Summary - Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks

Bowen Song[1]
August 15, 2018

***Abstract*— This report is a part of an independent study for set and string reconciliation problems in distributed systems. The paper [1] considers high-latency file synchronization using a reduced amount of message size in a server and client pair situation. This paper shows significant experimental improvements comparing the communication cost of its protocol with that of *rsync*.**

I. Introduction

The problem of maintaining large replicated files consistent under a small number of modifications in a bandwidth limited distributed system is convertible to a problem of string reconciliation. By regarding the files on different hosts as strings, if the two reconciling strings are within a small amount of edit distance, a string reconciliation protocol looks for a cost-efficient communication method to reconcile the two strings without knowing the details of their differences. The protocol considers a Server-Client setting where both the server and the client holds a copy of the same file and the client would like to update its local copy based on the updates from the server.

II. Algorithm Overview

The general approach is to reconcile two similar strings by partitioning a file into blocks of data, using multiple round-trips to figure out the common partition blocks, and sending the differences with the common block as the position reference. The algorithm only focuses on minimizing the number of rounds and transmitted message size to maximize partition matches between the two strings and regards sending string differences as a trivial procedure.

The protocol uses a recursive splitting technique to partition the server files with fixed block size. The protocol starts with a large partition block size that is powers of 2 and attempts to match with parts of the client. For unmatched blocks, the server recursively splits those blocks and seeks to match them in the next round until block length reaches the terminal size. The terminal block size is a tuning parameter which is experimentally shown to be best at 128 or 64 bytes. For each round of split partitioning, the algorithm uses either *continuation hashes* or *decomposable hashes* to minimize message size.

[1]B. Song is with Department of Electrical and Computer Engineering, Boston University, Boston MA, sbowen@bu.edu

A. Matching Partitions with Clients

At each round of communication, the server sends a set of weak hashes of file partition and seeks to match with parts of the client's file. For a fixed partition size, the client could coordinate with all possible substrings of the same length. For block length l and string size n, the clients would have $n - l + 1$ possible substring hashes. The client can locally mark the matched blocks and send the strong hash back to confirm with the Sever, which saves the confirmed block-match mappings. Once a confirmed match is found, the protocol tries to grow the matched region by *continuation hashes*; And for the unmatched blocks, the protocol uses *decomposable hashes* to confirm matching the subpartitions.

B. Continuation Hashes

In the case of a match is found, it is very likely that the neighboring unmatched block's subpartitions are also matching. Therefore, the server attempts to match the adjacent half of previous and the next block to that of the client. If the subpartitions are matching, the server then tries to expand the match for a quarter of the block, or else retrieve a quarter of block until the length of subpartitions is under terminal block size. In addition, the size of subpartition hashes can be decreased to $2log(l')$ for small scaled matching, where l' is the current length of the block.

C. Decomposable Hashes

For the unmatched blocks, the server breaks the partition by half and sends the client two subpartition hashes for further matching. The decomposable hashes are hash values that can be computed for half of the string knowing the hashes for the entire string and the other half of the string. With decomposable hashes, the server can send only one of the two subpartition hashes to the client while the client can compute the other half. Unfortunately, the paper does not provide the collision rate for the decomposable hashes.

III. Algorithm Performance Analysis

The algorithm is experimentally compared with *rsync* and *zdelta* data compressor. The total amount of transmitted message size is almost half of that of *rsync*. Using terminal block size as a tuning parameter, as expected, the larger the terminal block size, the bigger final string differences message is, and it trades off the rounds of communication between client and server and latency. When the terminal

block size is set to 32 bits, the final string differences message size is close to the size of compressed data from *zdelta*.

IV. Conclusion

The proposed protocol reduces the communication cost for file synchronization in comparing to the *rsync* and works well for large sized files. However, there is no analysis for the increased computation cost from the extra hashing. In addition, the protocol would need to guarantee that each partition block is unique. It is unclear how to resolve file blocks that are matched with multiple substrings or parts of the file from the client.

References

[1] T. Suel, P. Noel, and D. Trendafilov, "Improved file synchronization techniques for maintaining large replicated collections over slow networks," in *Data Engineering, 2004. Proceedings. 20th International Conference on*. IEEE, 2004, pp. 153–164.