Article

Data History in Decentralized Systems

Datum	Author	Bemerkungen
05.03.08 22:31:04	Bruno Feurer	Publish

Inhaltsverzeichnis

1 Introduction

How do you manage the history of data in an open, decentralized system, like for example the Internet?

CVS, Subversion, WebDAV or versioning file systems help you to remember old versions on one specific server. GIT, Mercurial and others let you do that in a decentralized, less dependent way, but the data needs to be managed in individual projects.

In documentation systems, like Wiki servers, CMS systems, data history is seen as a key feature. Even MS Word allows you to track changes to some extend. These histories are managed centralized and scoped on one document only.

In this article I'm going to analyze different aspects of the data history in a decentralized system and try to model a corresponding concept.

The first chapter defines some concepts in Data History and models the relations between them. The second chapter lays out some essential topics about Decentralized Data Management with influence on data history. In the Use Case chapter we find ourselves in the position of a user, who has concrete expectations on a decentralized data history. The last chapter concludes with some summarizing insights.

2 Data History

The history tells us how the data has evolved and how it looked at any point in time. It collects changes over time.

2.1 Scope

Here "data" can mean one specific entity, a collection of entities, a tree or any other structure of information. The corresponding history naturally models a similar structure.

Conceptually there is one history for all the data within a system. In the way we split a huge set of data into smaller, more manageable objects, we also want to split this one big history into smaller pieces. Like the smaller data objects, their histories are still connected with each other, but managed separately.

2.2 Changes

A change is a set of actions, that have manipulated the data. In a corresponding transaction all of these actions are performed or none at all (atomicity). So conceptually a change can be seen as one action.

Such a change is always based on a certain state of the data and transforms it into another one. It also provides all the information needed to reverse this transformation.

2.2.1 Actions

Actions that change the data of the system can include insert, update, delete, replace, copy, move, merge, rename, migrate, etc. While, for example, a merge action creates a new revision, a management action, like undo, redo or undelete, doesn't actually change the data and thus, is not included in the data history.

A change action should be informative in order to reconstruct the history as accurately as possible. In addition to the basic actions, like insert, update and delete, some higher level actions can provide additional, valuable information. For example a "move" action includes the relation to the source entity and thus is more informative than a separate insert and delete action.

2.2.2 Diff Algorithm

When the change is defined with the resulting, new revision instead of a set of actions, the system needs to calculate the difference and to transform it into a set of insert, update and delete actions.

2.3 Revisions

A revision is a state of the data between changes.

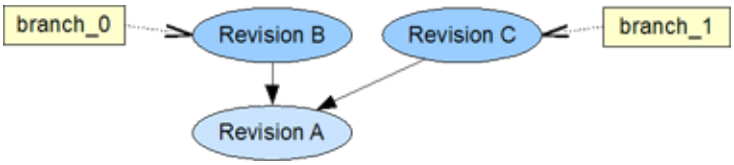
Every revision knows the identity of its immediate ancestor revision, usually referred to as its parent.

The head revision is the newest revision of the data, the one not serving as a parent for another revision. There can be multiple head revisions in a history, one for each branch.

2.4 Branches

A change to one revision produces a new revision, then a next change another one and so on. Such a chain of revisions can be thought of as branch in a tree.

If two separate changes applied to one revision produce two new revisions, we see a new branch forking from the existing one at the last common revision.



Two revisions, the same parent.

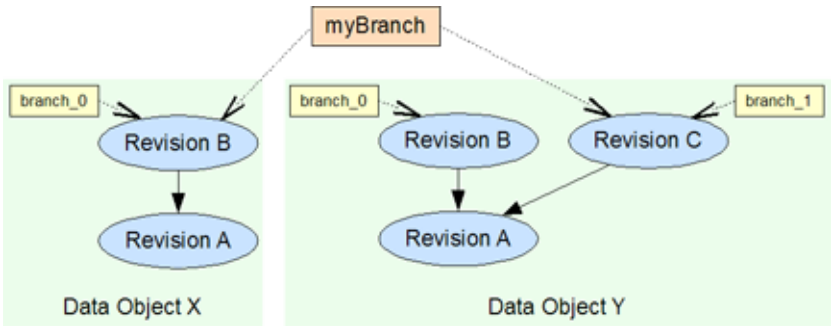
2.4.1 Anonymous Branch

In certain cases, like concurrent edits, the system needs to generate a new branch implicitly.

Such an anonymous branch has a generated ID and only covers the history of one data object.

2.4.2 Named Branch

To model a more enfolding, conceptual branch we can explicitly define a branch name. We then assign this name to an implicit branch in an object history.

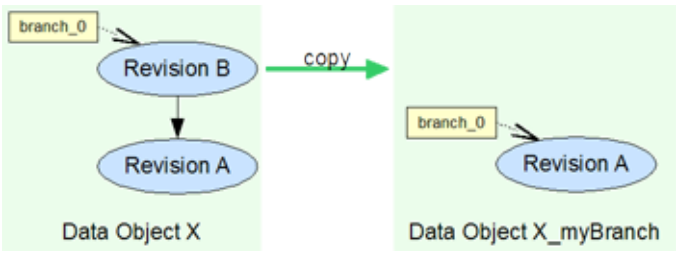


"myBranch" names two branches in two data objects.

A branch name can be assigned to branches in the histories of multiple objects through out the system, one branch per object. It can be seen as a named collection of implicit branches.

2.4.3 Copy

A copy of an object revision into a new object also forms a kind of a named branch.



Copy the data object to work on X_myBranch separately.

While such a copy is easier to address (just like any other object), as a branch, it covers the copied object only.

The independent history of the new object does not directly reflect any changes of the original data object anymore.

2.5 Forward Only

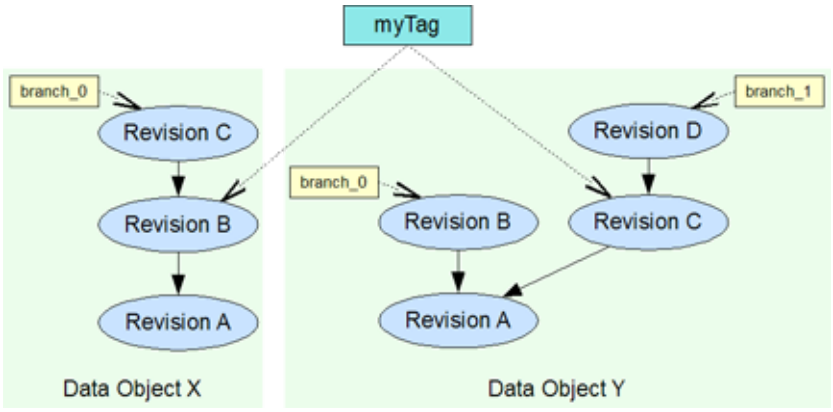
A revision is a snapshot of the data at a certain time. It can never be deleted or removed. Instead the data will be marked as deleted to see the history completed.

In case of a classic "undo" we want to reverse our last change. Still that change is part of the history and an undo only moves the head pointer one revision backwards. Future changes will branch from that older state.

Also "undelete" and "redo" are simple head pointer movements.

2.6 Tag

Like a revision, a history tag also represents a specific snapshot of the data. But as a conceptual marker it is not bound to a certain time and it can represent the state of multiple data objects. As such it can be seen as a collection of multiple revisions, one per data object at most.



"myTag" label for two data objects in a specific revision.

Similar to an explicit branch it can be explicitly defined and named. While an explicit branch definition points to the head revision of a certain branch in a history, a tag references a certain, fixed revision.

3 Decentralized Data Management

In a decentralized system multiple, individual systems connect to one, virtual concept. In comparison to a centralized system, no participant depends on a particular, central instance. An instance connects with others, they again connect with new ones and/or with each other. None of them plays a special role in the system.

An instance in such a system is independent from the system as a whole. Except for the shared data needed, it can run on it's own with it's complete feature set. It can connect, disconnect and reconnect to others at any time.

Such systems are also referred as "distributed" data management systems. But also centralized systems can be distributed and so the term "decentralized" seems to be more accurate.

In this article we focus on different aspects of decentralized systems with the data history in mind. Topics like scalability, reliability, flexibility, performance or system control don't directly influence the data history and are not discussed in this context.

3.1 Global Identifier

When two instances communicate, they need to identify the objects they are talking about. A globally unique identifier is needed.

Since in a decentralized system is no global, central instance to manage the data, every system needs to be able to generate such an ID independently.

3.1.1 Universally Unique Identifier (UUID)

Also called a Globally Unique Identifier (GUID), a UUID is a generated 128-bit value based on MAC addresses, time value and/or random numbers. Available algorithms offer a simple, practical way to independently generate unique identifiers.

Implementations can depend on hardware availability, like network cards, system clock, random number generator,.... Even though the chances for two equal identifiers are very low and practically insignificant, the uniqueness is not guarantied.

3.1.2 Content Hash

The identifier is based on the data itself. An MD5 or SHA-1 hash of the content can be used to identify an object.

Since such an ID depends on the content, it must not be changed without loosing it's identity. So objects with a content hash ID are read-only.

In some situations a content based identifier is not enough. For example a copy produces two objects with the same content and therefore sharing the same identifier. To distinguish such a copy, a content hash needs to be combined with a kind of context or another type of ID, like a managed URI (file names) or a time stamp.

In other situations, when two objects will be created independently with an equal content, in the same context, they are the same and content hash would identify both instances correctly.

3.1.3 Managed URI

At least some parts of an Uniform Resource Identifier (URI) must be managed by a central authority. Building hierarchical contexts the management is broken down into independent environments.

For example the domain names form the first part of an URL in the Internet and is globally managed in multiple levels. Then the paths, the second part of an URL, are managed within the different domains independently.

This type of global ID is the only one, guarantied to be unique, if managed correctly.

3.1.4 Time

The current time is a globally shared value and is often used to identify an object. Since two instances can create two IDs at exactly the same time, even in the exact same nanosecond, the uniqueness of this approach is limited. Another problem arises with the difficulty of measuring the time accurately enough.

The time serves as part of the UUID or can be combined with a content hash (except for two copies of the same object within the same nanosecond).

3.1.5 Position

Another globally unique value would be the exact position, the system is placed at. Because this value is very unpractical to measure exactly and systems can change position independently, this approach is more of a theoretical nature.

With the GPS technology available, we can measure the position pretty accurately. For example in combination with time or an locally managed ID, this value could be a simple way to identify a physical server running a system instance.

3.1.6 Combine

In a decentralized system we should combine these approaches to form the best suitable identification possible.

For example we start with a managed URI model, borrowed from the internet (domainName.com/...), to identify data object domains. Then we handle a more locally managed URI for the data object paths and apply a content hash to it's revisions.

While the managed URI for the data objects is human readable and guarantied to be unique, the content hash identifies two identical, but independently created revisions as one.

To identify a concrete physical system instance we use a UUID or, if available, a position/time based ID. We don't want to restrict our physical setup by any centralized management.

3.2 Nobody Has Got It All

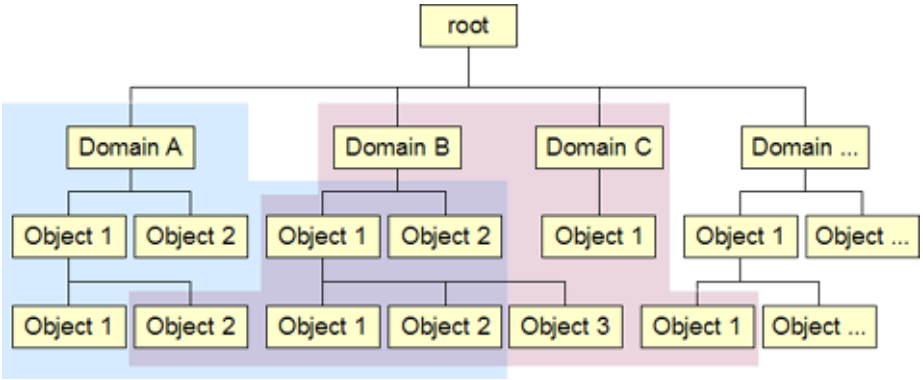
No system instance holds the complete database. Even if it would be theoretically possible for one instances to store all the data available, it wouldn't know it does.

As a result, a process cannot expect complete data sets. No transaction can instantly update all the possible storages, no query is able to return all possible results.

Even though the Google search engine, as an example, is able to list a huge collection of query results, it cannot return all the possible Internet pages, matching the search criteria.

3.3 Sharing Pieces

The complete, conceptual database of the system is formed by every participant holding a piece of the data. Like a cache this piece covers the topics the instance is most interested in.



The data storage of two system instances overlap.

Some parts of the data is stored only in the instance managing, while others are distributed redundantly. For backup and availability purposes it is recommended to share the data with multiple servers.

To comfortably share these redundant parts we need a process to automatically two-way synchronize the corresponding data.

3.4 No Locks

Since no instance is in control of the whole system, it is only hardly possible to request an exclusive lock on some specific data in a practical amount of time.

While within the system only optimistic locking is applicable, it can offer support to explicitly organize and manage a serialized workflow.

3.5 Security

Where in a centralized system access to all the data is controlled at one place, in a decentralized system the data and it's access control are distributed.

3.5.1 Closed System

Like centralized systems, also decentralized instances can be managed within certain "physical" boundaries. A big company could connect multiple servers to form the system and allow notebooks, also running an instance, to connect only to this setup.

These "physical" boundaries don't need to be real physical. With private connections (via Virtual Private Networks (VPN),...) these systems can be built geographically distributed and still being closed.

3.5.2 Open System

In an open, decentralized system architecture anybody could hold a copy of your private data. Even though some foreigner may not have access rights to do so, he could directly manipulate the data, working around any access control.

So private data must be encrypted before being placed in an untrusted storage. The decryption keys can be shaved explicitly and must be managed carefully.

4 Use Cases

From the user's perspective several use cases are related to data history and decentralized systems. The conceptional flow of action within these use cases leads into a basic feature set for data history management systems.

4.1 Reference an Old Revision

In some statements, for example about the history itself, we need to refer to a specific revision of the document.

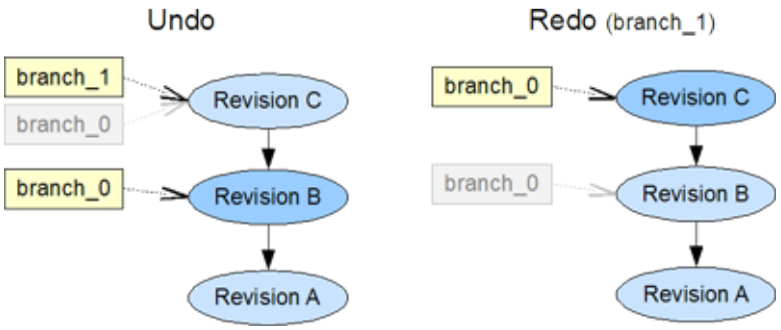
...in the old version, you were talking about $1000, whereas in the actual one, you ask for $1050!

Statement with links to different revisions of an object.

We need to be able to reproduce and reference every revision of the content.

4.2 Undo / Redo

Sometimes we are a little too quick with changing data and want to reverse our last change. Still working in the same branch, we simply move our head pointer one revision backwards. The system automatically creates a new branch for the undone revision.

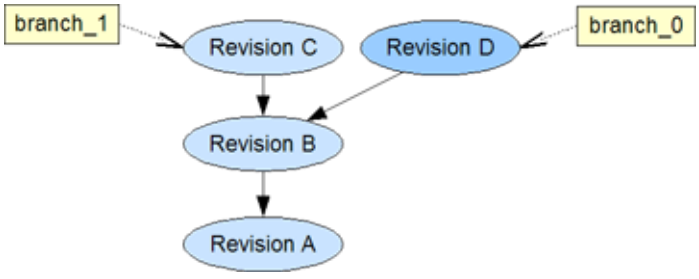


Undo and redo the revision C.

Later, we change our mind and are convinced, our undone change would still be good. So we want to redo it. The system moves the head pointer one revision along the previously created "undo" branch and reassigns the current branch to it.

After an undo, followed by a redo the history looks the same again.

A new revision, created after an undo, forks from the working branch and the automatically created "undo" branch stays in the history.



New revision D after undoing revision C.

A redo will not be possible directly anymore. We would have to explicitly switch to the "undo" branch.

4.3 Manual Undo / Redo

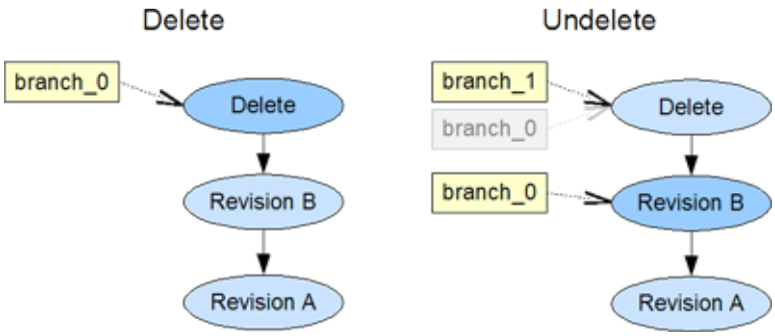
In some situations we "undo" a change by remembering the old state and simply update the content to reflect this older state again.

As a simple example we create a new chapter in a document, talking about an additional thought we are having... - no this thought does not fit in here, lets delete the chapter again... => no change in the document.

In the data history tree all the changes are represented in the different revision in a normal flow. No branch is created, no pointer is moved. There are only two revisions containing the same data.

4.4 Undelete

A data object can easily and quickly be deleted. Sometimes too quickly. Fortunately the history of the object is still available and we can move back to the last revision.



Delete and undelete revision B.

The delete mark is still part of the history in an automatically created branch.

4.5 Select a Branch

To work on an existing branch, we retrieve it's head revision. A change to this revision lets the selected branch grow accordingly.

A change to a head revision inherits the branch from it's parent, while changes to non-head revisions create their own branch.

4.6 Prepare a Draft

Sometimes we are not yet sure about the changes we are planning to make. We would like to forge a separate branch for our draft revisions.

When we base our draft on a non-head revision, a separate branch is created automatically. But to base it on a head revision, we need to tell the system, not to grow the branch of the parent revision, but to create and fork a new one.

"Null" change to fork at the head revision A of branch_0.

To fork at the head revision we can use a "null" change to duplicate a revision to form a new head. A null change does not change the content in any way and can be replaced by an actual change later.

4.7 Assign a Branch Name

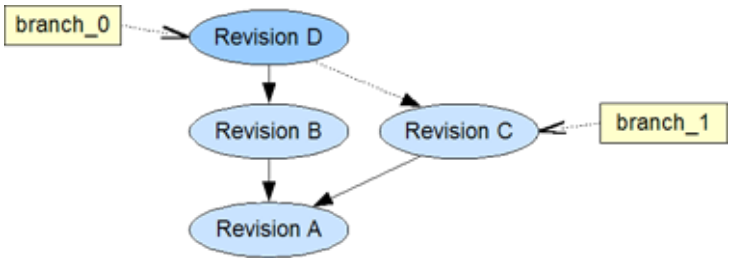
In our draft we are working on multiple data objects. We need a conceptual branch representing the different branches in the corresponding histories.

First we choose a unique name for our draft branch. Then we assign this name to the chosen anonymous branches within the histories of the involved data objects.

4.8 Merge two Branches

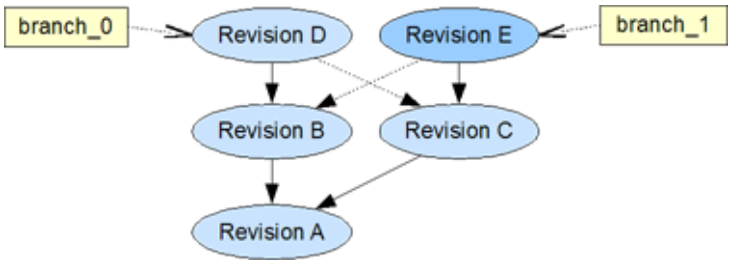
After we are satisfied with the work on our draft branch we like to merge the changes into a new revision of the original branch.

While a branch can be a source of a merge action, it is not actually merged together with another branch. The two branches still exist after the merge. The head of the merge target contains a combination of the two (merge result) and the merge source remains unchanged.



Revision C merged with revision B results in revision D.

The target branch could become the source of a following merge into the source branch of the previous merge.



Revision B merged into revision C results in revision E.

The resulting revisions of both merges don't necessarily need to be identical. In a manual merge, for example, one could favor some details in one branch.

4.9 Tag and Comment Certain Revisions

A specific revision of the data could mean something special to the user. He wants to comment such a revision and lets the readers know, what's so special about it.

We also want to comment a certain state of the data covering multiple objects. We can label or tag the revisions of the data objects to represent this state with a name and a comment.

With such a name tag we can retrieve the according state of the data objects as one consistent snapshot.

4.10 Edit Concurrently

Two authors read the same revision of an object. Both of them edit the data and commit it into the system.

The first commit simply creates a new revision. For the second a new branch is created automatically. The author with the second commit needs to decide if he wants to merge his revision into the existing head or to create a new, explicit branch.

4.11 Synchronize Instances

In a decentralized environment multiple histories of one object can be created. Like in the concurrent edit scenario, two authors create revisions based on the same origin. But this time they commit multiple revisions into their unconnected instances.

Before synchronizing we actually see two separate branches, one for each instance. Again one system creates automatically a branch for the instance to synchronize with and asks the user, if he likes to merge this new branch with his or to grow it separately.

The verb "synchronize" implies a bidirectional process and to find two identical states on both sides after such a process. But practically one side is always initiating the action and a unidirectional process is performed. The other side is maybe not interested in the other branch or wants to explicitly accept any merge results or new branches. So it also needs to explicitly synchronize it's side as well.

So to synchronize two instances, actually means to merge two branches, a local and a remote one (see Merge Branch).

4.12 Retrace Changes

Sometimes it's important to know about when, what and who has changed the data. This information could lead the reader to a quality attribute of the content (reliability of the author, age of the information) and/or provide a better comprehension why the content has been built this way. Also it would help to investigate any miss use of author access rights.

4.13 Reread Document

You have just read a certain, big document. Others have read it too and noticed some misunderstandings. After the author has corrected the corresponding topics, he lets you access the updated document. - Before you go through the hassle of reading the whole document again, maybe you would like to identify the topics changed and focus you efforts on these.

4.14 Control Access

Somebody is allowed to read a certain revision of an object only, an author is not allowed to change a specific branch or some stranger is not allowed to see the history at all. With the data history available the definition of access control becomes more extensive.

5 Conclusion

In a decentralized system we need to consider some aspects when managing the change history of the data.

Even though there is one, global, conceptual history for a system, in a decentralized system it cannot be managed as such. The scope of data, to manage a history for, should be split into smaller, more practical objects. These data objects, including their histories, can then be shared with other instances to build the complete, virtual database.

It would also be difficult and expensive to apply a global version number to identify the different revisions. A content based hash in combination with the decentralized data object ID would offer a more practical way to build and reference the revisions of the data.

While branches, in the history of centralized managed data, mostly serve the draft-validate-publish like workflow, in a decentralized environment they play a more important role for multiple, concurrent, independent edits of one revision. Every instance implicitly produces a new branch, that will be merged with the one of another instance in the synchronizing process.

From the user's point of view, all the recognized use cases could be handled with a data history model outlined in the beginning of this article. To find more, practice-proven use cases, we need to observe a decentralized data history in action and extend this article accordingly.

6 Resources

Git concepts in the Git User's Manual (for version 1.5.3 or newer)

Git for Computer Scientists, Tommi Virtanen, 2007

Subversion, decentralized version control, and the future, Karl Fogel, 2007

Mercurial: Behind the scenes in Distributed revision control with Mercurial, Bryan O’Sullivan, 2007

Distributed Systems Topologies, Nelson Minar, 2001

Wikipedia: Distributed Revision Control