Effective Synchronization of Data in Distributed Systems

P. Ocenasek
Brno University of Technology, Brno, Czech Republic

ABSTRACT: This paper proposes an approach for effective synchronization of data in distributed information systems. The first part of the paper deals with the synchronization. The second part presents the system that realizes the proposed approach. This approach supports its own protocol for exchanging data between nodes. The system is designed as platform independent.
KEYWORD: Synchronization, data, distributed system, backup, snapshot.

1 INTRODUCTION

Given the current increase in information technology and the gradual expansion of electronic documents and data, rises the need for sharing between different platforms and backup of the data. Data backup is a process which creates a copy of the data, in order to avoid the risk of loss. It is possible to return to backup data at any time. Synchronization and sharing of data is needed, where we work with different technologies, and even different platforms. Furthermore, it is needed when working in groups and in places where it is frequent to share data together. In both cases, you can automate these activities for the convenience of the user, which can also save much of the time.

This work aims to propose an approach (Martak, 2013) that should promote the general protocol for exchanging data, so that you can synchronize and backup data between different platforms. Nowadays, it is certainly the need to deal with the security of transmitted data, in order to prevent getting this data to unauthorized users. Therefore, when designing this tool, we put the emphasis on security and data retention. Another important factor for the design of this tool was the effectiveness. This would result in reduce demands on resources, such as bandwidth usage, network used and reduce the disk space required for backup data. (Tridgell, 2008; Martak, 2013)

2 SYNCHRONIZATION

Data synchronization should ideally bring the sharing of data between different places (Cohen, 2008). This process does not ensure data backup, since it does not provide the possibility of data recovery, but only duplication. The benefit of such sharing is that on all the places anyone can work with this data independently of each other.

When synchronizing with must ensure that out-of-date data was synchronized with the amended form of the same data from another location. This brings the issue of the conflict, so there will be the same data in other places. These conflicts can be solved in different ways and it just depends on the user and the tool, which is used for synchronization and what options for conflict resolution he offers.

These systems allow us to cooperate between the different participants and shall keep the status of the file system (most often in the form of a tree) as developed in time (Cougias et al., 2008). Users have the option at any time to return to previous states of the file system. An important feature is the file sharing solution and their conflicts between users (Pilato, 2008). The basis of such systems is a repository, which contains just mentioned various versions of files, and other information depending on the particular system.

One of the options, how can a user store different versions of his files is a local version control (Cougias et al., 2008). This form is applicable only to one user. Therefore, there is another form, called centralized version control system (Cougias et al., 2008; Pilato, 2008). This allows multiple users to work simultaneously. Centralization allows us to have the entire work in one place, and users download only the current version of the files. If we have to restore an older version of the file, we download the required version from the server. The disadvantage may be that the system is now in one place.

In the case of failure, the user does not have the opportunity to work with the remote system.
Another type of management systems represent distributed version control systems (Cougias et al., 2008). These systems keep the contents of the entire repository on the user's storage place. The user does not need to face problems like with the centralized systems and can still operate with the repository. If a repository contains large number of files and versions of files, it can be difficult to store at local medium.

3 ARCHITECTURE

We decided for peer-to-peer design of the concept. (Martak, 2013) Each node holds the entire history, including the current version, each node can use this data even in the event that the another node on the network is not available. The distributed system also provides duplicate data in case of damage. Thanks to peer-to-peer architecture, it appears similar topology to the mesh topology. On Figure 1 (Martak, 2013) we can see the example of such a topology. Node A synchronizes with node B. Both nodes C and D synchronize through node B. Both the nodes synchronize indirectly with the node A.

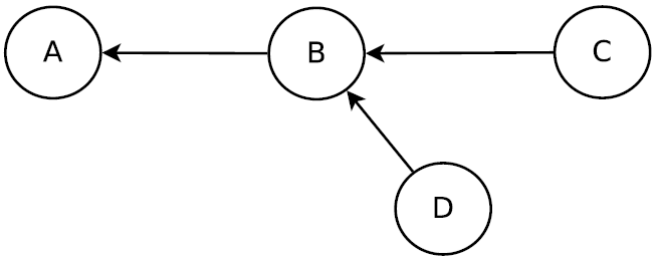

Figure 1. Possible synchronization topology.

For communication between two nodes, we need to determine their order. If the first node connects to another, we consider the first node as the client. This is due to that the BSD sockets are designed for a client - server architecture. During the whole communications, the nodes are equal. The system supports both IP versions due to the gradual expansion of IPv6. (McCabe, 2010; Martak, 2013)

4 COMMUNICATION PROTOCOL

For a description of the protocol we have used Protocol Buffers. It is ensured that the communication between tools supporting this protocol is platform independent. Anyone can start communication. Since both the peers (communication subjects) are equal, they can communicate with each other at once. (Martak, 2013)

Each message is created in the Protocol Buffers and is encapsulated in length-value structure like BER TLV 2 structure. Since the message types are contained in the structure of the Protocol Buffers, it is not necessary to indicate the type of message. We will create a simple length-value structure. The first four bytes of length-value structures represent the size of messages in the Protocol Buffers and are followed by the contents of the serialized message. (Tridgell, 2008)

Communication is divided into the following types: notification, request and response. Each type of request or response message contains a transaction identifier. At the beginning of communication both devices exchanged a message about extensions of the Protocol and introduce its own identifier and name, followed by exchange of information about volumes. After the initial exchange of information, there starts the request-reply communication. One request may have more answers. The request may be followed by the next request, if this is specified in the previous request. If the tool detects the presence of any of the volumes, the remote user can start synchronization. First, other party is requested the identifier of the last known snapshot. If the remote user does not have any snapshot for any volume, there will be no identifier in the message. This synchronization will end, since there is nothing to synchronize. (Martak, 2013)

Finally, the tool receives the last snapshot, it checks its own database if that snapshot already contains. If the snapshot has been found, the database already contains all of the changes to the other party. If the snapshot has not been found, the tool continues with the request on the next snapshot. After the receipt of this request, it has to find out whether a snapshot of parents. If the parent exists, and does it in a local database, the tool must continue polling for so long, until it finds the last synchronized snapshot or until it finds one that does not already have parents.

After finding the last of the new snapshot, which is no longer in a local database, the tool can begin querying metadata objects and data themselves. If the tool downloads all the objects bound to this snapshot, it can save it to our database as the last snapshot with the same identifier and continues on the next new snapshot, which does not yet have. This continues until all remote snapshots are synchronized. After synchronization is complete, the tool sends notification that it is synchronized.

When saving the snapshot to the database, it is important to maintain the snapshot identifier. This is because the tool, which originally belonged to, could detect its own snapshots in the foreign database profiles and does not need to download them again. Otherwise this could cause a recurrence of downloading own data.

When posting a pull message, the party can specify whether the remote node has to send changes that gradually appear while working with the working directory of the volume. Another message can stop sending these changes at any time.

If there is an error in the determination of the response, the tool has an option to send an error message together with a description of the error.

For efficient data transfer between the nodes, we have used the rsync delta algorithm. This can be used in cases where one party requests data for a particular file, has also available an earlier version of the file. In other cases, it is necessary to always send the entire file. The use of this algorithm is not mandatory, even in cases where this would be appropriate. Some of the parties may not have this extension implemented. (Martak, 2013)

5 STATUS AND DATA RETENTION

Files and directories of the current working directory are synchronized according to the defined volume. Each volume has a fixed path on the system and an optional name. The version history of files may be kept according to the user settings.

Next, it is needed to store all the information for the current status, all metadata for files and the file data itself. Each node (peer), the object and the volume will be identified using the UUID3, which should guarantee that none of the listed entities will have the same identification number. Object is data, metadata, and snapshot (snapshot). The snapshot represent a new version of the changes and can contain one or more items of metadata. (Tridgell, 2008; Martak, 2013)

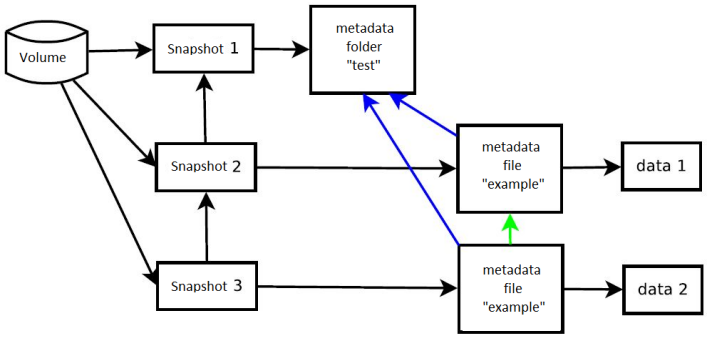

Figure 2. Object hierarchy.

We can see an example of the possible object hierarchy on the Figure 2 (Martak, 2013). There is a single volume named volume. In this volume, there were gradually made three changes, which represent the three snapshots. Snapshots are dependent on each other to make it possible to determine the sequence of changes. Snapshot 1 has no parents, so it is the first change. Snapshot 1 shows the object type metadata directory called test. The metadata of the test does not have any direct ancestor; this means that the directory was created. Since the directory itself cannot contain data, it cannot refer to a data object. The second snapshot indicates the type of metadata file named example. This metadata belongs to the directories test, this is expressed in the blue arrow. The metadata of the type file can contain data, which in our example is the arrow on the object data 1. There is also a snapshot 3, which shows the metadata for a type of file that is named example. This metadata have an ancestor, which expresses the green arrow and also contains custom data, which expresses the arrow on the data object 2.

Another option is to identify objects by using the SHA-1 thumbprint as it makes Git. This could be difficult and inefficient on the platforms with limited resources. For simple detection of whether the data in the file has changed, it would be possible to use SHA-1 fingerprint file. This could simply detect whether file data has changed or not. If the data has not changed, new metadata object with the original data is created. (Tridgell, 2008; Martak, 2013)

If we want to save binary data in any of the databases, we have to count with their limits. All the considered database systems can save a maximum of 1 GB of binary data into one record. If files are larger than 1 GB, the data must be split into multiple records.

The stored data is always full. The tool will always make a full back up of the data. For the efficient storage of data, it would have been able to save only the changes between the two objects of data. It might be challenging, so the implementation will always save the data in the full form.

6 IMPLEMENTATION

The application is designed as a multi-threaded, as is the need to react asynchronously at a networking event and the events in the file system. Furthermore, it is necessary to use a blocking operation for user input. Overall, at least three threads together with main thread are needed. (Tridgell, 2008; Martak, 2013)

The main thread manages volumes and should contain the logic for the management of the entire application. The main loop in the main thread also ensures collecting requirements of class FSMonitor and larger clusters are added through the class VolumeController to the individual volumes. This ensures that if there will be more changes at once, all added with only one snapshot. If they were added gradually changes one at a time, for each such amendment would be needed to create a new snapshot.

After starting the tool, the initialization of the user comes, and the tool generates a certificate with the private key. If the user owns some volumes, their checking follows. Then, the three threads are creat-

ed. One thread is for network communication and processing of remote requests, the other thread is for user input and the last thread is waiting on an event from the file system. For synchronization between threads, we have used mutexes and in some cases, a shared queues. The program has been used for the design on an object-oriented paradigm. (Tridgell, 2008; Martak, 2013)

7 SUMMARY

The aim of this work was the development of effective synchronization and backup system. The system was tested on a random generated sample data to test its functionality for backing up and synchronizing the data and to test its performance. The improvement of the transmitted data efficiency has been achieved using the rsync algorithm, delta. When using this algorithm, during the synchronization only the changed data is transmitted. This saves the synchronization time bandwidth.

Acknowledgements. This project has been carried out with a financial support from the Czech Republic through the project no. MSM0021630528: Security-Oriented Research in Information Technology and by the project no. ED1.1.00/02.0070: The IT4Innovations Centre of Excellence; the part of the research has been also supported by the Brno University of Technology, Faculty of Information Technology through the specific research grant no. FIT-S-14-2299: Research and application of advanced methods in ICT.

REFERENCES

Cohen, M. 2009. *Take Control of Syncing Data in Snow Leopard*, 1st Edition. Take control, TidBITS Publishing, Incorporated. ISBN 9781615420032.

Cougias, D., Heiberger, E., Koop, K. 2003. *The Backup Book: Disaster Recovery from Desktop to Data Center. Network Frontiers Field Manual Series*, Schaser-Vartan Books. ISBN 9780972903905.

Dubuisson, O. 2000. ASN. 1: *Communication Between Heterogeneous Systems*. ISBN 0-12-6333361-0.

Georgiev, M., Iyengar, S., Jana, S. 2013. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security*. URL <https://crypto.stanford.edu/~dabo/pubs/abstracts/ssl-client-bugs.html>

Martak, A. 2013. Synchronization and Backup of Data under OS Linux. FIT Brno University of Technology.

McCabe, J. 2010. *Network Analysis, Architecture, and Design. The Morgan Kaufmann Series in Networking*, Elsevier Science. ISBN 9780080548753.

Pilato, M., Collins-Sussman, B., Fitzpatrick, B. 2008. *Version Control with Subversion*. Holley Series, O'Reilly Media. ISBN 9780596510336.

Tridgell, A. 1999. *Efficient Algorithms for Sorting and Synchronization*, The Australian National University.