EXHIBIT W

Case No. 1:14-cv-00857-TSC-DAR

THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-96-05

The rsync algorithm

Andrew Tridgell and Paul Mackerras

June 1996

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-96-04 Peter Bailey and David Hawking. *A parallel architecture for query processing over a terabyte of text*. June 1996.

TR-CS-96-03 Brendan D. McKay. `autoson` *– a distributed batch system for unix workstation networks (version 1.3)*. March 1996.

TR-CS-96-02 Richard P. Brent. *Factorization of the tenth and eleventh Fermat numbers*. February 1996.

TR-CS-96-01 Weifa Liang and Richard P. Brent. *Constructing the spanners of graphs in parallel*. January 1996.

TR-CS-95-08 David Hawking. *The design and implementation of a parallel document retrieval engine*. December 1995.

TR-CS-95-07 Raymond H. Chan and Michael K. Ng. *Conjugate gradient methods for Toeplitz systems*. September 1995.

The rsync algorithm

Andrew Tridgell Paul Mackerras
Department of Computer Science
Australian National University
Canberra, ACT 0200, Australia

June 18, 1996

Abstract

This report presents an algorithm for updating a file on one machine to be identical to a file on another machine. We assume that the two machines are connected by a low-bandwidth high-latency bi-directional communications link. The algorithm identifies parts of the source file which are identical to some part of the destination file, and only sends those parts which cannot be matched in this way. Effectively, the algorithm computes a set of differences without having both files on the same machine. The algorithm works best when the files are similar, but will also function correctly and reasonably efficiently when the files are quite different.

1 The problem

Imagine you have two files, A and B, and you wish to update B to be the same as A. The obvious method is to copy A onto B.

Now imagine that the two files are on machines connected by a slow communications link, for example a dial up IP link. If A is large, copying A onto B will be slow. To make it faster you could compress A before sending it, but that will usually only gain a factor of 2 to 4.

Now assume that A and B are quite similar, perhaps both derived from the same original file. To really speed things up you would need to take advantage of this similarity. A common method is to send just the differences between A and B down the link and then use this list of differences to reconstruct the file.

The problem is that the normal methods for creating a set of differences between two files rely on being able to read both files. Thus they require that both files are available beforehand at one end of the link. If they are not both available on the same machine, these algorithms cannot be used (once you had copied the file over, you wouldn't need the differences). This is the problem that rsync addresses.

The rsync algorithm efficiently computes which parts of a source file match some part of an existing destination file. These parts need not be sent across the link; all that is needed is a reference to the part of the destination file. Only parts of the source file which are not matched in this way need to be sent verbatim. The receiver can then construct a copy of the source file using the references to parts of the existing destination file and the verbatim material.

Trivially, the data sent to the receiver can be compressed using any of a range of common compression algorithms, for further speed improvements.

2 The rsync algorithm

Suppose we have two general purpose computers α and β. Computer α has access to a file A and β has access to file B, where A and B are "similar". There is a slow communications link between α and β.

The rsync algorithm consists of the following steps:

1. β splits the file B into a series of non-overlapping fixed-sized blocks of size S bytes[1]. The last block may be shorter than S bytes.

2. For each of these blocks β calculates two checksums: a weak "rolling" 32-bit checksum (described below) and a strong 128-bit MD4 checksum.

3. β sends these checksums to α.

4. α searches through A to find all blocks of length S bytes (at any offset, not just multiples of S) that have the same weak and strong checksum as one of the blocks of B. This can be done in a single pass very quickly using a special property of the rolling checksum described below.

5. α sends β a sequence of instructions for constructing a copy of A. Each instruction is either a reference to a block of B, or literal data. Literal data is sent only for those sections of A which did not match any of the blocks of B.

The end result is that β gets a copy of A, but only the pieces of A that are not found in B (plus a small amount of data for checksums and block indexes) are sent over the link. The algorithm also only requires one round trip, which minimises the impact of the link latency.

The most important details of the algorithm are the rolling checksum and the associated multi-alternate search mechanism which allows the all-offsets checksum search to proceed very quickly. These will be discussed in greater detail below.

3 Rolling checksum

The weak rolling checksum used in the rsync algorithm needs to have the property that it is very cheap to calculate the checksum of a buffer $X_2..X_{n+1}$ given the checksum of buffer $X_1..X_n$ and the values of the bytes X_1 and X_{n+1}.

The weak checksum algorithm we used in our implementation was inspired by Mark Adler's adler-32 checksum. Our checksum is defined by

$$a(k,l) = (\sum_{i=k}^{l} X_i) \bmod M$$

[1] We have found that values of S between 500 and 1000 are quite good for most purposes

$$b(k,l) = (\sum_{i=k}^{l}(l-i+1)X_i) \bmod M$$

$$s(k,l) = a(k,l) + 2^{16}b(k,l)$$

where $s(k,l)$ is the rolling checksum of the bytes $X_k \ldots X_l$. For simplicity and speed, we use $M = 2^{16}$.

The important property of this checksum is that successive values can be computed very efficiently using the recurrence relations

$$a(k+1,l+1) = (a(k,l) - X_k + X_{l+1}) \bmod M$$

$$b(k+1,l+1) = (b(k,l) - (l-k+1)X_k + a(k+1,l+1)) \bmod M$$

Thus the checksum can be calculated for blocks of length S at all possible offsets within a file in a “rolling” fashion, with very little computation at each point.

Despite its simplicity, this checksum was found to be quite adequate as a first level check for a match of two file blocks. We have found in practice that the probability of this checksum matching when the blocks are not equal is quite low. This is important because the much more expensive strong checksum must be calculated for each block where the weak checksum matches.

4 Checksum searching

Once α has received the list of checksums of the blocks of B, it must search A for any blocks at any offset that match the checksum of some block of B. The basic strategy is to compute the 32-bit rolling checksum for a block of length S starting at each byte of A in turn, and for each checksum, search the list for a match. To do this our implementation uses a simple 3 level searching scheme.

The first level uses a 16-bit hash of the 32-bit rolling checksum and a 2^{16} entry hash table. The list of checksum values (i.e., the checksums from the blocks of B) is sorted according to the 16-bit hash of the 32-bit rolling checksum. Each entry in the hash table points to the first element of the list for that hash value, or contains a null value if no element of the list has that hash value.

At each offset in the file the 32-bit rolling checksum and its 16-bit hash are calculated. If the hash table entry for that hash value is not a null value, the second level check is invoked.

The second level check involves scanning the sorted checksum list starting with the entry pointed to by the hash table entry, looking for an entry whose 32-bit rolling checksum matches the current value. The scan terminates when it reaches an entry whose 16-bit hash differs. If this search finds a match, the third level check is invoked.

The third level check involves calculating the strong checksum for the current offset in the file and comparing it with the strong checksum value in the current list entry. If the two strong checksums match, we assume that we have found a block of A which matches a block of B. In fact the blocks could be different, but the probability of this is microscopic, and in practice this is a reasonable assumption.

When a match is found, α sends β the data in A between the current file offset and the end of the previous match, followed by the index of the block in

B that matched. This data is sent immediately a match is found, which allows us to overlap the communication with further computation.

If no match is found at a given offset in the file, the rolling checksum is updated to the next offset and the search proceeds. If a match is found, the search is restarted at the end of the matched block. This strategy saves a considerable amount of computation for the common case where the two files are nearly identical. In addition, it would be a simple matter to encode the block indexes as runs, for the common case where a portion of A matches a series of blocks of B in order.

5 Pipelining

The above sections describe the process for constructing a copy of one file on a remote system. If we have a several files to copy, we can gain a considerable latency advantage by pipelining the process.

This involves β initiating two independent processes. One of the processes generates and sends the checksums to α while the other receives the difference information from α and reconstructs the files.

If the communications link is buffered then these two processes can proceed independently and the link should be kept fully utilised in both directions for most of the time.

6 Results

To test the algorithm, tar files were created of the Linux kernel sources for two versions of the kernel. The two kernel versions were 1.99.10 and 2.0.0. These tar files are approximately 24MB in size and are separated by 5 released patch levels.

Out of the 2441 files in the 1.99.10 release, 291 files had changed in the 2.0.0 release, 19 files had been removed and 25 files had been added.

A “diff” of the two tar files using the standard GNU diff utility produced over 32 thousand lines of output totalling 2.1 MB.

The following table shows the results for rsync between the two files with a varying block size.[2]

block size	**matches**	**tag hits**	**false alarms**	**data**	**written**	**read**
300	64247	3817434	948	5312200	5629158	1632284
500	46989	620013	64	1091900	1283906	979384
700	33255	571970	22	1307800	1444346	699564
900	25686	525058	24	1469500	1575438	544124
1100	20848	496844	21	1654500	1740838	445204

In each case, the CPU time taken was less than the time it takes to run “diff” on the two files.[3]

[2] All the tests in this section were carried out using rsync version 0.5

[3] The wall clock time was approximately 2 minutes per run on a 50 MHz SPARC 10 running SunOS, using rsh over loopback for communication. GNU diff took about 4 minutes.

The columns in the table are as follows:

block size The size in bytes of the checksummed blocks.

matches The number of times a block of B was found in A.

tag hits The number of times the 16 bit hash of the rolling checksum matched a hash of one of the checksums from B.

false alarms The number of times the 32 bit rolling checksum matched but the strong checksum didn't.

data The amount of file data transferred verbatim, in bytes.

written The total number of bytes written by α including protocol overheads. This is almost all file data.

read The total number of bytes read by α including protocol overheads. This is almost all checksum information.

The results demonstrate that for block sizes above 300 bytes, only a small fraction (around 5%) of the file was transferred. The amount transferred was also considerably less than the size of the diff file that would have been transferred if the diff/patch method of updating a remote file was used.

The checksums themselves took up a considerable amount of space, although much less than the size of the data transferred in each case. Each pair of checksums consumes 20 bytes: 4 bytes for the rolling checksum plus 16 bytes for the 128-bit MD4 checksum.

The number of false alarms was less than 1/1000 of the number of true matches, indicating that the 32 bit rolling checksum is quite good at screening out false matches.

The number of tag hits indicates that the second level of the checksum search algorithm was invoked about once every 50 characters. This is quite high because the total number of blocks in the file is a large fraction of the size of the tag hash table. For smaller files we would expect the tag hit rate to be much closer to the number of matches. For extremely large files, we should probably increase the size of the hash table.

The next table shows similar results for a much smaller set of files. In this case the files were not packed into a tar file first. Rather, rsync was invoked with an option to recursively descend the directory tree. The files used were from two source releases of another software package called Samba. The total source code size is 1.7 MB and the diff between the two releases is 4155 lines long totalling 120 kB.

block size	**matches**	**tag hits**	**false alarms**	**data**	**written**	**read**
300	3727	3899	0	129775	153999	83948
500	2158	2325	0	171574	189330	50908
700	1517	1649	0	195024	210144	36828
900	1156	1281	0	222847	236471	29048
1100	921	1049	0	250073	262725	23988

7 Availability

An implementation of rsync which provides a convenient interface similar to the common UNIX command rcp has been written and is available for download from ftp://samba.anu.edu.au/pub/rsync.