Nonblocking Distributed Replication of Versioned Files

Lukáš Hejtmánek, Luděk Matyska
Institute of Computer Science, Masaryk University, Brno, Czech Republic
CESNET, z. s. p. o., Praha, Czech Republic
Email: xhejtman@ics.muni.cz, 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. We propose an algorithm for replica synchronization. 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.**

***Index Terms*— lock-free distributed replication, disconnected operations, file versioning, conflict avoidance**

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 or satellite links). 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 most operations are done only in one (the disconnected) state with explicit synchronization protocols run after re-connect. 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 [1] and Coda [2] 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 [3] 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 [4], SVN [5], 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 proposed architecture forms a part of a distributed storage framework that is usable as the Storage Elements that has been presented in Grid environments [7]. If a computing job fails, it is usually due to unavailability of the Storage Elements thus the storage systems with offline access support are most suitable here. Additionally, storage systems that incorporate file versioning allow easier resuming of failed jobs that work with the Storage Elements as particular file versions can be found in some defined state that is coherent for all the computing nodes.

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 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 all the others. Each participant is either running or down. We allow services and data to be replicated among 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, and

[1] `http://www.adobe.com/products/creativesuite/versioncue.html`

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 the disconnected state. Therefore, only the standalone approach is usable for offline clients, as it does not need to contact any other participant nor third party service.

As a disconnected client re-connects to the network, some synchronization must happen between the client and the distributed system. Our synchronization algorithm is based on a modification of the well known *Two-Phase Commit Protocol* [8], that is therefore described below. We have a set of participants that can commit or abort a transaction. If all the 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 commit or abort messages, 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.

Beside 2PC, we utilize the well know leader election algorithm [9] that we describe in the following, too. The leader election algorithm is concerned with bringing about a single leader so that everyone knows who is the leader. After each run, a single leader must be selected even if the protocol is initiated by multiple participants. There exist many variants of this algorithm [9]. We use *wave leader election* variant. Each participant is given a label so that all labels are distinct and comparable. When any participant sends any message then his label is included in the message. When a participant (initiator) decides that a new leader needs to be elected than the participant initiates voting. It means that a participant sends a message "I am the leader" to all other participants (election wave). Each participant replies to the initiator with an acknowledgement. The acknowledgement is positive if the participant has not yet received any message "I am the leader" or if the participant received the message "I am the leader" from the initiator with lower label; otherwise the acknowledgement is negative. Initiator that receives no negative acknowledgement and does not sent positive acknowledgement to any other initiator, is considered to be a leader. It is easy to see that participant with the highest label always can be the leader. This algorithm assumes that no messages are lost.

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, 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 use data replication in approach chosen for this work.

Data replication strategies [10] 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 and data availability problems due to blocking operations but it guarantees coherency of data. On the contrary, the optimistic replication strategy does not guarantee coherency of data immediately after update operation but it can be faster than the pessimistic approach and provides highly available data.

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 the network at any time. The disconnected clients use read-ahead 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.

The systems that support parallel read/write access to data must deal with two kind of conflicts: update and name conflicts. These conflicts can be either avoided or resolved after they occur. In the following, we describe these conflicts with their consequences in the systems with offline access support.

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 the clients are online then the update conflict is usually solved by a last-writer-wins rule or the system avoids update conflicts at all by using data locks.

If the client has updated data while being disconnected, the update conflicts may occur after the transition to the connected state. In such a case, the 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 ordering of commits does not need to be the same as the time ordering of updates. For usage of distributed data locking, the client must not be faulty (including disconnected state) or the client must periodically refresh soft-state locks. If we do not impose upper bounds on duration of the disconnected state then the soft-state locks cannot be used; otherwise

we have problems with clients that are disconnected for too long or the soft-state locks being prematurely freed.

2) Name Conflicts: Traditional file systems use a full file name (i.e., a file name together with the path to it) 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.

After introducing disconnected state, the system is unable to prevent creation of multiple identical full file names because full file names are generated by the clients. 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 to get the result of the operation immediately—which is unknown until transition to the connected state.

C. Concurrent Versions System with Replication

We use a 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 with each file version being 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 as a version conflict if we introduce file versioning together with immutable files. Replication algorithm must spread new file versions across replicas and spreading file versions may result in version 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 the version f_{n+1} is created on the replica R_1 and the version f'_{n+1} is created on the replica R_2. Both f_{n+1} and f'_{n+1} are versions of the same file with the same version number but they may have different content. We denote such a situation as the *version conflict*.

III. Architecture design

The model of our distributed file system consists of interconnected storage servers and clients that connect and disconnect arbitrarily. We do not distinguish between connected and disconnected clients. As we discussed in the previous sections, disconnected clients should not 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. Each file consists of two parts: metadata and data. Both data and metadata are stored on the storage servers. Data is stored in blocks of variable length, once stored, each data block is further immutable. The metadata resembles standard UNIX I-Node, as it contains references to the particular data blocks, their offsets in the file and their 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 may exist in several file versions. Every file version is further immutable and an update of the 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 us created after the file is closed. Every file version is given an UUID (Universally Unique IDentifier, represented by 16 bytes long number) [11] at the time of version creation. Algorithm used for UUID generation gives globally unique identifiers with high probability. A file with all its versions forms an 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 and 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 and updates based on file versioning. As immutable file cannot be changed, we completely avoid update conflicts.

B. Name Conflicts

In Section II-B.2, we discussed 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 to be returned 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, file `file.1` submitted by the client is renamed to `file.1#1`. Consequently, the client may not use file names as immutable identifiers because the system may change the file names without notifying all the users. 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 (UUID) with each

file (and its particular version) using which the user may access file directly without specifying the path and the file name. E.g., we bind the 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 the most cases. In addition, we present *checkpoints* which are abstract guarantees on immutability of the file names. We represent the checkpoints as natural numbers bound with 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) is fixed and will not be changed by the system.

C. Replication

Using our model, the replication is done at two distinct 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 conflicts may 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 the replica synchronization algorithm proposed in the following section.

D. Replica Synchronization Algorithm

Our replica synchronization algorithm is based on the well known 2PC algorithm and the wave leader election algorithm (both introduced in Section II). First, a leader is elected using the leader election algorithm and then the leader performs synchronization using the 2PC algorithm. To optimize our algorithm, we compound 2PC protocol messages into election messages as described below. More detailed description together with the proof of correctness can be found in [12].

A single versioned file with all its versions is replicated independently of all other files, therefore we can use abstraction of a single versioned file. Further, we suppose that each file version is given system generated name (UUID) which is globally unique. Using this UUID, we can distinguish file versions albeit having different version but having the same content (thus they also should have the same version number).

For a versioned file f, we denote a set $R_f = \{R_1, \ldots, R_n\}$ as the set of replicas that store the versioned file f. A single versioned file is an independent replication unit and a replication of a single versioned file does not depend on other versioned files. We denote file versions of a single versioned file as $f : \{1, \ldots, m\}$. We denote ${}_{R_i}f : \{1, \ldots, p\}$ as a set of file versions that are stored on a replica R_i. The set ${}_{R_i}f : \{1, \ldots, p\}$ does not always contain all the file versions which is a consequence of asynchronous version synchronization.

We define that ${}_{R_i}f : k = {}_{R_j}f : k$ if and only if the file version ${}_{R_i}f : k$ has the same UUID as the file version ${}_{R_j}f : k$. We define that ${}_{R_i}f : \{1, \ldots, n\} = {}_{R_j}f : \{1, \ldots, n\}$ if and only if for all $k \in \{1, \ldots, n\}$ it holds ${}_{R_i}f : k = {}_{R_j}f : k$. We define a $Checkpoint \in \mathcal{N}_0$ such that for all i, j it holds ${}_{R_i}f : \{1, \ldots, Checkpoint\} = {}_{R_j}f : \{1, \ldots, Checkpoint\}$. Checkpoint is *maximal* if there is no such $C > Checkpoint$ for which holds that $\forall i, j \ {}_{R_i}f : \{1, \ldots, C\} = {}_{R_j}f : \{1, \ldots, C\}$. In the following text, $Checkpoint$ denotes maximal checkpoint.

We define two operations that are requested by the client and performed by the replica. We also denote ancestor function $\pi(v_j^i)$, this ancestor function is used to track history of particular file version (i.e., for each file version, we can easily see its ancestors).

1) $Create(R_i)$—creates initial version ${}_{R_i}f : 1$ of a file on a replica R_i. The file name is automatically changed if initial version ${}_{R_i}f : 1$ already exists.
   We define $\pi({}_{R_i}f : 1) = nil$.
2) $Update({}_{R_i}f : j)$—creates a new file version derived from a single file of version ${}_{R_i}f : j$ on replica R_i. Operation $Update({}_{R_i}f : j)$ on non-existing version ${}_{R_i}f : j$ fails.
   We define $\pi(Update({}_{R_i}f : j)) = {}_{R_i}f : j$.

The set ${}_{R_i}f$ is built during synchronization or using operations $Create()$ and $Update()$. The set ${}_{R_i}f$ forms a tree with the root ${}_{R_i}f : 1$ using ancestor function $\pi()$.

When we want to synchronize a versioned file, we run replica synchronization algorithm. Algorithm is not required to start immediately after $Create()$ and $Update()$ operations but we start it immediately to spread updates as fast as possible. First, we elect only a leader that proceeds with synchronization and then we run the two-phase synchronization. Leader election is necessary to ensure that one instance of the synchronization algorithm is only running while synchronizing a single file. We allow to run multiple instances of synchronization algorithm for different files (not a different version but a file with a different name).

The traditional 2PC algorithm aborts if one of the participants is down (non-operational) thus all participants must be operational for 2PC algorithm to proceed. Similarly, leader election algorithm supposes that all communication channels are reliable. These properties are usually not met in real world. Therefore, we have modified leader election algorithm to work even with lossy channels. We have also modified 2PC algorithm to work with non-operational participants. As we use only leader election to elect participant that is allowed to proceed the synchronization algorithm, we do not require that all participants know who the leader is, they only must not be able to became the concurrent leader.

a) Leader election modification: We have a set of participants, each participant is labeled, all the labels are distinct and comparable. We can use, e.g., UUIDs for the labels. Each participant knows all other participants.

Whole leader election algorithm is related to a single file and for any different file (a file with different name) another instance of leader election algorithm may be running and elect possible different leader. Initiator (A) of election spreads a message "I am the leader" (voting message). The voting message contains identification of the initiator and time limit within which the voting message is valid. When a participant receives a voting message then there exist several possible scenarios:

- A participant has not yet received any message in this voting or the received message is not valid any more (due to the time limit). In this case, the participant replies "True".
- A participant has already received message from initiator with *higher* label. In this case, the participant replies "Cancel".
- A participant has already received message from an initiator (B) with a *lower* label. In this case, the participant sends revocation message to the initiator (B). The initiator (B) decides whether he resigns or not and replies "True" or "Cancel", resp. The participant forwards the reply from the initiator (B) to the initiator (A).

We allow some participants to be crashed or disconnected from a network. Such participants do not reply to a voting message. Initiator that has acquired "True" replies from majority of participants and no "Cancel" reply is the leader. Majority means strictly more than a half of all participants (we remind that each participant knows all other participants). It is easy to see that only one participant is able to acquire majority of "True" replies. Voting is finished when an initiator collects replies from all running participants. When the initiator does not want to be the leader any more—typically after performing synchronization algorithm—he sends his resignation to all participants. When participant has given "True" reply to any initiator then the participant does not start voting until she receives resignation of the leader or time limit bound with voting message expires (regarding a single file).

If the elected leader crashes during leader election then his voting messages eventually expire and another leader election may be started. In such case, there is no leader until next leader election. This means that in this case, our synchronization algorithm is postponed until the next leader election.

b) Two-phase Commit modification: We bind timeout with any message sent using this algorithm. If participant does not reply within this timeout then it is considered to be non-operational. Our modification of 2PC is based on majority approach. If initiator of 2PC collects no "Abort" reply and the "Commit" reply from strictly more than a half of all participants then the result is to commit, otherwise the result is to abort. We require majority here to be able to do global agreement which cannot differ from agreement of a small group of separated participants that were also allowed to proceed 2PC. It is guaranteed that there can exist only one majority group.

At the beginning, the synchronization algorithm checks whether another instance of synchronization is running on the same replica synchronizing the same file. If another instance is detected than after the $Update()$ operation, the new instance of synchronization is terminated and after the $Create()$ operation, the new instance renames the name of versioned file and starts again. Synchronization algorithm continues in two parts: (1) leader is elected and the first phase 2PC synchronization is done, (2) synchronization is performed if leader election was successful and then the leader sends her resignation to all participants. Resignation is sent even in case of unsuccessful leader election to release votes. Participant agrees to resign to be the leader only if he performs phase one of 2PC. If participant performs phase two of 2PC then he must refuse to resign. Our proposed synchronization algorithm has two slightly different separate parts. One part of the algorithm is run after the $Create()$ operation and the other part is run after the $Update()$ operation. We describe both parts separately.

Goal of the synchronization algorithm after the $Create()$ operation 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) election wave together with verifying that file version does not exist on any of replicas and "locking"[2] the replicas so that file of this name cannot be created, (2) file distribution and "lock" release.

1) Coordinator (A) verifies whether the given file name does not locally exist and it atomically locks the local replica so that the given file name cannot be created any more. Than it sends election wave together with *CreateRequest* to all other replicas. If any replica refuses to lock (due to the already existing lock or the given file already exists), the coordinator renames the file and restarts the synchronization algorithm. If the coordinator has collected majority of replies from election wave and has received no cancel reply then it proceeds to phase 2 of the synchronization algorithm. Otherwise, it releases all the locks, renames the file and restarts the synchronization algorithm.
   A replica that receives multiple *CreateRequests* (let us suppose that another *CreateRequest* is from a coordinator (B)), behaves as follows: if an UUID of a coordinator (A) is lower then the UUID of the coordinator (B) then the replica replies *Cancel*; if the UUID of the coordinator (A) is higher then the UUID of the coordinator (B) then the replica asks the coordinator (B) whether its lock can be overridden. If the lock cannot be overridden then the replica replies *Cancel*. The coordinator (B) allows

[2]The term "locking" means that a replica is only "locked" to prevent concurrent synchronization.

lock overriding if it is in phase 1; otherwise it denies lock overriding.

2) At the beginning of phase 2, coordinator checks whether any replica has requested him to release lock, if so then it releases the lock and restarts the synchronization algorithm in phase 1. Coordinator spreads the newly created file between replicas and releases locks. It may happen that spread file already exists on some replica. This can be only unsynchronized file as a product of Create synchronization algorithm. In this case, the already existing file on target replica is renamed to non-conflicting name. The non-conflicting name is non-conflicting only within particular replica. It is not necessarily globally non-conflicting. Renaming operation converges as each name conflict means that one concurrent run of renaming operation has succeeded. The rename operation starts another instance of synchronization algorithm using the different file name.

Goal of the synchronization algorithm after the $Update()$ operation is to synchronize file versions. In the case of concurrent coordinators, the leader coordinator proceeds and the others terminate. The synchronization algorithm has two parts: (1) election wave together with snap-shooting replicas with "locks" and (2) a synchronization and "lock" releasing. We assume that coordinator is running from replica R_a.

1) Coordinator (R_a) checks whether another coordinator has not already locked this replica, if so then it terminates. In phase 1, the coordinator sends election wave together with *ObtainUpdateSet* request. This request returns for all $R_i \in R_F$ sets $B_i = {}_{R_i}f : \{Checkpoint + 1, \ldots, n\}$, i.e., B_i contains all file versions with a version number higher then $Checkpoint$. If any of replicas has replied *Cancel* then it means that another instance of synchronization algorithm is running and the coordinator releases all its locks and terminates.
   A replica that receives multiple *ObtainUpdateSet* (let us suppose that another *ObtainUpdateSet* is from a coordinator (R_b)), behaves as follows: if an UUID of a coordinator (R_a) is lower then the UUID of the coordinator (R_b) then the replica replies *Cancel*; if the UUID of the coordinator (R_a) is higher then the UUID of the coordinator (R_b) then the replica asks the coordinator (R_b) whether its lock can be overridden. If the lock cannot be overridden then the replica replies *Cancel*. The coordinator (B) allows lock overriding if it is in phase 1; otherwise it denies lock overriding.
2) If coordinator has collected replies from the majority of replicas and none is *Cancel* then it proceeds to phase 2. At the beginning, the coordinator checks whether any replica has requested him to release lock, if so it releases the lock and terminates. The coordinator (running on the replica R_j) creates a set $f : \{Checkpoint + 1, \ldots, n\}$ by merging all the sets B_i (see the Figure 1). All file versions which parents that are not already synchronized nor are in the set B_i, are omitted from merging. The set $f : \{Checkpoint + 1, \ldots, n\}$ is distributed to all replicas. If distribution has succeeded to the majority of replicas then the coordinator sets the new $Checkpoint$ to all replicas. All file versions that are not present in the set $f : \{Checkpoint + 1, \ldots, n\}$ are given a new version higher then the $Checkpoint$.

```
1  proc Merge(Checkpoint, B_1, ..., B_n)
2    B := B_1 ∪ B_2 ∪ ... ∪ B_n
3    B' := ∅
4    V' := ∅
5    x := Checkpoint + 1
6    foreach R_i f : j ∈ B do
7      if ∃R_k f : l ∈ B' | R_i f : j ≐ R_k f : l
8        then
9              foreach v such that π(v) = R_i f : j do
10               π(v) := R_k f : l
11             od
12       else
13             B' := B' ∪ {R_i f : j}
14             f : x := R_i f : j
15             V := V ∪ f : x
16             x := x + 1
17     fi
18   od
19   Checkpoint := x
20   return(V)
21 end
```

Figure 1. Merge operation. We define ${}_{R_i}f : j \doteq {}_{R_k}f : l$ iff the file version ${}_{R_i}f : j$ has the same UUID as the file version ${}_{R_k}f : l$.

When a crashed replica R_i is operational again then it obtains current *Checkpoint*. Then it renames all unsynchronized file versions to be beyond the current global *Checkpoint*. This step is required because unsynchronized versions below the *Checkpoint* will be overwritten in the next step. In the following step, the crashed replica fetches and merges file versions that are missing between the last synchronized version and the global checkpoint. After this operation, it may happen that the set ${}_{R_i}f$ contains ${}_{R_i}f : j$ and there exists ${}_{R_i}f : k$ such that $k > j$ and the ${}_{R_i}f : j$ has the same UUID as the ${}_{R_i}f : k$. Then the ${}_{R_i}f : k$ is removed from the set ${}_{R_i}f$. (I.e., the file versions in the set ${}_{R_i}f$ have distinct UUIDs, the file versions with duplicate UUIDs are removed.) This case is a result of a crashed replica that has contained unsynchronized file version which has been synchronized by other replicas meanwhile. We remind that if two files possess the same UUID then through this property the files have been marked by client as the same. At this point, the set ${}_{R_i}f$ contains only items with different UUIDs. And finally, it starts the synchronization algorithm. This procedure is required because the crashed replica could assign new version numbers to already synchronized (by other online replicas) file versions. As we have mentioned,

once synchronized file version may not change its version number.

When a coordinator of the synchronization algorithm crashes then her locks eventually expires and another instance of the synchronization algorithm may run. It could happen that an update synchronized by this crashed coordinator was the very last and no further update will occur. It means that the last update will not be synchronized. To solve this problem, each replica periodically search its unsynchronized file versions and starts the synchronization algorithm.

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 [13]. 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 of metadata handling that utilizes our proposed replica synchronization algorithm, is done in Java language and provides API for metadata storage, retrieval, and update.

Preliminary experiments have been run on several servers equipped with two Pentium 4@3.0 GHz processors, 3 GB RAM, and 1 Gbps NIC. In our testbed, we evaluated latency of file create and update distribution. We evaluated only a single file create distribution because according to our algorithm, file create distribution is never postponed. We evaluated latency of the file create distribution for 2 to 10 nodes, the results are shown in Figure 2. Contrary to file create distribution, the file update distribution can be postponed therefore, we evaluated several numbers of the postponed update distributions. In particular, we evaluated latency of the file update distribution for 2 to 10 nodes with 1 to 160 pending updates at each replica, the results are shown in Figure 3. In both Figures, we can see that synchronization scales well. The number of nodes corresponds to the number of replicas of a file and we do not expect that there will ever be significantly more replicas.

Number of transfered messages after *update* operation is linearly dependent on the number of replicas:

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

where the *messages* is the total number of transfered messages, the *replicas* is the number of participating replicas, and the *newversions* is the number of unsynchronized versions of the file. The number of transfered messages is derived from implementation of the algorithm and closely follows its proposal which means that our implementation does not send significantly different number of messages.

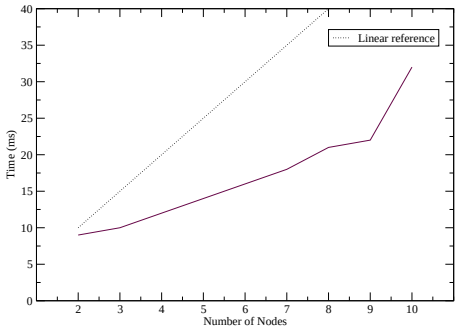

Figure 2. Latency of file create distribution among specified number of replicas with outlined linearity.

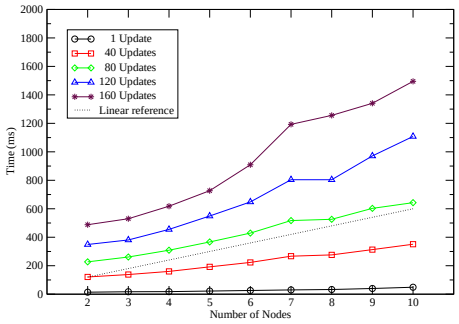

Figure 3. Latency of file update distribution among specified number of replicas with outlined linearity.

V. Related Work

We discuss systems that provide either offline support and/or file versioning. The CVS [4] 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 [1] file system aims to be very large-scale replicated distributed file system, it uses optimistic replication strategy [10] and allows to operate in disconnected mode [3]. 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 [2] is a heir to the 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 [14]. 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] R. G. Guy, "Ficus: A Very Large Scale Reliable Distributed File System," Los Angeles, CA (USA), Tech. Rep. CSD-910018, 1991.

[2] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[3] J. S. Heidemann, T. W. P. Jr., R. G. Guy, and G. J. Popek, "Primarily Disconnected Operation: Experiences with Ficus," in *Workshop on the Management of Replicated Data*. IEEE, 1992, pp. 2–5.

[4] B. Berliner, "CVS II: Parallelizing software development," in *Proceedings of the USENIX Winter 1990 Technical Conference*. Berkeley, CA: USENIX Association, 1990, pp. 341–352.

[5] M. Pilato, *Version Control With Subversion*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2004.

[6] L. Torvalds, "GIT: Fast Version Control System," 2005, http://git.or.cz/.

[7] EGEE: Site Access Control Architecture DJRA3.2. 2005. https://edms.cern.ch/document/523948.

[8] S. Mullender, Ed., *Distributed systems (2nd Ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993.

[9] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[10] Y. Saito and M. Shapiro, "Replication: Optimistic Approaches," HP Laboratories Palo Alto, Tech. Rep. HPL-2002-33, 2002.

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

[12] L. Hejtmánek, "Distributed Storage Framework with Offline Support," 2007, PhD. Thesis, Masaryk University, Brno, Czech Republic.

[13] M. Beck, T. Moore, and J. S. Plank, "An end-to-end approach to globally scalable network storage," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 339–346, 2002.

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

Luděk Matyska is an Associate Professor in Informatics at Faculty of Informatics, and he also serves as a vice-director of Institute of Computer Science, both at Masaryk University in Brno, Czech Republic. He got a PhD in Chemical Physics from Technical University Bratislava, Slovakia. His research interests lie in the area of large distributed computing and storage systems, with a specific emphasis on their management and monitoring. He also works in high speed network applications, with a specific emphasis on collaborative work support and use of all these technologies in various e-learning activities. He lead national Grid infrastructure projects and participates in several EU funded international projects including the EGEE and the CoreGRID Network of Excellence.

Lukáš Hejtmánek graduated in computer science and got PhD in Informatics from Faculty of Informatics, Masaryk University in Brno, Czech Republic. His main research interests include: high speeds networks, network data storage and peer to peer storage, distributed storage with client–server semantics, data replicas, and replicas management. Beside data storage systems, his research interests also include HD video conference tools and high performance and parallel computing.