A dual-time vector clock based synchronization mechanism for key-value data in the SILENUS file system

Max Berger
MTA KFKI

Michael Sobolewski
Texas Tech University

***Abstract*— The SILENUS federated file system was developed by the SORCER research group at Texas Tech University. The federated file system with its dynamic nature does not require any configuration by the end users and system administrators.**

The SILENUS file system provides support for disconnected operation. To support disconnected operation a relevant synchronization mechanism is needed. This mechanism must detect and order events properly. It must detect also possible conflicts and resolve these in a consistent manner.

This paper describes the new synchronization mechanism needed for providing data consistency. It introduces dual-time vector clocks to order events and detect conflicts. A conflict resolution algorithm is defined that does not require user interactions. It introduces the switchback problem and how it can be avoided. The synchronization mechanisms presented in this paper can be adapted to synchronize any key-value based data in any distributed system.

I. Introduction

Under the sponsorship of the National Institute for Standards and Technology (NIST) the Federated Intelligent Product Environment (FIPER) ([1], [2], [3]) was developed (1999-2003) as one of the first service-to-service (S2S) metacomputing environments. The Service-Oriented Computing Environment (SORCER) ([4], [5]) builds on the top of FIPER to introduce a federated computing environment with the basic service infrastructure to support service object oriented metaprogramming. [6] It provides an integrated solution for grid computing.

SORCER provides a centralized File Store Service (FSS) [7]. It supports filtering out information from remote files, thus reducing the amount of data transfers between providers. However, it is usually provided as a single service in the network and as such not a true S2S application. To improve reliability and performance replication services [8] where developed for SORCER. These services allow the replication of file data on different nodes in the computing grid.

SILENUS [9] [10] [11] completes the step from a traditional client-server application to a network-centric application. Instead of storing data on one particular node or in a particular service, it is the federation of several small distributed services that provide the file system. SILENUS provides a true data grid solution to complement a SORCER computing grid. Unlike existing file store solutions, the SILENUS system is completely network-centric. It allows access to files in the file system even when a node is disconnected from the rest of the network, but still keeps consistent data. This paper discusses the synchronization mechanisms needed to provide required data consistency.

This paper is organized as follows: Section I introduces the SILENUS file system and its main services and gives an overview over the data to be synchronized. Section II refreshes on the context of time-vector clocks. Section III introduces a new dual time vector clock algorithm. Section IV describes our approach to conflict resolution. The findings are summarized in section V.

A. The SILENUS File System

The SILENUS file system provides a grid data storage solution using loosely coupled replicated services. Each service is run independently on any number of machines. Services can discover themselves dynamically. These services federate together to provide one storage system to the user. Two of these services are byte store and metadata store. We assume that metadata about files is relatively small, with respect to large content of files.

A byte store service stores the actual file data. In the basic hardware analogy this would be the actual hard drive. Files in a byte store are identified uniquely by the ID of the byte store and an entry ID in the byte store. These ID numbers never change. This makes the file storage independent from file metadata such as the file name. The byte store services provide nothing but support for file storage. The advantage is that this service can be then optimized for performance.

A metadata store provides attributes for the files stored in a file system. By analogy to a traditional storage system a metadata store can be considered itself as the file system. The metadata information creates the well-known hierarchical structure. Files in the metadata store are identified by universally unique identifiers (Uuid). The metadata provides mapping from and to file names.

Metadata stores are synchronized while connected. All metadata stores contain the same information. Should a metadata store be disconnected while its information changes, it will be resynchronized when it discovers the other operational metadata stores after the reconnection. This paper describes the synchronization mechanism.

Other notable services are the SILENUS facade service and the legacy adapters. The SILENUS facade service provides an

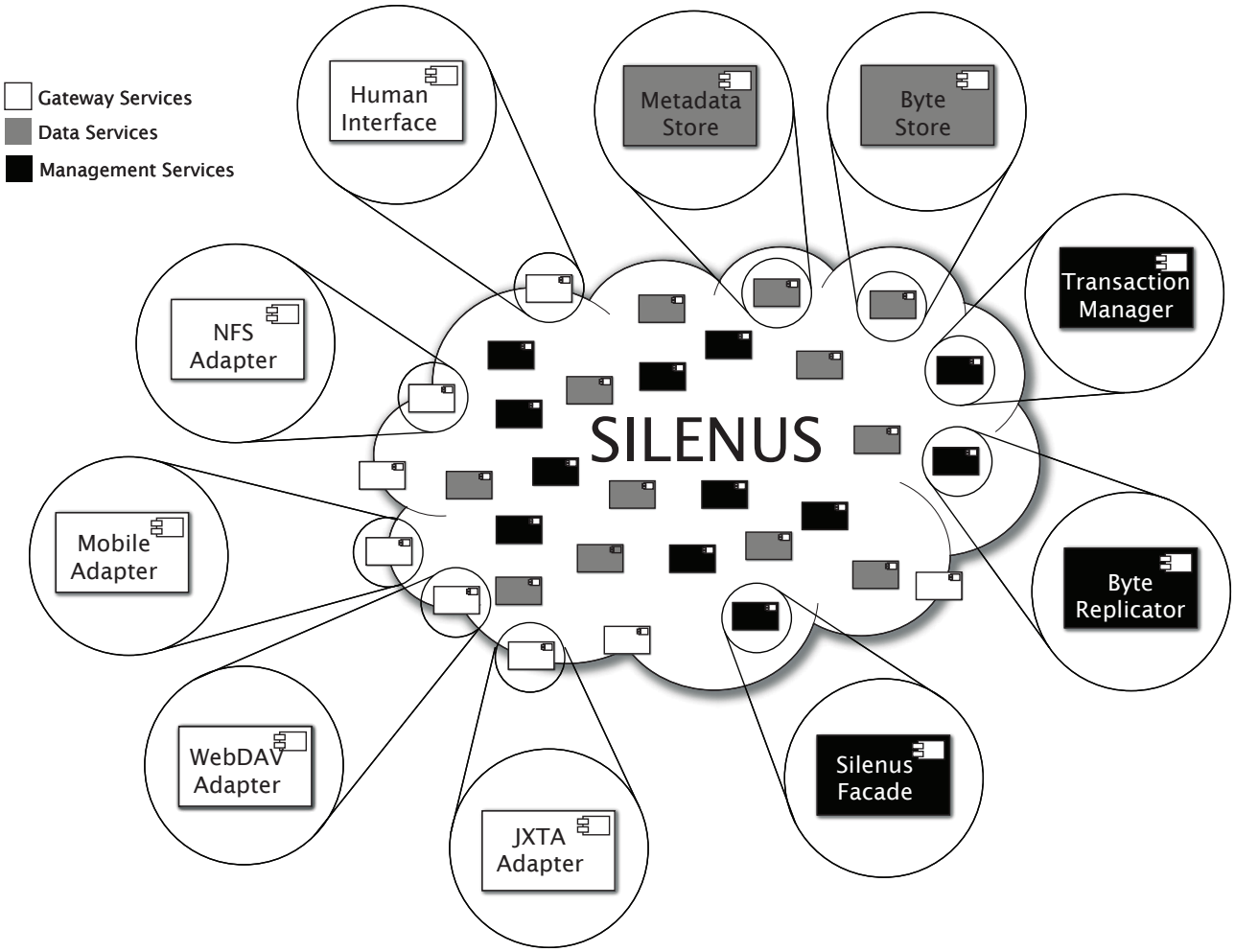

Fig. 1. SILENUS architecture.

entry point into the SILENUS system. Instead of accessing the metadata stores and byte stores separately, a requestor application can contact a facade service, which will setup the S2S internal communication via smart proxying. To provide scalability, there may be any number of facade services in the network. The legacy adapters provide access to the SILENUS file system through well-defined existing file system protocols, such as NFS or WebDAV. The full SILENUS architecture is depicted in Figure 1.

While traditional file systems have the luxury of using a locking system to prevent concurrent access from occurring, the SILENUS file system can not employ locks. When a lock is requested, the information must be sent to all nodes, therefore requiring all participating nodes being always available and reachable. One of the main properties the SILENUS system, however, is disconnected operation: Users are free to modify files no matter if they are connected to the main system or not. Instead of preventing concurrent modifications, the system expects them and resolves them with the synchronization mechanism presented in this paper.

B. Synchronization requirements

As stated in the description of the metadata stores, all metadata stores should contain the same information. Whenever a change event occurs, the information is sent to all connected nodes, which can then update their own state. However, as the system is supposed to support disconnections, not all metadata stores may be available for updates when a change occurs. In this case, the change information cannot be sent immediately to these disconnected nodes. Therefore, a node must be brought up-to-date with the current information when it is reconnected with the rest of the system. A synchronization mechanism must:

- detect what needs to be synchronized,
- merge data on all nodes, and
- resolve synchronization conflicts.

C. Metadata contents

To describe which data needs to be synchronized, it must first be looked at what type of data is stored in the metadata

Key	Value	Since
Uuid	`1234-5678`	
Name	`test.txt`	2
Mime-Type	`text/plain`	1
Parent	`2345-6789`	1
Last changed on	`3456-7890`	2

Fig. 2. Example of Metadata in SILENUS Metadata Store. The file "test.txt" is a plain-text file. It has the internal Uuid of "1234-5678". The Uuid of its parent directory is "2345-6789". The metadata was last changed on the metadata store with the id "3456-7890" at the global time "2".

stores, as synchronization mechanisms for different type of data will be different.

The file metadata is stored as key-value pairs. Figure 2 gives an example of file metadata structure in SILENUS. Thus, we assume that the synchronized data is represented as key-value pairs. Each pair contains additional information at which time it was last changed. Older information is kept as well to provide versioning.

Every SILENUS metadata store keeps two counters called local time counter and global time counter. The global time counter is incremented whenever any metadata change occurs. This change may have originated locally, or on a remote metadata store. This provides a mechanism to trace all the modifications on this particular metadata store. The global counter therefore counts every change event in the whole file system. Whenever an event occurs that originated at this local metadata store, both the local and the global time counter are incremented. The local time counter therefore counts only the events that occurred at this particular metadata store.

To support synchronization, the global time counter of the last change and the information where the last change occurred is stored along with the file metadata. The global time of the last change is needed to recreate change-log information while the local time counters are used for conflict resolution. It is assumed that only the attributes with a time-stamp larger than a given time-stamp have changed. The last-changed-on information is stored to solve the switchback problem which will be shown later in this paper.

II. Time Vector Clocks

In a distributed system, a notion of time is mandatory to describe the current state of the overall system. To perform synchronization, it is essential to provide a notion of ordered time that provides a "happened before" relationship. A notion of time is needed to define an order of events, create a change log of events, and decide which events to apply. Time must fulfill the following requirements:

- every event e is annotated with a time t_e,
- t must be strictly monotonically increasing,
- $t_{e_1} < t_{e_2}$ iff e_1 occurred before e_2,
- $t_{e_1} > t_{e_2}$ iff e_1 occurred after e_2,
- $t_{e_1} = t_{e_2}$ iff e_1 and e_2 describe the same event on the same node,
- $t_{e_1} || t_{e_2}$ (parallel) iff e_1 and e_2 occurred at the same time on different nodes.

The native approach for time is to use the real time clock present in most machines. However, it does not provide a good enough notion of time. First, the time may not be accurate. To be useful for synchronization it would have to match exactly on all nodes involved. Thus, it does not provide an adequate notion of events occurring in parallel.

One possibility to provide time is to have a global clock. Every node would get the time from the global clock. It would then be easy to find out which events occurred in the past and which events are newer and can safely update older events. Unfortunately, this would require all nodes to keep the exact same time and reconnect to the global clock often enough. This assumption can be made for a reliable local network, but it is impossible to hold it up for disconnected nodes.

Instead of using a global absolute clock a logical clock is used. A logical clock is a monotonically incrementing software counter, starting at zero. It is required that there is at least one clock tick between two events. If all events are timestamped, it becomes possible to reconstruct the order in which events occurred. This works very well for a single node, but shows limitations when applied to a distributed system. [12]

To order events in a distributed system, a timestamp on the local process is not sufficient. Every event needs to be timestamped with the global time from every node. This can be implemented with vector clock timestamps. A vector clock contains the logical clock for every connected component. [13]

Unfortunately, a system with fully working vector clocks would need a reliable observer. In a truly distributed system this is impossible. Instead, the vector time is approximated with the best knowledge of a system. In a vector clock system, each node keeps the knowledge of its own logical clock and the logical clocks of all its peers. This clock vector is appended to all network messages. Other nodes can then use this information to update their own vector clock and to check if the received message was current. Vector clocks can be used to provide total ordering of events in a distributed system.

The vector clock approximation algorithm is defined as follows:

- If a change event occurs locally, increment the own clock.
- If a change event is received from another node, set each clock to the maximum of the clock received and increment your own clock by one.

Using this algorithm we can now compare vector clocks and define an absolute ordering of events. We will compare the received time vector V_r with the local time vector V_l. $n, m \in [1..|V|]$ are used as counter for the elements of V.

- If $V_r \neq V_l \wedge \forall n : V_r(n) \geq V_l(n)$ then the received message is newer than the local state. Receive all change events from the remote node and apply them accordingly.
- The case $V_r \neq V_l \wedge \forall n : V_r(n) \leq V_l(n)$ is not possible. A metadata store will increment its own clock before sending out events, therefore at least the clock component $V_r(sender)$ must be greater than the component stored at the receiver side.
- If $\exists m, n : V_r(n) > V_l(n) \wedge V_r(m) < V_l(m)$ then some events have happened in parallel. In this case, the

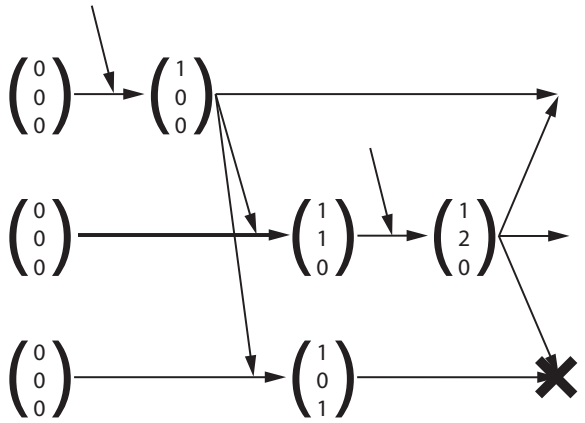

Fig. 3. Vector clock approximation problem. In this particular example, a change event is generated on the first node. It increments its own clock and sends information about the change event to all other nodes. These nodes update to the change and increment their own time vector. Another change event occurs on the second node. The node increments its time vector accordingly and notifies about the change all other node. The first node will just apply the changes. The third node, however, assumes that a conflict occurred as its logical clock time-stamp is greater than its time-stamp received.

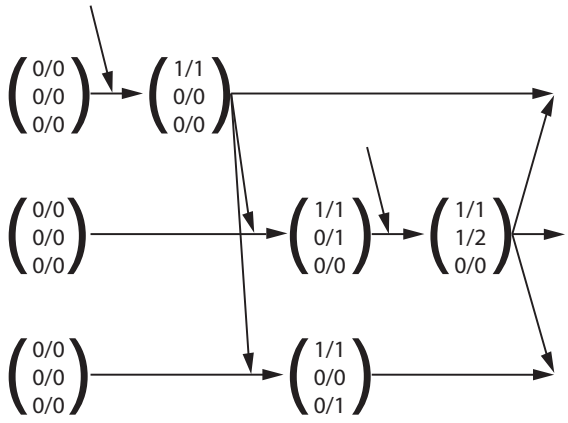

Fig. 4. Dual vectors with local and global counter. A change event is generated on the first node. It increments both counters and sends information about the event to all other nodes. These nodes apply the change and increment their own global counter. Another change event occurs on the second node. The node increments both counters and notifies all other nodes. Both other nodes will see this new event as newer and apply corresponding non-conflicting changes.

receiving metadata store needs to retrieve all events from the sending metadata store and merge the contents. A potential conflict has occurred that must be resolved.

The problem with this vector time clock approximation algorithm is that it does not keep accurate track of the actual changes, but rather of the messages received. According to the algorithm the own logical clock is incremented every time a change event is received. If this time vector is then sent to a third party, the party could not distinguish if the time was increased because a new local event occurred or because another remote node indicated a change. Figure 3 gives a relevant example. We therefore needed to enhance the existing vector-clock algorithm in order to manage proper synchronization of changes in distributed nodes.

III. Dual Time Vector Clocks

To solve the conflict detection problem with single-time vector clocks we use a novel dual time system introduced for the metadata stores to keep track of the changes in the key-value attributes. A local timer counts only events that originated locally, whereas the global timer counts both local and external change events. The time vector now contains two pieces of information for all nodes: timing of local and external changes.

Using only the local time counters in the time vector clock algorithm seems to solve the conflict detection problem. If the original time vector clock algorithm is used with local time counters instead of global time counters, fewer false conflicts are detected. A potential conflict notification only occurs, when there where actual changes originating on different nodes. However, the global time counter is crucial for correct ordering of events. Avoiding the global time counter will mark most events as parallel, although they may have actually occurred in a specific order. To resolve this problem, both the local and global time counters are used in the time vector. The local time counter component is used for conflict detection, while the global time counter component is used to order the change events.

When comparing the dual-time vector clocks, only the local components of the time vector are compared for conflict detection. We will denote the time vector for the local clocks as Vl. The following for possibilities can be considered:

- If $Vl_r \neq Vl_l \wedge \forall n : Vl_r(n) \geq Vl_l(n)$ then the received message is newer than the local knowledge. All changes can be safely applied and the time vectors merged by setting each component, both local and global, to the maximum of the current value and the received value.
- If $Vl_r \neq Vl_l \wedge \forall n : Vl_r(n) \leq Vl_l(n)$ then the received message is older than the local knowledge. It can be safely ignored, the changes have been applied earlier. The time vectors are still merged, some global counters may be different. The originating node can be notified that it should update.
- If $\forall n : Vl_r(n) = Vl_l(n)$ both nodes have the same information. The time vectors are still merged, some global counters may be different.
- $\exists m, n : Vl_r(n) > Vl_l(n) \wedge Vl_r(m) < Vl_l(m)$ then some event have happened in parallel. A potential conflict occurred that must be resolved.

This mechanism leads to a lower rate of false conflict detection than a single vector clock. Figure 4 illustrates the same sequence of change events as presented for a single vector clock in Figure 3. Since no actual conflict occurred, none is really detected.

A. Properties of the Dual Time Vector Clock algorithm

Since the dual-clock time vectors are a new proposed solution, we have to prove its properties. The requirements for time given in Section II were defined as follows:

- every event e is annotated with a time t_e,
- t must be strictly monotonically increasing,
- $t_{e_1} < t_{e_2}$ iff e_1 occurred before e_2,
- $t_{e_1} > t_{e_2}$ iff e_1 occurred after e_2,
- $t_{e_1} = t_{e_2}$ iff e_1 and e_2 describe the same event on the same node,
- $t_{e_1}||t_{e_2}$ (parallel) iff e_1 and e_2 occurred at the same time on different nodes.

Each of these required properties is now investigated. For each property, local events and remote events have to be investigated. The dual-clock time vector can without loss of generality be defined as follows:

$$V = \begin{pmatrix} l_1/g_1 \\ \vdots \\ l_{self}/g_{self} \\ \vdots \\ l_n/g_n \end{pmatrix}$$

To prove that t is strictly monotonically increasing it must be shown that $V_{new} > V_{old}$. The proposed algorithm states that in the case of a local event both the local and global clock of the current system have to be increased, therefore:

$$\begin{aligned} V_{new} &= V_{old} + \begin{pmatrix} \vdots \\ 1/1 \\ \vdots \end{pmatrix} \\ &= \begin{pmatrix} l_1/g_1 \\ \vdots \\ l_{self}/g_{self} \\ \vdots \\ l_n/g_n \end{pmatrix} + \begin{pmatrix} \vdots \\ 1/1 \\ \\ \vdots \end{pmatrix} \\ &= \begin{pmatrix} l_1/g_1 \\ \vdots \\ l_{self}+1/g_{self}+1 \\ \vdots \\ l_n/g_n \end{pmatrix} \end{aligned}$$

We can immediately see that $\forall n : Vl_{new} >= Vl_{old}$ and $V_{new} \neq V_{old}$. Therefore the requirement $V_{new} > V_{old}$ is satisfied.

The second case is the case of received remote events. There are four subcases:

1) $V_r = V_l$. In this case, no event has happened; V_l does not have to increase.
2) $V_r < V_l$. In this case, the received event is older; V_l does not have to increase.
3) $V_r > V_l$. In this case, the received event is newer; V_l must increase.
4) $V_r||V_l$. In this case, events have happened in parallel; V_l must increase.

In the subcases 3 and 4 V_{new} is defined as:

$$\begin{aligned} V_{new} &= max(V_{old}, V_r) \\ &= \begin{pmatrix} max(l_{old,1}, l_{r,1})/max(g_{old,1}, g_{r,1}) \\ \vdots \\ max(l_{old,self}, l_{r,self})/max(g_{old,self}, g_{r,self}) \\ \vdots \\ max(l_{old,r}, l_{r,r})/max(g_{old,r}, g_{r,r}) \\ \vdots \\ max(l_{old,n}, l_{r,n})/max(g_{old,n}, g_{r,n}) \end{pmatrix} \end{aligned}$$

The use of the max function satisfies the condition $\forall n : Vl_{new} >= Vl_{old}$. $V_{new} \neq V_{old}$ follows directly from the subcase selection, would $Vnew = V_{old}$ then sub case 1 would have been selected. This proves that the dual vector clock algorithm satisfies the requirement: t must be strictly monotonically increasing for every event.

The next property that must be proven is that $t_{e1} < t_{e2}$ iff e1 happened before e2. This property is a direct result from t being strictly monotonically increasing for every event.

The properties $t_{e1} > t_{e2}$ iff e1 happened after e2 and $t_{e1} = t_{e2}$ iff e1 and e2 are the same event on the same machine also follow directly from the property that t is strictly monotonically increasing.

The property $t_{e1}||t_{e2}$ iff e1 and e2 happen at the same time on different machines can be proven as follows: Assuming two nodes N1 and N2 have the same vector V_{start} at some point:

$$V_{start} = \begin{pmatrix} l_1/g_1 \\ \vdots \\ l_{N1}/g_{N1} \\ \vdots \\ l_{N2}/g_{N2} \\ \vdots \\ l_n/g_n \end{pmatrix}$$

After two events happened in parallel on both machines, the time vectors for nodes N1 and N2 will be:

$$V_{N1} = \begin{pmatrix} l_1/g_1 \\ \vdots \\ l_{N1}+1/g_{N1}+1 \\ \vdots \\ l_{N2}/g_{N2} \\ \vdots \\ l_n/g_n \end{pmatrix}$$

$$V_{N2} = \begin{pmatrix} l_1/g_1 \\ \vdots \\ l_{N1}/g_{N1} \\ \vdots \\ l_{N2}+1/g_{N2}+1 \\ \vdots \\ l_n/g_n \end{pmatrix}$$

When comparing the two time vectors, we find that $V_{N1,l}(N1) > V_{N2,l}(N1)$ and $V_{N1,l}(N2) < VN2, l(N2)$. Given the definition of parallelism these vectors are correctly detected as $V_{N1}||V_{N2}$.

The other direction is to provide that if $V_{N1}||V_{N2}$ then there must be events on more than one host. This can be proven by looking at the algorithm: The only time the own local clock increases is if an event happened locally. Therefore, if more than one local clock has changed, there must be events that happened on more than one host.

The dual-clock time vector algorithm still supports all the properties that where required originally. It can therefore provide a reliable order of events for a synchronization mechanism.

IV. Conflict resolution

We now have a sophisticated mechanism to detect changes and potential conflicts. The next step to data synchronization is to use this information to resolve the actual conflicts. But first, it must be determined if the potential conflict is an actual one.

Even if there is a potential conflict, in most cases will not be an actual one. The ordering of events using the dual vector clocks only gives the indication that two or more events have happened at the same time.

These events must be related to each other in order to create an actual conflict. In the case of SILENUS metadata, events are only related if they apply to the same metadata node. Change events applied to different nodes can therefore be updated without any problems. Some events make no sense together, such as deletion of a directory and creation of a new file in the same directory, but they do not create conflicts.

The relation can even be specified more exactly on a field basis. If two different fields on the same node have changed, these changes can be merged without conflicts. For example, if a file is renamed on one node, and modified on another, both changes are not in conflict with each other.

The last-changed-on metadata time-stamp should never create any conflicts. This data is updated every time any data on any node changes. As such, it would always lead to a conflict. However, this data is only used for conflict resolution and is ignored for conflict detection.

If none of these criteria match, then an actual conflict has occurred. To solve a conflict, the SILENUS system uses virtual duplication. Virtual duplication addresses the issue of local consistency and requires no direct user interaction.

	start time	
MS 01 contains:	`bla.txt`	version 1
MS 02 contains:	`bla.txt`	version 1
	metadata stores get disconnected	
	file gets modified on store 01	
	file gets modified on store 02	
MS 01 contains:	`bla.txt`	version 2/01
MS 02 contains:	`bla.txt`	version 2/02
	stores get reconnected	
MS 01 will show:	`bla.01.txt`	version 2/01
	`bla.02.txt`	version 2/02
	`bla.txt`	→ `bla.01.txt`
MS 02 will show:	`bla.01.txt`	version 2/01
	`bla.02.txt`	version 2/02
	`bla.txt`	→ `bla.02.txt`

Fig. 5. Example of virtual duplication. The File `bla.txt` gets modified on two different meta stores (MS), or two different sets of meta stores.

An automatic conflict resolver should require no user interaction. If a file is modified in multiple places, the system should be able to provide a conflict handling strategy. This strategy should not require user interaction. In most environments it is impossible or impracticable to ask the user which conflicting option to choose. Conflict resolution must therefore be handled automatically.

One issue with automatic conflict management is that it can break local consistency. As an example, assume a change may be made to a local metadata store. This metadata store is then synchronized with another metadata store where a conflict occurs. The users on both metadata stores expect their own action to take precedence over the conflicting action from the other user.

The Coda distributed file system introduced virtual duplications. [14] They are used in the Coda file system to resolve conflicts between two versions of the same file with its updated file content. In SILENUS this method is not only applied to file content, but it is also applied to all changes in file metadata. Changing the file content adds a new version and therefore triggers a change in file metadata. But other changes in file metadata are also possible and they may need to be resolved.

Virtual duplication provides a file under three different names: A name with the appended `version` to the files depending on which store it was modified on. It will also provide the file under its original name as a soft link pointing to the original version created on this particular node. Figure 5 illustrates a related example.

One hardship with independent synchronization is a switchback problem. Figure 6 gives a relevant illustration. The switchback problem occurs if two distinct stores contain the same information and then synchronize and merge at the same time with two other stores containing another set of information. If they resolve the conflict differently then they again create different versions, which lead to a follow-up conflict.

This problem can be solved if both metadata stores resolve a

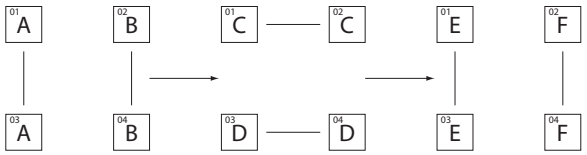

Fig. 6. The switchback problem. Stores 01 and 03 have information A. Stores 02 and Store 04 have information B. Then these stores get disconnected. Store 01 connects with store 02, detects a conflict and resolves it by creating information C. Store 03 connects with store 04, detects a conflict and resolves it by creating information D. The stores get disconnected again and reconnected in the original configuration. Now store 01 and store 03 resolve their conflict by creating information E, where as stores 02 and 04 resolve their conflict by creating information F. And the same process starts over…

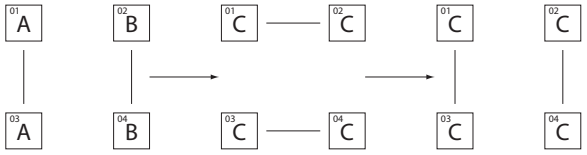

Fig. 7. Solving the switchback problem. Stores 01 and 03 have information A. Store 02 and Store 04 have information B. Then these stores get disconnected. Store 01 connects with store 02, detects a conflict and resolves it by creating information C. Store 03 connects with store 04, detects a conflict and resolves it by creating the exact same information C. Now the stores can disconnect and reconnect in any configuration without a conflict.

conflict in the exact same manner and arrive at the exact same solution. This requires two conditions to be satisfied when using virtual duplication: The file names must be exactly the same and the file generated Uuids must be exactly the same. Figure 7 shows an example of the solved switchback problem.

The first issue is that the names of the files must be exactly the same. In the algorithm outlined above, the id of the synchronizing metadata store is used as an extra file name. This is insufficient, as it leads to different file names depending on the nodes involved in the synchronization. Instead, the id of the node that has last changed the metadata is used. This information is stored in the last-changed-on attribute.

Generating the same new ids is the second issue. To provide support for this, the original id is extended with the last-changed-on information. However the link is kept to the original id. The two conflicting versions will get the original id with the last-changed-on information appended. An id may have multiple last-changed-on informations appended, since there may again be a conflict in one of those files. Figure 8 shows an example of the complete conflict resolution process.

V. Conclusion

The algorithm utilizing dual-time vector clocks presented in the paper can be applied to synchronizing any replicated key-value based data. It supports disconnected and concurrent data modification. The algorithm has been successfully deployed in the SILENUS federated file system and it is the core functionality required to satisfy disconnected operations in the SORCER federated meta computing environment.

Merging and managing changes of key-value based data on disconnected nodes without user interaction is a never-ending topic. There are many applications for this algorithm. The presented federated data-grid storage is just one of them. Other applications include synchronizing personal information, such as address books and calendars with mobile devices. [15]

The algorithm proposed in this paper satisfies all requirements of the SILENUS data grid system. It provides a stable, autonomic conflict resolution mechanism that is only applied if necessary. It has complete and consistent support for dynamically federating services and changing overlay networks.

References

[1] M. Sobolewski, “Federated P2P services in CE environments,” in *Advances in Concurrent Engineering*. A.A. Balkema Publishers, 2002, pp. 13–22.

[2] ——, “FIPER: The federated S2S environment,” in *JavaOne, Sun’s 2002 Worldwide Java Developer Conference*, San Francisco, 2002, http://sorcer.cs.ttu.edu/publications/papers/2420.pdf.

[3] R. Kolonay and M. Sobolewski, “Grid interactive service-oriented programming environment,” in *Concurrent Engineering: The Worldwide Engineering Grid*. Tsinghua Press and Springer Verlag, 2004, pp. 97–102.

[4] S. Soorianarayanan and M. Sobolewski, “Monitoring federated services in CE,” in *Concurrent Engineering: The Worldwide Engineering Grid*. Tsinghua Press and Springer Verlag, 2004, pp. 89–95.

[5] SORCER, “Laboratory for Service-Oriented Computing Environment,” Mar. 2007, http://sorcer.cs.ttu.edu/.

[6] M. Sobolewski, “Metacomputing with Federated Method Invocation,” SORCER Technical Report SL-TR-11, July 2007. [Online]. Available: http://sorcer.cs.ttu.edu/publications/papers/FMI.pdf

[7] M. Sobolewski, S. Soorianarayanan, and R.-K. Malladi-Venkata, “Service-oriented file sharing,” in *CIIT conference (Communications, Internet and Information Technology)*. Scottsdale, AZ: ACTA Press, Nov. 2003, pp. 633–639.

[8] V. Khurana, M. Berger, and M. Sobolewski, “A federated grid environment with replication services,” in *Next Generation Concurrent Engineering*, ISPE. Omnipress, 2005.

[9] M. Berger, “SILENUS – a service oriented approach to distributed file systems,” PhD Dissertation, Texas Tech University, Department of Computer Science, Dec. 2006.

[10] M. Berger and M. Sobolewski, “SILENUS – a federated service-oriented approach to distributed file systems,” in *Next Generation Concurrent Engineering*, ISPE. Omnipress, 2005.

[11] ——, “Lessons learned from the SILENUS federated file system,” in *Complex Systems Concurrent Engineering*, G. Loureiro and R. Curran, Eds., ISPE. Springer, 2007, pp. 431–440.

[12] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[13] F. Mattern, “Virtual time and global states of distributed systems,” in *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989.

[14] M. Satyanarayanan, “Coda: A highly available file system for a distributed workstation environment,” July 15 1999. [Online]. Available: http://citeseer.ist.psu.edu/239688.html; http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/wwos2.pdf

[15] M. Berger, “Integrated PIM data management with SyncML,” Master’s thesis, Technische Universtät München, Munich, Germany, June 2001. [Online]. Available: http://max.berger.name/research/syncml/

Key	Value	Since
Uuid	1234	
Name	test.txt	2
Mime-Type	text/plain	1
Last changed on	5678	2
Content-Version	1	2
Location	2345: 6789	2

(a) Original Data

Key	Value	Since
Uuid	1234	
Name	test.txt	2
Mime-Type	text/plain	1
Last changed on	5678	2
Content-Version	2	3
Location	2345: 7890	3

(b) Change on MDS 5678

Key	Value	Since
Uuid	1234	
Name	test.txt	2
Mime-Type	text/plain	1
Last changed on	7890	2
Content-Version	2	3
Location	3456: 90AB	3

(c) Change on MDS 7890

Key	Value	S
Uuid	1234.5678	
Name	test.5678.txt	4
Mime-Type	text/plain	4
Last changed	5678	4
Cont-Version	2	4
Location	2345: 7890	4

(d) Merged data from 5678

Key	Value	S
Uuid	1234	
Name	test.txt	2
Mime-Type	link	4
Last changed	5678	4
Link-To	1234.5678	4

(e) Merged link

Key	Value	S
Uuid	1234.7890	
Name	test.7890.txt	4
Mime-Type	text/plain	4
Last changed	5678	4
Cont-Version	2	4
Location	3456: 90AB	4

(f) Merged data from 7890

Fig. 8. Complete conflict resolution process. 8(a) shows the original file with the Uuid 1234. 8(b) and 8(c) show the same file, modified on two different metadata stores. A conflict occurs and is resolved by MDS 5678. It will now show both versions (8(d), 8(f)) and a link to its original version (8(e)).