zsync — Optimised rsync over HTTP

Colin Phipps

<cph@moria.org.uk>

(work in progress)

Abstract

This document describes the thinking behind zsync, a new file transfer program which implements efficient download of only the content of a file which is not already known to the receiver. zsync uses the rsync algorithm, but implemented on the client side, so that only one-off pre-calculations are required on the server, and no special server software or new protocol is required to use zsync.

Table of Contents

1. The Problem

- File Transfer
- Existing Methods for Partial File Transfer
- Compressed Files
- The Ideal Solution

2. zsync Theory

- Rsync on the Client Side
- The zsync Control File
- Checksum Transmission
  - Weak Checksum
  - Strong Checksum
- Match Continuation
- rsync Speed
  - Negative Hash Table
- Networking
- Comparison with rsync
- Empirical Results

3. Compressed Content

- Looking Inside
- Mapping Deflated Files
- Is It Worthwhile?
- Compressed Delta Transfer
- Compression Results

Recompression

4. Implementation

libzsync
Control File Generation
zsync Client
Security Considerations
Work To Do

Bibliography

Chapter 1. The Problem

Table of Contents

File Transfer
Existing Methods for Partial File Transfer
Compressed Files
The Ideal Solution

File Transfer

A large amount of the traffic on the Internet today consists of file downloads of one kind or another. The rapid growth in the size of hard drives, and the wide spread of first CDs and now DVDs for distributing files in hard form, has led to a rise in the size of files generally. While one result of the tech boom has been to leave us with plentiful and cheap bandwidth available to most people, the inexorable rise in file sizes means that there is always potential in technology that reduces the time taken to transfer data over the network.

In the days of modems, anything to reduce the volume of data being transferred was gratefully received. The rise in ADSL, cable modems and other broadband Internet connections has temporarily relieved the problem. But it has also raised expectations about download times — where I was happy for the latest security update to take an hour to download over a modem, I now begrudge the few minutes taken for the same task on a broadband connection.

Other things being equal, there will always be advantages in reducing the total amount of data that must be transferred:

- Reduces the time taken for the transfer to complete.
- Reduces the total data transferred — important if there are fixed data transfer limits (as with many hosting packages) or costs per byte downloaded.
- Reduces contention for network bandwidth, freeing up network capacity at both ends for other tasks.

There is a significant category of file downloads where it would seem that the volume of data moved over the network could be reduced: where the downloading machine already has some of the data. So we have technologies like download resuming for FTP, and Range support in HTTP, which allow partial file content to be transferred. These are only effective when we know precisely which content we already have, and (hence) which parts we still need to download.

There are many circumstances where we have partial data from a file that we intend to download, but do not necessarily know what. Anywhere where a large data file is regenerated regularly, there may be large parts of the content which are unchanged, or merely moved around to accommodate other data. For instance, new Linux kernel source are made regularly; changes are scattered widely over a large number of files inside the archive, but between any two given releases the total amount of changes is tiny compared to a full download. But because the changed sections and the unchanged are intermixed, a downloader will not be able to selectively download the new content.

Existing Methods for Partial File Transfer

HTTP already provides the Range header for transferring partial content of files. This is useful only if you are able to determine from some other source of information which are the changed sections. If you know that a file is a log and will only ever grow — existing content will not change — then Range is an effective tool. But it does not solve the problem by itself.

There are alternative download technologies like BitTorrent, which break up the desired file into blocks, and retrieve these blocks from a range of sources [BitT2003]. As BitTorrent provides checksums on fragments of file content, these could be used to identify content that is already known to the client (and it is used for this, to resume partial downloads, I believe). But reusing data from older files is not a purpose of this data in BitTorrent — only if exactly matching blocks could be identified would the data be any use.

The best existing solution from the point of view of minimising data transfer is rsync. rsync uses a rolling checksum algorithm that allows the checksum over a given block length at all points in a file to be calculated efficiently. Generally speaking, a checksum would have to be run at every possible start point to achieve this — the algorithm used in rsync (see [Rsync1998]) allows the checksum window to be rolled forward over the file and the checksum for each new location to be trivially derived from the previous checksum and the values at the window edges. So rsync can calculate the checksum at all points in the input file by streaming through the file data just once. While doing so it compares each calculated checksum against the list of checksums for the existing data file, and spots any chunks from the old data file which can be reused.

So rsync achieves a high level of data reuse. It comes at a high computational cost, however. The current rsync implementation calculates the checksums for a set of blocks on the client, then uploads these to the server; the server them uses the rsync algorithm to work out which blocks the client has and which it needs, and pushes back the blocks it needs. But this approach suffers many drawbacks:

- The server must reparse the data each time. It cannot save the computed checksums. This is because the client sends just the checksums for disjoint blocks of data from its pool of known data. The server must calculate the checksum at all offsets, not just at the block boundaries. The client cannot send the checksum at all points, because this would be four times larger than the data file itself — and the server does not want to pre-compute the checksums at all points, because again it would be four times larger, and require four times as much disk activity, as reading the original data file. So CPU requirements on the server are high. Also the server must read the entire file, even if the final answer is that the client requires only a small fragment updated.

- Memory requirements for the server are high - it must store a hash table or equivalent structure of all the checksums received from the client while parsing its own data.

- The server must receive and act on a large volume of data from the client, storing it in memory, parsing data, etc — so there is the opportunity for denial of service attacks and security holes. In practice rsync has had a remarkably good security record: there have been a few vulnerabilities in the past few years (although at least one of these was actually a zlib bug, if I remember rightly).

The drawbacks with rsync have prevented it being deployed widely to distribute files to the general public. Instead, it has been used in areas closer to the existing use of cvs and sup, where a limited community of users use an rsync server to pull daily software snapshots. rsync is also very widely used inside organisations for efficient transfer of files between private systems, using rcp or scp as a tunnel. rsync also has very powerful functionality parallelling **cp -a** and tar's abilities, with transfers of file permissions, directory trees, special files, etc. But public releases are rarely made with rsync, as far as I can tell.

I should also mention rproxy. While I have not used it myself, it is an attempt to integrate the rsync algorithm into the HTTP protocol [RProxy]. An rproxy-enabled client transmits the rsync checksums of blocks of data it already has to the server as part of the HTTP request; the server calculates the rolling checksum over the page it would have transmitted, and transmits only the blocks and the meta-information needed for the client to construct the full page. It has the advantage of integrating with the existing protocol and working even for dynamic pages. But it will, I suppose, suffer the same disk and CPU load problems as rsync on large files, and is an unwelcome overhead on the server even for small files. Since server administrators are rarely as concerned about bandwidth and download time as the client, it is hard to see them wanting to put extra work on their servers by offering either rsync or rproxy generally.

Finally, there are the mechanisms traditionally used among programming projects — version control and diffs. The Linux kernel, for instance, is distributed by providing patches to get from one version to the next ([LKML FAQ]). For comparison with the other methods discussed, we can say that this method effectively pre-computes the changes between versions and then sends only the changes to the client. But it only works with a given *fixed* starting point. So to get from, say, 2.4.19 to 2.4.27, the user has to download the patch 2.4.19 -> 2.4.20, the patch 2.4.20 -> 2.4.21, and so on. This method is efficient if there are clear releases and the frequency of releases is smaller than the frequency with which users check for updates — it is less efficient when releases in the affected files are frequent, as there are then large numbers of patch files to manage and download (and these files contain enough data to construct not only the final file, but every intermediate revision).

CVS and subversion provide a specialised server programs and protocols for calculating diffs on a per-client basis. They have the advantage of efficiency once again, by constructing exactly the diff the client needs — but lose on complexity, because the server must calculate on a per-client basis, and the relatively complicated server processing client requests increases the risk of security vulnerabilities. CVS is also poor at handling binary data, although subversion does do better in this area. But one would hardly distribute ISO images over either of these systems.

Hybrid protocols have been designed, which incorporate ideas from several of the systems above. For instance, CVSup [CVSup1999] uses CVS and deltas for version-controlled files, and the rsync algorithm for files outside of version control. While it offers significantly better performance than either rsync or CVS, due to efficient pipelining of requests for multiple files, it does not fundamentally improve on either, so the discussion above — in particular the specialised server and high server processing cost per client — apply.

Compressed Files

There is another drawback to partial file downloading. Transferring partial content has some similarities to the compression problem, in that we must be able to spot data patterns that occur in both the target file and the existing copy known to the client. Perhaps as a consequence, it interacts very badly with files that are already compressed.

In a compressed data stream, the representation of any particular fragment data will vary according to the overall compression algorithm, how aggressively the file has been compressed, options used to the compression tool, and, most importantly, the surrounding file content. For instance, the zip compression used in common compression tools uses backreferences in the compressed data stream to avoid duplicating data. The Huffman codes chosen by the compressor to represent individual bytes in the uncompressed stream will vary depending on the frequency of that character in the surrounding block of data, as well as just according to the arbitrary choice of the compression utility. From the first point at which two files differ, their compressed versions may have no data in common at all. The output of a compression program is, roughly speaking, not possible to compress further, because all redundancy and structure from the original file is gone — precisely the structure that might have been useful for working out partial file transfers. For this reason, rsync is usually ineffective on compressed files.

There has been an attempt to address this problem — patches have been made available for gzip, for instance, to make the format more friendly to rsync [GzipRsync]. By forcing the compression program to start a new block of compressed data at certain intervals, particularly based on the content of the underlying data, it is possible to get compressed files which will get back into step after a difference in data, making rsync effective again. But even in the patched version of gzip this option is not a default: it makes the compression less efficient, for no benefit except to users of programs like rsync, which as already noted is not used for most file distribution. So the --rsync option is not widely used.

Of course, compression is the best solution to the problem when the client knows no data from the file. So people will still want to distribute compressed files. For files where the client knows nearly everything, with just very small changes to files, it is more efficient to access the uncompressed data and get only the blocks you need. There is a crossover somewhere in the middle. There is a crossover area where it is more efficient to transfer partial file content from the compressed data stream: if you have a long text file to which large new blocks of text are added daily, then is is certainly best to use rsync on the compressed file — rsync on the uncompressed file would waste less local data, but transferring the new data uncompressed would be inefficient (assuming rsync is being used over a data channel which is not itself doing compression).

The Ideal Solution

So what, ideally, would we want? A system requiring no special server support — preferably nothing more complex than, say, HTTP Range: support. We want no per-client calculations at the server end at all. Anything that is required at the server end should be pre-calculated. It should also address the problem of compressed files.

Chapter 2. zsync Theory

Table of Contents

Rsync on the Client Side
The zsync Control File
Checksum Transmission

- Weak Checksum
- Strong Checksum

Match Continuation
rsync Speed

- Negative Hash Table

Networking
Comparison with rsync
Empirical Results

Rsync on the Client Side

Essentially, we already have a solution — rsync. The problem is that rsync does the hard work on the server, and requires server support. This is not essential to its algorithm. The algorithm merely requires that one side calculates the checksums of each distinct block of data, and sends it to the other end; the other end then does a rolling checksum through its file, identifying blocks in common, and then working out which blocks are not in common and must be transmitted.

So, we make it the server which calculates the checksums of each *distinct* block. Because it need calculate only one checksum per block of data, and this is not specific to any given client, the data can be cached. We can save this data into a metafile, and the client requests this data as the first step of the process. This metafile can simply be at another URL on the same — or even a different — server.

The zsync client will pull this metafile. It then runs through the data it already has, applying the rsync rolling checksum and comparing with the downloaded checksum list. It thus identifies the data in the target file that it already has. It then requests the remaining data from the server. Since it knows which data it needs, it can simply use HTTP Range requests to pull the data.

The server has no per-client calculations. The metafile can be calculated in advance. No state is needed on the server. An ordinary web server, plus a program to generate the metafile (the zsync control file, from now on), provides everything we need.

The actual data in the control file will consist of the simple checksum (for comparison with the checksum produced by the rolling checksum method on the client) for each block, plus a strong checksum (currently MD4) used to eliminate false positive matches occurring with the simple checksum. I have simply followed rsync in this area; the rolling checksum is effective; the strong checksum is fairly arbitrary, provided it is resistant to collisions.

The zsync Control File

Apart from the checksums, what data should go into the control file? The blocksize must be transmitted, so that the client calculates the checksums on the same size of block. A fixed value could be hard-coded, but I prefer to keep it tunable until we can prove in common use that one value is always best. Andrew Tridgell's paper on rsync [Rsync1998] suggests that a value of around 500-700 bytes is optimal for source code (so perhaps textual data more generally); but for transmitting ISO images of Linux distributions, or other very large and often binary content, there is likely to be less movement of small blocks of data and more large blocks of either matching or non-matching data, where a larger blocksize to the algorithm is appropriate. For now it can be configurable.

The file length must be transmitted, so that we know the total number of blocks. Also, the final block of data will often extend past the end of the file, which will need to be padded when calculating checksums. So zsync must truncate the file once the block downloading is done.

The control file could include file permissions and other data, in a similar way to subversion's file properties. This is more important within organisations, and hence where the user often has logins on both machines. In this situation, there is little wrong with the existing solution of rsync. For situations where zsync is more useful, there is usually no trust between the distributor and the downloader, so permissions data is not useful. I have not attempted any features in this area.

The URL from which the unknown blocks are to be retrieved can also be part of the metafile. We could code in the assumption that the metafile is always alongside the normal content — but this would be an unnecessary restriction. By putting the URL inside the control file, we give the chance to host the control file outside of the normal directory tree, which will be convenient at this early stage of zsync's development.

The control file header will not exceed a few hundred bytes. The block checksum data will be some certain and fixed number of bytes per block in the file to be transferred; the precise content is discussed next.

Checksum Transmission

The main content of the control file will be the checksums for blocks of data in the file to be downloaded. What checksums should be used, and how should they be transmitted? The choice of the rsync algorithm gives us part of the answer: a weak checksum, which can be calculated in a rolling manner over the data already held by the client, must be transmitted, to allow the client to easily identify blocks. Then a stronger checksum, strong enough to prevent blocks being identified as in common with the target file incorrectly, must be used for verification.

Weak Checksum

rsync transmits a 4-byte weak checksum, and I have used the same formula, but we could shorten the amount of data, to reduce the size of the control file. For small files, 4 bytes might be more than is needed to efficiently reject false matches (see, for example, [CIS2004]).

There is a tradeoff here between the download time and the processing time on the client. The download time (for the checksums) is proportional to the amount of checksum data we transmit; part of the processing time on the client is proportional to the number of strong checksums that need to be calculated. The number of false matches on weak checksums (and hence unnecessary strong checksum calculations) are proportinal to the number of hashes calculated by the client (which is roughly the file size on the client) times the likelyhood of a false match. Adding this to the download time for the weak checksums, we have:

$$N\frac{N}{b}2^{-d}t_2+\frac{d}{8}\frac{N}{b}t_3$$

(where d is the number of bits of weak checksum transmitted per block, for a file of N bytes and b bytes per block, t_2 is the time to calculate a strong checksum, and t_3 is the download time for each weak checksum.) This is a simple minimisation problem, with minimum time given by:

$$-\log 2\, N\, 2^{-d}\, t_2 + \frac{t_3}{8} = 0$$

$$\Rightarrow d = \log_2 N + \log_2 \left(\frac{8 t_2 \log 2}{t_3} \right)$$

t_3 is easy to know for any given client, but of course varies between clients - a client on a modem will view download data as more expensive, and so prefer a shorter weak checksum and more CPU time used on the client instead. Similarly t_2 varies depending on the speed of the client computer; it also depends on the blocksize (OpenSSL on my computer can calculate the MD4 checksum of ~200MB per second - although this drops for very small blocksizes). The server will have to assume some reasonable default for both values. For 512kbps transfer speeds (standard ADSL) and a typical desktop computer able to calculate strong checksums at 200MB/s, we have (for large blocksizes, say >512 bytes):

$$d \approx \log_2 N + \log_2 b - 8.6$$

For the moment, I have chosen to send a whole number of bytes; and it is better to err on the side of sending too many weak checksum bits than too few, as the client performance degrades rapidly if the weak checksum is too weak. For example, on a 600MB ISO file, a 3-byte weak checksum causes the client to take an extra minute of CPU time (4 minutes, against 3 minutes when 4 bytes of weak checksum were provided) in a test conducted with zsync-0.2.0 (pre-release). In practice, this means that 4 bytes of weak checksum is optimal in most cases, but for some smaller files 3 bytes is better. It may be worth tweaking the parameters of this calculation in specific circumstances, such as when most clients are fast computers on very slow network connections (desktop computers on modems), although in these circumstances the figures here will still be small relative to the total transfer size.

Strong Checksum

The strong checksum is a different problem. It must be sufficiently strong such that, in combination with the weak checksum, there is no significant risk of a block being identified in common between the available local data and the target file when in practice the blocks differ. rsync uses an MD4 checksum of each block for this purpose.

I have continued to use MD4 for the moment. There are probably alternatives which would be more efficient with CPU time, but this is not a scarce quantity for zsync. What is of interest is the amount of data that must be transmitted to the client: a full MD4 checksum requires 16 bytes. Given a file of length N, blocksize b, there are `N/b` blocks in a file; and assume that the client also has, for simplicity, N bytes of potential local data (and so ~N possible blocks of local data). If there is no data in common, and k bits of checksum transmitted, and the checksums are uniformly and independently distributed, then the chance of no collisions (data incorrectly believed to be in common, and ignoring the weak checksum for now):

$$p \geq \frac{\left(2^k - \frac{N}{b}\right)^N}{\left(2^k\right)^N} = \left(1 - \frac{N}{b}\frac{1}{2^k}\right)^N$$

(bound derived by assuming the worst case, `N/b` distinct hashes from the server; the hashes on the client must not include any of these from the server.) To derive a value for k, we want this probability to be arbitrarily close to 1; say for some d:

$$\left(1-\frac{N}{b}\frac{1}{2^k}\right)^N > 1-2^{-d}$$

Approximating the LHS and manipulating:

$$1-N\frac{N}{b}2^{-k} > 1-2^{-d}$$
$$\Rightarrow \frac{N^2}{b} < 2^{k-d}$$
$$\Rightarrow k > d+\log_2 N+\log_2\frac{N}{b}$$

To get a reasonable certainty of no false matches, say one in a million, we can have say `d=20`, so this formula then gives us an easy way to calculate how many bits of the strong checksum have to be included in the .zsync file. It can be rounded up to the nearest byte for convenience; but by keeping this value low, we reduce the size of the .zsync. This reduces the storage requirement on the server, and the total amount that the client must download.

Match Continuation

Another source of information that can help in determining a match is the matching status of neighbouring blocks. There is no reason to believe that matching data in the target file will end neatly on block boundaries - quite the opposite, we will expect to see that after one block matches, neighbouring blocks of the source data will match corresponding neighbours in the target data, giving long areas in the source file that can be copied to the target file.

One way to use this is by rejecting matches unless a certain number of consecutive neighbouring blocks also match (see [CIS2004]). If we insist on, say, 2 matching blocks in sequence, we greatly reduce the chance of false positives - assuming the checksums of these blocks remain independent, then we can halve the number of bytes of strong checksum transmitted per block. The only matches we lose by this restriction are single-block matches - but these are rare anyway, and are the least serious matches to miss (because we will normally have unmatched data either side that needs to be fetched, so the loss of transmitting the data for the extra block is partially offset by the reduced overhead of downloading a single range instead of the two ranges either side). (Alternatively, one can think of this as splitting a larger matching block into two parts and allowing half-block aligned matches, as discussed in [CIS2004].)

The first match of a sequence will involve two neighbouring blocks matching together; assuming this is equivalent to a single block matching with the hash lengths combined, then we can directly halve the required checksum bits from the previous section. For subsequent blocks, while we are testing with a reduced hash length, we are only testing against a single, specific location in the target file, so again the chance of a false match is reduced. So, to avoid false positives in these two cases, we must have enough hash bits to satisfy the following two conditions:

$$2k > d+\log_2 N+\log_2\frac{N}{b}$$
$$k > d+\log_2\frac{N}{b}$$

(from [CIS2004]; I have checked that this seems right, but haven't verified in detail)

At the block sizes that zsync uses, the latter inequality is usually the stricter one, but the difference is small, so the saving in transmitted data is near, if not quite, 50%. For zsync in particular — which, unlike rsync, must always calculate and transmit the strong checksum data for every block in the target file — this is a worthwhile saving.

Note that we can also reduce the amount of weak checksum data transmitted, by assuming that matches for consecutive blocks are needed - in testing it proved more efficient to calculate the weak checksum for both blocks, rather than testing the weak checksum of only the first and then calculating the strong checksum for both (because in a situation where a given block occurred very often in a file, for example an all-null block in an ISO image, a prior block match provides much weaker information about the likelyhood of a following block match). Once we are calculating both weak checksums, we can halve the amount of weak checksum data transmitted. In testing this was upheld, and checking both weak checksums did not significantly harm performance, while providing a definite reduction in the metadata transferred.

rsync Speed

Moving the work to the client relieves the server, but the client then has to deal with the problem of computing the rolling checksum over the old file. As explained in [Rsync1998], although it is necessary to calculate the weak checksum at every possible offset in the old file, due to the choice of checksum the checksum at offset `x+1` can be calculated using the checksum at offset x in combination with the bytes at `x` and `x+blocksize`.

Despite this, when working on large files, for instance ISO files, the calculation can take some time — my Athlon XP 1600+ took roughly 3 minutes to pass over an entire ISO file in zsync-0.2.0. Various optimisations in the implementation helped get it to this level. The client skips forward to the end of a block once a match is obtained (there would not be a match overlapping with an existing match except if the target file contains unusual redundancy), allowing it to parse files faster when there is a large amount in common with the target. The half-block alignment data transfer optimisation also helps speed up strong checksum checking, because often only the first of the two blocks needs to be checksummed in order to get a rejection (whereas using a larger blocksize, we would have to strong checksum the entire block to get a rejection).

Negative Hash Table

To further improve performance, it was necessary to reconsider the use of a hash table to locate target blocks with a matching checksum value. Normally, the size of a hash table should be of the order of the size of the number of records to be stored. But, normally, we are more interested in the speed of sucessful lookups than failed ones. In the rsync algorithm, the vast majority of lookups are negative (that is, they are only to prove that no such entry exists); consequently, what we need is a data structure which makes negative lookups very quick.

I opted to retain the existing hash table for locating positive matches, but adding an additional compact hash table as a pre-filter. This extra, negative hash table just contains one bit per entry (so 8 entries per byte), with the bit indicating whether there is any entry with this hash value or not. Whereas the main hash table has at most 2^16 entries (which, as each is a 32-bit pointer value, is 0.25MB in size), the negative hash table is allowed 2^19 entries (so taking just 65536 bytes, but providing a higher resolution than the main hash table). I experimented with the number of extra bits that the negative hash table should have - each extra bit roughly halved the number of hash hits on large files, as expected, and on large test files it cut over a minute from the total processing time (figures for zsync-0.2.2, before the optimisation was added, shown for comparison):

Version	Extra bits resolved by negative hash	Total time (s) to process 640MB ISO file
zsync-0.2.2	n/a	191
zsync-0.2.3 (pre-release)	1	165
zsync-0.2.3 (pre-release)	2	142
zsync-0.2.3 (pre-release)	3	129
zsync-0.2.3 (pre-release)	4	133
zsync-0.2.3 (pre-release)	5	146

Note that this table is the CPU time used, including downloading and final SHA-1 checksum verification, so the improvement in the core rsync stage is larger proportionally than shown by the table. Elapsed time did not improve by quite as much as CPU time, as amount of time when the process was disk-bound increased.

Networking

HTTP is widely deployed and accepted, and supports Range: requests. But is it optimal from our point of view? HTTP's control data is text key: value pairs, and some control data is sent for every distinct block of data to be transferred. If a file is downloaded all at once, there is only one set of HTTP headers, so the overhead is negligible; once we begin transferring lots of disjoint blocks, this overhead must be quantified.

At its most basic, HTTP transfers one block of data per connection. Each request has a header like `Range: bytes=1024-2047` and each response contains a header Content-range: bytes 1024-2047. But the full set of headers can be 6 or 7 lines:

```
HTTP/1.1 206 Partial Content
Date: Sat, 30 Oct 2004 17:28:36 GMT
Server: Apache/2.0.51p1 (Eduserv/Unix) PHP/4.3.8
Last-Modified: Thu, 16 Sep 2004 04:35:27 GMT
ETag: "3a0c62-27dbe000-935ae1c0"
Accept-Ranges: bytes
Content-Length: 1024
Content-range: bytes 1024-2047
```

This totals 265 characters — a 25% overhead on a 1024 byte block. There are also the overheads at lower layers: most web servers send the headers in a separate packet, so there is an extra set of TCP/IP headers to allow for too (some servers optimise this away and arrange to transmit the headers and data at once, but fewer will do so for requests with unusual headers like Range).

HTTP allows the client to request multiple ranges at once. It then replies with a single set of headers for the reply as a whole, and encodes the content as "multipart/byteranges", using a multipart MIME encoding. This encoding results in an extra header being emitted in front of each block, but this extra header is smaller, with just 3 lines per block:

```
HTTP/1.1 206 Partial Content
Date: Sat, 30 Oct 2004 17:28:36 GMT
Server: Apache/2.0.51p1 (Eduserv/Unix) PHP/4.3.8
Last-Modified: Thu, 16 Sep 2004 04:35:27 GMT
ETag: "3a0c62-27dbe000-935ae1c0"
Accept-Ranges: bytes
Content-Length: 2159
Content-Type: multipart/byteranges; boundary=3e7ad816c6e011b3

--3e7ad816c6e011b3
Content-type: application/octet-stream
Content-range: bytes 1024-2047

[...]

--3e7ad816c6e011b3
Content-type: application/octet-stream
Content-range: bytes 3072-4095

[...]
--3e7ad816c6e011b3--
[end]
```

This reduces the overhead per block to around 90 bytes, a significant saving (but with the full HTTP headers once per request, so the total overhead remains higher). There is the risk that this encoding puts more load back on the server — it would not be advisable to request very large numbers of ranges in a single request. This area needs discussion with some web server developers, to decide where the balance lies between less traffic and more server overhead.

Using multiple ranges alleviates the network-level problems too — it means fewer requests, and servers (Apache, at least) do not issue the boundary headers in separate packets, so the total number of packets will fall too. Note that HTTP servers are required not to issue a `multibyte/ranges` response if there is only a single range given.

HTTP/1.1 allows a further improvement, because the client and server can hold a connection open and issue multiple requests. The client can send multiple requests (each of which can include multiple ranges, as described above) over the same connection. This saves the overhead of connection setup and shutdown. It also allows the TCP stacks to get up to their best data transfer speed: TCP implementations usually use a slow start algorithm, where data is transmitted slowly at first, then increasing the speed until packet loss begins; this is a way of feeling out the available bandwidth between the two ends. Transfer speed is important, because even though zsync transmits less data, it could still take longer than a full transfer if the speed was much lower. TCP stacks are also free to perform other optimisations, like the Nagle algorithm, where packets are delayed so that ACK packets can be merged with outgoing data packets.

Finally, HTTP/1.1 allows pipelining. This allows the client to submit multiple requests without waiting for responses to each request before issuing the next. This is the difference between a full-duplex and a half-duplex connection between the two ends — while the client will be transmitting little to the server, it would clearly be less than ideal if the server has to pause and wait after finishing one block before receiving instructions for the next. While this could be worked around by having multiple connections to the server (so while one was waiting the other would still be transmitting), this would be far more complicated to implement and would be subject to the arbitrary choice of the server and of the network as to which connection used the most bandwidth.

zsync-0.0.1 used HTTP/1.0, with 5 ranges per request, a single connection to the server, and a new connection for every request. It could manage 200-350kbps per second on my ADSL line. zsync-0.0.2 uses HTTP/1.1, keeping the connection open as long as possible, and pipelining its requests, as well as issuing requests for up to 20 ranges per request — it achieves a sustained 480kbps — which is about the normal limit of my 512kbps ADSL line.

To minimise network load and maximise transfer speed, it is essential for any zsync implementation to use multiple ranges per request, HTTP/1.1 persistent connections and pipelining. See [Pipe1999] for more discussion of the performance advantage of HTTP/1.1 - although much of this paper is concerned about links between documents and retrieving links from small, partially downloaded files, some of the HTTP/1.1 and pipelining material is very relevant.

Comparison with rsync

Really this section is only of theoretical value. The relevant difference between zsync and rsync is that rsync requires special server support, and uses this to enable a richer data transfer, with file permissions, tunnelling over SSH, etc. Whereas zsync can be used with no active server support. But it is interesting to compare the effect that this has on their data transfer abilities.

zsync incurs overheads due to HTTP headers. rsync must wrap the data in its own protocol, but has presumably chosen an efficient protocol for this purpose. rsync also has the overhead of rsh, ssh or whatever protocol it uses to talk to rsyncd with, but again this will be rather smaller than the overhead for HTTP.

More interesting is the difference caused by the metadata being downloaded, instead of uploaded. rsync can transmit the weak checksums to the server, and the server then requests only the needed strong checksums, saving on their transmission (I have not checked that this optimisation is actually implemented by rsync). zsync cannot easily do this: while we could separate the weak checksums, so the client could download only those, and then selectively retrieve strong checksums from the server, in practice selective retrieval of strong checksums is not efficient when we have only HTTP Range: requests to work with — each distinct range carries an overhead, as seen above.

More interestingly, with rsync the bulk of the metadata (checksums) travels upstream (from client to server) and only minimal metadata comes downstream with the file data. zsync must download the metadata. As the developers of Bittorrent have noted, the client's upstream bandwidth is usually free [BitT2003], whereas downstream bandwidth is the bottleneck — thus rsync's upstream traffic is negligible in many practical situations. Surprisingly, rsync makes relatively little use of this: empirically, rsync seems to default to quite large blocksizes if the data files being transferred are large, which tends to result in less data uploaded and more downloaded. But rsync and similar protocols do benefit from good bidirectional use of bandwidth (see for example [CVSup1999]).

Empirical Results

As zsync develops, I am performing a number of test runs, and cataloguing the results here. The numbers here must be taken in the context that the current implementation is still being optimised.

Numbers given here reflect application layer traffic only - I have not attempted to account for TCP/IP headers. Generally speaking, provided the algorithm does not result in *more* data being transmitted, and provided it does not needlessly fragment packets or require lots of separate connections, there should be no extra overhead at the network level relative to a full download. zsync-0.0.2 and up satisfy these requirements in my view. I have done some empirical verification of this, but not to the same precision as the other numbers here.

Numbers for zsync are the figures given by zsync itself when exiting - this includes only downstream traffic (upstream traffic is typically negligible with zsync - necessarily so, as the client is doing all the work). Numbers for rsync are downstream, but with upstream traffic given in brackets afterwards, as returned by `rsync -vv` (note in particular that rsync's figures appear to neglect the transport overhead or rsh/ssh, although for rsh I assume this overhead would be negligible anyway). zsync downloads the checksums and then downloads the data, whereas rsync uploads the checksums and then downloads the data, so roughly speaking the up+down data for rsync should equal the down data for zsync, if all is well.

This section deals with data files which are not generally compressed, perhaps because the data they contain is already compressed, albeit not in a form recognisable to file handling tools - e.g. ISO files containing compressed data files, or JPEG images.

The first test file is `sarge-i386-netinst.iso` (Debian-Installer CD image), with the user updating from the 2004-10-29 snapshot (md5sum ca5b63d27a3bf2d30fe65429879f630b) to the 2004-10-30 snapshot (md5sum ef8bd520026cef6090a18d1c7ac66a39). Inter-day snapshots like this should have large amounts in common. Both files are around 110MB.

I tried various block sizes (rsync's default for files of this size is around 8kB). I have included zsync prior to the checksum length optimisations, for historical reference. Bear in mind that zsync-0.2.0's block sizes are not directly comparable to rsync or earlier zsync, because it requires 2 consecutive matches; hence zsync-0.2.0 with a block size of 1024 may be more directly comparable to rsync with a block size of 1024.

Block size (bytes)	512	1024	2048	4096	8192	16384
zsync-0.0.6	13278966	11347004	10784543	10409473	10357172	10562326
rsync		9479309 (+770680 up)	9867587 (+385358 up)	9946883 (+192697)	10109455 (+96370)	
zsync-0.2.0 (pre-release)	10420370	10367061	10093596	10111121	10250799	10684655

zsync transferred more file data as the block size was increased, as expected. At a block size of 512, the .zsync file was around 1.5MB - this fell to 660kB, 330kB and so on for the larger blocksizes. All the results were very close, however: the most obvious feature of the results is that in all cases only about 10MB was transferred, a saving of around 90% on the full download of 113MB.

Next, I tested an update from a Fedora Core 3 test2 iso image (668MB, md5sum) to Fedora Core 3 test3

(640MB) (two Linux distribution CD images, with significant differences between them).

Blocksize (bytes)	512	1024	2048	4096	8192	16384
rsync		339684147 (+5224424 up)	345822571 (+2612232 up)	353812835 (+1306136 up)	363311939 (+571457)	374611439 (+285752 up)
zsync-0.0.6					366356894	
zsync-0.2.0	347181962	347151941	352041787	359541472	369585481	380574374

zsync closely parallels rsync's result here. Roughly 50% of the files are in common I guess from these results, and somewhere around 60% is being transferred. zsync (paired with apache 2.0.52) took about 6 minutes in a local to local transfer, while rsync took about 7 minutes (over rsh).

For reference, here are the CPU times used corresponding to the table above, in seconds. These are just indicative, as they include downloading the control files and the final checksum verification (except for rsync, which does not do this), and the machine was not idle, nor did I flush disk cache etc between runs. Nonetheless, this gives an indication of how expensive the smaller block sizes are, which is an important consideration for larger files.

Blocksize (bytes)	512	1024	2048	4096	8192	16384
rsync		1113	570	418	314	205
zsync-0.2.0	1785	931	520	297	219	158

zsync appears to be very close to rsync, both in CPU usage and transfer efficiency.

Finally, here is an example with a different type of data file. I did an update between two Debian Packages files of a few days apart. These files consist of textual data (some key: value lines, and some text descriptions), with only a few entries changed each day. The files in my test were each 12.1MB; the diff between them was 59kB.

Blocksize (bytes)	256	512	768	1024	1536	2048	4096	8192
zsync-0.1.0 (pre-release)		564709		353690		279580	306050	468092
rsync-2.6.3 (down only)	247512	175004	156772	161452	162108	190048	258128	403776
rsync-2.6.3 (total)	579758	317418	251732	232682	209608	225686	275970	412720
rsync-2.6.3 (total, with compression)	349311	165121	120567	102686	83033	81638	85591	117775
zsync-0.2.1	405520	257388	n/a	204028	n/a	204266	280286	487790

(rsync's default block size for this file is around 3.5kB, giving a sub-optimal 245kB transferred.) Note that zsync is ahead of rsync in total data transferred - in the default blocksizes, the optimal blocksizes, and all of the smaller blocksizes. rsync remains a clear winner for smaller blocksizes if we ignore the upstream data, and is ahead at larger blocksizes (although it mmight be fairer to compare zsync with rsync at twice the blocksize, due to the match continuation optimisation - in which case the result is reversed, with rsync better at the smaller sizes and zsync for the larger). The optimum total data transferred is similar for both. Note that zsync-0.1.0, which lacked the checksum size and match continuation optimisations, is very inefficient by comparison and particularly for small blocksizes.

With compression (the -z option - zlib compression of the comminication channel) turned on — to which zsync has no equivalent — rsync is about 60% more efficient.

Chapter 3. Compressed Content

Table of Contents

Looking Inside
Mapping Deflated Files
Is It Worthwhile?
Compressed Delta Transfer
Compression Results
Recompression

Looking Inside

As discussed earlier, generally speaking the rsync algorithm is ineffective for compressed data, unless the new data is only (at least predominantly) appended to the existing content. The **--rsync** option for gzip is not widely used; if zsync succeeded widely then **--rsync** might become widespread, but that is a distant prospect.

So zsync could work just for uncompressed and **--rsync** files, but this would limit its use, given that so much existing content is distributed compressed. There is no fundamental reason why we cannot work on compressed files, but we have to look inside, at the uncompressed data. If we calculate the block checksums on the uncompressed data stream, store these checksums on the server, and apply the rolling checksum approach on the uncompressed data on the client side also, then the basic algorithm is effective.

Having looked at the checksums for the uncompressed data, the normal rsync algorithm tells us which blocks (from the uncompressed stream) we have, and which are needed. Next we must get the remaining blocks from the server. Unfortunately, HTTP Range headers do not allow us to select a given spot in a compressed data stream, nor would it be desirable from the point of view of the server to implement such a feature. So we must have a mechanism for retrieving blocks out of the compressed data stream.

Mapping Deflated Files

For now, let us restrict ourselves to deflated files. By this I mean the deflate algorithm as defined in RFC1951, and implemented in zlib and in the popular gzip format on Unix systems. More complex formats, like zip files, would introduce too much complexity into the client, as it is unclear what one would do with the uncompressed stream corresponding to such a file: it needs the metadata from the compressed archive to be useful. Simple deflated streams are ideal, in that the compressed file wrapper contains no information of interest to us, so we can ignore it and look at only the stream of data that it contains.

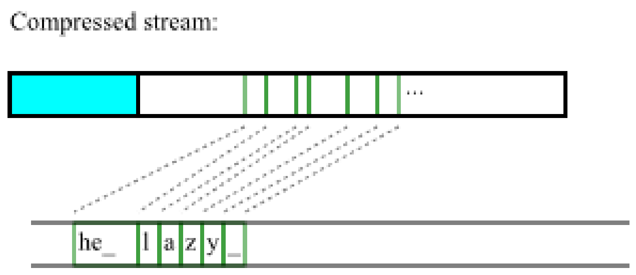
We have some blocks of data from the uncompressed file locally; we want the remaining blocks from the server; the server offers only the deflated stream and only allows us to access it at offsets in the deflated stream. We cannot read the entire compressed stream from the server, because that means there is no use knowing any data locally. So we must have a map which allows us to work out where in the deflated stream a given block of the underlying uncompressed data is — and enough data to allow us to pull this data out of the middle of the deflated stream and inflate it.

This information is available at the time that the zsync metadata is calculated. The program to write the block checksums can also record the location in the deflated stream where a given block occurs. This is not enough by itself, however. The deflate algorithm works by writing data out in blocks; each block's header indicates either that the block is merely stored (so that blocks of data that do not compress well are stored as-is in the gzip file), or it gives the code lengths and other data needed to construct the decoding tree. A typical deflate program like gzip will calculate the optimum compression codes as it goes along, and will start a new block with new codes whenever it calculates that the character distribution of the data stream has altered enough to make a change in the encoding worthwhile. We cannot merely retrieve the compressed data at a given point: we must also have the preceding block header, in order to construct the decoding tree.

So we construct a table that contains the offset of each block header in the deflated stream, and the offset in the uncompressed data that this corresponds to. We can also store similar offset pairs away from block headers, but the client will need to get the preceding block header before it can use one of these.

A simple implementation on the client side can then work out, for each block of data that it needs, which block(s) in the deflated data contains it. It then retrieves these blocks and inflates them, and takes out the chunk of data that it wanted. It will end up transferring the whole deflated block, which will contain more data than it needs — but it will benefit from the data being compressed. The client must also be intelligent about spotting overlaps and merges in the ranges to be retrieve: for instance, two non-adjacent blocks from the uncompressed stream might lie in the same deflate block, or in adjacent deflate blocks, so the client should retrieve a single range from the server and decompress it in one pass. zsync-0.0.1 up to zsync-0.1.0 implemented this approach.

A more sophisticated implementation can use the pointers within blocks. In order to decompress part of the stream, we need to know: the header for the block (or its location in the file, so we can download it); an offset into the compressed stream for that block (which is an offset in bits, as the gzip format used variable-length codes); and a number of leading bytes of output to skip (because it may not be possible to provide an index in the compressed stream corresponding to the desired offset in the uncompressed stream, for example when a backreference generates several bytes of output for a single compressed code). Given this information, we can download the header and the relevant section from the compressed stream, and inflate it; it is no longer necessary to download the whole compressed block.



This image illustrates a hypothetical compressed stream for "The quick fox jumped over the lazy dog.", showing the inflated version of part of the stream (underscores are used to denote spaces in this, for clarity). Note that the first compressed code shown expands to 3 characters, "he " (it is a backreference to the occurrence of the same 3 characters earlier in the sentence).

To extract a block starting at "e lazy dog.", we need to know the offset of the code containing the backreference for "he " in the compressed stream, and the offset at the end of the block, so that we can download the compressed data; *and* the offset (1 character) from the start of "he " to the character we want; and the location of the block header (shown in blue).

There is a final difficulty with deflate streams: backreferences. Deflate streams include backwards references to data earlier in the stream within a given (usually 32kB) window, so that data need not be duplicated. The zsync client will need to know 32kB of data of leading context before trying to inflate any block from the middle of the deflate stream. Provided the zsync client requests blocks in order, it can inductively guarantee that it knows all prior content, and so can construct the window required by the inflate function.

zsync-0.1.2 and up have implemented this more sophisticated algorithm. The zsyncmake program constructs a sufficiently detailed map of the compressed file so that zsync can download only the parts of compressed blocks that it needs.

Is It Worthwhile?

Given that this is relatively complex, and could be made obsolete if --rsync or something similar were more widespread. But technologies do not exist in an ideal world; if the existing content is not adapted for rsync, then it must be allowed for. Some downloads may be more efficient using --rsync and not looking inside the compressed data, while others might be more efficient when looking inside the file. I think it is enough of an open question to warrant implementing something, and seeing whether it proves useful. The basic zsync functionality is not tied to this feature, and it could be easily dropped.

To test the usefulness of this feature, I have benchmarked zsync with some data files which are normally transferred compressed. There are more combinations to consider in the case of compressed files. I have broken them down by how the file to be transferred is compressed (none, gzip, or gzip --rsync) and whether zsync's look-inside-gzip functionality was used. I have also included numbers for zsync-0.1.x, and for rsync-2.6.3 (current at this time). I have also included numbers for rsync with the -z option, which enables compression of deltas with the deflate algorithm on the server.

I took two Debian Packages files, downloaded a day apart, as the source and target files. The target file was 12.1MB, or 3.1MB gzipped. I have included the transferred data as (file data + control data), where control data is just the size of the .zsync file (which clearly cannot be neglected as it must be downloaded, so it is an overhead of the algorithm). For rsync, I have shown numbers both for total data transferred and for just the downstream data (as noted earlier, upstream data is comparativaly cheap, so rsync has an advantage because most of the metadata goes upstream).

Debian Package files contain textual data. This is about half and half between plain English package descriptions, and key:value pairs of text data containing package names, versions, and such. The changes week to week are widespread and very scattered. The diff of the two files was about 58kB.

Several methods were used. Firstly, for comparison, working on the full 12.1MB:

Blocksize (bytes)	256	512	768	1024	1536	2048	4096	8192
zsync-0.1.0 (pre-release)		564709		353690		279580	306050	468092
rsync-2.6.3 (down only)	247512	175004	156772	161452	162108	190048	258128	403776
rsync-2.6.3 (total)	579758	317418	251732	232682	209608	225686	275970	412720
rsync-2.6.3 (total, with compression)	349311	165121	120567	102686	83033	81638	85591	117775
zsync-0.2.1	405520	257388	n/a	204028	n/a	204266	280286	487790

Next, on the file compressed with **gzip --best**. For a fairer comparison with rsync, and to show the difference that the look-inside method makes, zsync without the look-inside method is shown too. As expected, without look-inside or with rsync, almost the entire 3.1MB compressed file is transferred.

Blocksize (bytes)	256	512	768	1024	1536	2048	4096	8192
zsync-0.1.2 with look-inside		613532		339527		217883	190338	230413
zsync 0.1.0 (pr) without look-inside		3134061		3074067		3044591	3033427	3033999
rsync-2.6.3 (down)	3013037	3012749	3012877	3013241	3014117	3014045	3018029	3026169
rsync-2.6.3 (total)	3086271	3048759	3037319	3031581	3026361	3023235	3022647	3028501
zsync-0.2.2 without look-inside	3096931	3054736		3031354		3023228	3022746	3028652
zsync-0.2.2 with look-inside	559703	304058		185237		140735	149467	209643

Finally, the file is compressed before-and-after with **gzip --best --rsync**.

Blocksize (bytes)	256	512	768	1024	1536	2048	4096	8192
zsync 0.1.2 with look-inside		625590		351942		228179	263135	354503
zsync 0.1.0 (pr) without look-inside		496567		449475		444663	492377	607225
rsync-2.6.3 (down only)	400794	390142	394190	392290	407498	417422	471982	581186
rsync-2.6.3 (total)	476020	427778	419292	411128	420072	426864	476726	583578
zsync-0.2.1 without look-inside	449153	415905		406514		422712	485402	617931
zsync-0.2.1 with look-inside	571679	316116		197467		151031	222331 (error)	343906

gzip --rsync does fairly well, with both rsync and zsync transferring about 410kB at the optimum point. zsync with the look-inside method does much better than either of these, with as little as 140K transferred. At this optimum point, zsync transferred 75kB of (compressed) file data - close to the diff in size - and 65kB of the .zsync.

For this example, where the difference between the two files is small, working on the uncompressed data does quite well. With the uncompressed files, rsync transfers about 210kB, and zsync around 200kB. zsync with look-inside on the compressed data is ahead of this - having the data to download compressed saves a lot, even if we are having to transmit a map of the compressed file with it.

The clear winner is rsync with compression (**rsync -z**), transferring only 80kB. Here rsync combines the advantages of all the methods — by working on the uncompressed data and then compressing the deltas, rsync gets the equivalent of zsync's look-inside method, but without having to transmit a full map of the compressed data. But this is at a high cost, since in addition to the usual overhead of reading the entire source data file and doing the checksum calculations for each client, the rsync server has to compress the deltas per client. zsync's look-inside, on the other hand, causes hardly more server load than a normal HTTP download.

Compressed Delta Transfer

The comparison with **rsync -z** suggests another way of looking at the problem. We want to apply the rsync algorithm to the uncompressed data — which is what zsync's look-inside does — and then transfer the needed blocks compressed. While zsync with look-inside combined with **gzip --best** is effectively doing this, it is far from optimal - the code for decompressing from the middle of a compressed block is a workaround, not an optimal solution. For instance, zsync is often forced to transfer extra bytes at the start and end of blocks, and must often make a separate range request for the zlib block header, if that is some distance away from the needed compressed data.

This suggests that we should try compressing blocks individually. One approach to this would be to use on-the-fly compression of the needed data by the server (effectively what rsync does), by using a web server module like mod_gzip ([ModGzip]) or mod_deflate ([ModDeflate]). But this would be against zsync's main aim, to avoid server load; in any case, on-the-fly content compression with these modules is not that widely deployed, and servers might choose not to enable it for partial file transfers. So relying on this to improve zsync's transfer efficiency would not benefit many users.

So ideally blocks should be individually compressed and stored on the server in advance. Compressing blocks separately is a special requirement which gzip is not designed to meet, so to do this zsync will need to do the compresion itself. It is still possible for the result to be a gzip file though: it could be a gzip file which happens to have a new zlib block for each zsync block in the file.

So it is sufficient to compress a file 1024 bytes (or whatever the blocksize is) at a time, telling zlib to start a new block after each input block, and then apply zsync's look-inside method. With this optimised gzip file, zsync should never need to request data from other blocks than the ones it is downloading; the map of the .gz file will point it straight to the start of the compressed data for that block.

Note that this technique is totally unrelated to **gzip --rsync**.

Compression Results

The table below takes the best results from the previous comparison of transfer results with and without compression, and adds one new line, for zsync with the optimised gzip file just discussed.

Blocksize (bytes)	256	512	768	1024	1536	2048	4096	8192
zsync-0.1.0 (pre-release), no compression		564709		353690		279580	306050	468092
rsync-2.6.3 (total, no compression)	579758	317418	251732	232682	209608	225686	275970	412720
rsync-2.6.3 (uncompressed file, compressed data transfer)	349311	165121	120567	102686	83033	81638	85591	117775
zsync-0.2.2, gzip --best, with look-inside	559703	304058		185237		140735	149467	209643
zsync 0.3.3 (pre-release) with optimised .gz		292684		169582		115396	106654	149950
zsync-0.2.1, with gzip --rsync, without look-inside	449153	415905		406514		422712	485402	617931
rsync-2.6.3, with gzip --rsync, (total)	476020	427778	419292	411128	420072	426864	476726	583578

Note that the optimised gzip file gives a further reduction of almost 25% in the total amount transferred (comparing at the optimum block size for gzip --best and for the optimised .gz). The reduction in the block transfer phase is more pronounced, of course; the benefit accrues entirely in the block transfer phase, as it makes no difference to the size of the .zsync.

In fact, zsync, at 106kB, has now almost caught up with **rsync -z**, at 82kB. The difference between the two is almost entirely made up of the 23kB in the .zsync file for the map of the compressed data (essential to zsync, since the server is not compressing the data for it, so zsync needs a map to get it out of the middle of the file).

To extend the testing to a wider range of data types, I performed tests with a binary data file (data files from two different editions of a computer game), and with a large source code tarball (part of the source code for the X window system, ~170MB). Blocksize 2048, which seems to give good performance for all the cases above, was used throughout, and I compared zsync with no compression, with gzip --best, and with its own optimised .gz. Total data transferred is shown.

	zsync, uncompressed	zsync, gzip --best with look-inside	zsync, optimised compression & look-inside	**rsync -z (total)**
Textual data file	279580	140735	115396	81638
Binary data files	9021714	3715979	3647387	3641657
Large source code tarballs	21873310	5630022	4488535	3015492

So zsync still trails **rsync -z**, but the difference is relatively small. And, importantly, zsync does this without any special load on the server.

Recompression

Another problem with zsync's look-inside method is that the end result is the uncompressed data. This is a drawback because many applications require an exact download of the .gz file — for instance, the FreeBSD ports system contains MD5 checksums of every upstream .tar.gz, and will reject downloads if the checksum does not match. One could argue that this is a mistake in the design of the system, and it is the content that should be checksummed (so it would then accept semantically equivalent compressed files) — but since nothing except rsync and zsync are likely to want to transfer a different .gz from the original, it is understandable why this is not allowed for.

However, we can observe that, in practice, it is quite possible to recreate the .gz file — provided the file is compressed with the same options to **gzip**, and the gzip header is reproduced. Clearly this is not guaranteed to work — any system's gzip program could choose to compress a file slightly differently — but, in practice, most Linux and FreeBSD systems at least are using an identical version of **gzip**, and so they can reproduce a .gz file by just compressing with the same options.

The main obstacle here is determining what those options are. There is no field in the gzip format for storing the compression level selected at compression time. But it is possible to decompress the file and then recompress it with a variety of options, until a set of options is found that produces a file identical to the original. Fortunately, it seems that gzip with different options produces files that normally differ within the first few hundred bytes of output, so it seems sufficient to check just a small leading segment of the file. And almost all gzip files are either compressed with the defaults, or with **gzip --best**, so there are few combinations to try.

Chapter 4. Implementation

Table of Contents

libzsync
Control File Generation
zsync Client
Security Considerations
Work To Do

libzsync

Initially I wrote a partial client in python, but I am not an advanced enough python programmer to get the necessary speed for checksum calculations in native python. So I implemented the underlying checksum algorithm and supporting functions for tracking the ranges of known/unknown blocks in C, as a small library. Doing so proved useful in keeping the algorithm and the client implementation separate.

libzsync implements the rolling checksum, and provides functions for checksumming blocks and the stronger MD4 checksum. It is pulled in for both the client, and the program to generate the control file, so they use the same code for calculating checksums.

Control File Generation

The zsyncmake program generates a .zsync file for a given data file. It calculates the checksums of each block, and then prepends the header with file name, length, and the download URL. I chose a simple "key: value" format for the header. The header data is all text (with the exception of Z-Map2, described below), so the administrator can easily edit filenames and URLs if they change. A typical .zsync file header is like this:

```
zsync: 0.0.1
Filename: Packages
Blocksize: 1024
Length: 12133882
URL: http://localhost/~cph/Packages
SHA-1: 97edb7d0d7daa7864c45edf14add33ec23ae94f8
```

I have also chosen to include a SHA-1 checksum of the final file in the control file. Firstly, this serves as a safety check: early versions of zsync will doubtless have some bugs, and a final checksum will help catch any failure and flag then before they cause a problem. Secondly, I am aware of the parallel with bittorrent, which (I think) provides a similar final check. Downloading a file in chunks and merging odd chunks from the local system gives plenty of chance for mistakes, and I would not blame users for being sceptical about whether the jigsaw all fits together at the end! A final checksum gives this assurance. In any case, it only inconveniences the client — the program to make the control file has to read the data through once and only once, and can easily calculate a checksum while doing so.

zsyncmake automatically detects gzip files, and switches to inflating the contained data and recording blocks for this uncompressed data instead. It also adds a Z-Map2 line followed by another block of data in the header, which provides the map between the deflated and underlying data. I have had to include a locally customised copy of (part of) zlib to manage this. Each block of data in the zmap is a pair of offsets: one in the deflated stream, and the corresponding offset in the inflated stream. 2 bytes are used for each offset, except that one bit of the inflated offset field is used as a flag to indicate whether this marks a block header. Together, this gives the client enough information to read from the compressed file: it finds the block header preceding the data it wants, and a pair of offsets in the inflated stream that enclose the required data, and then downloads between the corresponding offsets in the compressed stream.

zsyncmake also has an option to compress a file, in a way optimised for zsync, before starting to make the .zsync. I chose to build in the gzip compressor, as it is too specialised to be of interest outside of zsync.

The zsync version is included to allow future clients to be backward compatible with older .zsync files. There is also a Min-Version header which warns off clients which are too old to use this file, and a Safe header which gives a list of other headers which older clients can safely ignore if they choose.

zsync Client

The client program is also in C. While intending to write it in python, it proved useful to write a C version in order to test libzsync directly. The tie in with zlib was also complicated and required some fairly low level changes to that code. So in the end I decided to simply continue with the C version of the client and make that usable. The HTTP client built into it is fairly crude, but should suffice.

The client is for end users, and so had to meet certain key usability goals:

- It should make it easy to update a local file, but must not corrupt or change the local file until it has completed and verified the new downloaded version. I have implemented the client to construct the new file separately, and replace the old file only as a final step. For now, the old file is moved aside when this happens, so one old copy is always preserved.
- It should never throw away downloaded data. It stores downloads in progress in a .part file, and if a transfer is retried it will reread the .part file and so reuse any content that is already known from the previous transfer.
- The client must be able to get the .zsync control file over HTTP. Even a technically savvy user would find having to pipe the control file from wget or curl to be inconvenient, even though it might be the optimal Unix-like solution to the problem. The client must have a minimal HTTP client anyway, so this is no great inconvenience.
- The client must not retrieve data from servers not supporting Range:. Nor should it continue to hit a server if it notices that the data received from the server is not what it expects. I have implemented this carefully I hope; the client checks all blocks downloaded against the same MD4 checksum used to check local blocks, and it will stop connecting to any server once it has a block mismatch. A mismatch will usually indicate that the file on the server is updated, or the wrong URL is present in the control file, so an abort is appropriate.

The client currently supports the gzip mapping for retrieving blocks out of a deflated stream. It supports requesting only the leading segment of a deflated block where that is sufficient to cover the uncompressed data it needs. It does not implement skipping data inside a block at the moment - if it needs data inside a deflated block, it reads all of the deflated block up to where it needs data.

The client supports multiple URLs for both compressed and uncompressed content; if it gets a reject on one URL it will try another. It chooses among such URLs randomly, providing a crude load balancing option for those that need it. It supports both compressed and uncompressed URLs for the same stream; it currently favours compressed data URLs but investigation is needed about what the optimum choice here is.

Security Considerations

These are mercifully small, compared to rsync or cvs for instance.

- The server shows the zsync version in use in the control file. But the client has no interaction with the zsyncmake program, so showing its version does no real harm. Vulnerabilities in zsyncmake will only be an issue if you try to offer a .zsync for a maliciously constructed archive, and it should be easy to avoid doing anything dangerous with this data. zlib is more likely to be the weak link here.

- The client transmits no checksums to the server. But it does implicitly reveal the checksums of blocks of data that it possesses by its subsequent pattern of requests to the server. Splitting the download traffic between multiple servers operated by different organisations will help here. But the blocks that you request from the server will always reveal those that you need; for instance, downloading a Debian package file, the blocks you download could reveal how long it was since you last updated your system. We could retrieve extra blocks to try and add noise to this information. I do not see a good solution to this. We are at least giving the server far less information than rsync does. And it is all academic if, as with a Debian Packages file, you have to download the new packages too.

- The client must be robust against malicious .zsync files and malicious HTTP servers. The client will only make GET requests, and does not use cookies or SSL, so we are safe from being asked to connect to sensitive URLs at least. We must be wary of buffer overflows in HTTP header parsing and .zsync parsing.

- The client must not modify local files unexpectedly. A policy of only allowing writes to files with the same name as the .zsync file will satisfy the principle of least surprise here. And filenames are not allowed to contain slashes.

- We must not harm web servers. The client aborts if it gets a 200 OK when it wanted a ranged response. So we cannot be fooled into making large numbers of requests for large files by a malicious .zsync file. zsync makes a lot of requests, so we really do not want it to connect to any URL with dynamic content. Most servers will reject Range: requests for dynamic content anyway I suppose? The client does not understand chunked encoding, which is just as well, so users should not find it too easy to point zsync to something dynamic and thrash a server to death regenerating the page every time. We could limit zsync to connect to the same server hosting the .zsync file if we got a lot of administrators complaining - but I do not want this restriction at this stage of development.

Work To Do

- Supporting multiple files, directory structures, and so on. It's not clear whether this is something we want zsync to understand, or whether it should just zsync a tar file and let local tools handle the work. zsync is designed for a very asymmetric, client-server arrangement; syncing directory trees is more of a peer-to-peer operation.

- Composite files containing compressed data. Debian package files, for instance, contain two deflate streams surrounded by a wrapper (an ar file with tar .tar.gz inside). We could have .zsync files describe multiple streams which are to be merged as the final step after transfer. It's not clear if this work if worthwhile, or whether the same efficiency is achieved by using --rsync when packages are built.

- It must be tested against a wider range of servers. There are bound to be some unusual response encodings that defeat the current client.
- Blacklisting of servers with inefficient support for Range:, so users do not work them to death.
- Do what we can to help webservers. We should already be aligning our range requests on block boundaries (for uncompressed content), which will save the server having to read across multiple blocks on the filesystem or disk. Should we issue one range per request, or five, or hundreds at once? Apache seems to calculate the content length etc for the whole request up front, does that mean it does so having already constructed the response, in which case we should not ask for anything too long? And ditto for other servers. But at a network level, fewer connections is better, and allows the TCP stack to get the connection up to speed better if they last longer.
- Work out what should go in the library, to be useful to other client programs. In theory, any web browser could have a zsync plugin and use it to save on large, regular downloads. A web proxy could even implement this transparently.
- Integrate my local modifications back into zlib.

Bibliography

[Rsync1998] *The rsync algorithm*. 1998-11-09. Andrew Tridgell and Paul Mackerras. http://rsync.samba.org/tech_report/.

[CVSup1999] *How Does CVSup Go So Fast?*. 1999-02-10. John D. Polstra. http://www.cvsup.org/howsofast.html.

[Pipe1999] *Network Performance Effects of HTTP/1.1, CSS1, and PNG*. 1999/10/18. The World Wide Web Consortium. Henrik Frystyk Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. http://www.w3.org/TR/NOTE-pipelining-970624.

[BitT2003] *Incentives Build Robustness in BitTorrent*. 2003-05-22. Bram Cohen. http://bittorrent.com/bittorrentecon.pdf.

[RProxy] *rproxy*. 2002/10/03. Martin Pool. http://rproxy.samba.org/.

[GzipRsync] *(gzip --rsync patch)*. http://ozlabs.org/~rusty/gzip.rsync.patch2.

[CIS2004] *Improved Single-Round Protocols for Remote File Synchronization*. 2004-09. Utku Irmak, Svilen Mihaylov, and Torsten Suel. http://cis.poly.edu/suel/papers/erasure.pdf.

[ModGzip] *mod_gzip - serving compressed content by the Apache webserver*. Michael Schröpl. http://www.schroepl.net/projekte/mod_gzip/.

[ModDeflate] *mod_deflate*. The Apache Software Foundation. http://httpd.apache.org/docs-2.0/mod/mod_deflate.html.