Remote JabDrive storage

This describes the synchronization to JabDrive. JabRef Online also implements the JabDrive interface.

The setting is that clients synchronize their local view with a server. The server itself does not change data on itself. If it does, it needs to create a separate client connecting to the server. Thus, all changes are finally triggered by a client.

The following algorithm is highly inspired by the replication protocols of CouchDB and RxDB. For the explanation, we focus on the synchronization of entries. Nevertheless, the synchronization of other data (such as the groups tree) works similarly.

From a high-level perspective, the sync algorithm is very similar with git: both the server and the client have their own change histories, and the client has to first pull and merge changes from the server before pushing its new state to the server. The sync process is incremental and only examines entries updated since the last sync.

We call this the “pull-merge-push cycle”.

Data structures

We start by providing information on data structures. There are some explanations of data structures included if they are short. Longer explanations are put below at “The ‘pull-merge-push cycle’”.

Metadata for each item

In order to support synchronization, additional metadata is kept for each item:

- `ID`: An unique identifier for the entry (will be a UUID).
- `Revision`: The revision is a “generation Id” being increasing positive integer. This is based on Multiversion concurrency control (MVCC), where an increasing identifier (“time stamp”) is used.
- `hash`: This is the hash of the item (i.e., of all the data except for `Revision` and `hash`).
- (Client only) `dirty`: Marks whether the user changed the entry.

`ID` and `Revision` are handled in `org.jabref.model.entry.SharedBibEntryData`.

DIRTY FLAGS

Using dirty flags, the client keeps track of the changes that happened in the library since the last time the client was synchronized with the server. When the client loads a library

into memory, it computes the hash for each entry and compares it with the hash in the entry's metadata. In case of a difference between these hashes, the entry is marked dirty. Moreover, an entry's dirty flag is set whenever it is modified by the user in JabRef. The dirty flag is only cleared after a successful synchronization process.

There is no need to serialize the dirty flags on the client's side since they are recomputed upon loading.

Global time clock

The idea is that the server tracks a global (logical) monotone increasing "time clock" tracking the existing revisions. Each entry has its own revision, increased "locally". The "global revision id" keeps track of the global synchronization state. One can view it as aggregation on the synchronization state of all entries. Similar to the revision concept of Subversion.

Tombstones

Deleted items are persisted as tombstones, which contain the metadata `ID` and `Revision` only. Tombstones ensure that all synchronizing devices can identify that a previously existing entry has been deleted. On the client, a tombstone is created whenever an entry is deleted. Moreover, the client keeps a list of all entries in the library so that external deletions can be recognized when loading the library into memory. The local list of tombstones is cleared after it is sent to the server and the server acknowledged it. On the server, tombstones are kept for a certain time span (world time) that is strictly larger than the time devices are allowed to not sign-in before removed as registered devices.

Checkpoints

Checkpoints allow a sync task to be resumed from where it stopped, without having to start from the beginning.

The checkpoint locally stored by the client signals the logical time (generation Id) of the last server change that has been integrated into the local library. Checkpoints are used to paginate the server-side changes. In the implementation, the checkpoint is a tuple consisting of the server time of the latest change and the highest `ID` of the entry in the batch. However, it is better to not depend on these semantics.

The client has to store a checkpoint `LastSync` in its local database, and it is updated after every merge. The checkpoint is then used as the `Since` parameter in the next Pull phase.

The "pull-merge-push cycle"

Each sync cycle is divided into three phases:

1. `Pull phase`: The server sends its local changes to the client.
2. `Merge phase`: The client and server merge their local changes.
3. `Push phase`: The client sends its local changes to the server.

We assume that the server has some view on the library and the client has a view on the library.

STRAIGHT-FORWARD SYNCHRONIZATION

When the client connects to the server, one option for synchronization is to ask the server for all up-to-date entries and then using the `Revision` information to merge with the local data. However, this is highly inefficient as the whole database has to be sent over the wire. A small improvement is gained by first asking only for tuples of `ID` and `Revision`, and only pull the complete entry if the local data is outdated or in conflict. However, this still requires to send quite a bit of data. Instead, we will use the following refinement.

Pull Phase

The client pulls on first connect or when requested to pull. The client asks the server for a list of documents that changed since the last checkpoint. (Creating a checkpoint is explained further below.) The server responses with a batched list of these entries together with their `Revision` information. These entries could also be tombstones. Each batch includes also a checkpoint `To` that has the meaning “all changes to this point in time are included in the current batch”.

NOTE

Once the pull does not give any further changes, the client switches to an event-based strategy and observes new changes by subscribing to the event bus provided by the server. This is more an implementation detail than a conceptual difference.

Merge Phase

The pulled data from the server needs to be merged with the local view of the data. The data is merged on a per-entry basis. Based on the “generation ID” (`Revision`) of server and client, following cases can occur:

1. The server’s `Revision` is higher than the client’s `Revision`: Two cases need to be distinguished:
   a. The client’s entry is dirty. That means, the user has edited the entry in the meantime. Then the user is shown a message to resolve the conflict (see “Conflict Handling” below)
   b. The client’s entry is clean. That means, the user has not edited the entry in the meantime. In this case, the client’s entry is replaced by the server’s one (including the revision).
2. The server’s `Revision` is equal to the client’s `Revision`: Both entries are up-to-date and nothing has to be done. This case may happen if the library is synchronized by other means.
3. The server’s `Revision` is lower than the client’s `Revision`: This should never be the case, as revisions are only increased on the server. Show error message to user.

If the entry returned by the server is a tombstone, then:

- If the client's entry is also a tombstone, then we do not have to do anything.
- If the client's entry is dirty, then the user is shown a message to resolve the conflict (see "Conflict Handling") below.
- Otherwise, the client's entry is deleted. There is no need to keep track of this as a local tombstone.

CONFLICT HANDLING

If the user chooses to overwrite the local entry with the server entry, then the entry's `Revision` is updated as well, and it is no longer marked as dirty. Otherwise, its `Revision` is updated to the one provided by the server, but it is still marked as dirty. This will enable pushing of the entry to the server during the "Push Phase".

After the merging is done, the client sets its local checkpoint to the value of `To`.

Push Phase

The client sends the following information back to the server:

- The list of entries that are marked dirty (along with their `Revision` data).
- The list of entries that are new, i.e., that do not have an `ID` yet.
- The list of tombstones, i.e., entries that have been deleted.

The server accepts only changes if the provided `Revision` coincides with the `Revision` stored on the server. If this is not the case, then the entry has been modified on the server since the last pull operation, and then the user needs to go through a new pull-merge-push cycle.

During the push operation, the user is not allowed to locally edit these entries that are currently pushed. After the push operation, all entries accepted by the server are marked clean. Moreover, the server will generate a new revision number for each accepted entry, which will then be stored locally. Entries rejected (as conflicts) by the server stay dirty and their `Revision` remains unchanged.

Start the "pull-merge-push cycle" again

It is important to note that sync replicates the library only as it was at the point in time when the sync was started. So, any additions, modifications, or deletions on the server-side after the start of sync will not be replicated. For this reason, a new cycle is started.

Scenarios

Having discussed the general algorithm, we discuss scenarios which can happen during usage. In the following, `T` denotes the "global generation Id".

We focus on JabRef as client and a "user" using JabRef.

Sync stops after Pull

1. JabRef pulls changes since `T = 0`
2. JabRef starts with the merge and the user (in parallel) closes JabRef discarding any changes.
3. User opens JabRef again.
4. JabRef pulls changes again from `T = 0` (since the checkpoint is still `T = 0`) and JabRef has to redo the conflict resolution.

This is the best we can do, since the user decided to not save its previous work.

However, consider the same steps but now in step 2, the user decided to save their work. The locally stored checkpoint is still `T = 0`. Thus, the user has to redo the conflict resolution again. The difference is that the local version is the previously merge result now.

Future improvement: We could send checkpoints for every entry and after each conflict resolution set the local checkpoint to the checkpoint of the entry.

Sync stops after Merge

1. JabRef pulls changes since `T = 0`
2. JabRef finishes the merge (this sets the checkpoint `T = 1`).
3. User closes JabRef with discarding any changes (in particular, the checkpoint is not persisted as well).
4. User opens JabRef again.
5. JabRef pulls changes again from `T = 0` (since the checkpoint is still `T = 0`) and has to redo the conflict resolution.

This is the best we can do, since the user decided to not save their previous work.

If the user decides in step 3 to save their changes, then in step 5 JabRef would pull changes starting from `T = 1` and the user does not have to redo the conflict resolution.

Sync after successful sync of client changes

1. JabRef modifies local data: `{id: 1, value: 0, _rev=1, _dirty=false}` to `{id: 1, value: 1, _rev=1, _dirty=true}`. `id` is `ID` from above, `value` summarizes all fields of the entry, `_rev` is `Revision` from above, and `_dirty` the dirty flag.
2. JabRef pulls server changes. Suppose there are none.
3. Consequently, Merge is not necessary. JabRef sets checkpoint to `T = 1`.
4. JabRef pushes its changes to the server. Assume this corresponds to `T = 2` on the server. On the server, this updates `{id: 1, value: 0, _rev=1, updatedAt=1}` to `{id: 1, value: 1, _rev=2, updatedAt=2}` and on the client `{id: 1, value: 1, _rev=1, _dirty=true}` to `{id: 1, value: 1, _rev=2, _dirty=false}`.
5. Client pulls changes starting from `T = 1` (the last local checkpoint). Server responds with `{id: 1, value: 1, _rev=2}, checkpoint={2}`.

6. Client merges the ‘changes’, which in this case is trivial since the data on the server and client is the same.

This is not quite optimal since the last pull response contains the full data of the entry although this data is already at the client.

Possible future improvements:

- First pull only the `IDs` and `Revisions` of the server-side changes, and then filter out the ones we already have locally before querying the complete entry. Downside is that this solution always needs one more request (per change batch) and it is not clear if this outweighs the costs of sending the full entry.
- The server can remember where a change came from and then not send these changes back to that client. Especially if the server’s generation Id increased by one due to the update, this is straight-forward.

FAQs

Why do we need an identifier (`ID`)? Is the BibTeX key not enough?

The identifier needs to be unique at the very least across the library and should stay constant in time. Both features cannot be ensured for BibTeX keys. Note this is similar to the `shared_id` in the case of the SQL synchronization.

Why do we need revisions? Are `updatedAt` timeflags not enough?

The revision functions as “generation Id” known from Lamport clocks and common in synchronization. For instance, the Optimistic Offline Lock also uses these kinds of clocks.

A “generation Id” is essentially a clock local to the entry that ticks whenever the entry is synced with the server. As for us there is only one server, strictly speaking, it would suffice to use the global server time for this. Moreover, for the sync algorithm, the client would only need to store the revision/server time during the pull-merge-push cycle (to make sure that during this time the entry is not modified again on the server). Nevertheless, the generation Id is only a tiny data blob, and it gives a bit of additional security/consistency during the merge operation, so we keep it around all the time.

Why do we need an entry hash?

The hash is only used on the client to determine whether an entry has been changed outside of JabRef.

WHY DON’T WE NEED TO KEEP THE WHOLE REVISION HISTORY AS IT IS DONE IN COUCHDB?

The revision history is used by CouchDB to find a common ancestor of two given revisions. This is needed since CouchDB provides main-main sync. However, in our setting, we have a central server and thus the last synced revision is *the* common ancestor for both the new server and client revision.

Why is a dirty flag enough on the client? Why don't we need local revisions?

In CouchDB, every client has their own history of revisions. This is needed to have a deterministic conflict resolution that can run on both the server and client side independently. In this setting, it is important to determine which revision is older, which is then declared to be the winner. However, we do not need an automatic conflict resolution: Whenever there is a conflict, the user is asked to resolve it. For this it is not important to know how many times (and when) the user changed the entry locally. It suffices to know that it changed at some point from the last synced version.

Local revision histories could be helpful in scenarios such as the following:

1. Device A is offline, and the user changes an entry.
2. The user sends this changed entry to Device B (say, via git).
3. The user further modifies the entry on Device B.
4. The user syncs Device B with the server.
5. The user syncs Device A with the server.

Without local revisions, it is not possible for Device A to figure out that the entry from the server logically evolved from its own local version. Instead, it shows a conflict message since the entry changed locally (step 1) and there is a newer revision on the server (from step 4).

More Readings

- CouchDB style sync and conflict resolution on Postgres with Hasura: Explains how to implement a sync algorithm in the style of CouchDB on your own
- A Comparison of Offline Sync Protocols and Implementations
- Offline data synchronization: Discusses different strategies for offline data sync, and when to use which.
- Transaction Processing: Concepts and Techniques
- Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery