Versionable, Branchable, and Mergeable Application State *

David H. Lorenz[1,2] †
[1]Open University, Raanana 43107, Israel
[2]Technion–Israel Institute of Technology,
Haifa 32000, Israel
dhlorenz@cs.technion.ac.il

Boaz Rosenan[3]
[3]University of Haifa,
Mount Carmel,
Haifa 31905, Israel
brosenan@gmail.com

Abstract

NoSQL databases are rapidly becoming the storage of choice for large-scale Web applications. However, for the sake of scalability these applications trade consistency for availability. In this paper, we regain control over this trade-off by adapting an existing approach, *version control* (VC), to application state. By using VC, the data model is defined by the application and not by the database. The consistency model is determined at runtime by deciding when to merge and with whom. We describe the design of a VC system named VERCAST that provides fine-grained control over the consistency model used in maintaining application state.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Version control; H.2.4 [*Database Management*]: Systems—Distributed databases.

General Terms Design.

Keywords Source control management (SCM); Version control (VC); Git; Optimistic replication; NoSQL; Consistency; Availability; Conflict resolution; Transactions.

1. Introduction

Imagine a magic show in the two-dimensional world of Flatland [1]. The magician, appearing to be a triangle, says the magic word "threedimensionality," and suddenly, in front of the amazed audience, his body gets disfigured until he takes the shape of a circle. He says the magic word again and becomes smaller and smaller, until his body vanishes into thin air. Then a tiny triangle appears on stage, growing back into the magician's original size. To a Flatland resident, this might look like magic, but for us this is just a cone rotating and moving in space.

In this work we say the word "threedimensionality" to persistent application state. Abstractly, application state can be considered a data store implementing a two dimensional mapping,

$$\sigma : key \times time \rightarrow value$$

in which stored values are indexed by key and vary over time. In practical distributed network implementations, however, the trade-off between consistency and availability manifests to the user as what could be considered unexpected behavior of the data store. We show that this problem may be resolved by introducing a third dimension, placing the state of an application under *version control (VC)* [22]. In the NoSQL world this might look like magic, but in the software engineering world this is just like *source control management (SCM)* [18].[1]

1.1 Consistency vs. Availability

Consider an on-line booking system, similar to the one used for SPLASH.[2] Two participants from two different cities may attempt to book on-line the last discounted hotel room at roughly the same time. What should the system do if during these simultaneous booking attempts the network between the two data centers handling the two requests become temporarily disconnected?

For handling such a case, the developers of the booking system have a choice. One option is to design the system for *consistency*, the default choice when using a relational database. In such a case, at least one of the users will be notified that the booking system is temporarily unable to book the room, and asked to try again later. However, this

* This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 1440/14 and by an Open University Research Grant No. 502672.

† Work done in part while visiting the Faculty of Computer Science, Technion—Israel Institute of Technology.

Onward! 2014, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2661136.2661151

[1] The term SCM is also known as *source code management, revision control*, and *version control* (VC).

[2] The example is adapted from Sadalage et al. [19].

option is sometimes unacceptable. In the modern world of e-commerce, "customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornadoes." [8]

Another, more business-appropriate option is to prefer *availability*, a choice made possible by choosing certain NoSQL databases [19]. In such a case, the booking system will optimistically approve both booking requests. At some point, after communication is restored, the system will discover that the same room has been double-booked, and shall issue a cancellation note to one of the customers apologizing for the mix-up. This option may be satisfactory in case the business environment is flexible enough. However, what if the customer already booked a nonrefundable flight knowing he or she would have a place to stay? As one is often reminded by one's fiancée, there are times in life where one needs to commit.

This trade-off between consistency and availability of stored values is expressed by the CAP theorem [12]. In practical terms, CAP states that, at the event of a network partition (P), a key-value store σ that is replicated across the network can remain either consistent (C) or available (A) but never both [19]. Our "mental model" expects $\sigma(\kappa, \tau)$ to retrieve exactly one value for a particular $\kappa \in key$ and $\tau \in time$. However, at the event of a network partition, $\sigma(\kappa, \tau)$ seems to have either more than one value (when consistency is compromised in favor of availability), or no value at all (when availability is compromised to ensure consistency), which is surprising and unexpected.

1.2 Could We Be Living in Flatland?

The reason $\sigma(\kappa, \tau)$ does not always provide exactly one value is that σ has an implicit third dimension: a *replica*. That is, each time one consults the store, one actually consults a replica of the store. The store's implementation hides these replicas from the user, and attempts to keep the various replicas in sync. Unfortunately, keeping them in perfect sync is sometimes impossible due to network failures, or undesired due to synchronization latencies. As a result, the store can be unavailable or inconsistent at times.

Making replica an explicit parameter of a three dimensional mapping,

$$\sigma : key \times time \times replica \to value$$

can redress this issue, returning a single, definite value $v = \sigma(\kappa, \tau, \rho)$, where ρ represents the replica. Unfortunately, supplying the replica as an argument is mostly impractical. Replication is often considered an internal implementation detail not exposed to users. Moreover, for optimization reasons, the data store may need to dynamically choose a replica based on location or load considerations. For these reasons and others, actual databases take different measures for hiding the replica, keeping the user trapped in a two-dimensional Flatland-like world.

1.3 The Missing Dimension

In this paper we suggest a practical way to make the third dimension explicit in a way that will make sense to users, as it is already handled in a different field of software, namely, *source control management (SCM)*.

Returning to the booking system example, the source code of the booking system itself can also be considered replicated data that is distributed across machines and global locations. For example, the developers of the booking system may be collaborating on the code from distant locations. The network they use is the same Internet used to run the application, and prone to suffer from the same failures. Interestingly, network failure may temporarily delay software integration across development sites, but not hold back the software development all together. This is thanks to distributed SCM tools, that, on the one hand, allow each developer to work on a private own copy of the source code, uninterrupted by other developers or by network failures; and, on the other hand, allow synchronization in form of merges. This begs the question: could an SCM-like system provide the same kind of freedom for application state?

What makes SCM so powerful is its three dimensional nature. Like data stores, it has a key dimension (file path) and a time dimension (version number). However, it also has a *branch* dimension, which can be considered a "virtual" replica dimension. Indeed, SCM systems are typically queried using all three dimensions. In many SCM systems (both distributed and multi-site), a branch *belongs* to a replica. In Git, for example, the master branch in one replica is a different branch than another replica's master. In such systems, specifying the three coordinates of key, time and *branch* implies key, time and *replica*, and uniquely identifies a version.

SCM is a special case of VC [22]. In this paper we use the term VC more broadly than SCM to denote versionable content of any kind, not limited to source code. VC can be as simple as keeping track of history, allowing the retrieval of past versions, but it becomes much more useful when branching and merging capabilities come to play. For the purposes of this paper, VC refers to the whole bunch, branching and merging included.

2. Background

In this section we provide necessary background on the problem domain. The reader that is familiar with NoSQL may wish to skip this section.

2.1 SQL vs. NoSQL

For many years, relational databases [7] dominated the database landscape for nearly any kind of application. They had everything: a well defined data model based on well established mathematical principles, a wide selection of open-source and commercial implementations that guaranteed the right performance for the right price, and to top it all, strong

standards, such as ODBC and SQL, which made it possible for users to migrate from one database to another with relative ease.

However, relational databases fail to scale with the Web. Today, *NoSQL* databases are often considered the paradigm of choice for many emerging applications [19]. While relational databases are required (by standardization) to be fully consistent, NoSQL databases allow different levels of consistency, which in turn, allows different levels of availability and performance. However, this variety comes with no standardization whatsoever. Each NoSQL database offers its own data model, consistency model, and API (Application Programming Interface).

2.2 Data and Consistency Models

NoSQL databases are often classified by their *data model* into four main categories [19]:

Key-value Store: a database that stores opaque *values* under *keys*.

Wide-column Store: a database that stores data in *cells* with two keys: a *row* key and a *column* key.

Document Store: a database that stores *documents* such as JSON (JavaScript Object Notation) objects and often allows partial update or retrieval of such documents, as well as indexing documents based on their content.

Graph Database: a database that stores *graphs*, which are highly efficient in finding nodes and paths on a graph (e.g., finding mutual friends in a social network).

NoSQL databases can also be classified by their *consistency model*. Roughly speaking, they can be divided into three categories:

Sequential: a database in which, as in relational databases, operations are performed sequentially with regards to the entire database. ACID (Atomicity, Consistency, Isolation, Durability) transactions are often used to achieve this sequentiality. Examples include Neo4J [24] and other graph databases.

Strongly Consistent (SC): a database in which each operation is performed atomically with regards to a subset of the database (such as a *row*, or a *document*). No order can be assumed between operations on different such subsets. Examples include BigTable [6] and MongoDB [2].

Eventually Consistent (EC): a database that guarantees an order on operations, given that enough time has passed between them. In particular, when updating a database entry κ to some value x, and then querying κ, the queried value is guaranteed to be equal to x only if enough time has passed between the update operation and the query, and if no other update has been performed to κ during that time. Examples include Dynamo [8] and Cassandra [11].

2.3 Tight Coupling Between the Models

In many NoSQL databases, the API is tightly coupled with both the data and consistency models. For example, MongoDB, supporting SC, guarantees atomic changes on documents. To facilitate this, it provides a language for transmitting reified modification requests to the server. This language (part of the MongoDB API) is tightly coupled with the MongoDB data model on the one hand, and its SC behavior, on the other hand. As a consequence, application developers need to choose a database (and hence, a consistency model) at an early stage of development. Afterwards, moving from one NoSQL database to another is hard, since the data model and API are unique to each database. This is in contrast to relational databases, where application development can start with one database (typically, simple and cheap), and gradually migrate to another (more expensive) database without making fundamental changes to the software, that already "speaks" SQL.

3. Motivating Example

Let us consider an alternative implementation for the booking system discussed in Sect. 1. Assume it runs (like many such systems) on many unrelated Web servers that are scattered in different data centers all over the world. Unlike most Web applications, these servers do not store their state on a common database, but rather, each Web server stores its state in a deep hierarchy of directories and text files on its local disk. Let us assume that for the purpose of storing vacancies in hotel rooms we use the following path:

`/hotels/`⟨country⟩`/`⟨state⟩`/`⟨city⟩`/`⟨hotel⟩`/`⟨room⟩

where ⟨room⟩ is a file consisting of lines of the form: ⟨date⟩: ⟨customer⟩. For example, the file

```
/hotels/US/OR/Portland/Marriott/1405
```

represents the availability of room 1405 in the Marriott Downtown Waterfront Hotel in Portland, OR. If the file has the following content:

```
2014−10−19:
2014−10−20: Boaz Rosenan
2014−10−21: Boaz Rosenan
2014−10−22: Boaz Rosenan
2014−10−23: Boaz Rosenan
2014−10−24: Boaz Rosenan
2014−10−25:
```

it means that the room has been booked by Boaz Rosenan for the duration of SPLASH'14, and is vacant for one night before and after.

Since each server stores information on its own file system, information cannot be shared across servers. To solve this problem (typically solved by a shared database), we will use a VC system. Specifically, we will use Git [5].

Let us assume that after each change, each server commits the modified files to the local repository (`git commit`

`-a`). Each server has a list of the other servers, and every few seconds it picks a peer at random and pulls changes from that peer (`git pull`). Git's pull operation merges changes committed by the peer to the local file system. This serves the purpose of an eventually consistent database (EC, Sect. 2.2): if no conflicts are present, a change made on one server will *eventually* propagate to all other servers. It also features the main advantage of EC databases: a failure in one of the servers, or in the network connection between them, will not harm the availability of the service, just its consistency.

3.1 Conflict Resolution

Now assume Boaz Rosenan and David Lorenz try to book the same hotel room at roughly the same time. Here "roughly the same time" means that the server serving Boaz did not get an update about David's booking from the server serving David, before committing Boaz's booking, and vice versa. In such a case, sometime down the road when both changes eventually meet on a certain server, a merge conflict is identified by Git, and the `git pull` operation fails. When that occurs, Git can report the names of the conflicting files, e.g., `git status` would list the conflicting files as *both modified*. Inside each conflicting file, Git marks the conflict in the text, in a similar manner to other SCMs. Using `git annotate` we can determine the ID of the local change that caused the conflict.

Our application will resolve the conflict by first undoing the merge (`git reset --hard`) and then rolling back the local change that caused it by calling `git revert` with the change ID. This will undo the change by applying the opposite change. Pulling from that same peer will not conflict anymore. All that is left to do is to notify the user whose booking was canceled (either Boaz or David) of the need to reserve a different hotel room. We can use the commit comment in Git to convey the information of what needs to be done when commits are reverted.

3.2 Who Decides?

The conflict resolution mechanism described in Sect. 3.1 should run on all servers, since any server may experience conflicts. However, we can designate a particular server to be the "decider." When one books a hotel room on the file system of that decider server, the booking is guaranteed.

One example of how this can be implemented is by giving each server s a unique number $u(s)$, e.g., some hashing of its IP address. When a conflict between two servers s_1 and s_2 is identified, if $u(s_1) < u(s_2)$, s_1 wins, and s_2 needs to roll-back its change. Otherwise, s_2 wins, and s_1 needs to roll-back. The server $s_{\min}$ for which $u(s_{\min})$ is the smallest value across all servers is the *decider*.

Now assume, for those customers who need to know for sure (e.g., those of us who need to buy nonrefundable airline tickets), we add a special button in the booking page: "*Get confirmation now*." When a user pushes this button, the server handling the request opens a connection to the decider server, and asks it to pull changes from it. If the pull operation succeeds without conflicts, the user is notified that the booking is confirmed. If a conflict occurs, the user is notified immediately. If the decider server is unavailable for some reason (and that may happen because, after all, we are counting on a single server here), the user is told the reservation could not be confirmed, and asked to try again later.

Some EC databases provide similar behavior, by supporting *tunable consistency* [11]. Tunable consistency states that for each read or write operation, the user may decide how many replicas should be contacted before considering the operation successful. A confirmation button can be implemented over such a NoSQL database by making an update to the room availability data, requiring all replicas to be contacted. If we do so, any conflict will be discovered before the operation completes. Of course, any replica residing on a computer to which we have no connection will cause a temporary failure, just like when using Git.

3.3 All or Nothing Group Booking

So far we demonstrated with VC features similar to the ones we have in state-of-the-art NoSQL databases (up to performance, obviously). But we can do better.

Imagine we wish to allow customers to book more than a single room. The rationale is to allow groups to travel together, booking several rooms in a single hotel, or allowing travelers to book rooms in different hotels, along the path of their trip. If one of the rooms is unavailable, the entire order should be canceled.

To support this, we allow users to modify any number of files. The server will commit the changes to Git in a single `git commit` operation. This way these changes will be a part of the same commit, with the same commit ID. If a conflict occurs, the same conflict resolution applies, only that this time it would undo all changes in all files involved in that commit.

Achieving group booking in a typical NoSQL database is much harder. In many such databases, each change to a data element (document, row, value, etc.) stands on its own. Group booking cannot be done atomically, and it is possible for an observer to see at a certain point in time some of the rooms booked, and some vacant. Moreover, if a conflict occurs in booking one of the rooms, all the bookings that were already done must be undone. An observer may see a room being booked and then immediately unbooked.

In the VC solution described here, we solved the group booking problem by introducing *transactions*, a concept largely abandoned by NoSQL databases [19]. These transactions are similar in some ways to their *ACID* counterparts in relational databases: they are *Atomic*, *Consistent* and *Isolated*, all thanks to the atomicity of Git merges. However, they are not *Durable*. A transaction can be undone after having been committed successfully. In return, commits are done locally and therefore fast, unlike two-phase commits

in the ACID world, which involve all replicas. Durability, of course, can be achieved by contacting the decider server. Once committed successfully to the decider, our transaction will not be undone. The advantage we have here over relational databases and ACID transactions is that here *we* have the power: we can decide when we want to wait for the decider server and when we are willing to take our chances; when we want certainty at the risk of unavailability; and when availability is critical, and we are willing to risk having our updates reverted.

3.4 Summary

While Git could handle the state of a simple booking system, SCM systems like Git are designed for source code projects, sized in Megabytes. They are not appropriate in real-life applications for persisting application state sized in Terabytes or even Petabytes [14]. Git, like many SCM systems, merges text files on a line-by-line basis. This forced us to design the booking example in a way that each line stands on its own, with files representing rooms and lines representing nights. In real-life booking systems, however, booking is not done to particular rooms but rather to room types. What matters in these systems is the total number of rooms available of each type. If Git had support for versionable objects, e.g., *counters*, our job could have been much easier.

4. Solution Domain

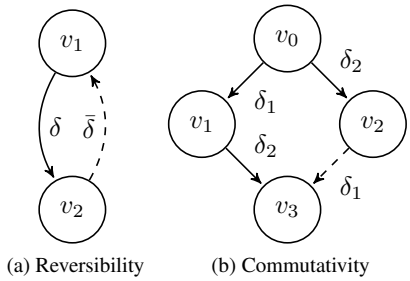
In this section we review the building blocks needed for designing a VC system intended for application state.

4.1 Versionable Objects and Patches

In SCM systems, the term *versionable objects* typically refers to files and directories. Many of these systems hold a single version of each object in the user's file system. This version is usually called the *working version* (denoted w). When the user commits changes to a file or a directory tree, the SCM compares (*diffs*) the working version w with the latest committed version v_1 of the object, creating a *patch* $\delta = w - v_1$. Patches represent changes or differences (deltas) in the state of the repository. Applying patch δ to v_1 will result in version v_2, which is identical to the working version $w = v_1 + \delta = v_2$.

When we consider versionable application state, we refer to versionable objects more loosely than SCM, and they can take any shape or form. However, we do not hold a working version. Instead, the application creates patches directly, and applies them to the state. This means that we do not need to define a *diff* operation on objects, just *transformation* operations, detailing how patches are applied to objects.

Reversibility and Commutativity Versionable objects can be compared to objects in OOP, with patches acting as messages. However, versionable objects must fulfill two requirements, namely: *reversibility* and *commutativity* (Fig. 1). We expect patches to be invertible and their application to versions reversible (Fig. 1a). That is, if $v_1 + \delta = v_2$, we expect that there exists a patch $\bar{\delta} = inv(\delta)$ computed based on δ alone, such that $v_2 + \bar{\delta} = v_1$. A patch therefore needs to contain not just the information needed to perform the change expected from it (postconditions), but also the information needed to calculate its inverse (preconditions) [22].



(a) Reversibility (b) Commutativity

Figure 1: Required properties of versionable objects

For example, consider an *atom* object, which contains some opaque value. Since the value is opaque, the only way we can modify it is by replacing it. Consider a set patch that replaces the atom's content. To allow modification, we need the set patch to contain the new value we would like to store. To support inverting the patch, the patch also needs to contain the value we are replacing. If we wish to replace the value 1 with 3, our patch can be written as: $\delta = set\,\langle 1, 3\rangle$. The inverse patch can be easily calculated: $\bar{\delta} = set\langle 3, 1\rangle$. In case the application of the patch cannot be reversed, i.e., when the preconditions do not hold, the transformation cannot be performed. We consider such a failure as a *conflict*.

A patch δ_2 is said to be *independent* of patch δ_1 *with respect to* version v_0, if $v_0 + \delta_1 + \delta_2 = v_3$ and $v_0 + \delta_2 = v_2$ can both be applied without conflict. In such a case we require commutativity (Fig. 1b), i.e., that $v_0 + \delta_2 + \delta_1 = v_3'$ can also be applied without conflict, and that $v_3' = v_3$. Note that commutativity implies that patch independence is a symmetric property, i.e., δ_1 is independent of δ_2 if and only if δ_2 is independent of δ_1.

As a counter-example to this requirement let us consider an object that represents a number and accepts four patches: δ_1 which increments the number, δ_2 which doubles it, and their inverses $\bar{\delta}_1$ and $\bar{\delta}_2$. It is obvious that for any valid state (i.e., any number) any of the four patches can be applied without conflict, and that the object satisfies the reversibility property. However, the commutativity property does not hold. Applying δ_1 and δ_2 in a different order will result in a different state. Therefore, such an object is not considered to be a valid versionable object.

4.2 Persistent Trees

Woelker [25] points out that three systems (Git, CouchDB, and Clojure), while being three different sorts of things (an SCM, a NoSQL database, and a programming language, respectively), have at least one thing in common: they all

use persistent trees. A *fully persistent tree* [16] is a tree comprising immutable nodes. Once a tree node has been created, it is never changed. Instead, changes to the tree are performed by *path copying*, a technique that involves copying all nodes along the path from the modified node to the root, thus creating a new tree. The old and the new trees share all nodes except those on the updated path.

Operations on persistent trees typically have the same big-O time complexity as their non-persistent counterparts, but they make non-destructive updates. For CouchDB, as well as other NoSQL databases, persistent trees offer a way to look at snapshots of the database as of a certain time. This is needed to implement Multi Version Concurrency Control (MVCC), which is common in the NoSQL world. For functional programming languages (such as Clojure), persistence is required by the nature of the language. Clojure also uses this property to implement MVCC [13], in order to support Software Transactional Memory (STM) [21].

Persistent Trees for VC Git uses persistent trees to represent the repository's directory structure. Git holds an internal key-value store, where objects (representing files and directories) can be retrieved by a unique ID. Git uses SHA-1 hash of the content of an object as its key. Directories reference subdirectories and files by storing their hash. If the content of a file changes, its hash value changes. This way, the directory containing it needs to change (update of the hash), and therefore its own hash changes. Eventually, the root directory's hash values changes. This hash of the new root becomes the ID of the new version.

The use of persistent trees provides Git with an efficient way to store and retrieve multiple versions of the same directory tree. Unlike many other SCMs that store a single version of each file along with patches that can lead to all other versions, Git holds *snapshots* of all versions. When a patch is applied (by performing a commit), a new root is created, and its ID is stored. From this point on, retrieving that version is as simple as traversing the directory tree from *that* root.

Similar to files and directories in Git, general versionable objects can also be stored in a persistent tree. Each object version v can have a unique *version ID,* $[v]$, that is derived from its value (e.g., by using a hash function). Object versions may reference versions of other objects by specifying their version ID. This way, a version ID $[v]$ does not only specify a concrete version v of an object, but rather it also specifies concrete versions for all objects in its underlying sub-tree.

An important special case is the *root object*, which represents the entire application state. The version ID of the root object represents the version of the entire state. For the rest of this section, we discuss only the state of the root object.

4.3 Version Graphs and Merging

We define the *version graph* of an application as a directed graph whose vertexes are all the versions taken by its root object, and its edges represent the patches that caused the transformation between these versions. If we were to use a relational database to maintain our application state, our version graph would have been linear due to its sequential nature. However, since we allow patches to be applied to *any* version, our version graph becomes more complex. Version graphs are represented in different ways in different SCM systems, but their common primary role is to support *merges*.

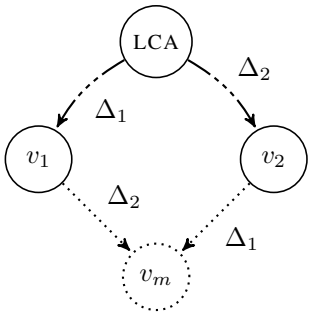


Figure 2: Merging v_1 and v_2 into v_m

The algorithm we need for merging application state is somewhat different than that commonly used in SCM systems. Typically, SCM systems perform merges by first identifying the effected files, and then performing three-way textual merges on these files. This method is inapplicable for our purposes, because we use general objects rather than text files.

The first step in merging two versions v_1 and v_2 of the application state is finding their lowest common ancestor (LCA) in the version graph (Fig. 2). The edges marked Δ_1 and Δ_2 in Fig. 2 indicate the paths from the LCA to v_1 and v_2, respectively, that exist in the version graph before the merge. The next step is to apply to v_1 all the patches in the path Δ_2. If no conflicts occur, the resulting version, v_m, is the result of applying to the LCA all patches in both Δ_1 and Δ_2. To record the merge in the version graph we add the two dotted lines, indicating both the patches that we applied during the merge, and the patches we would have applied had we chosen to apply Δ_1 on top of v_2 instead. The correctness of this last step follows from the commutativity requirement. The choice of the direction in which to apply the patches (apply Δ_2 to v_1 or Δ_1 to v_2) is arbitrary, since both will yield the same merged version at the absence of conflicts.[3]

4.4 Branches

One concept that we ignored so far is the notion of the *current version*. Obviously, a good VC system can cope with any number of current versions. For example, it is common that when working on a software project, each developer has a private version of the product, each feature and each project has an integration version, there is a staging version

[3] We extend this algorithm to handle conflicts in Sect. 7.2.

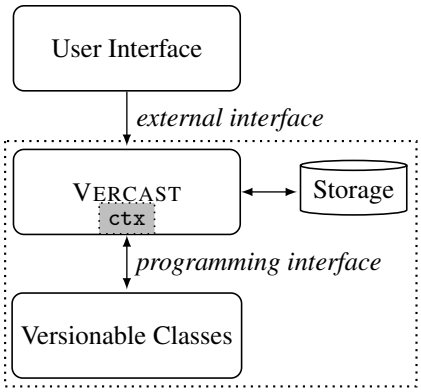


Figure 3: VERCAST interfaces

and, of course, a released version. In modern VC systems each of these current versions is a *head* or a *tip* of a *branch*.

Branches can be seen as directed paths on a version graph. In Git, branches are stored as mutable pointers to versions of the tree.[4] At any given point in time, for every branch, Git holds a single *head* version. When a patch is applied to a branch, it is actually applied to the tree version the branch points to. Then, if no conflicts are detected, the branch head is updated with the new version.

5. VERCAST

VERCAST is a VER*sion* C*ontrolled* A*pplication* ST*ate* framework. It is a VC system designed for managing versionable, branchable, and mergeable application state. VERCAST has two interfaces (Fig. 3). Its *programming interface* (Sect. 5.1) lets application developers implement *versionable classes* needed for constructing versionable applications. Its *external interface* (Sect. 5.2) lets clients interact with the application state (queries and modification), as well as perform synchronizations.[5]

5.1 Programming Interface

Versions are snapshots of versionable objects. Each version is thus an instance of a *versionable class*. The verionable class provides a constructor for producing an initial version, and transformer methods [15] for handling various patches. For example, Fig. 4 illustrates a class definition for a versionable Counter.

Syntactically, a construction of a version has the form:

$$r \leftarrow C.init\,(ctx, args)$$

where C is a class, ctx is a context (Sect. 5.1.2), and $args$ is an argument list. For example, the Counter constructor in Fig. 4 takes a context `ctx`, an initial value `val`, and a Boolean flag `bounded` telling the counter whether or not it should conflict once its value drops below 0. It initializes the `value` and `bounded` instance variables with their respective values.

An invocation of a transformer method has the form:

$$r \leftarrow v.t\,(ctx, \delta, u)$$

where v is a version, t is a method name, ctx is a context, δ is a patch to be applied, and u is a Boolean flag determining whether to apply or "unapply" δ (i.e., apply $\bar{\delta}$). In the Counter class, the *get* method simply returns the value of the counter. Based on the u flag, the *add* method either adds to or subtracts from the counter a given value.

5.1.1 Transformation Semantics

VERCAST interprets changes to the state of version v as a creation of a new version v' rather than a modification of the existing version v. Semantically, a transformer method is a functional mapping T of the form:

$$(v', r, c, E) \leftarrow T\,(v, \delta, u)$$

where v is the version of the object before applying the patch δ. It returns a quadruple comprising the version v' of the object after applying δ (or $\bar{\delta}$), the result r returned by the patch (could be *nil*), a Boolean flag c that indicates whether or not the transition conflicted, and an effect set E of patches.

Effect Set The *effect set* allows objects to communicate changes to the global state of the application, effecting objects they have no direct reference to. They are needed for binding a change in one place in the tree to other places.[6]

5.1.2 Context

To maintain their pure functional semantics, constructors and transformer methods are not permitted to interact with code not under VC. VERCAST provides such constructors and transformer methods with a *context,* which is an object serving as a single entry point for all the interactions they *are* allowed to make. As such, it serves two purposes. One purpose is to maintain the state of the transformation, holding the conflict flag c and the effect set E. The other purpose is providing a portal to the repository, for construction and manipulation of other objects.

A context, `ctx`, supports the following operations:

1. $[v_0] \leftarrow ctx.init(C, args)$: returns the ID of an initial version v_0 of an object of class C, based on arguments $args$.
2. $([v'], r) \leftarrow ctx.trans([v], \delta)$: applies patch δ to v, returning both the ID of the resulting version v' and the result r.

[4] They are actually mutable pointers to *commits*, which point to the root of a tree and to the previous commit. This structure in necessary for tracking the history of a branch. We do not need history tracking for our discussion, so we can ignore the commit level and treat branches as if they point directly to the tree.

[5] The external interface is typically used by the application's user interface, but can also be used by other external interfaces, such as interfaces to other systems.

[6] We show a usage example in Sect. 7.1.

```
class Counter:
 variables: value, bounded;
 constructor init(ctx,val,bounded)
  this.value ← val;
  this.bounded ← bounded;
 method add(ctx,δ,u):
  if (u)
   then this.value ← this.value − δ.amount
   else this.value ← this.value + δ.amount;
  if (this.bounded ∧ this.value < 0)
   then ctx.conflict();
 method get(ctx,δ,u):
  return this.value;
```

Figure 4: Versionable Counter

3. ctx.*conflict*(): reports that a conflict has been encountered.
4. ctx.*effect*(δ): adds δ to the effect set of the current transition.

5.1.3 Application State as a Persistent Tree

The application will typically organize its objects in a form of a persistent tree (Sect. 4.2). Patches originating from user interaction will be directed to the application's root object, invoking one of its methods. This method will propagate the patch to one of the child objects, using ctx.*trans*(). The context will then call one of the child object's methods, which will propagate the call through the tree until a leaf is reached. In case of a query, the return value will typically be returned as-is all the way back to the root. In case of mutation (modification), the invoked methods will update the state of each object along the path with the new value or new child version ID. This will cause the framework to create a new version of each object, and eventually, a new version of the root.

5.1.4 Hotel Booking Example

Fig. 5 shows an example of a versionable Hotel class, representing the availability of rooms in a single hotel. Its constructor receives a map, `rooms`, mapping a room type (`single`, `double`, or `suite`) to an integer specifying how many of those types of rooms the hotel has. It is also given `initial` and `nights` that define the range of dates (by some enumeration) that are open for booking. It initializes an array of counters for each room type, one counter per day. Since all counters in the array are initialized to the same value, we can create just one counter and provide its version ID to the array's constructor. The array will initialize all of its entries with this ID.

The *book* method receives a patch δ comprising a room type, a date range and an amount of rooms, and builds a patch δ_3 to be applied to the relevant array. We use a JSON-like notation to represent a patch, with a `_type` field

```
class Hotel:
 variables: rooms,initial,vacancy;
 constructor init(ctx,rooms,initial,nights)
  this.initial ← initial;
  this.vacancy ← map();
  for type ∈ rooms.keys() let
   counter ← ctx.init(Counter,rooms[type],true);
  in
   this.vacancy[type] ← ctx.init(Array,nights,counter);
  end
 method book(ctx,δ,u)
  let
   δ1 ← {
    _type: add,
    amount: −(δ.numRooms) };
   δ2 ← {
    _type: applyRange,
    from: this.initial + δ.start,
    to: this.initial + δ.end,
    patch: δ1 };
   δ3 ← u? inv(δ2): δ2;
  in
   (this.vacancy[δ.roomType], _) ←
    ctx.trans(this.vacancy[δ.roomType],δ3);
  end
```

Figure 5: Versionable Hotel

specifying which method should be invoked. The patch δ_3 is constructed in three steps. First, we construct a patch δ_1 that can be applied to a counter for decrementing the amount of rooms we would like to reserve. Second, we construct a patch δ_2 that can be applied to an array, for applying δ_1 to a range of its entries. Finally, if u is true, we invert patch δ_2 using the *inv* utility function. The *book* method applies the resulting patch δ_3 on the array corresponding to the requested room type. The array version ID is then updated in the map. Note that Hotel does not explicitly check for conflicts. When constructing the counters it sets `bounded` to true, so that they will conflict when overbooked. As a result, invoking a *book* patch on a Hotel will conflict if the hotel does not have enough rooms at the specified dates.

A single hotel is obviously just one piece of the full application state. In the following we will assume that the full state of the application is represented by a versionable map, where hotels and possibly other data elements are referenced by unique keys. We will assume that the map responds to all patches that contain a `_key` field by propagating them to the corresponding object.

5.2 External Interface

The external interface is intended for the parts of the system not under VC. This interface lets the application and its users query and change the application's state, as well as control

its consistency vs. availability trade-off using transactions and synchronization.

VERCAST's external interface supports the following operations:

1. $[v_0] \leftarrow$*init*(C, A): similar to the *init* method described in Sect. 5.1. Typically, used to initialize the full application state.
2. $([v_2], r) \leftarrow$*trans*$([v_1], \delta)$: similar to the *trans* method described in Sect. 5.1, but can also take a transaction object (see #7 below) in place of $[v_1]$, in which case it will update it with patch δ.
3. *fork*$(\beta, [v_0])$: creates a new branch named β, starting at initial version v_0.
4. $[v_h] \leftarrow$*head*(β): returns the last known head (tip) of branch β. It does not synchronize and may yield a somewhat stale head, but offers high availability.
5. $s \leftarrow$*push*$(\beta, [v])$: merges the version v to the head of branch β, and updates the head atomically. Returns status s which may be one of the following: `success` if all goes well, `conflict` if a merge conflict was encountered, or `unavailable` if the server holding β's head was unavailable. In the latter two cases, the branch head is not updated.
6. $[v_m] \leftarrow$*pull*$([v_1], \beta)$: merges v_1 with the head of branch β, yielding the resulting version ID. `pull` is supposed to always succeed. It uses the highly-available `head` method to access β's head, and in case of a merge conflict, it resolves it by preferring β (Sect.7.2).
7. $t \leftarrow$*beginTransaction*$([v_0])$: returns an empty transaction object t, that corresponds to version v_0 of the application state.
8. $[v] \leftarrow$*commit*(t): commits transaction t: applies all its underlying patches as a single patch to the transaction's v_0. Returns the resulting version ID. Note that it only replays previously applied patches, and therefore will not conflict.

Fig. 6 shows how this API can be used to implement a part of the user interface of the booking application. Recall that we would like to support group booking. For this reason, we would like to have a user-level object similar to a shopping cart, where users can add one or more bookings, and then book them all together. If one or more of the rooms cannot be reserved, the entire order is to be canceled.

To implement this, Fig. 6 lists three functions that are to be called in response to three kinds of user interactions: creating a new cart, requesting a room in a specific hotel, and booking everything. Creating a cart is done by creating a transaction object, based on the head of some branch β_0. Recall the `head` function returns the *last known* head, so it can run very fast, at the expense of providing a somewhat

```
1  procedure newChart()
2   t ← beginTransaction(head(β0))
3
4  procedure addToCart(hotelID,roomType,start,end)
5   let
6    δ ← {
7     _type: book,
8     _key: hotelID,
9     roomType: roomType,
10    numRooms: 1,
11    start: start,
12    end: end}
13  in
14   try
15    t ← trans(t, δ);
16   on_conflict
17    print("No rooms available");
18   end
19  end
20
21 procedure book(β)
22  [v] ← commit(t);
23  case push(β, [v]) of
24   "success" =>
25    print("Booking successful");
26   "conflict" =>
27    print("Rooms are not available");
28   "unavailable" =>
29    print("Cannot confirm. Please try again later");
30  end
```

Figure 6: External API usage example

old version. The branch that we choose here, β_0, is the application's *main* branch.

When the user books a room in a specific hotel, we create a patch for that hotel, and add the hotel key to the `_key` field, so that the map holding the full application state could forward this patch to the right Hotel instance. If the hotel is already fully booked, this transition conflicts and the user is notified. Otherwise, the patch is added to the transaction object.

When the user finally wants to seal the deal, the transaction is committed. This replays all the patches in a single operation. To make it visible to other users (and hence, to make the reservations take effect) we push the state to a certain branch. The function *book* takes the branch name as a parameter. Indeed, in different scenarios we would like to use different branches. Because the `push` operation requires atomic updates, the operation needs to be performed where the branch is maintained, so typically, we would like to choose our branch β_n based on locality and availability. However, if we need immediate confirmation and are willing to wait for it and risk unavailability, we would like to use the decider branch, β_0.

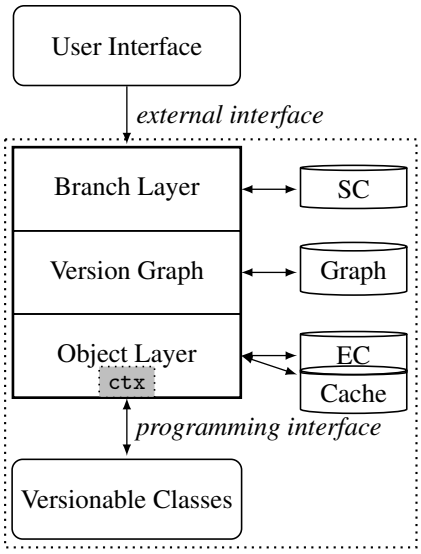


Figure 7: VERCAST layered architecture

For now we do not describe how the different branches are synchronized. We return to that later, in Sect. 7.3.

6. Design and Implementation

In this section we discuss the design and implementation status of VERCAST.

6.1 Architecture

VERCAST's design has a layered architecture, comprising three layers, each of which uses a different type of NoSQL database (Fig. 7):

Object layer The object layer implements the programming interface (Sect. 5.1). It works with individual objects, that together construct the full application state. It handles storing versions in persistent storage (which can be an EC data-store) and caching them for fast access.

Version graph layer The version graph layer works with the application's root object, and hence handles the full application state. It maintains a version graph (Sect. 4.3). Its main role is to support merges by finding, for two versions in the graph, the LCA, and the path from the LCA to each version needed for the merge algorithm. It uses a graph database that is expected to work efficiently even for large graphs.

Branch layer The branch layer implements VERCAST's external interface and tracks individual branches. A branch holds a notion of a *current* state in form of a mutable variable holding a version ID. This value needs to be maintained as a single value (per branch) regardless of which physical server serves it. To accommodate this, we use a SC datastore, that supports atomic updates.

6.2 Optimizations

The need to store every version of every object and the need to re-apply patches during merges may incur a high overhead in terms of both disk space, I/O operations and computation. To minimize this overhead we apply several optimizations. To improve merge performance we cache transition results in the object layer. If a transition that already occurred needs to be performed again (typically, as part of a merge), we fetch the resulting version from the cache.

To avoid storing all versions of all objects, we use the inherent redundancy VC has to offer. VERCAST has access to both the content of each version, and the patches that constructed it. Storing the content of all versions is good for read performance but bad for write performance and disk space consumption. Storing only the patches require much less disk space and improves write performance, but requires calculating the state of each object by applying patches on top of each other when performing read operations.

To balance between these two representations, we take a combined approach. First, we cache recently accessed object versions, for fast reads. Second, we split the state of the application into pieces we call *buckets*. Buckets are limited in both space and time. They are limited in both the number of objects they contain (a sub-tree of the application state rooted at a single object), and in the time-frame they represent (a certain number of versions of the bucket's root object). On disk, we store for each bucket its initial state (full object content), and the patches that are needed to transform it to all its other versions.

When reading a version, we first look it up in the cache. If not there, we extract the bucket that contains it, by reading the initial versions into the cache, and then applying the patches to the bucket's root version. The bucket size becomes an important parameter for tunning performance, as it determines the balance between disk and CPU consumption.

6.3 Implementation Status

A prototype implementation of VERCAST in Javascript over Node.JS is in progress.[7] At the time of writing this paper, our implementation consists of 1593 lines of Javascript code, backed by 157 unit tests. We implemented the three layers discussed in Sect. 6.1, including the optimizations discussed in Sect. 6.2. Unit-tests verify, among other things, the merging algorithm (Sect 4.3) and the conflict resolution algorithm (Sect. 7.2). Currently, the prototype implementation does not access any real database. Instead, to facilitate unit tests, we have in-memory stubs that implement the interface designated for the storage associated with each layer. The implementation includes several generic data structures that are useful as basic building blocks for applications, e.g., atom, counter, array and map.[8]

[7] `https://github.com/brosenan/vercast`

[8] The hotel booking example discussed in this paper is not included in the repository at this point.

7. Discussion

Next, we discuss some of the considerations that led to our design.

7.1 Database or Software Framework?

An important design decision we made was to implement VERCAST as a software framework that application developers can use to implement *user-defined* versionable classes. This is in contrast to an alternative design where one provides a fixed set of versionable storage primitives, and user code manipulates these primitives from the outside. Such an approach would resemble a database supporting VC on its values.

We now describe a scenario where the latter design will fail. Imagine we wish to provide our hotel booking system the capability of listing all hotels with vacancies in a certain city on a certain day. One approach would be to go through all hotels in the desired city and find the ones for which the counter corresponding to that day contains a positive value. Unfortunately, in cities with many hotels this can take a long time. A more appropriate approach would be to maintain, for each city and each day, a set of available hotels. This practice of holding redundant information in a database to accommodate fast query is called *de-normalization*, and is common practice in large-scale applications, especially in conjunction with NoSQL databases [10]. When some hotel vacancy counter reaches zero for one or more dates, the application removes that hotel from the sets corresponding to these dates.

Now imagine a scenario where there are initially two available rooms in a hotel. Two users in two remote geographic locations book a room simultaneously. Some time afterwards, the two transactions performed by the two users merge. Because two rooms were available, these transactions do not conflict, and the reservation of both users is confirmed. The question is, now that the number of vacancies got down to zero, will the hotel be removed from the set of available hotels?

If we chose the option of using storage primitives (or a database with VC), the code that decides whether to remove the hotel from the set executes twice, as response to each user's booking. Because each user made the booking when there were two available rooms, neither one of the executions would remove the hotel. The merge operation will not remove the hotel either, because it is performed between two sets that contain the hotel.

However, if we placed this logic in a versionable class such as the *Hotel* class in Fig. 5, the code that decides whether or not to remove the hotel from the set would run three times: once per each booking (with vacancies going down from 2 to 1), and a third time during the merge, when one of the patches is re-applied on top of the other. In this last application, the number of vacancies will go from 1 to 0, causing the hotel to be removed from the set.

```
let
 δq ← {
  _type: applyRange,
  from: this.initial + δ.start,
  to: this.initial + δ.end,
  patch: {_type: get} };
 (_,vacancies) ← ctx.trans(δq);
in
 for i ∈ 0...size(vacancies) do
  if vacancies[i] = 0
  then let δe ← {
    _type: remove,
    _key: (this.city,
           δ.roomType,
           this.initial + i),
    item: this.name };
   in
    ctx.effect(δe);
   end
end
```

Figure 8: De-normalization example

In VERCAST, such logic can be implemented using the effect set. For example, the code in Fig. 8 can be added to the *book* method of the *Hotel* class. This code first fetches the counter values for all effected counters, and then, for each counter that reached 0 emits an effect patch that will remove the hotel's name from the set identified by the tupple (city name, room type, date).[9]

7.2 Merge Conflict Resolution

Conflicts are inherent in VC. In order to allow large applications to be built around VC, we need a systematic way of handling merge conflicts.

Avoiding Conflicts Undoubtedly, the best way to handle merge conflicts is to avoid them altogether. Data structures such as *Conflict-free Replicated Data Types* (*CRDTs*) [20] have well-defined behavior when merging two instances, and therefore, they never conflict. With VERCAST, we can implement similar conflict-free objects. One example is the counter in Fig. 4, with `bounded` set to false.

Resolving Conflicts In some cases, such as the hotel booking example, conflicts are used to assure the system state remains intact. This gives rise to the need for a robust conflict resolution algorithm.

Fig. 9 depicts how conflicts are resolved in VERCAST. Recall that the *pull* operation defined in the external interface (Sect. 5.2) resolves conflicts by preferring the version it takes as second argument (which we call the *superior*, denoted as v_s in Fig. 9) over its first argument (which we call the

[9] This code assumes we initialized two extra member variables: *name* containing the name of the hotel, and *city*, containing the name of the city the hotel is located in.

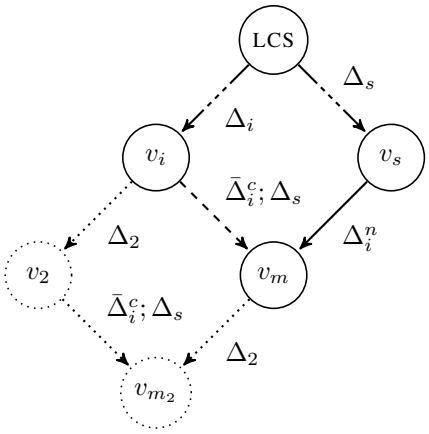


Figure 9: Conflict resolution

inferior, denoted v_i). The underlying merge operation, as described in Sect. 4.3, attempts to apply the patches Δ_i that constructed v_i, on top of v_s. If some of them conflict, they are ignored, and the merged version (v_m) is achieved by only applying Δ_i^n, the non-conflicting subset of Δ_i. When recording the merge, the edge from v_s to v_m records the patches that were actually applied, and the edge from v_i to v_m records v_i's defeat: it first contains the inverse of all conflicting patches Δ_i^c in Δ_i (in reverse order), and then the patches that constructed v_s.

When a second merge is being performed (in this case, between v_m and v_2, a descendant of v_i), the decision to roll-back Δ_i^c remains in place. Even if v_2 is superior in the second merge, the first patches it will need to apply are $\bar{\Delta}_i^c$, undoing Δ_i^c.

The correctness of this algorithm (in particular, the correctness of the label of the edge from v_i to v_m) can be proven by induction on the length of Δ_i.

7.3 Eventual Consistency

We showed how merged versions maintain the decision as to which patches need to be undone. However, for any two conflicting versions v_1 and v_2, two possible merged versions can be produced: one preferring v_1, and the other preferring v_2. Eventual consistency [4] requires that each conflict shall always be resolved the same way. In VERCAST, we cannot guarantee that random merges will always result in the same decisions, but we can guarantee that each conflict is resolved only once.

In applications such as the hotel booking example, where eventual consistency is required, we can apply hierarchy to branches (as we did in Sect. 3.2), where each branch has exactly one parent, except for the root branch (β_0) that has none. We can assure that each conflict is resolved only once if we allow *pull* operations to only be performed from a direct ancestor. This way, conflicts are always resolved by preferring the parent. For example, the algorithm in Fig. 10 will complete the hotel booking example by adding propagation between branches, assuring eventual consistency. This is quite similar to the pull/push, or update/commit sequence that is common when working with SCM systems. Note that we ignore the status returned by both *push* operations. If a conflict is encountered, it will be resolved the next time around. If a branch is temporarily unavailable, we will automatically try again later.

```
1 while true let
2   β ← choose a non-root branch at random;
3   β_p ← β's parent in the hierarchy;
4   [v] ← pull(head(β), β_p);
5 in
6   push(β, [v]);
7   push(β_p, [v]);
8 end
```

Figure 10: Patch propagation sequence

8. Related Work

Finally, we compare VERCAST to the state-of-the-art.

8.1 Optimistic Replication

Our work falls under the category of *optimistic replication*, a collection of methods in distributed systems designed to provide some consistency guarantees without the use of locks. The first important work in the field, with respect to persistent state, was Bayou [23], a distributed system that gave its users the view of a single relational database, where in fact it consisted of multiple instances, synchronizing by sending each other SQL updates. The system used complex algorithms to ensure the correct order of applying these updates, and often needed to undo an redo changes, to get the order right. Bayou turns to the user to provide conflict resolution methods. In comparison, in our work the user defines how objects respond to change in general, and does not have to worry about merge conflicts. For that, we use a general algorithm (Sect. 7.2).

More recent work includes *Conflict-free Replicated Data Types* (*CRDTs*) [20], *Cloud Types* [4] and *Isolation Types* [3]. While different in purpose and in the design decisions that led to their conception, these are all data types designed to hold concurrent replicas of data, with well-defined merging behavior. These solutions focus on providing primitives such as counters, sets and maps. To achieve a fully mergeable state applications are expected to construct their state based on these primitives. Unfortunately, creating custom types is considered hard [26, Sect. 2.2.1]. In comparison, in our work we allow such primitives to be provided by the framework and as software libraries, but we also allow users to implement their own versionable classes, such as the versionable Hotel class (Fig. 5). This is essential for maintaining the integrity of the state when de-normalizing the data (Sect. 7.1). The key idea that makes custom objects practical in our

model is that in our model objects only need to define their behavior in response to patches applied to them. Specifically, objects do not need to specify a merging behavior. VERCAST implements merges by applying patches to objects, based on a version graph (Sect. 4.3). In fact, version graphs can be seen as a generalization of *revision diagrams* [3] used in the implementation of isolation and cloud types. VERCAST's branch hierarchy model (Sect. 7.3) provides similar guarantees (eventual consistency), but is still more permissive than revision diagrams.

It is worth mentioning that Cloud Types implement a transaction mechanism similar to our own. Their *yield* operation bundles some updates together in a transaction, guaranteeing atomicity. Their *flush* operation adds forced synchronization, similar to our pulling from and pushing to β_0. However, because merges are designed in Cloud Types to always succeed, it is hard to tell how they can fail gracefully, e.g., in the conflict scenario described in Sect. 3.

8.2 Operational Transformations

Operational Transformations (*OT*) [9] are a method that allows concurrent edits to documents, implemented in systems such as Google Drive and Apache Wave (formerly, Google Wave). The idea is that in order to support concurrent updates, updates must take the form of patches. When describing changes to a document, these patches must reference locations in the document, such as line numbers. However, since concurrent updates may move parts of the document (e.g., inserting a line may shift all following line numbers), re-applying the original patches will not work as expected. OT proposes to transform all patches to be applied based on patches already applied to the state.

In VERCAST we currently do not support OT. As a result, a patch addressing an object that moved will conflict. Finding ways in which OT can be introduced to VERCAST to provide good support for document-like state is a topic for future work.

8.3 NoSQL over Git

A somewhat complementary approach to our own, taken by some practitioners [14, 17], involves using an existing SCM to store data, similar to our example in Sect. 3. The approach presented by Keepers [14] uses Git's internals directly to maintain a persistent tree similar to the one in our design. However, he reports a performance hard limit derived from the need to store each new version on disk. This hard limit makes his solution inadequate for large datasets. In our work we start where they left off, and provide a solution that endures large amounts of data and rapid modifications. We achieve that by using heavy-duty NoSQL databases for storage, in concert with the optimization described in Sect. 6.2.

9. Conclusion

In this paper we present an approach in which version control techniques can provide application developers with fine-grained control over the consistency model used in maintaining application state. This allows developers to explicitly express their preferences regarding the CAP [12] consistency vs. availability trade-off. We discuss how placing the state of applications under VC can be made practical. For concreteness, we describe a design of a system we name VERCAST.

A significant benefit of the approach is in relieving application developers from needing to choose *a priori* a NoSQL database. Currently, due to the lack of standardization in the NoSQL world, and the tight coupling between consistency models, data models and APIs, application developers need to choose a NoSQL database for each task at an early stage. Moreover, they must get it right upfront. The price of migrating from one database to another is high, and gets higher as the software grows in size.

By using VC, the data model is defined (either directly or through a software library) by the application and not by the database. The consistency model is determined at runtime by deciding when to merge and with whom. Application developers do not have to rely on choices made by NoSQL database developers: VERCAST invests the power in their hands.

While we focused on VC as a way to control CAP tradeoffs, there are other advantages to the use of VC for maintaining application state. One such advantage is in improving usability of client/server applications, and mainly of Web applications, by making modifications non-destructive. When a Web application stores state in a database, any modification to a record destroys the previously stored value. Features such as history tracking or even an undo/redo stack require special treatment on the infrastructure level, and are not common in applications in general.

VC can fundamentally change this. When using VC, modifications are stored as new versions, and do not destroy previous versions. Users can view the state of the system at any historical date, or ask to roll-back to that state. This can support undo and redo operations in any application with very little effort on the developers' part.

VC can also offer the same benefits SCM provides software development teams. Client/server applications are used today for a wide range of uses, some of which are creative work that involves entire teams. Two examples are content management, with systems such as WordPress or MediaWiki, and requirement management (or application lifecycle management), with systems such as Rational DOORS. Both tasks require cooperation between people, and the content that goes into these systems often involve a significant investment. VC can offer a better protection of the investment in the sense that any change is reversible, and work can continue from any point in history. VC can also enable better collaboration in the sense that each developer can see a consistent view of the state on his or her own branch, and integration can be done at the time of their choosing using explicit merges.

References

[1] E. A. Abbott. *Flatland: A Romance of Many Dimensions*. Seeley & Company, 1884.

[2] K. Banker. *MongoDB in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[3] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'10)*, pages 691–707, Reno/Tahoe, Nevada, USA, Oct. 2010. ACM SIGPLAN Notices 45(10) Oct. 2010.

[4] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*, number 7313 in Lecture Notes in Computer Science, pages 283–307, Beijing, China, June 2012. Springer.

[5] S. Chacon. *Pro Git*. Expert's voice in software development. Apress, Berkeley, CA, USA, 2009.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[7] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, pages 205–220, Stevenson, WA, Oct. 2007. ACM SIGOPS Operating Systems Review 41(6) Dec. 2007.

[9] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD'89)*, pages 399–407, Portland, Oregon, USA, 1989. ACM SIGMOD Record 18(2) June 1989.

[10] E. Evans. Cassandra by example. `http://www.rackspace.com/blog/cassandra-by-example/`, 2010. [Online; accessed 17-July-2013].

[11] D. Featherston. Cassandra: Principles and application. *University of Illinois at Urbana-Champaign*, 2010. `http://d2fn.com/cassandra-cs591-su10-fthrstn2.pdf`.

[12] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[13] R. Hickey. Clojure: Refs and transactions. `http://clojure.org/refs`, 2014. [Online; accessed 18-June-2014].

[14] B. Keepers. Git: the NoSQL database. `https://speakerdeck.com/bkeepers/git-the-nosql-database/`, 2012. [Online; accessed 15-March-2014].

[15] D. H. Lorenz and J. Vlissides. Designing components versus objects: A transformational approach. In *Proceedings of the 23th International Conference on Software Engineering (ICSE'01)*, pages 253–262, Toronto, Canada, May 12-19 2001. IEEE Computer Society.

[16] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[17] R. Pollock. We need distributed revision/version control for data. `http://blog.okfn.org/2010/07/12/we-need-distributed-revisionversion-control-for-data/`, 2010. [Online; accessed 19-March-2014].

[18] M. J. Rochkind. The source code control system. *IEEE Trans. Software Eng.*, 1(4):364–370, 1975.

[19] P. J. Sadalage and M. Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, 2012.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[21] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'95)*, pages 204–213, Ottawa, Ontario, Canada, Aug. 1995. ACM.

[22] W. Swierstra and A. Löh. The semantics of version control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*, Portland, OR, USA, Oct. 2014. ACM.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'95)*, pages 172–182, Copper Mountain Resort, Colorado, Dec. 1995. ACM SIGOPS Operating Systems Review 29(5) Dec. 1995.

[24] J. Webber. "Wavefront" workshop: A programmatic introduction to Neo4j. In *Proceedings of the 3rd International Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12)*, page 217, Tucson, AZ, USA, Oct. 2012. ACM.

[25] M. Woelker. Persistent trees in git, Clojure and CouchDB. `http://eclipsesource.com/blogs/2009/12/13/persistent-trees-in-git-clojure-and-couchdb-data-structure-convergence/`, 2009. [Online; accessed 14-March-2014].

[26] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. Technical Report INRIA/RR–8347, Inria Rocquencourt Research Centre, Paris, France, Aug. 2013. *CoRR*, abs/1310.3107, Oct. 2013.