Multiround Rsync

John Langford

January 31, 2001

Abstract

I present an optimization of the Rsync algorithm[1][2] for file transfer which trades computational performance for network performance in the low-alteration limit. The amount of improvement is application dependent although a worst case analysis shows that $O(\sqrt{n})$ network utilization can be reduced to $O(\log n)$ network utilization.

1 Introduction

Transferring bytes is a fundamental building block of network communication. The rsync algorithm is a technique for optimizing file transfer in the special case where a similar version of the file happens to exist at the destination. This restricted domain - transfer of bytes when the destination already has a similar copy is still large enough to be quite interesting. Examples include distributing source code and reloading web pages.

In the limit of no alterations between the old file and the new file, the rsync algorithm transfers only a small fraction of the file size (in bytes) across the network. This behaviour degrades robustly against any many forms of small alterations - mutation, deletion, and insertion of bytes into the file. However, 'small' here is not necessarily good enough when working with large files. The multiround rsync algorithm improves on the value of 'small' from (essentially) $\sqrt{n}$ to $\log n$. When synchronizing an 80 gigabyte system image this implies a ratio of nearly 10000 between the network utilization of rsync and multiround rsync.

The layout of the paper is as follows:

1. The rsync algorithm and an analysis of its strengths and weaknesses
2. The multiround rsync algorithm and its analysis.
3. Experiments
4. Discussion of various uses

2 The Rsync algorithm

Before presenting the rsync algorithm we will cover a few preliminaries for future reference. There are two fundamental parameters of the rsync problem:

- o - the size of the old file
- n - the size of the file we want to copy

In addition, there is one fundamental parameter of the rysnc algorithm.

- k - the block size

For convenience, we will assume that all filesizes are a multiple of the block size. This assumption is not fundamental to the implementation of the algorithm, nor does alter the analysis significantly.

2.1 Algorithm

The rsync algorithm works in three phase. The first phase works on the destination host as follows:

1. On the destination host, the old version of a file is divided into blocks of length k.
2. For each block, calculate a weak and strong checksum. The reason for the weak checksum will be clear in the next phase.
3. send the weak and strong checksums to the source machine.

The network utilization require for the first phase is $\Theta(o/k)$. In practice, the constant varies from 2 to 20 bytes depending on the implementation.

The second phase works on the source host. It is the most computationally demanding phase and can be thought of as calculating a compressed form of the source file using references to blocks in the old file.

1. Recieve the weak and strong checksums from old file blocks.
2. Insert all the pairs into a hashtable indexed by the weak checksums.
3. Match as many parts of the source file as possible. This is done efficiently by
   (a) checking if a block matches by
      i. calculating the weak checksum
      ii. If there is a hit in the hash table
         A. calculate a strong checksum
   (b) If there is a hit record it and goto (a) for the next block

(c) otherwise there is a miss so:

   i. calculate the weak checksum for the block of size k which is 1 byte advanced from the previous block (this is quickly computable given the previous checksum)

   ii. Check if the block matches

      A. If there is a hit, record the bytes which were unmatched, the hit, and go to (a) for th next block

      B. otherwise there is a miss so go to i.

4. Transfer the sequence hits and missed strings to the destination

The network utilization of the second phase is dependent upon the amount and nature of the differences between the old file and the new file so a precise estimate is problem dependent. The analysis later will show that if c bytes change, then the network utilization of the second phase is no worse then $O(ck)$.

The third phase simply reconstructs the source file using the sequence of hits (from the old file blocks) and missed strings calculated in the second phase.

1. Recieve the sequence of hits and missed strings.

2. Reconstruct the source by:

   (a) For every hit, copy the block from the old file.

   (b) For every missed string, copy the string directly into the old file.

2.2 Analysis

We do a worst case and best case analysis of the rsync algorithm because it is difficult to model the average case. In both the worst and the best case, we will assume that the files differ by a parameter c which describes the number of inserted, deleted, or altered bytes. Under this assumption, the difference in size between the old file and the new file must be no larger then c.

2.2.1 Best Case

In the best case all the differences between the old file and the new file are done in a maximally local manner resulting in deletion, alteration, or insertion of c consecutive bytes. In this setting, all but approximately c/k blocks will match blocks in the old file. Transmitting the matches takes $O(\ln(o/k))$ network utilization which is negligible in comparison to the first phase network use. Therefore, network utilization is essentially $O(\text{first round } + \text{second round}) = O(o/k + \max(c,k))$. Optimizing the choice of k, we get $k = \sqrt{o}$ for $c \leq \sqrt{o}$ and network utilization of $O(\sqrt{o})$ for the optimal k.

2.2.2 Worst case

In the worst case all differences between the old file and the new file are as unlocal as possible. We can model this by assuming that the alterations occur every k bytes. For the worst case, we will have $\Theta(c)$ unmatched blocks resulting in network utilization of $O(\frac{o}{k} + kc)$. Assuming we knew that the there would be c unmatched blocks before starting rsync, we could optimize k:

$$c = \frac{o}{k^2}$$

$$\Rightarrow k = \sqrt{\frac{o}{c}}$$

The optimal k thus behaves something like the square root of the file size which implies that we transfer $O(\sqrt{oc})$ bytes. This is good, but as mentioned earlier it is not good enough for gigabyte size files.

3 Multiround Rsync

The multiround rsync algorithm is a refinement of the rsync algorithm which reduces the $\sqrt{o}$ terms above to $\ln o$. Multiround rsync has 3 parameters:

- The starting block size s
- The minimum block size m
- The base b

3.1 The algorithm

The multiround rsync algorithm consists of 3 phases which are cycled through up to $\ln n$ times. Initially, we have:

- block size = starting block size
- old holes = the entire old file
- source holes = the entire source file

The first phase works exactly the same as for the rsync first phase except that we only block and checksum bytes which are within the current 'old holes'.

1. On the destination host, every hole is divided into blocks of length k.
2. For each block, calculate a weak and strong checksum.
3. send the weak and strong checksums to the source machine.

The network utilization require for the first phase is $\Theta(l/k)$ where l is the number of bytes in the holes and we assume every hole is larger then k.

The second phase works on the source host. It differs from the second phase in the rsync algorithm by only operating on the current set of source holes and by only sending the size and location of the missed portions of holes.

1. Recieve the weak and strong checksums from old file hole blocks.
2. Insert all the pairs into a hashtable indexed by the weak checksums.
3. Match as many parts of the source file holes as possible. This is done efficiently in the same manner as for rsync.
4. If the block size is the minimum block size then transfer the sequence of hits and mssed strings to the destination.
5. Otherwise, transfer the sequence of hits and misses to the destination.

The third phase is again similar to the rsync algorithm except that we must deal with misses as well as hits and strings.

1. Recieve the sequence of hits, misses, and strings
2. Reconstruct the source by moving to each source hole in turn and:
   (a) For every hit, copy the block from the old file.
   (b) For every miss, copy a null block the size of the miss.
   (c) For every missed string, copy the string directly into the old file.

At this point, we have completed one round of the multiround protocol. The next round is initiated with the blocksize divided by the base ($k_{next} = k_{last}/b$). The set of source holes is assigned the set of misses and the set of old holes becomes the set of blocks which were not used by hits in the third phase.

The multiround rsync protocol terminates when no source holes exist.

3.2 Analysis

3.2.1 Network

The best case for the multiround rsync does not differ significantly from the best case for rsync. However the *worst case* performance changes dramatically. In the worst case, we have c bytes altered resulting in c unmatched blocks for the rsync algorithm. Optimizing the block size k in the rsync algorithm resulted in $O(\sqrt{oc})$ network utilization. The multiround rsync protocol has more parameters, but we will not optimize them at all. Instead, we will set the minimum block size to $m = 40$ bytes based upon the idea that the minimum blocksize should be somewhat larger then checksum information. The starting blocksize will be set to the number of bytes in the source file $s = o$[1].

The worst case analysis is straightforward now. Every altered byte will cause at most one unique miss in each round of the protocol. Since misses become the holes of the next round, each round of the protocol will send at most $O(bc)$ bytes from the destination to the source. The number of bytes sent from the source to the destination with run length compression will be $O(c)$ bytes for every round except the last. In the last round, the number of bytes sent will be $\Theta(mc)$ since each corrupted byte will cause a miss in a block of the minimum block size. The number of rounds is approximately $r = \log_b \frac{o}{m}$. Pulling all these details together we get a total network utilization of $O(bcr + mc) = O(c(b\log_b \frac{o}{m} + m))$. Since m is similar to the checksum size, the worst case simplifies to $O(c \ln o)$. Since the rsync algorithm is a specialization of the multiround rsync algorithm with $k = s = m$ the worst case analysis for the multiround rsync algorithm is never worse then for the rsync algorithm.

The effect of the parameters can be summarized as follows:

1. b: Increasing the base leaves has unpredictable effects on the traffic from the destination since it depends upon the granularity of alterations.
2. m: Increasing the minimum block size increases traffic from the source and decreases traffic from the destination.
3. s: decreasing the starting block size results in increased traffic from the destination to the source.

3.2.2 Computation

In general, the decreased network utilization of the multiround rsync algorithm is traded for increased computation.

In the worst case (no hits), the amount of computation is multiplied by the number of rounds $r = \log_b \frac{s}{m}$ on the destination side because every part of the file must have a checksum calculated for every blocksize. This computation could be conceivably be reduced by using a checksum with an additive property. Let c_1 and c_2 be the checksums of blocks 1 and 2. Then it would be desirable for a fast way to calculate the checksum of the concatenated block, $c_1 c_2$. This optimization has not yet been investigated, but it could result in reducing the computation on the destination side to something essentially equivalent to the rsync algorithm.

On the source side, the quantity of computation will again be multiplied by $r = \log_b \frac{s}{m}$ and the trick suggested above could not feasibly be used to reduce the quantity of computation.

In the best case (few hits), the amount of extra computation is essentially negligible because only the bytes near the alterations are included in the holes

[1]This is chosen for analysis simplicity. In practice, it is desirable to use $s = min(n, o)$ or smaller.

of each round. It is important to notice that the best case computationally is the same as the best case for network utilization.

4 Optimization

4.1 Network utilization optimizations

The original rsync algorithm includes significant extra optimizations including:

1. A specialized compression and decompression algorithm for source to destination traffic.
2. Safe reduced signature sizes. Send only some of the 20 checksum bytes per block and make sure the resulting file is reconstructed correctly by sending a strong signature for the entire source file to the destination. Rerun with a larger signature if this fails.
3. Generic compression.

The improvement from these optimizations does not affect the worst case analysis except through constant reduction. In particular, (1) reduces source to destination traffic to be essentially only the missed strings and (2) reduces the signature size to 4 or 5 bytes (rather then 20) in the typical case. (3) often results in reduced network utilization on the source to destination direction.

All of these optimizations were also done for the multiround rsync implementation. Optimization (1) is very similar to the rsync algorithm. It uses run length encoding on the difference between block numbers and bit packing techniques. The language used for network transport contains one other optimization. Instead of sending "Hit for block i", hit runs of the form: "Hit for block i through j" is sent.

Optimization (2) is done in a somewhat more sophisticated manner then the original rsync. There are two differences:

1. A strong signature is sent at the end of each round in the multiround protocol. The signature is a checksum of all the matched blocks in that round. The signature is checked on the destination side and if it fails to match, the round is rerun with an additional byte sent for every block signature.
2. The source side is capable of predicting the number of signature bits required using the approximate formula:

   $$\Pr(\text{false hit}) \simeq \frac{cm}{2^b}$$

   where b is the number of signature bits, c is the number of block signatures, and m is the number of lookups in the hash table. The cost of rerunning a round is approximately just the number of bytes that would be resent which implies that the source side should request more signature bits if:

$$\frac{cm}{2^b} \geq \frac{1}{k}$$

where k is the number of bytes sent in this round. In practice, this formula worked well.

Optimization (3) was implemented in a less sophisticated manner then the original rsync algorithm - we simply use generic compression in the transport algorithm (ssh).

4.2 Time optimizations

Pipelining, Multiscale

4.3 Parameter optimization

The multiround rsync algorithm has significantly more parameters then the rsync algorithm. In general extra parameters are not good for useability because they imply more effort may be required for tuning. There are two approaches to counter this.

First, we can show that there exists some reasonable choice for the parameters and simply stick with that choice. This appears to be the case with the maximum block size. The maximum block size should be a multiple of the base times the minimum block size ($s = b^n m$) in order to avoid the inefficiencies inherent in disretization. The exact value of the maximum block size is an implementation issue. We chose $s < 2^{22}$ bytes.

The second technique is optimizing away the parameter. In our particular case, experiments showed that a base of 3 seemed near optimal. Optimizing the minimum block size was trickier however because the "right" minimum varied significantly depending on the dataset under consideration. However, as will be shown below, the benefit (in terms of network utilization) of decreasing the block size turns out to be fairly easy to estimate. In particular, the relation appears to be near linear. Motivated by this, we start with a minimum block size of 20 bytes and then predict the benefit of an additional round of rsync according to:

$$\text{(matched blocks)} * \text{blocksize}/3$$

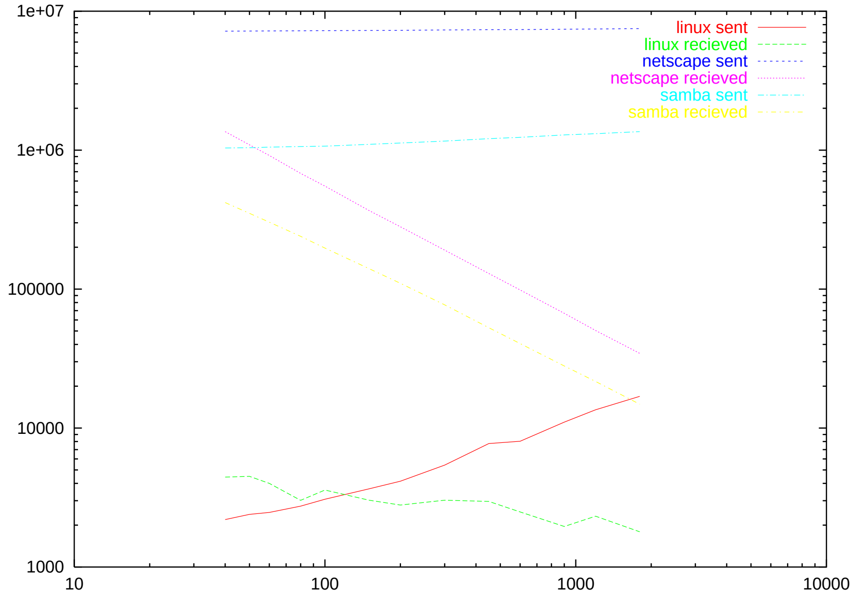
The division by 3 is a heuristic that models the benefits of generic compression and the fact that not every new block will be a match. This expected benefit is compared to the cost of sending the signatures and the round proceeds if the expected benefit is greater or the cost is trivial (1/100 or less) in comparison to transferring the unmatched blocks.

5 Experiments

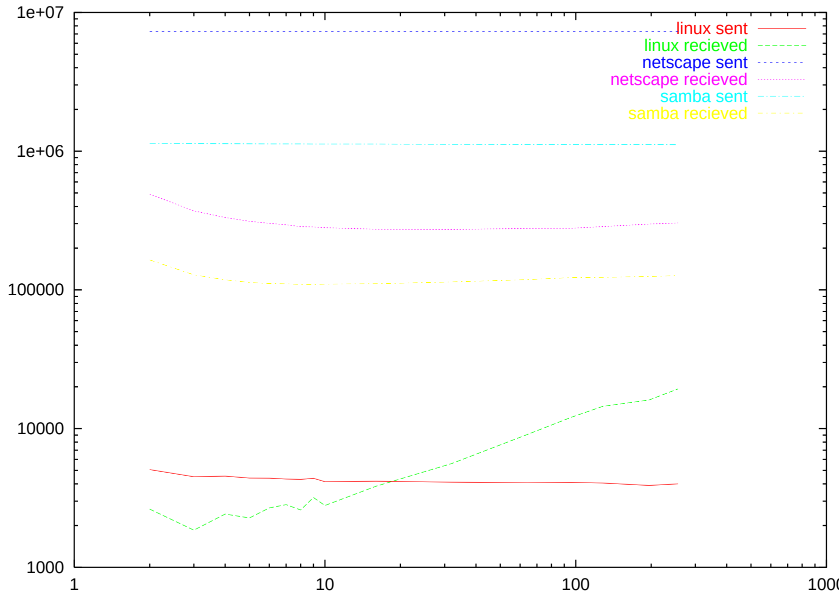
The experiments are run using the rsync dataset from ftp://samba.org/pub/tridge/rsync_data. The dataset consists of 3 datasets for exchanges:

1. The source of linux kernel 2.0.9 and 2.0.10
2. The source code of samba 1.9.18p10 and 2.0.0beta1
3. The binary of netscape communicator 4.06 and netscape communicator 4.07

For each of these datasets we vary the base and the minimum block size resulting in two plots.



This plot varies the minimum block size vs. the number of bytes sent and recieved. The base was set to 4 and the maximum block size to 4 MB. the most strongly noticeable feature in this graph is the near linearity which motivates our optimization heuristic.



This plot varies the base vs. the number of bytes sent and recieved. The minimum block size was set to 200 bytes and the maximum block size to 4 MB. A base of 3 performs best on the 'linux' dataset but poorly on the less matcheable 'netscape' and 'samba' datasets due to signature overhead.

It is interesting to compare the results from multiround rsync with the results from rsync. This can be done in a pessimistic way by choosing the optimal (minimal number of bytes sent and recieved) setting for blocksize in a single round rsync for each dataset. We then compare the results with multiround rsync with only default parameters. A pessimistic comparison is made because multiround rsync has more tuneable parameters and so it is not fair to compare a fully tuned multiround rsync to a fully tuned rsync.

In the following table R abbreviates Rsync and MR abbreviates Multiround Rsync.

dataset	block size	R sent	R rec.	MR sent	MR rec.	R total	MR total
linux	4500	27128	30500	2490	3183	57628	5673
netscape	1000	7419783	107652	7374410	47131	7527435	7421541
samba	300	1171234	118326	1132736	141084	1289560	1273820

From this table we can conclude that a default multiround rsync empirically never requires more network utilization then an optimized rsync and is sometimes much superior ($\times 10$ lower network traffic for the linux dataset). It is also interesting to note that the total network utilization by multiround rsync is comparable to the size of a patch file.

dataset	MR total	diff	diff + gzip
linux	5673	4566	1921
samba	1273820	15532295	3764962

The diff program is not optimized for compresssion but it is certainly encouraging that we observer performance on the same order of magnitude.

6 Conclusion

I presented the multiround rsync algorithm and showed that it can significantly improve network utilization with respect to the rsync algorithm when the number of changes is small. This improvement was first shown by comparing a worst case analysis and then empirically on the linux data set. The multiround rsync algorithm *is* rsync for a specific degenerate setting of the parameters so when doing a full optimization multiround rsync always does better then rsync. However, in standard use we do not have the luxury of first optimizing the parameters and so it was also shown then multiround rsync with reasonable parameter settings does much better ($\times 11$) to slightly worse ($\times 1.1$) then an optimized rsync in terms of network utilization.

There are two approaches for future work.

1. The multiround rsync algorithm seems like a worthwhile improvement to rsync (the program) so it would be interesting to integrate this into the program available at http://rsync.samba.org

2. It may be possible to reduce the computational burden on the destination side of running the multiround rsync by using a multiscale checksum. This requires investigation

References

[1] Andrew Tridgell and Paul Mackerras, “The Rsync Algorithm”, technical report

[2] Andrew Tridgell, “Efficient Algorithms for Sorting and Synchronization”, thesis