d2	1:49:51	3408	922188	24080071062	-	-
d3	1:56:46	3408	901052	24081243535	-	-
d4	2:00:49	3408	870040	24081042956	-	-
d5	2:12:54	3412	842132	24081705105	-	-
d6	41:56	3408	717240	11656952804	-	-

burp-2.0.1
A one line patch was applied to burp-1.3.36 in order to turn off network compression.

		max mem (kb)				
test	time	client	server	network (b)	disk space (kb)	nodes
a1	49:55	964824	222620	6333908776	5134556	1047
a2	5:26	546768	15616	459379807	5275764	1862
a3	7:30	546796	23460	982353947	5886076	2701
a4	7:01	546792	23440	470919618	6027296	3516
a5	17:36	547928	88208	631650761	6169096	4332
a6	1:57	184516	14256	151701404	6215876	4604
b1	2:03:56	3480	333452	18070868859	-	-
b2	2:02:49	3464	333448	18070889282	-	-
b3	2:00:17	3464	333452	18069833763	-	-
b4	2:00:25	3460	333248	18070389579	-	-
b5	1:59:24	3464	332836	18070507134	-	-
b6	1:04:34	3460	333032	5919135548	-	-
c1	46:07	166628	27076	15968721238	14442396	1317
c2	0:59	3836	23276	10660	14544832	2113
c3	33:22	166520	28208	380791042	14704548	2912
c4	33:30	137996	25148	351607673	14817148	3709
c5	33:30	137996	25264	349251193	14929748	4506
c6	16:18	122168	23076	169103345	14982380	4898
d1	1:08:10	3476	369672	24077829873	-	-
d2	1:08:05	3464	369660	24076534634	-	-
d3	1:10:13	3480	368820	24077401653	-	-
d4	1:14:52	3476	368608	24078404161	-	-
d5	1:13:54	3480	368612	24077496825	-	-
d6	26:55	3460	362164	11655881312	-	-

Appendix H - Items to be completed before releasing the new software

This is a raw list from my notes of the the things that need to be done before an initial release of the newly developed software. I expect these items to take roughly two months to complete. As this is a raw list, much of it relates to obscure parts of the software, and is provided for interest only.

```
* Merge in burp1 code and make it possible to switch between burp1 and
  burp2 modes. This will help transition, and enables support to continue
  for situations where burp1 is suitable where burp2 may not be.

* Reduce the number of blocks that the client holds in memory at a
  time.  Make sure this doesn't affect speed. Start by halving the
  number.

* Get a lock before writing to a data file.

* Get a lock before regenerating a sparse index.

* Review how the sparse indexes are generated. Probably too many of them
  are made at the moment.

* Sorting the hooks at the end of a backup probably uses too much memory.

* Separate phase1/phase2 again.

* Fix counters (needs phase1/2)

* Make recovery from partially complete backups work (needs phase1/2).
  Hooks need to be generated from already transferred manifests. Need
  to forward through already written 'changed/unchanged' manifest.

* Make include/exclude/include work again (needs phase1/2 to make it
  easy).  By this, I mean including something inside a directory that is
  excluded.

* Don't store fingerprints and md5sums as strings.

* Make verify work.

* Make CMD_INTERRUPT work (on restore, maybe others too).

* Make it possible to delete unused data files.
```

* Make the status monitor work.

* Make the status monitor use sdirs.

* Add data compression.

* Add data encryption.

* Make acl/xattrs work as far as burp1 does.

* Make Windows EFS work.

* Make Windows 'image' backups work.

* Make asyncio use iobuf for everything instead of just being a wrapper
  around the old stuff.

* Make src/client/list.c use sbuf.

* champ_chooser: figure out a way of giving preference to newer
  candidates.

* Fix conf/cntr init problem.

* Check notifications work (warnings are turned off because of conf/cntr
  init problem.