rsync in http

Andrew Tridgell
tridge@linuxcare.com

Peter Barker
pbarker@samba.org

July 6, 1999

Abstract

The integration of rsync into HTTP provides a way to significantly reduce network bandwidth usage. The rsync algorithm allows for client or proxy caching of all content, rather than only caching static content as is done using current techniques. This can be done without any additional round trips and without additional storage requirements on the server. It also fits in well with the existing WWW infrastructure and does not require any changes in the way web content is developed or deployed.

NOTE: This is a very brief description of the proposed protocol extension and does not cover several important issues. We intend to produce a more complete description in the near future.

Introduction

The web is moving towards dynamic content. While right now most web pages still consist of static content there is an obvious trend towards content that is generated on demand from back-end databases or personalized for individual users preferences. The problem with this trend is that it is rendering ineffective the traditional web caching techniques that have been so important in keeping internet traffic down.

This paper describes a solution to the problem of dynamic content caching. The solution is based around the rsync algorithm, an efficient remote differencing and data transfer algorithm. It does not require the server to store copies of generated pages or any change in the way web content authors work.

Traditional web caches only save bandwidth if the old and new pages are identical[1]. With the rsync extensions the savings can be made in proportion to the amount of common data with a block-wise granularity. This is effective because most dynamically generated pages have a large amount of boilerplate content that does not change between requests.

The rsync algorithm

The rsync algorithm consists of the following basic steps[4]:

signature generation A signature block describing an existing file is generated by breaking a file up into equally sized pieces and generating two

[1] E-tags allow for a "near enough" concept, but are problematic.

checksums for each block. These checksums are concatenated to form the signature block.

signature search The differences between the data that the signature block was generated on and and the new data are found by a one-pass search on the the new data, computing a rolling checksum at each byte offset and using a hash table to find blocks of data at any alignment in the new data that match any block in the old data. The differences are encoded in terms of literal data and block references and may be compressed using traditional stream compression algorithms.

reconstruction The computed differences can be applied to the old data to generate the new data by alternatively writing literal data and blocks from the old data to form the new data.

The details of the algorithm and the more complicated extensions will not be described here, instead I suggest you look at the complete descriptions available in the rsync documentation[5].

Integrating rsync into HTTP

Integration of rsync into the existing HTTP protocol is fairly easy, and fits in well with the existing web infrastructure. In the following I will refer to a single client and server and will discuss the integration of chains of proxy servers in the next section.

In the following I will assume that the client has a cached copy of the data from a previous fetch of the same, or a related, URL.

- The client generates a rsync signature block for the old data. This is base64 encoded.
- The client sends a normal HTTP GET or POST request but adds a "Rsync-Signature:" header containing the signature of the old data.
- The server does the normal page generation, generating the new data.
- The server computes the difference between the new data and the old data using the supplied rsync signature. The server adds rsync to the Content-Encoding header, creating the header if necessary.
- The client receives the reply and decodes it by referring to the old data.

It is important that all of the above steps be performed in a streaming fashion so that the latency is not increased. This requires some careful coding but is quite possible, as demonstrated in our prototype implementation.

As with the rsync algorithm, this algorithm is probabilistic in nature. The probability of failure can, however, be made arbitrarily small. With reasonable signatures sizes failures will happen only on geological or universal time-scales[2].

[2]Around one failure every 10^{11} years or so if widely used throughout the Internet. The universe is thought to be about 10^{10} years old.

Proxy servers

The previous section described only the simplest case where the HTTP client and server both understand the rsync extensions to the protocol. For rapid adoption and integration with existing infrastructure it is likely that integration into existing or new web proxy servers would be preferable. That is the approach we have taken in our prototype implementation.

When a rsync enabled proxy server receives a request it first determines if the request contains a Rsync-Signature header. If it doesn't and a cache file is available for a matching URL then a Rsync-Signature header is generated and appended to the request before it is passed onto the upstream server.

Upon receiving the reply from the upstream server the proxy then has to handle four situations. Each is handled in a streaming fashion to minimize latency.

- The downstream client supplied a Rsync-Signature header and we got a Rsync-Encoded reply from the upstream server. We send on the reply to the client as is.
- The downstream client didn't supply a Rsync-Signature header and we didn't get a Rsync-Encoded reply. We send the reply on to the client as is.
- The downstream client didn't supply a Rsync-Signature header but we got an Rsync-Encoded reply. We need to decode it before sending it on to the client.
- The downstream client did supply us with a Rsync-Signature header and we got a non-encoded reply. We need to rsync encode the reply and send it on to the client.

In each case above the proxy should save in its cache the unencoded data of any reply from the server.

With the above system the benefits of reduced bandwidth consumption are seen on any sections of the link between the first and last elements which understand the rsync extensions. This allows for the gradual introduction of the extensions without disruption of existing systems.

Signature size

The critical variable in the tuning of this protocol extension is the size of the signature block generated by the client. A larger signature block allows the old data to be divided more finely and produces larger savings in the encoded reply as smaller pieces of common data between the old and new data can be found.

Using the signature algorithms from our prototype the signature block will be 8 bytes long for each block in the old data, so for a 500 byte signature block we would be able to divide the old data into at most 46 blocks, once the overheads of base64 encoding are taken into account. This means that if the old and new data differ only in one place that we will save about 98% of the data in the encoded reply.

Disadvantages of the proposal

The primary disadvantage of the proposed integration of rsync into HTTP is the increased CPU load on the server. A rough rule of thumb is that the algorithm will consume CPU at about the same level as a fast compression algorithm, although there are techniques for reducing this. For servers that are not heavily loaded this will be no problem but it may be a problem for busy servers. This disadvantage needs to be weighed against the considerable bandwidth savings in order to decide if the extensions are worthwhile for a particular server.

A secondary disadvantage is the overhead of sending the rsync signature header in the request packet. In our current prototype we have limited the signature header to less than 500 bytes of data which usually results in a request packet less than the size of a IP segment. Even so, this additional header will more than double the size of a typical HTTP request. The use of a smaller signature, with correspondingly reduced bandwidth savings in the returned data, may be an attractive option.

The size of the signature in the request packet is particularly important when talking to servers that do not understand the rsync extensions as this additional data will be wasted. There are a number of ways of addressing this problem:

- The client can remember which servers do not send rsync encoded replies to rsync-enabled requests and can avoid sending rsync signatures to those servers. The list could be flushed or cycled over a time scale of a few days to pick up servers that add the extension.
- The client could use a very small signature for servers which are not known to support the rsync extensions. If the server does send a rsync encoded reply then the client could remember that server and use a larger signature block for future requests.
- Server generated signatures could be used, as described later in this paper.

Cache file selection

One nice property of a rsync enabled web cache is that the cache content does not have to exactly correspond to a requested URL in order to be used successfully. Using a cache file that differs widely from the requested data will reduce the efficiency of the algorithm but will not affect its correctness.

This allows the cache some flexibility in choosing a cache file to match a request. For example, the cache may choose to trim requests at the first '?' in order to use a single cache file for a CGI script, rather than a separate cache file for all possible script parameters. This reduces the cache size and greatly increases the chance that a cache file will be available for a particular request.

Similarly, the cache may choose to prioritize cache files according to the degree that they match the requested URL, using an exact match if available or falling back on another cached file from the same server (or even a different server) if no cache file is found matching the exact URL. It is expected that cache implementors will develop heuristics for the choice of cache files so as to maximize the bandwidth savings.

Server generated signatures

One way of avoiding the overhead of sending the signature to servers that don't understand the rsync extensions is to use a server generated signature scheme. This also happens to avoid some potential patent problems[3].

The way it works is this:

- The client adds a "Accept-Encoding: rsync" header to all requests.
- A server receiving a request containing "Accept-Encoding: rsync" will generate a "Rsync-Signature" header for the reply along with the normal reply data.
- The client stores the Rsync-Signature supplied by the server along with the normal page data in its cache.
- When the client next requests that URL (or a related URL) the client sends back the signature to the server in a Rsync-Signature header.
- The server can generate differences in the same way as described earlier in this paper.

The advantage of this scheme is that the rsync signature is not transmitted at all unless both the client and server understand the rsync extensions to the HTTP protocol. It also means that the client does not do any signature generation, which avoids some patent issues.

Comparison to other schemes

The concept of moving only the differences between two files over HTTP is not new, although all the schemes we are aware of have some serious shortcomings. The primary problems with previous schemes is that they either require web authors to change the way they develop content by adding additional markup information or they require the server to store outgoing pages. Rysnc avoids these problems.

Mogul et al. [3, 2] have prepared a draft proposal to modify the HTTP protocol to support a Delta-HTML system and it will be possible to use at least some of this framework in a rsync based implementation. They propose two server based difference generating schemes, one using diff(3) and the other a differencing system called vdelta. Both require that the server store outgoing pages which is likely to be prohibitive in terms of storage requirements for large sites.

Williams [6] proposes two schemes to implement http-deltas:

- use sgml-style DYNAMIC tags to specify which parts of a document may change between document retrieval
- "Bytewise Difference Encoding", basically a character-by-character diff(3).

[3] We are investigating some possible patent problems at the moment.

The use of DYNAMIC tags requires web-content developers to modify their content to take advantage of the scheme. This would like be a serious impediment to widespread adoption.

"Bytewise difference encoding" also requires a copy of each document served to be kept on the server.

Hoff et al.[1] discuss "Differential Download", the use of gdiff to retrieve diff-format changes to make to a local file to bring it up-to-date. This would also require the server to cache all (or a large number of) served documents.

A working prototype - rproxy

We have implemented a working prototype called rproxy. The prototype is a simple fork-per-request HTTP proxy server that implements the proposed rsync extensions. It it written in C and currently runs only on Unix-like systems.

The prototype has the following features:

- it can be chained with other proxy servers.
- It uses a maximum signature size of 512 bytes.
- Files smaller than 2k are not cached.
- Cache files are based on a hash of the URL truncated at the first '?'.
- All requests are pipelined for minimum latency impact.
- The proxy is implemented on top of librsync, a simple rsync library implementation.
- zlib is used for deflate compression of the rsync encoded data.

The prototype is available via anonymous CVS from the module rproxy on samba.org. See http://samba.org/cvs.html for details.

Initial results

I have been running rproxy over my dialup connection for the last few days. I have configured it so that my web browser talks to a local rproxy which talks to a rproxy on the PPP server at the other end of the modem link, which talks to a squid server which talks to the world. This setup allows for simple testing of rproxy.

Overall, rproxy has reduced the repeat URL traffic on my link by 76%. By repeat URL traffic I mean any requests for URLs that rproxy could find a cache file for, which is my case is most URLs that I visit. Note that this saving is on top of the savings already afforded by the squid server. Nearly all the savings came from the following sites:

Site	Saving %
linuxtoday	81
slashdot.org	82
linux.com	93
excite.com	93

These sites either make extensive use of generated content or change by small amounts between visits.

Conclusion

This paper has given a very brief overview of a proposal to integrate rsync into the HTTP protocol. While there is a lot more work to be done the initial results shown by the working prototype are promising and certainly encourage further investigation. I strongly believe that for the bandwidth requirements of the web to be kept to reasonable levels in the future this system or something like it will need to be widely adopted.

References

[1] Hoff, Giannandrea, Hapner, Carter, and Medin. Note-drp: The http distribution and replication protocol. Technical report, August 1997. http://www.w3.org/TR/NOTE-drp-19970825.html.

[2] Mogul, Douglis, Feldman, Balachander, and Krishnamurthy. Wrl research report 97/4a: Potential benefits of delta encoding and data compression for http. Technical report, Digital Equipment Corporation, Western Research Laboratory, 1997.

[3] Mogul, Douglis, Feldman, Balachander, Krishnamurthy, Goland, and Hoff. Internet-draft: Delta encoding in http. Technical report, IETF, February 1999. http://www.ics.uci.edu/pub/ietf/http/draft-mogul-http-delta-00.txt.

[4] A. Tridgell. *Algorithms for Efficient Sorting and Synchronization.* PhD thesis, February 1999. http://samba.org/ tridge/.

[5] A. Tridgell and P. Mackerras. rsync. http://rsync.samba.org.

[6] Williams. Http: Delta-encoding notes. Technical report, January 1997. http://ei.cs.vt.edu/ williams/DIFF/prelim.html.