A Client-Server File Synchronisation System

Rui Hu, Zhangbin Cheng
The University of Melbourne

1. Introduction

File synchronisation is a term used to describe the management of copies of files which are kept in different locations. The simplest and perhaps most widespread method is manual synchronisation, in which users remember which files they have changed on which computers and manually keep track of which files are up-to-date and which are out-of-date, but this method is highly error prone and inconvenient.

In this context, a number of technologies has arisen in the last decade, such as file-level synchronisation which transmits the whole new file across the network to another machine, on which the old version of file is hosted. Obviously, this idea is bandwidth and time consuming, especially when it is utilised for synchronising a large collection of files over a slow network. In contrast to the file-level technique, an optimised method called block-oriented synchronisation makes it possible to detect the changes to files and only propagate those changes from computer to computer. The details of this method will be discussed in later sections.

In this report, based on the block-oriented technique, we implement a distributed system for updating files, which is particularly designed for client-server structure. In the meantime, we critically review a number of existing synchronisation techniques in terms of their concepts, merits, demerits and the differences with our system.

As a whole, the report will proceed as follows: Existing file synchronisation methods and techniques are discussed and contrasted in section 2. A detailed description of our system is presented in section 3. Conclusions are given in section 4.

2. State of Art and Related Work

2.1. Rsync

In practice, the most widely used protocol for file synchronisation is the *rsync* algorithm, which is employed within a widely used open-source tool for the same name. The basic approach in *rsync* is described as followings. 1) Destination file is divided into blocks of a fixed size and hashes of these blocks are computed. 2) These hashes are then sent to the source machine, where the recipient attempts to find matching blocks in its own file. This is achieved by comparing received hashes with all substrings of the same block size in source file. 3) Thus, a number of blocks matching received hashes are found on the source machine, which blocks are basically the unchanged data. In between these blocks, the data that is not a part of any blocks can be regarded as the *changes*, either modified or new data. 4) Source then sends instructions back to destination on how to reconstruct a copy of the source file. Particularly, this is done by sending a list of unchanged blocks references and the full data of changed parts. 5) After that, the new version of file is reconstructed on destination machine based on the received instructions.

Our system applies the same guiding concepts as *rsync*, which are particularly the block decomposition and hash comparison methodologies. However, a number of differences exist. Firstly, rather than splitting destination file, our system decomposes the source file into blocks to detect differences. More details regarding to our difference detection method will be narrated in section 3. Secondly, instead of sending all blocks at once, our system sends and processes blocks in a queue. Specifically, the first received block will be looked up in destination file according to its hash value and if it can be found, then the source can send next block, otherwise, the destination will request the source to send a new block with full data which can be then used to reconsolidate the destination file. Therefore, it can be said that *rsync* operates a single round of messages exchange, whereas our system is a multiple rounds operation. Thirdly, *rsync* is a stateless method but our system is stateful. To illustrate, in our system, after a successful synchronisation, the hash values of the synchronised file will be stored locally on the source machine, which means in next synchronisation, system can first compare the modified source file against the hash values of the source file before the modifications. By doing so, transfer of block hash values will be decently saved, in other words, system performance will get greatly improved.

As for the similarities between *rsync* and our system, apart from the similar basic concepts, it

can be clearly seen that the choice of block size is critical to the performance of both of them, and to a great extent, the best choice would depend on the number and granularity of changes between the versions. For instance, if a single character is changed in each block of source file, then no match will be found by the destination which in turn making both *rsync* and our system completely ineffective. On the other hand, if all changes are collected in a few areas of the file, the performance will be pretty satisfactory even with a large block size.

2.2. Content-Dependent File Partitioning

Using a fixed block size as in our system means that in order to find a match, the recipient has to compute hashes for all possible alignments in its file, which requires not only more computation, but also more bits in each hash value.

In this context, a partitioning technique, which allows us to find many common blocks in similar files, is highly expected. Typically, there is one partitioning technique called *2-way min*, which can be defined as follows: Similarly to *rsync* and our system, hashing technique is fist applied to convert all substrings of a fixed length *l* to integer values. And then a block boundary is defined before position *p* of the file if the *hash[p,p+l-1]* is strictly smaller than the hash values of the *w* preceding and following positions, the average length of blocks thus will be about *w*. Since boundaries are chosen in a local manner, any large substrings common to both files are partitioned in the same way and result in identical blocks. After identifying the boundaries, the resulting blocks are hashed to integers as part of the synchronisation protocols.

From the comparison statistics [1] between content-dependent partitioning technique and *rsync* which utilises fixed block size, it is obvious that as block size decreases, all methods can identify more data redundancy. And overall, *rsync* even performs the best, nevertheless, it is notable that *rsync* requires more bits for each hash value, which is offset against its benefits. To illustrate, if the file synchronisation is operated over a low bandwidth network, it is evident that content-dependent partitioning would be a better choice than both of *rsync* and our system.

2.3. Low-bandwidth Network File System

To be practical over the wide-area network, a file synchronisation system must consume significantly less bandwidth than most current file systems, both to maintain acceptable performance and to avoid monopolising network links in use for other purpose. In this context, a new file synchronisation method was proposed by [2], which significantly improves on previous approaches, such as *rsync*, in terms of bandwidth usage. Overall, this method partitions the whole synchronisation into two phases, namely *map construction* and *delta compression*. Specifically, in *map construction* phase, multiple roundtrips are applied in two files to create an approximate representation of the identical parts. To be specific, a *map* of the latest file f_{latest} is generated based on hash values sent by the destination, which specifies the known and unknown parts of f_{latest}. In the meantime, the destination will maintain a shadow map that keeps track of the map, such as which parts of f_{latest} are known to the client. The goal of this phase is to minimise the size of the unknown parts of f_{latest}. In *delta compression* phase, delta compression technique is applied to transmit the unknown parts of f_{latest} to the destination. In particular, the source creates a reference file f_{ref}, consisting of all known parts of f_{latest} and a target file f_{target} of all other parts. After that, the destination will use its map to recreate f_{ref}, and then decode delta into f_{target}.

For the map construction, it is apparent that there is not much to do apart from repeatedly sending hashes like what our system actually does, but in order to fit the slow network context, it is possible to minimise the number of bits needed for the hashes. A technique, which is called *Optimised Match Verification*, could be employed to achieve so. Specifically, the idea is to first send a fairly weak hash that can be used to identify a possible match, and then an optimised protocol is utilised based on ideas from group testing to filter out any false matches. For instance, according to [2], if the length of destination file is *n*, at least lg(n) bits would be required for a hash that is compared to all positions in the source file. Hence, the destination could initially send only slightly more than lg(n) bits to allow client to identify candidates for matches. After that, by having the source send

verification hashes of its matches back to the destination, these candidates can be verified in an optimised manner.

Compared with our system, it is similar that both of them stand on a multi-rounds block-oriented synchronisation system, however, due to the different changes-detection techniques, this method requires more communication rounds but consumes a lower bandwidth than our system. There is always a trade-off between the number of communication rounds and the bandwidth consumption and it actually depends on the operational context. An example would be that extra rounds might increase the communication latencies for small files, whereas these latencies could be hidden on larger collections.

2.4. Decomposable Hash Functions

A hash function is decomposable if hash(f[m+1,r]) can be efficiently computed from the values h(f[l,r]), h(f[l,m]), r-1, and r-m-1 [2]. A decomposable hash function allows system to significantly save on the cost of the hashes for indentifying candidate for matches, since a hash for the parent block has already been transmitted. What needs to do is just to send one additional hash per pair of sibling blocks and then the other sibling's hash can be computed from these two. In addition, a decomposable hash function should be rolling so that h(f[l+1,r+1]) can be computed in a constant time from h(f[l,r]).

Similarly to the *Optimised Match Verification* technique mentioned above, decomposable hash functions are also aimed to minimise the number of bits required for hashes. However, this method is less implementable than *Optimised Match Verification*, due to some obstacles in practice. For instance, the frequency of same mapping hash of strings obtained from each other via permutation should not be too high; as such cases are quite common in reality. Moreover, compared with *rsync*, both of them can achieve rolling hash functions, for instance, the "rolling checksum" version of Adler-32 utilised in *rsync* is basically a kind of decomposable hash functions. In contrast with our system, the main difference is still the efficiency of bandwidth consumption and the number of communication rounds. Furthermore, from practical point of view, the algorithm of decomposable hash functions is much more complex than that of our system.

3. Our System Design

The proposed Client-Server system employs a synchronous request-reply-acknowledge (RRA) communication paradigm to effectively update the destination's file to make it identical to the source's file. The system is capable of fulfilling different levels of functionality from basic one-way client-server transmission to bidirectional operations with negotiable roles and block size.

At first, the server is started and listening on a port to which a server socket is bound; after that, the client can make a TCP connection request to the server using the hostname (such "*localhost*" or IP address) of server machine and the port number the server is listening on. Once the connection is accepted by the server, the client and server can now communicate by writing to or reading from their socket. Generally, the communication protocol is implemented as follow: the server initialises synchronisation negotiations and the client replies it by involving user's interactions. Then the selected sender begins to send instructions. Upon receiving an instruction, the destination tries to process it and sends acknowledgement back if the process is successful or sends new block request if referred blocks are not available. Importantly, next instruction will not be issued until the source receives acknowledgement of last instruction from the destination.

To be specific, the server initialises a JSON object querying who will be the source/destination, the block size to be utilized in subsequent communication and client file's last modified time. An example of the initial JSON may look like: *("BeSource":"Yes","LastModified":"05/01/2013 16:51:59","BlockSize":"1024").* This JSON object is then converted into string and sent to the client for response. When the client receives the JSON query, it will prompt user input to decide who is going to act as source or destination and specify intended block size for later transmission. Then the user's choices are recorded and sent back to the server. The response JSON may look like: *("BeSource":"Yes", "LastModified":"05/02/2013 12:11:59","BlockSize":"2048").* It is worth noting

that, no matter which role the client will be, the server is always told the block size so that communication is guaranteed consistently. Furthermore, the client fetches the last modified time of the file on server and replies its own file last modified time in the response JSON. By this means, both the client and server know each other's file state. If the last modified times on both sides proved to be the same, the system will just skip current synchronisation round. Once the last modified times do not match, the server will send the client a confirmation and start synchronising.

For instance, if the client is chosen as the source (sender) by replying the server a JSON like *("BeSource":"Yes","LastModified":"05/01/2013 16:51:59","BlockSize":"3000")*, the server (destination) will then send confirmation *("OK, you are source")* and starts waiting for instructions to come, meanwhile the source begins to check its file state and build the instruction queue as soon as the confirmation arrives. On top of that, every synchronisation process is preceded by a *StartUpdate* instruction and finished by an *EndUpdate* instruction. When the server receives the start instruction, it processes it and sends acknowledgement message *("Process Acknowledged")* to the client. Once acknowledged, the client can continue to transmit block instructions. There are two types of block instructions, one is called *CopyBlock* and the other is *NewBlock*. A CopyBlock is produced if the considered block is not modified at the source side since last file state check and hence only block hashes are included in instructions. On the other hand, a NewBlock instruction is produced when changes are detected and actual bytes of the block are included. As a CopyBlock instruction arrives, the server first looks up whether it has the bytes referred by the block hash or not; if it happens to have, the server copies its existing block into a temporary file and sends acknowledgment to inform the client. If for some reason the block is not available at the server, a *BlockUnavailableException* will be thrown and the server will issue a request asking for upgrade from the CopyBlock instruction to a NewBlock instruction. When the client receives a message *("NewBlock Request")*, it updates the instruction and resends it to the server. Now the server can insert the new block bytes into the temporary file and sends acknowledgment back. As mentioned before, an instruction will not be sent until acknowledgment of last one is received. Finally, an *EndUpdate* instruction is encountered, which indicates a synchronisation round is completed and the server will suspend for client message. At this moment, the client prompts user to determine whether he/she wants to quit the system or continue to another synchronisation. If "exit" is typed, the client messages *"Exit"* to the server and terminates itself. The server will be terminated too after exit message is captured. Otherwise, another synchronisation round is activated and going to perform exactly the same as the previous procedures, that is, user is allowed to choose a different role (source or destination) and specify a different block size every time.

In the other case, if the client is chosen as destination by replying the server a JSON message like (*"BeSource":"No", "LastModified": "05/01/2013 16:51:59", "BlockSize":"5000")*, the server likewise will send a confirmation to the client *("OK, you are destination")* to grant. However, this time it is the server that who wants to check its file state and build the instruction queue while the client is waiting for instructions to come; the rest processes will perform the same as previous situation (where the client is source and server is destination). One thing to note is that the power to make decision of who will be the source or destination and what kind of block size to use is on the client side (namely the user). Also, the decision to determine either to quit the system or going for another synchronisation process.

Eventually, either way depicted above will result in a mirror from the source file to the destination file.

A typical communication procedure (the client acts as source and server as destination) is illustrated in Figure 1.

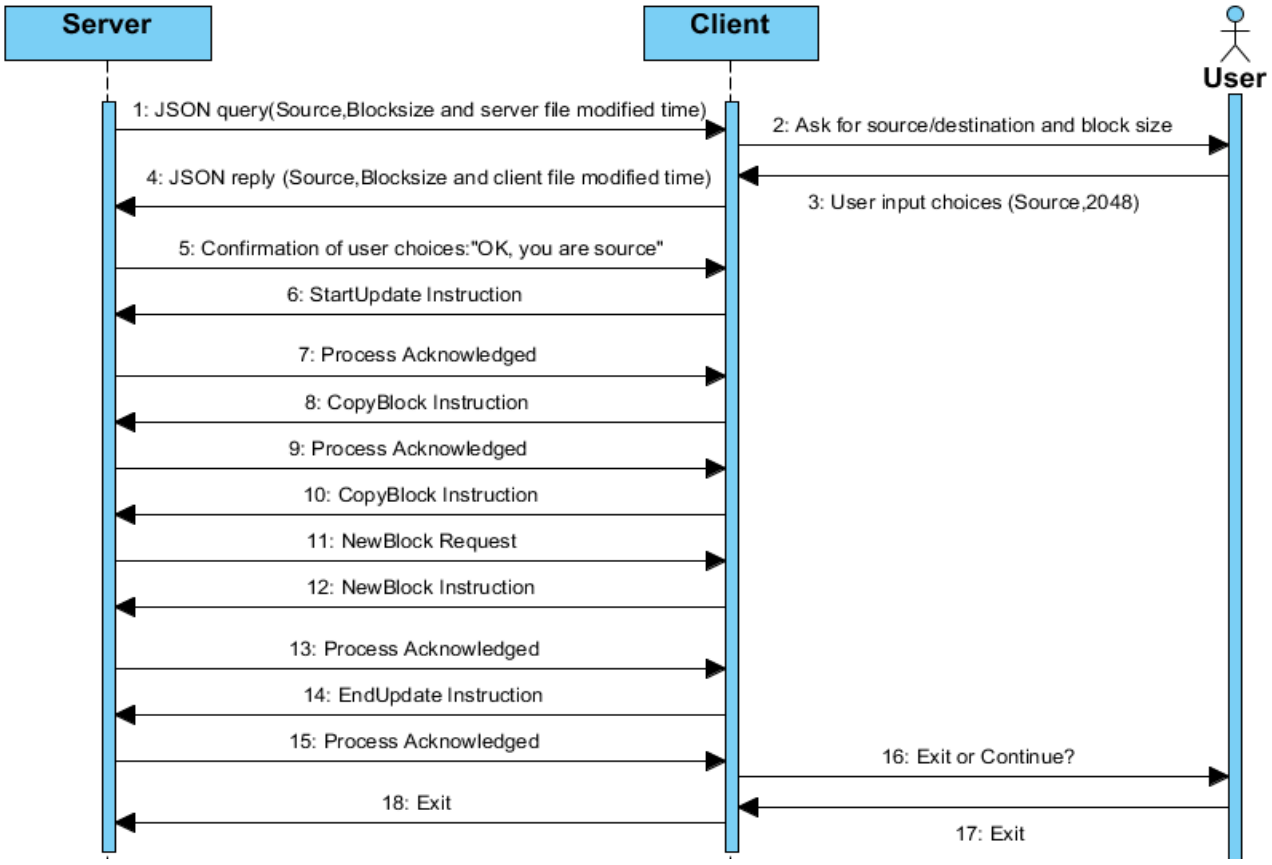

Figure 1

4. Conclusion

We have implemented a client-server system which effectively synchronise files distributed across network. The system features by breaking files into successive blocks of fixed size and identify each block by its hash value, in which way, the bandwidth can be significantly saved as only hash values of blocks are actually transmitted and receiver can take advantage of the hash indexes to subsequently look up blocks to reconstruct its file without requesting the same data from the source. A Request-Reply-Acknowledge protocol is implemented in our system to model a well-defined client-server communication scheme by ensuring instructions emitted from source will be sequentially processed by the destination and the sender also will be timely informed of successful process acknowledgement from the receiver, through which, every block is guaranteed to be operated correctly. In addition, the system accepts user input to determine client/server roles as well as block size in a synchronisation round, making the procedure more flexible. On the other hand, there is still much room for us to improve when compared with some other methods studied from literature.

We believe that the lessons learned from our work are as follows: First, the problem with sing-round synchronisation algorithms like *rsync* is that performance greatly depends on the choice of the block size. But unfortunately, selecting a good block size sounds easier than it is. One obstacle would be that it is crucial to know the best compressibility of the unmatched parts of the new file to determine the best block size, while the results in [1] indicate that the compressibility ranges widely in different data sets. Second, in contrast to the fixed block partitioning in *rsync* and our system, the content-dependent block partitioning technique [5] performs much better and provides an interesting alternative to fixed-size blocks because of its enhanced ability of finding commonalities between two files. Third, Optimised Match Verification and Decomposable Hash Functions are more suitable for low network synchronisation due to its capability of minimising the number of bits required for hashes, in other words, bandwidth consumption can be saved decently.

References

[1] Hao Yan, Utku Irmak, Torsten Suel. 2008. *Algoritms for Low-Latency Remote File Synchronization*. In IEEE INFOCOM 2008 - the 27th Conference on Computer Communications; 2008, p156-160, 5p

[2] Torsten Suel, Patrick Noel, Dimitre Trendafilov. 2004. *Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks*. In Proceedings of the International Data Engineering; 2004:153-164

[3] Athicha Muthitacharoen , Benjie Chen, David Mazieres. 2001. *A Low-Bandwidth Network File System*. In Proceedings of the eighteenth ACM symposium on Operating systems principles; Pages 174 - 187

[4] Russ Cox, William Josephson. 2005. *File Synchronization with Vector Time Paris*. Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory

[5] D. Teodosiu, N. Bjomer, Y, Gurevich, M. Manasse, J. Porkka. 2006. *Optimizing File Replication over Limited Bandwidth Networks Using Remote Differential Compression.* MSR-TR-2006-157, Microsoft.