Distributed Data Storage with Strong Offline Access Support

Lukáš Hejtmánek[*]
[*]Faculty of Informatics,
Masaryk University,
Botanická 68a, 602 00 Brno,
Czech Republic
e-mail: xhejtman@mail.muni.cz

Luděk Matyska[*][†]
[†]Institute of Computer Science,
Masaryk University,
Botanická 68a, 602 00 Brno,
Czech Republic
e-mail: ludek@ics.muni.cz

***Abstract*—In this paper, we propose a distributed data storage framework that supports unrestricted offline access. The system does not explicitly distinguish between connected and disconnected states. Its design is based on a lock-free distributed framework that avoids update conflicts through file versioning. The feasibility of this framework is confirmed by a proof-of-concept implementation. We also demonstrate that the proposed lock-free replica synchronization algorithm scales well. A future work will include also direct support for non-versioned files.**

I. Introduction

As the mobility is becoming a more and more important aspect of work pattern of contemporary users, the ways in which data is processed in a distributed system that supports mobility are gaining more practical interest. When considering mobility, we expect mobile clients to be connected to the network from different places. However, as the network is not yet omnipresent and does have very differing properties at different places, we have to consider situations when network is not available—users have to work in a disconnected mode—or network has very limited throughput or very high latency (e.g., when using the GPRS). The distributed data storage must be able to support usual work patterns even in such cases, hiding the actual network quality (or even existence) from the users as much as possible.

The primary goal of our work is to present a distributed data storage system which does not distinguish between connected and disconnected states and in fact operates only in one (disconnected) mode. Our secondary goal is to present a system which does not need to use locking and we demonstrate that this is the needed property to fulfill our primary goal. We propose fast lock-free synchronization algorithm and we avoid update conflicts using file versioning.

Ficus [7] and Coda [12] are prominent examples of systems that introduced the so called disconnected operations. The Coda system distinguishes whether the client is connected or disconnected. The Ficus system supports primarily disconnected mode [8] but uses complex synchronization algorithms and does not support file versioning. File versioning is popular in the programming and text/file editing fields and is usually supported at the application level through tools like CVS [3], SVN [4], or GIT [6]. Recently, file versioning has begun to be popular also in the field of computer graphics, with, e.g., Adobe Version Cue [1] which is an application-level file versioning tool. These systems usually support disconnected operations (e.g., editing a file independently and making explicit synchronization with the server) and their supported modes of operations are very similar to the use of file systems in disconnected state.

The rest of this paper is organized as follows. In Section II, we discuss problems related to data replication, file versioning, and problems related to offline support. Architecture of the proposed framework is presented in Section III. This is followed, in Section IV, by the information about a prototype implementation and first experimental results. Section V summarizes related work and Section VI gives concluding remarks and work summary.

II. Distributed Systems

In this section, we discuss some common problems related to distributed storage systems with replication and offline support and concurrent versions system with replication. We use a distributed system model with a set of participants. Each participant can communicate directly with others. Each participant is either running or down. We allow services and data to be replicated between participants, i.e., services or data are redundantly hosted by multiple participants.

To distinguish individual objects moving in a distributed system, some global unique identification is needed. Three different approaches are usually used for such a purpose: centralized service, peer to peer approach, and standalone. The first two approaches rely on availability of either the central service or the peers at the moment of issuing the unique identifier. This requirement is not valid for our situation when new object can be created in a disconnected state. Therefore, only the standalone approach is usable for offline clients, as it does not need contact with any other participant or third party service.

As a disconnected client re-connects to the network, some synchronization must happen between the client and the distributed system. We will describe here the basics of the well known *Two-Phase Commit Protocol*, as its modification is used in our synchronization algorithm. We have a set of participants that can commit or abort a transaction. If all participants commit then the result is to commit. If any of the participants aborts then the result is to abort. Two-phase commit protocol (2PC) decides whether to abort or to commit. The 2PC consists of two parts. The first, message "prepare to commit" is broadcasted to all participants by one of the participants. If any of the participants does not answer until certain timeout (broadcast message is lost or a participant is down) then the result is abort. After collecting answers with to commit or to abort, the second part is started by broadcasting the result. The second broadcast is supposed to be a reliable broadcast. If initiator of the transaction does not receive acknowledgement of the second phase from any of the participants than it is up to the initiator to retransmit the request of the second phase.

A. Distributed Data Systems and Replication

Large scale distributed systems are prone to failures. If we are given a distributed system consisting of hundreds or even thousands of elements, it is almost certain that some elements are non-operational. If we are to secure a reliable service, the possible way out lies in replication. We can replicate elements or services, in the case of data systems, we replicate storage servers or orthogonally we can replicate stored data. We are using data replication.

Data replication strategies [11] can be divided into two groups. The first, pessimistic replication strategy blocks update operations until the update is spread over all replicas. The second, optimistic replication strategy does not block update operations and spreading is not synchronous with update. Consequently, the pessimistic replication strategy can have performance problems due to blocking operations but it guarantees coherency of data. The optimistic replication strategy does not guarantee coherency of data immediately after update operation but it can be faster then the pessimistic approach.

B. Distributed Storage with Offline Access Support

We adopt a model which consists of online servers and possibly offline clients. The servers are all interconnected, the clients can connect to and disconnect from network at any time. The disconnected clients use prefetch cache to be able to read data and write back cache to store updated data. Write back cache is synchronized with servers after the transition from disconnected to connected state. Write back cache can serve as prefetch cache in the case of reading previously stored data.

1) Update Conflicts: We denote a situation when two or more distinct clients want to update the same data as an *update conflict*. If all clients are online then the update conflict is usually solved by last-writer-wins rule or system avoids update conflicts using data locks.

If the client has updated data while being disconnected then update conflicts may occur after the transition to connected state. In such a case, last-writer-wins rule is ambiguous because the time stamp bound to the update relies on real time clock of the client. However, it is infeasible to synchronize real time clock of all participants in distributed environment with offline support. Moreover, the updates are committed after transition from disconnected to connected state. The time order of commits may not be the same as the time order of updates. For usage of distributed data locking client must not be faulty (including disconnected state) or the client must periodically refresh soft-locks. If we do not bind upper bounds with duration of disconnected state then soft-locks cannot be used, otherwise we have problems with clients that are disconnected for too long.

2) Name Conflicts: Traditional file systems use full file name (i.e., a file name together with a path) as an unique and immutable identification of the file. Consequently, these file systems prohibit the creation of two or more identical full file names for different files.

Introducing disconnected state, system is unable to prevent creation of multiple identical full file names because full file names are client generated. We are unable to check full file names created in disconnected state. The name conflicts may occur after transition from disconnected to connected state if we allow to create and rename files in the disconnected state. Moreover, file creation or file renaming are synchronous operations expecting result state which is unknown until transition to connected state.

C. Concurrent Versions System with Replication

We use model of a file system with versioned files. Besides traditional directory structure, we bind a version number to every file. A single file may have several distinct versions and each file version is immutable. Update made to a particular file version results in a new file version that is further immutable. We extend this model using replication: we use a file with all its versions as an independent replication unit and updates may be performed on any of the replicas.

File replication of immutable files does not pose problem with conflicting updates because every file is unique and once written, it may receive no updates. However, the update conflicts return if we introduce file versioning together with immutable files as versions conflict. Replication algorithm must spread new file versions across replicas and spreading file versions may result in versions conflict.

More precisely, denote a set $F = \{f_1, \ldots, f_n\}$ of versions of a particular file that are spread over all replicas. Assume that version f_{n+1} is created on the replica R_1 and version f'_{n+1} is created on the replica R_2. Both f_{n+1} and f'_{n+1} are versions with the same version number of the same file but they may have different content. We denote such a situation as *version conflict*.

III. Architecture Design

Model of our distributed file system consists of interconnected storage servers and clients that connect and disconnect at their will. We do not distinguish between connected and disconnected clients. As we discussed in the previous sections, disconnected clients cannot use data locking and thus our model avoids data locking completely. The disconnected clients use prefetch cache to be able to read data and write back cache to store updated data. Write back cache is synchronized to servers after the transition from disconnected to connected state. Prefetch and write back cache stores data blocks instead of whole files. File consists of two parts: metadata and data. Data is stored in blocks of variable length, once stored data block is further immutable. The metadata resembles standard UNIX I-Node, as it contains references to particular data blocks, their offsets in the file and lengths. The metadata supports replication of data blocks, i.e., particular offset may be referenced by multiple data blocks. The metadata is maintained in a directory structure. Files can exist in several file versions. Every file version is immutable, an update of a file creates a new file version. We adopt the so called open-close semantics where metadata of a particular file is published to network after the file closing. Consequently, a new file version arises after the file is closed. Every file version is given an UUID (Universally Unique IDentifier, represented by 16 bytes long number) [10] at the time of version creation. Algorithm used for UUID generation gives with high probability globally unique identifiers. A file with all versions forms independent replication unit, every file can be replicated. Replication model embodies multiple master (peer to peer) approach, i.e., no replica has master role, and all replicas are read-write accessible. Each replica is given an UUID. Each replica knows all other replicas. Replication is performed by a storage server.

A. *Update Conflicts*

As we presented in Section II-B.1, systems with offline support may suffer from update conflicts. Our model is based on immutable files. As immutable file cannot be changed, we completely avoid update conflicts.

B. *Name Conflicts*

We discussed in Section II-B.2 that systems with offline support may have problems with name conflicts. As we already mentioned, the file creation and file renaming are synchronous operations expecting result status synchronously but the result status is unknown till transition to the connected state. We use optimistic approach which means that if a new file name is not conflicting with cached file names then it is not globally conflicting. Using this approach, we keep synchronous nature of creating and renaming operations but we do not completely avoid name conflicts. If a conflict occurs after transition to the connected state, we change the conflicting name. E.g., let us assume that offline client creates a file `file.1`. After transition to the connected mode, the metadata of the file `file.1` is stored on metadata manager but let us assume that there already exists a file of the same name. In such a case, client's file `file.1` is renamed to `file.1#1`. Consequently, the client cannot use file names as immutable identifier because system may change the file names without all users notification. In our example, the client may not use `file.1` for the file identification because it was changed to `file.1#1` in background. We resolve such situation by binding globally unique identifier [10] to each file (and a particular version) using which the user is able to access file directly without specifying path and the file name. E.g., we bind UUID `ccb8c47c-709c-40a9-906e-8383aacef173` with `file.1#1`. Using this identifier, the file is always accessible regardless of its actual name. The user can always access the file using the file "name" `?uuid=ccb8c47c-709c-40a9-906e-8383aacef173`. However, using UUID for accessing files is not user friendly and thus we support the use of ordinary file names for accessing files for most cases. In addition, we present *checkpoints* which are natural numbers bound to every file and initially set to zero. If a checkpoint of any file is non-zero then we guarantee that file name (including file version) will not be changed by the system.

C. *Replication*

Using our model, replication is done at two levels: data replication and metadata replication. For data replication, we can easily adopt optimistic replication strategy because our model assumes that stored data blocks are immutable. Consequently, no update conflict can occur. We allow updates of immutable files using file versioning. Replication of versioned files does not pose update conflicts as versioned files are immutable. However, version conflicts as discussed in Section II-C may occur. We solve this problem by proposed replica synchronization algorithm which is presented in the following section.

D. *Replica Synchronization Algorithm*

Our proposed replica synchronization algorithm is based on the well known 2PC algorithm. As we mentioned, a single file with all its versions is an independent replication unit, therefore we can use abstraction of a single file.

For a file F, we denote a set $R_F = \{R_1, \ldots, R_n\}$ as the set of replicas that store the file F. A single file is an independent replication unit and a replication of a single file does not depend on other files. We denote file versions of a single file as a set $V = \{v_1, \ldots, v_m\}$. We denote a set $V_{R_i} = \{v_1^i, \ldots, v_p^i\}$ as the set of file versions that are stored on a replica R_i. The set V_{R_i} does not always contain all the file versions which is a consequence of asynchronous version synchronization. We denote ancestor function $\pi(v_j^i)$.

For each set V_{R_i}, we define a number $C_{R_i} \in \mathcal{N}_0$ and a set $A_i = \{v_j^i \in V_{R_i} \mid j \leq C_{R_i}\}$. We define that $v_j^i = v_j^k$ if and only if the file version v_j^i has the same UUID as the file version v_j^k. We call the C_{R_i} a *checkpoint* if and only if $C_{R_1} = C_{R_2} = \ldots = C_{R_n} \wedge A_1 = A_2 = \ldots = A_n$, i.e., all the replicas are synchronized. Initially, we set $C_{R_1} = C_{R_2} = \ldots = C_{R_n} = 0$. In the following text, $Checkpoint$ denotes single maximal checkpoint, i.e., the maximal C_{R_i} for which holds $Checkpoint = C_{R_1} = C_{R_2} = \ldots = C_{R_n} \wedge A_1 = A_2 = \ldots = A_n$.

We define two operations that are requested by the client and performed by the replica.

1) $Create(R_i)$—creates initial version v_1^i of a file on a replica R_i. Operation fails if initial version v_1^i already exists. We define $\pi(v_1^i) = nil$.
2) $Update(v_j^i)$—creates a new file version derived from a single file of version v_j on replica R_i. Operation $Update(v_j^i)$ on non-existing version v_j^i fails. We define $\pi(Update(v_j^i)) = v_j^i$.

After any of these operations, replica synchronization algorithm must be started and it is asynchronous to these operations. A client may request multiple operations on multiple replicas at once. Consequently, multiple instances of replica synchronization algorithm may be synchronizing a single file. Such a situation must be detected and resolved because only one coordinator for a single file can exists to make this algorithm work correctly.

The set V_{R_i} is built during synchronization or using operations $Create()$ and $Update()$. The set V_{R_i} forms a tree with the root v_1^i using ancestor function $\pi()$.

The replica that has performed operation $Create()$ or $Update()$ is the coordinator of the synchronization algorithm. We remind that each replica knows all the other replicas. If any replica receives requests from different coordinator while synchronization algorithm has not finished then concurrent run of synchronization algorithm is detected. Leader is elected from all concurrent coordinators using UUIDs of replicas—the replica with a maximal UUIDs is the leader. The leader proceeds with the synchronization algorithm. Behavior of the other coordinators depends on performed operations as shown below.

Goal of the synchronization algorithm after operation $Create()$ is to spread newly created version across all replicas. Name collision may occur if initial file version already exists on any replica. In such a case, the newly created file must be renamed. The synchronization algorithm has two parts: (1) replicas lock and (2) file distribution.

1) Coordinator requests all replicas to prevent further creation of the given file. If any of replicas already has the file then coordinator renames the newly created file and terminates. Rename operation actually triggers $Create()$ operation on a different file.
2) Coordinator spreads the newly created file between replicas.

Goal of the synchronization algorithm after operation $Update()$ is to synchronize file versions. In the case of concurrent coordinators, the non-leader coordinators terminate the synchronization algorithm. The synchronization algorithm has two parts: (1) snap-shooting replicas and (2) a synchronization. We assume that coordinator is running from replica R_j.

1) Coordinator of a synchronization obtains sets $B_i = \{v_j^i \in V_{R_i} \mid j > Checkpoint\}$ for all $R_i \in R_F$, i.e., B_i contains all file versions which version number is higher then $Checkpoint$. This is done by sending request by a coordinator to other replicas.
2) Coordinator (running on replica R_j) creates a set V' by merging all the sets B_i (see the Figure 1). All file versions with parents not present in A_j (a set with already synchronized file versions) nor B_i are omitted from merging. The set V' is distributed to all replicas. Each replica computes the new C_{R_i} value according to the set V'. All file versions that are not present in the set V' are given a new version higher then the $Checkpoint$.

```
proc $Merge(Checkpoint, B_1, \ldots, B_n)$
  $B := B_1 \cup B_2 \cup \ldots \cup B_n$
  $B' := \emptyset$
  $V' := \emptyset$
  $x := Checkpoint + 1$
  foreach $v_j^i \in B$ do
    if $\exists v_l^k \in B' \mid v_j^i \doteq v_l^k$
      then
          foreach $v$ such that $\pi(v) = v_j^i$ do
            $\pi(v) := v_l^k$
          od
      else
          $B' := B' \cup \{v_j^i\}$
          $v_x := v_j^i$
          $V' := V' \cup v_x$
          $x := x + 1$
    fi
  od
  $Checkpoint := x$
  $return(V')$
end
```

Fig. 1. Merge operation. We define $v_j^i \doteq v_l^k$ iff the file version v_j^i has the same UUID as the file version v_l^k.

The 2PC algorithm aborts if one of the participants is down (non operational). However, such behavior is problematic in large distributed systems where the probability of any node going unexpectedly down (or being disconnected) is too high. Therefore, we bind a time limit on request in phase 1. Replicas not answering within the time limit are considered to be down. When a crashed replica R_i is operational again then it fetches and merges file versions in the range $(C_{R_i}, Checkpoint >$ from other replicas and then it starts the synchronization algorithm.

To perform the phase 2, more then half of replicas must be operational. If this precondition is not met, then all replicas are unblocked and the synchronization is postponed. This precondition is needed to guarantee that file versions less than checkpoints will not be changed by subsequent merging operations.

IV. Prototype Implementation

Our proof-of-concept implementation splits data storage into two independent parts: data and metadata. The data is stored using logistical networking approach [2]. The metadata is handled by our metadata manager. The metadata manager supports the following operations: `create`, `update`, and `list`. The `create` operation creates initial version of a file and replicates metadata between replicas. Replication is done asynchronously. The `update` operation creates a new version of a given file and runs asynchronously the proposed replica synchronization algorithm. The `list` operation returns a list of files that are stored on a particular replica.

Our prototype implementation is done in C language and provides standard UNIX I/O API. `Open`, `Close`, `Read`, `Write`, and `Seek` operations are supported. We also have a preliminary implementation using FUSE [5] providing a generic file system for the Linux operating system.

Preliminary experiments have been run on several servers equipped with Pentium 4@2.0 GHz processors, 1 GB RAM, and 1 Gbps NIC. The results show good scaling of the synchronization algorithm which can be seen in the Figure 2. The Figure shows duration of the synchronization algorithm which was triggered by operation update. Number of transfered messages is linearly dependent on the number of replicas:

$$messages = replicas * 4 + newversions - 1$$

where the *messages* is total the number of transfered messages, the *replicas* is the number of participating replicas, and the *newversions* is the number of unsynchronized versions of the file. Number of transfered messages is derived from implementation of the algorithm.

We have done some preliminary performance tests which show that extracting the sources of Linux kernel (approx. 250 MB total in approx. 18000 files) is quite comparable to NFSv3.

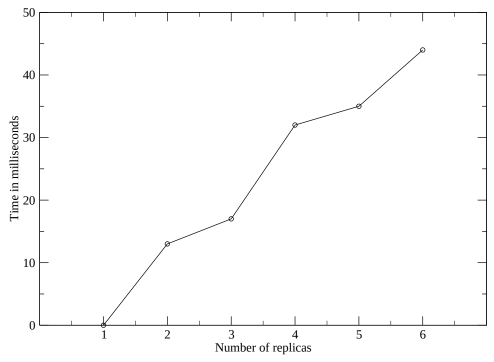

Fig. 2. Scaling of the synchronization algorithm

V. Related Work

We discuss systems that provide either offline support and/or file versioning. The CVS [3] system provides file versioning and offline support, but it is not a distributed system when we consider the way server storage is organized. The GIT [6] is a distributed approach to file versioning similar to the CVS. Instead of simple file versions, it uses hash values to identify particular file versions to simplify distributed design. Users can access particular file versions using the hash values which makes it more difficult than using natural numbers. Natural numbers allow easier identification of particular file versions. Our proposed approach uses natural numbers to identify file versions while preserving distributed approach and using UUID to uniquely identify individual files. The Ficus [7] file system aims to be very large-scale replicated distributed file system, it uses optimistic replication strategy [11] and allows to operate in disconnected mode [8]. However, the Ficus does not provide file versioning, requiring rather complex synchronization algorithm to solve the update conflicts. Another limitation of the Ficus is that it does not support large files as it uses NFSv2 as transport and storage layer. The Coda file system [12] is a heir to AFS file system, it provides full replicas (read/write), provides disconnected operations, and it is also using optimistic replication strategy. Update conflicts are detected and either automatically resolved or reported to the user. However, nor the Coda file system provides file versioning, and it uses leases (which are basically time-limited locks) to maintain cache coherency.

Similar approach has been studied to support disconnected operations also in AFS [9]. It is based on journaling operations performed when connection to a file server is unavailable. When the connection is available, the journal is replayed and possible conflicts are reported to the user. However, this attempt has never been adopted by AFS community.

VI. Conclusions and Future Work

In this paper, we propose a distributed framework capable of file versioning in the way of CVS while being fully distributed and replicated. Together with file versioning, our system possesses strong offline support, i.e., we do not distinguish between connected and disconnected state. We have designed a prototype implementation and performed preliminary experiments. The results show that idea of strong offline support and simple file versioning is feasible and our proposed replica synchronization algorithm scales well. We have done some preliminary performance tests which show that our framework is quite comparable to NFSv3.

Our further work is directed to support work with non-versioned files including algorithms for distribution of updates and conflicts resolution. We relax open-to-close semantics of access to non-versioned files. We also plan to support more operations on the metadata manager to meet the requirements of fully compliant POSIX I/O interface.

Acknowledgments

This research is supported by a research intent “Optical Network of National Research and Its New Applications” (MŠM 6383917201) and by the CESNET Development Fund project 172/2005. We would also like to thank to David Antoš and Petr Holub for kindly supporting our work and for stimulating discussions.

References

[1] Adobe Version Cue. `http://www.adobe.com/products/creativesuite/versioncue.html`.
[2] M. Beck, T. Moore, and J. S. Planck. An end-to-end approach to globally scalable network storage. In *SIGCOMM’02*, 2002.
[3] B. Berliner and J. Polk. Concurrent Versions System (CVS), 2001. `http://www.cvshome.org`.

[4] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version Control with Subversion, 2004. `http://svnbook.red-bean.com/en/1.0/svn-book.html`.

[5] FUSE: Filesystem in Userland. `http://fuse.sourceforge.net`.

[6] GIT – Fast Version Control System, 2005. `http://git.or.cz/`.

[7] Richard G. Guy. Ficus: A Very Large Scale Reliable Distributed File System. Technical Report CSD-910018, Los Angeles, CA (USA), 1991.

[8] John S. Heidemann, Thomas W. Page Jr., Richard G. Guy, and Gerald J. Popek. Primarily disconnected operation: Experiences with ficus. In *Workshop on the Management of Replicated Data*, pages 2–5, 1992.

[9] L. B. Huston and P. Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, pages 1–10, Cambridge, MA, 1993.

[10] P. Leach, M. Mealling, and R. Salz. RFC4122: A Universally Unique IDentifier (UUID) URN Namespace, 2005. `http://www.ietf.org/rfc/rfc4122.txt`.

[11] Yasushi Saito and Marc Shapiro. Replication: Optimistic Approaches. Technical report, HP Laboratories Palo Alto, 2002. `http://www.hpl.hp.com/techreports/2002/HPL-2002-33.html`.

[12] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.