Bitbox: eventually consistent file sharing

Erwan Le Merrer, Nicolas Le Scouarnec and Gilles Straub

Technicolor, France

Abstract. Bitbox is an application that synchronizes distributed repositories of data. It can be used as a backup or sharing application similarly to popular cloud-based storage systems. Bitbox supports arbitrary and changing topologies, thus allowing residential gateways to be used as caches for synchronizing nomadic devices that connect only periodically. In this article, we describe the data-structure and algorithms powering Bitbox. We prove its correctness by showing that its synchronization scheme achieves *strong eventual consistency.*

1 Introduction

To share, backup and access their content from all their devices (e.g., smartphones, tablets, laptops), people often rely on manual management of files (*e.g.*, sending photos by email or backing them up manually). However, such process is error-prone (*e.g.*, overwriting the newest files with older ones or forgetting files during backups). Sharing, backing up and providing pervasive access can all be boiled down to the problem of synchronizing multiple copies. Indeed, if a user updates some content, she wants her backups to be updated, her friends to receive updates of the shared content, and all her other devices to be updated with the new content in a transparent manner [1].

Current systems (*e.g.*, Dropbox [2]) require to synchronize against a central set of tightly synchronized servers. All synchronizations rely on these servers, thus requiring the clients to be frequently connected to them. A decentralized synchronization scheme furthermore allows to take into account the specificities of the network topology. For example, in home networks, the Internet gateway could be used as a cache since it stands between the slow Internet connection and the fast LAN; two devices can then synchronize their updates way faster than by leveraging remote servers.

In this paper we propose Bitbox, a distributed synchronization application. Bitbox relies on the concept of *convergent replicated data types* [3] for correctness. We describe the application and its core algorithms in Section 3 and give a proof that it correctly achieves strong eventual consistency in Section 4.

2 Background on Related Data-Synchronization Tools

Dropbox [2] partially leverages local connections by a feature called LanSync that allows downloading (*i.e.* reading) content from local devices instead of downloading from the cloud. However, this functionality does not support uploading (*i.e.*

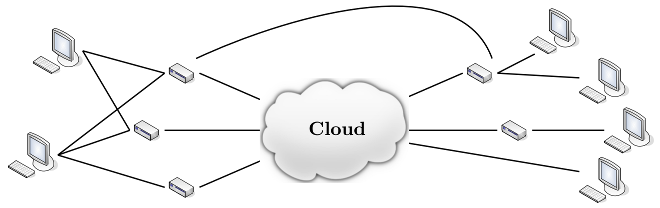

Fig. 1. Bitbox allows arbitrary synchronization topologies.

writing) since the central server is mandatory for it is in charge of maintaining the reference state.

To allow for complex replication policies facing device heterogeneity, Anzere [4] leverages a centralized and well-provisioned node among the user's devices. That special node is in charge of hosting the system's state and runs a conflict solver. We design symetric roles in our system, for less advanced policies.

Bitbox is related to distributed version control tools (*e.g.*, Git [5] or Mercurial), which are replacing traditional central version control (*e.g.*, SVN or CVS). These new tools are designed to allow arbitrary workflow departing from the single centrally managed version. They keep track of the history including all the successive versions of the content of each file, and assist the developer in performing semi-automatic merges between divergent versions. Bitbox differs in that it stores only the latest content of files (which are assumed to be large binary files such as videos or photos), and is designed so as to allow frequent automatic and seamless merges of divergent repositories.

3 The Bitbox Core

Bitbox is an application that keeps various repositories in sync. To this end, whenever a repository synchronizes against another, it lists the changes that occurred remotely since the repository was last synchronized, and applies them locally to reach an equivalent state.

To be able to list these changes, Bitbox keeps track of past states of the repository. A particular state is identified by a version number and consists in a set of file descriptors $\langle \textit{file name}, \textit{file hashes}, \textit{unique file id} \rangle$, which is later referred as *content*. This is a compact representation since file hashes are much smaller than the actual file content. The unique file id is generated by the device which detects that a file has been added (*e.g.*, device identifier concatenated with a local timestamp). The state's version number allows distinguishing two states having the same content but having observed modifications (*e.g.*, a file is removed and then added back). The version number is also a compact representation of a state, allowing Bitbox to trivially detect whether a remote repository does not contain new changes.

Whenever a repository synchronizes against another, Bitbox needs to detect their last common state. To this end, the states are organized in a directed

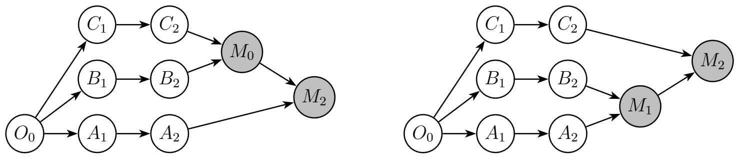

Fig. 2. The Merge operation builds a version number not directly from previous versions but from the previous update (P_U) of versions being merged. Here, $\mathrm{P}_U(M_0) = \{C_2, B_2\}$ and $\mathrm{P}_U(A_2) = \{A_2\}$. As a consequence, the order of Merges has no impact on the version number (*i.e.*, $A2 \oplus (B_2 \oplus C_2) = (A_2 \oplus B_2) \oplus C_2$).

acyclic graph (DAG) as depicted on Figure 2. An edge from state A_1 to state A_2 exists if and only if A_2 results directly from a user's action on A_1, or A_2 results directly from the merge of A_1 with some other state. The last common state of two repositories is their common ancestor in the DAG. Bitbox only considers deletion and addition of files since renaming and updating files can be expressed as a combination of deleting and adding. If O_0 is the last common state between the local state B_2 and the remote state C_2, then Bitbox applies locally all deletions that occurred between O_0 and B_2, and all additions that occured between O_0 and B_2. There cannot be conflicts on a local addition and a remote deletion; indeed, a file is either in O_0 in which case it can be deleted, or not in O_0 in which case it can be added. Furthermore, two deletions applied to the same file in O_0 are necessarily identical.

In some rare cases, conflicts may be observed between additions. They are resolved seamlessly as explained hereafter. In the first scenario, it can happen that users concurrently add two different files under the same file name, at two different devices. The filename f is then associated with the hash h_1 on one device, and h_2 on the other device. In the second scenario, two users may concurrently add the same file under two different versions; this results in the filename f being associated with two different unique file id u_1 and u_2. Since a filesystem cannot store multiple files under a single filename, Bitbox performs a deterministic renaming: the file `fn.ext` of hash h_1 and unique id u_1 is renamed to `fn-`h_1`-`u_1`.ext`. The renames performed during a merge are saved in the DAG so that Bitbox is able to detect *hidden* conflicts and perform renames appropriately. An hidden conflict happens when three conflicting version $\{(\mathtt{f}_1\mathtt{.ext}, h_1)\}$, $\{(\mathtt{f}_1\mathtt{.ext}, h_2)\}$ and $\{(\mathtt{f}_1\mathtt{.ext}, h_3)\}$ are merged. After the first merge, we obtain $\{(\mathtt{f}_1\text{–}h_1\text{–}u_1\mathtt{.ext}, h_1), (\mathtt{f}_1\text{–}h_2\text{–}u_2\mathtt{.ext}, h_2)\}$ and $\{(\mathtt{f}_1\mathtt{.ext}, h_3)\}$. The second merge could be performed without renaming but to ensure that the merge operation is associative, the hidden conflict must be detected and the rename performed.

As the Bitbox synchronization operation is not atomic, changes on the local file system may occur in the middle of a synchronization. To deal with such a case, Bitbox takes a snapshot of the filesystem (*i.e.,* the list of file names, and file hashes) just before running the synchronization process. All changes to the local filesystem occuring during a merge or an update are subsequently treated as if they occurred afterwards (*i.e.*, they are considered at the next update).

Algorithm 1 Updating and Synchronizing

```
    $\mathcal{G}_L$ local history graph.
    $v_L$ current local version.
    $c_L$ local content description.
    $\mathcal{G}_R, v_R, c_R$ the remote ones.
    $c'_L$ new local content descr.
    H SHA applied on the **sorted** set
    $\cdot$ concatenation operator
    $\mathrm{P}_U(x)$ returns $x$ or the last
        ancestors of $x$ resulting from
        an UPDATE (cf. Fig. 2).

 1: def MERGE($(v_L, c_L, \mathcal{G}_L), (v_R, c_R, \mathcal{G}_R)$)
 2:     $\mathcal{G}'_L \leftarrow \mathcal{G}_L + \mathcal{G}_R$
 3:     if $v_L \geq_{\mathcal{G}_L} v_R$ then
 4:         return($v_L, c_L, \mathcal{G}'_L$)
 5:     else if $v_R >_{\mathcal{G}_R} v_L$ then
 6:         return($v_R, c_R, \mathcal{G}'_L$)
 7:     else
 8:         $c_A \leftarrow$ COMMONANC($\mathcal{G}'_L, v_L, v_R$)
 9:         $c'_L \leftarrow$ MERGECHANGES($c_A, c_L, c_R$)
10:         $v'_L \leftarrow \mathrm{H}(\mathrm{P}_U(v_L) \cup \mathrm{P}_U(v_R)) \cdot \mathrm{H}(c'_L)$
11:         $\mathcal{G}'_L \leftarrow \mathcal{G}_L + \{v'_L\} + \{v_L \to v'_L, v_L \to v_R\}$
12:         return($v'_L, c'_L, \mathcal{G}'_L$)
13:     end if
14: end def

15: def UPDATE($(v_L, c_L, \mathcal{G}_L), c'_L$)
16:     $v'_L \leftarrow H(\mathrm{P}_U(v_L)) \cdot \mathrm{H}(c'_L)$
17:     $\mathcal{G}'_L \leftarrow \mathcal{G}_L + \{v'_L\} + \{v_L \to v'_L\}$
18:     return($v'_L, c'_L, \mathcal{G}'_L$)
19: end def
```

4 Proof of Correctness

In this Section, we show that our algorithm ensures that repositories are strongly eventually consistent, thus achieving our protocol goals.

Definition 1 (strong eventual consistency [3]). *The following properties must hold. (i) eventual delivery: an update delivered at some correct replica is eventually delivered to all correct replicas. (ii) convergence: correct replicas that have delivered the same updates eventually reach equivalent state. (iii) termination: all method executions terminate. (iv) strong convergence: correct replicas that have delivered the same updates have equivalent state.*

Theorem 1 (convergent replicated data type [3]). *Assuming eventual delivery and termination, any **state-based object (SBO)** that satisfies the **monotonic semilattice property** is strongly eventually consistent.*

Theorem 2. *A Bitbox object is a convergent replicated data type.*

Proof (sketch). The SBO in Bitbox is the tuple $\langle version, content, graph \rangle$. Replicas are equipped with *query* (trivial and omitted due to space constraints), *update* and *merge* operations (Alg.1) that work on the SBO state. SBO payload is its current tuple state, while its initial state is defined to be a share with no content (*e.g.*, an empty repository).

We need to show that this SBO is a monotonic semilattice (definition follows).

(i) payloads of the SBO form a join-semilattice ordered by $\leq$: the order $\leq$ operates on version identifiers. As they are not directly meaningful (as constituted by a hash value returned by operation at Alg.1 l.10 & l.16), an explicit

order is available by maintaining a DAG of version numbers (as on Fig.2), starting from the SBO's initial state. A DAG forms a partial order on its vertices. Next, we need to show that the SBO's structure exhibits:

- **idempotency**: $\textsc{Merge}(x, x)$ returns x, same graph and version (Alg.1 l.4).
- **commutativity**: $\textsc{Merge}(x, y)$ or conversely results in the same tuple (due to the sort operations at the hash functions Alg.1 l.11, and as additions of graph edges and states are commutative).
- **associativity**. First, operations on $\mathcal{G}$ are associative, involving only edge additions. Second, we define operations on content (*e.g.* $\textsc{MergeChanges}$ Alg.1 l.9) to be solely additions and deletions of files; a file modification is then a deletion of the original file and the addition of the modified version. In this light, by assuming unique additions (*i.e.* unique file ids) and deletion operations occurring causally after corresponding additions, those operations on files actually characterize a *U-Set*, which is itself a convergent replicated data type [6]. We then directly obtain associativity. Finally, we look at the merge operation minus the $H(c'_L)$ operation (discussed at last step) for simplicity: $\textsc{Merge}(x, m = \textsc{Merge}(y, z)) = H(\mathrm{P}_U(x) \cup \mathrm{P}_U(m)) = H(\mathrm{P}_U(x) \cup (\mathrm{P}_U(y) \cup \mathrm{P}_U(z)) = H(\mathrm{P}_U(x) \cup \mathrm{P}_U(y) \cup \mathrm{P}_U(z)) = \textsc{Merge}(\textsc{Merge}(x, y), z)$

With these three identities, the SBO object forms a join-semilattice structure.

(ii) State of the SBO is monotonically non-decreasing across updates: each update creates a new version (Alg.1 l.16) that depends on previous version. A new version is appended to the DAG after the sink state, then becoming the new sink state. Consequently, since the partial order on state is defined by the DAG, the SBO's state is monotonically-non decreasing. □

References

1. J. Strauss, C. Lesniewski-Laas, J. M. Paluska, B. Ford, R. Morris, and F. Kaashoek, "Device transparency: a new model for mobile storage," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, pp. 5–9, Jan. 2010.
2. "Dropbox," http://www.dropbox.com.
3. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free Replicated Data Types," in *SSS*, 2011.
4. O. Riva, Q. Yin, D. Juric, E. Ucan, and T. Roscoe, "Policy expressivity in the anzere personal cloud," in *SOCC*, 2011.
5. "Git: distributed source control management," http://git-scm.com.
6. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," INRIA, Tech. Rep. 7506, 2011.