Orthogonal Synchronization for Software Services

Sebastián Krynski
National University of Quilmes (UNQ)
skrynski@gmail.com

March 7, 2019

Abstract

In the past several years as the internet became ubiquitous, application development transitioned from a *(*stand alone) desktop model to a (web and mobile) *service oriented* model. The first model was usually self contained, easier to write, difficult to upgrade and (only) locally available. In the second model, applications are always updated and available from anywhere in the world at anytime. However, they became a lot harder to write and usually will not work with unreliable or nonexistent connections. *Software services* as conceived by Bracha [Bra06] seek to combine the advantages of traditional client applications and Internet services. In this work we intend to materialize and further develop the cited work. We will describe the architecture and design of a platform for software services in the context of object oriented languages.

1 Introduction

*W*eb and mobile service applications (or *Internet services*, in general) have shown advantages over the traditional model of stand alone desktop applications. Internet services are always available and always up to date.

Availability means the service can be reached from anywhere in the world (it doesn't matter where the service is hosted physically) at any time.

Up to date means that the user or developer is always implicitly communicating with/receiving the latest incarnation of the service. There is no concept of version since there's always only one version available at any given time, so that burden is taken away from the user.

However, this model has its downsides.

On web applications

- The network connection may not be available or robust enough, rendering the service unusable.
- Considering the current web standards (HTML, JS, AJAX, etc), creating a rich interactive application which communicates with a service is a complex task.
- It is not always clear for the user what the application lifecycle is (e.g., has it ended once the browser tab is closed?).
- The interaction with the host operating system is limited.
- Updating the application is done by reloading the page, which is the typical shut down-install-restart cycle. There is no live update (even when Javascript would support such capability), hence the current user's work needs to be interrupted in the process.

On mobile applications

- Mechanisms for data synchronization and local persistence need to be carefully implemented by the developer in every application.

- The application update mechanism is handled by the operating system. Even when the user is not explicitly aware of the current application's version, the process involves restarting the application in order to install the new version. We will discuss these aspects in detail later in Section 7,

Software services, as defined in [Bra06], seek to combine the best of both models: traditional stand alone applications and *service-based* applications.

This model enables applications to be usable even when there is not a reliable network connection. Since applications must be usable offline, they need to be able to run on a client device and have the necessary data to work locally (e.g., a local object data store). The moment the application gets back online a synchronization process will take place. Local data will be sent to a server so it can be accessible from other devices or users, and reliably backed up. Likewise, new data (produced on other devices) will be received from the server and installed locally.

Since multiple users can modify the same piece of information simultaneously, the server needs to be capable of reconciling and maintaining a unified version of the information. Typically, this process would require complicated application-specific code that would pollute the core business logic. The synchronization process is also responsible for the update of the application itself and the corresponding data format that each version assumes, so both application and data will evolve simultaneously.

The reader might associate the description given above with a typical mobile application architecture. However, there are important differences. For instance, the developer of the application will not need to get involved with aspects such as synchronization, reconciliation of information and the persistence of local data. From the user's perspective, we intend to have an application's update mechanism that is not invasive with respect to the user's current work (e.g. not having to restart the application on every update).

The aim of this work is to go deeper into the concept of *Software Services*. We intend to model and implement a platform that would allow applications developed in this environment to benefit from it, both during development and in production.

This work will focus on *Orthogonal Synchronization* and *Program as Data* as its main features.

- *Orthogonal Synchronizability*. This is a concept introduced in [Bra06]. Synchronization focuses on reconciling two parts of the same data that are stored in different places. Orthogonal synchronizability implies that any object in an application can be synchronized without the need of any *ad hoc* or application-specific development. The synchronization process should be part of the programming framework and ready to use, resulting in more robust and performant systems, with less effort and in less time.
- *Programs as Data*. As mentioned earlier, during synchronization new information is sent from the server to the client in order to keep the application up to date. This new data may include modifications to the program itself. In this regard, we are considering programs as data that can be manipulated and transferred. Said modifications will need to be applied without affecting the current user's work (the program cannot be stopped or restarted).

  The language should support all the needed reflective capabilities and support for live modification. Languages like Smalltalk [GR83], Self [US87], Lisp [KdRB91] and other dynamic languages have these abilities. However the update process is not straightforward.

  The concept of *Program as Data* is fundamental for accomplishing the goal of a *version-free* application.

We are currently implementing the ideas described here in a prototype using *Newspeak* ([Bra], [BvdAB^{+}10]) (a descendant of Smalltalk [GR83] and influenced by Self [US87]). We will now present the core pieces of the platform.

1.1 Roadmap

The rest of this document is structured as follows. Sections 2, 3 and 4 describe the core pieces of the platform which are the main contributions of this work. Section 2 outlines the design of the Versions Server component, in charge of storing the information and communicating with clients. Section 3 describes the mechanism of client-server synchronization in detail. Section 4 presents the Client Infrastructure and how it interacts with the Versions Server. Section 5 briefly describes key features of Newspeak and the rationale for implementing the platform using this language. Section 6 describes the current status of the project, followed by a discussion of related work (section 7) and conclusions (section 8).

2 Versions Server

The fundamental component in the platform is the *Versions Server*. The Versions Server is a repository composed out of *stores*.

A *store* is a universe of objects. Objects inside a store can be of any kind, from module definitions to plain objects. A store can be thought of as being analogous to a working directory, and the objects within it as analogous to files. As a store changes and evolves over time, *Store Versions* are created to reflect that evolution.

Versions A *version* represents the state of an object at a specific point in time. It is created whenever a new change occurred on an object. Formally, a version is either the null version, ε, or the pair (o,t) where o is an object and t a timestamp. Conceptually, any kind of object susceptible to change can be versioned, from entire stores to individual plain objects.

We propose the existence of two levels of versioning within a store:

- At a higher level, *store versions* which evolve representing specific states in a store.
- At the lower level, *object versions* are created as objects within a store are modified. They are responsible for storing the actual data that has changed.

Whenever a new object (store or regular object) is registered on the server, an *initial version* is created on its behalf. At this moment, this object will be associated with a fresh *global unique identifier* (or *guid*). The object's guid will be used universally across different clients to refer to it. Only the *Versions Server* is entitled to create such identifiers, so there is no risk of collision. Subsequent modifications of the same object will produce new versions. Versions will also be associated with a sequential *version number* that will be used to identify a specific version of the same object.

Every version chain will be preceded by an initial version. The initial version will be responsible for holding the guid and for generating the version number for future versions of the same object. The initial version will not hold any information related to the actual object that is being versioned.

Store versions

A store version can be thought of as being responsible for keeping track of a particular set of object versions. It represents a snapshot of the store at a particular moment. When an object inside a store changes, it means that both the store and that particular object have changed. In consequence, a new store version and a new object version will be created. Since we are only modifying the store through the synchronization process, typically a set of incoming object changes will be associated to a new store version.

On the implementation level, store versions will only be aware of the object versions created in that version. This will avoid having to associate a store version with a large group of object versions that were created in prior versions. Moreover, a store version will also have information about the objects that have been garbage collected.

Object Versions

We want to keep track of different versions of the same object. We have decided to store the information in separate structures independent of the actual objects that are being versioned. One might be tempted to represent the versions of an object as different instances of the object's class. This is not a good idea. Since said objects will be all coupled with the same class, any kind of modification to the class would directly affect the objects. For example, removing a slot in a class would cause the slot to be removed in all the instances, hence losing information.

We have considered the following options regarding the storage of the information on the repository: take a snapshot of the new state of the object, store the changes relative to its parent versions, or do both.

We decided to separate the representation of objects over time into an initial object version and a series of changes relative to the parent version.

Storing changes instead of snapshots may be convenient for different purposes:

- To avoid storing the complete set of information in every version. This will be at the expense of an increase in processing time for recovering the actual information.
- To have valuable information to understand what has changed in between different versions, that might not be possible to recover otherwise. This will vary depending on the nature of the information that is tracked. For instance, the renaming of a class cannot be detected by comparing two versions of the code. A deletion and a creation will be detected instead. Moreover, this will also depend on how the entity is identified on the platform (e.g. is the class identified by its name or by a guid?).
- As a way to compute differences between versions without the need of comparing the whole set of information

The type of the stored changes will be different based on the type of information being described. For example, the basic object changes will be different from program changes. We have not implemented program changes yet. However, we consider them to be essential to have a relatively useful prototype. Moreover, we think that the platform should be open to allow custom defined ways of storing the information based on the type of the objects. This comprises not only a method for storing the evolution of objects, but also a way for computing changes between versions and serializing those changes to a client.

Next, we will describe *simple-object* changes.

Simple-object changes

Simple-object changes consist of an initial *Object Definition* and subsequent *Object Change Record*s.

- *Object definition*. It contains a reference to the defined object's class and the values for every slot in the object.
- *Object change record*. It has the new values for the slots that have changed.

Values stored within changes can be primitive objects (e.g. numbers, strings, booleans, etc.) or object references. An *object reference* is just a carrier of an object's guid. Object changes will refer to other objects by their guid and not by the object version number. This decision avoids coupling object versions unnecessarily. It also means that to recover the state of a certain object, every reference must be resolved by accessing the current store version.

Example: A Simple Store

Figure 1 shows a simple scenario with store versions at the top and object versions at the bottom. Dotted lines represent the parent relationship between versions. Dashed lines are the direct connections between store versions and object versions.

At store version 1, an object (guid 1) instance of *Bar* class was added with *nil* in slot "bar" . Then at store version 3 the same object changed the content of slot "bar" to a reference to the object with guid 2.

At store version 2, an object (guid 2) instance of *Foo* class was added with 'hello' in slot "foo". Then at store version 4 the same object changed the content of slot "foo" to 'bye'.

The reader should notice that:

- The parent relationship of object versions varies independently to the parent relationship of store versions.
- New version chains are preceeded by an initial version containing the object's guid.

The figure only depicts how information is stored within store versions. However, there is more information that can be derived from this schema. For example, object with guid 2 at version 1 is also included in store version 3 even if they are not directly connected.

3 Synchronization Scheme

As mentioned before, we want the client to always be *up-to-date*. Of course, since the client can have periods being *offline* this is not completely feasible. What can be done is to get the client updated to the latest version on every synchronization. We call each such version the *anchor* of a store.

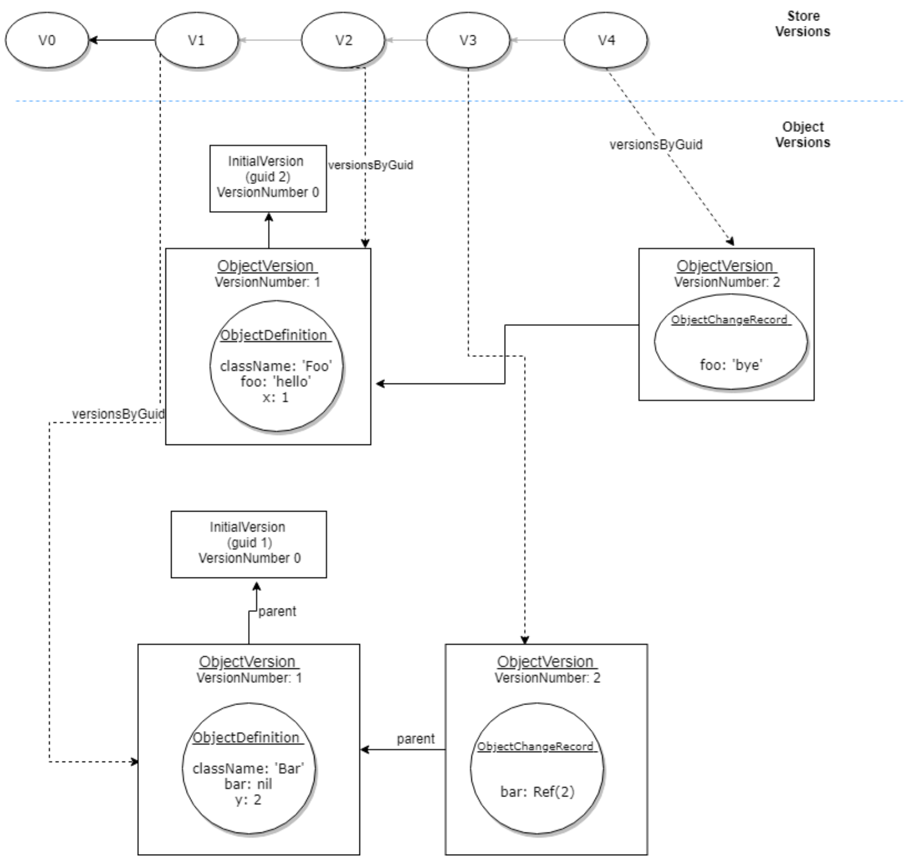

Figure 1: A Simple Store.

We will now describe the synchronization process. First, the client sends to the server only the changes that occurred since the last synchronization. Using these changes, the server (which remembers the previous state of the client) will reconstruct the client's current state. After that, the newly computed client state is reconciled with the current anchor, creating then a new store version which will be set as the new anchor. Finally, the server will compute the differences between the anchor and the state on the client's device, and those changes will be sent to the client. Only changes or *deltas* will be transferred between the client and server, allowing faster synchronization times and hence the possibility of doing frequent syncs.

Since the reconciliation process is not a simple task, it will always occur at the server to avoid consuming client's device resources.

Example: A Simple Synchronization Scenario

Figure 2 depicts a simple synchronization process. At t=1, client 1 is working on Version 2. At t=2, client 2 submits changes, creating Version 3. In that moment, Version 3 will become the new anchor. Later on (t=3) client 1 decides to upload its changes. Since the Server knows that Client 1 was at Version 2 in the previous sync, Version 2' will be created having Version 2 as its parent. At this point (t=4), a merge between Version 2' and Version 3 is performed, creating Version 4.

In the last part of the process, client 1 will be sent the proper changes to update from version 2' to version 4. The server is responsible for computing this change set. Two things should be noticed in this part. First, computing this changeset does not require the creation of any additional versions. Second, since version 4 has more than one parent and possibly merge decisions were taken in that process, computing such a changeset is not a trivial operation. For example, just

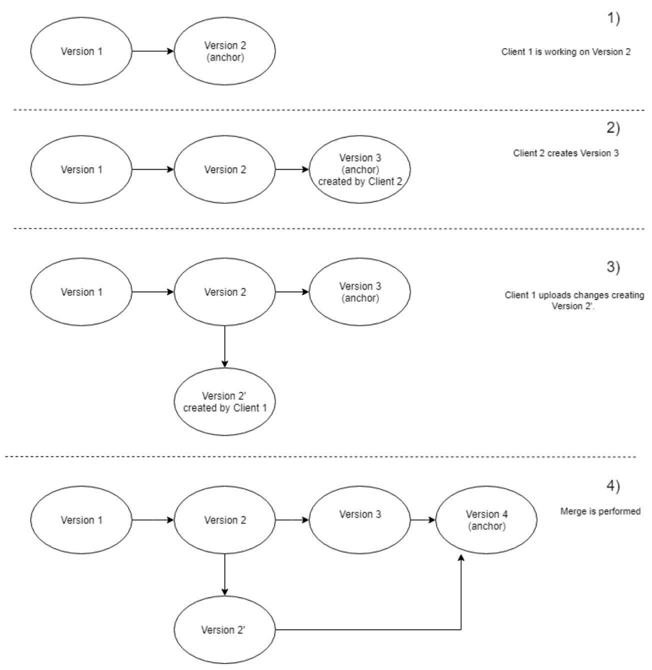
1)
Version 1
Version 2 (anchor)
Client 1 is working on Version 2
2)
Version 1
Version 2
Version 3 (anchor) created by Client 2
Client 2 creates Version 3
3)
Version 1
Version 2
Version 3 (anchor)
Version 2' created by Client 1
Client 1 uploads changes creating Version 2'.
4)
Version 1
Version 2
Version 3
Version 4 (anchor)
Version 2'
Merge is performed

Figure 2: Sync schema.

sending back the changes done at Version 3 may not be adequate.

Merging Versions

Two versions, P1 and P2, can be *merged* to produce a new version C. We say that P1 and P2 are *parents* of C. Only versions of the same object can be merged. The resulting version will be of the same type as the parents. We can now think about what the history of a version means. Formally:

The history of the *empty* version is the *empty* history.
History$(\varepsilon) = ()$

History(V) = (V, History(P1), History(P2)) where P1, P2 are the parents of V

The Tip of a History of a version V is V.
Tip(History(V)) = V

The concept of *history of a version* is conceptually relevant because it represents how information evolved over time until reaching the version's state.

For example, the merge operation between two revisions in a typical Source Control System not only merges the two selected versions, but also takes into consideration the history of those versions. This is to say that two *histories* are being merged. In this case, the merge function will be able to make decisions and resolve conflicts with more information.

As we can expect when reconciling two versions of the same object, different problems may arise. We refer to such situations as *merge conflicts*. Depending on the type of conflict presented, different strategies or actions can be taken. Some strategies might be able to solve conflicts automatically (e.g last version wins) and other more complex situations may require the interaction of a user. Conflict detection and resolution in a merge process is a complex topic and exceeds the scope of this presentation. However, we intend to put considerable effort in this direction to develop something that is useful and consistent.

Merging store versions should have the following outcome:

- A new store version is created. This version should have the merged versions as its parents
- Object versions with conflicts will be merged to create a new object version which reconciles the information of its parents.
- The new store version will be directly associated with the new object versions created in this process. Objects that are no longer part of the new store version will be identified.
- No new objects (i.e. object versions associated with object definitions) will be created as a result of this process.

Transient Slots

When programming in object oriented languages, it is a common practice to have objects which hold in instance variables values that can be derived from its other instance variables. They may be stored for performance reasons or even for convenience. Usually this data is computed on demand and cached for future calls. We call such data *transient*.

Since transient data can be derived from others, it does not make sense to store it in a persistent medium. Regarding synchronization, we want to avoid sending transient data over the wire and let the clients perform the recomputation when needed.

We introduce *transient slots* to store transient data so that the system can identify such data.

The declaration of a transient slot in Newspeak can be done as follows:

```
transient m1 = (1 + 2).
```

We are declaring the transient slot `m1`, which externally will behave (and be invoked) as any regular slot or method. Sending the message `m1` to an object o (e.g. `o m1`) will return the value 3. However, since `m1` is a transient slot, the actual computation `1+2` will only be performed on the first send of `m1`. Subsequent calls to `m1` will not compute `1+2`, but only return the cached value `3`.

Cached values in transient slots are not synchronized. Since certain values in an object may result in a different value for a transient slot, transient slots caches are reset at the end of the synchronization process. This will force such values to be recomputed from the new state.

In the context of this work, we have incorporated transient slots into Newspeak [1]. As with any modification to the syntax of a language, a considerable amount of work is needed to make the new feature interact and cohabit naturally with the rest of the environment.

4 The Client Side

4.1 Client Application Infrastructure (CAI)

We will now describe how a user would interact with an existing service. There needs to be a platform component on the client end that is in charge of subscribing to applications located on the server. We call this component *CAI*. CAI can be thought of as being equivalent to the *AppleStore* or *PlayStore* application installed on mobile devices, which enables a user to install/subscribe and uninstall applications. As of the moment, we have not implemented this component, hence the following description will be brief. For simplicity, we are assuming CAI is installed on the client device..

The responsibility of this component is to allow users to subscribe to different stores or applications. It will be aware of all the applications the user has subscribed to.

At the moment of the subscription, an initial synchronization will be performed where user data and the application itself are going to be retrieved. From that moment, the application is installed and the platform is in charge of the

[1] Newspeak Programming Language Draft Specification: `http://newspeaklanguage.org/spec/newspeak-spec.pdf`

application's life cycle. As mentioned before, synchronizations will be handled by CAI. It is important to restate that future application upgrades as well as user data will be part of the same synchronization process.

After the initial synchronization, an object identified as the *root* will be retained by the platform. This object will contain a reference to the application that is to be executed as well as user data objects. Because this root object is the one that references (transitively) every other object in the application, the platform itself needs to retain said object (otherwise the whole object graph would be garbage collected).

We do not yet have a clear idea as of when the periodic synchronizations are going to be performed, or who needs to be the initiator of the synchronization. Most likely the platform will need to find moments where the application is *quiescent* [Bra06] to trigger the action. Also, probably the platform would need to provide the application (at the moment of the instantiation) with an object to enable certain type of communication from the application to the platform. For example, it would be desirable that an application user had the ability to force the application to sync.

4.2 Change Detection

In order to synchronize with the server, the client needs to be able to detect what has changed on its end. Detecting changes in an application is not an easy task and it's usually done with a custom implementation per every application. A common (not very efficient) approach is not to detect changes at all; instead the entire graph of objects is sent back and forth over the wire.

We want to relieve the programmer of this burden and let the platform handle this situation as a benefit of subscribing to the platform. This is what *Orthogonal Synchronization* is intended for.

During this process, changes to known objects have to be detected and recently created objects have to be discovered. To put more context into the problem, after doing an initial synchronization with the server, the client will have a set of objects with a corresponding guid. From this set of objects only one will be marked as the *persistent root object*. Changes will be detected in objects that belong to the transitive closure of the root. In this scenario, new objects created on the client will be reached from existing well known objects (otherwise they would haven been garbage collected and hence not interesting). New objects will not have a corresponding guid before synchronization.

We have considered different strategies for detecting changes:

1 Detect changes in the end. After every synchronization keep an entire copy of every object's state in a separate structure. At the moment of detecting changes, traverse the graph and compare the objects with their prior state. During the traversal, new objects can be detected too. Drawbacks: it is slow and requires a duplication in space in order to save the initial state of every object

2 Mark changed objects as they change by means of delegation. After every synchronization wrap every object with another object which will intercept any message sent to a write accessor and register the object as changed (after saving the initial state). Drawbacks: There are two main issues. First, it would consume twice the memory as each object would have its own wrapper. Second, as with any other wrappers in a class based language, the "self reference problem" may easily lead to inconsistencies.

3 Mark changed objects as they change by means of inheritance. Each client class would have a corresponding (dynamically generated) persistent class, which will inherit the real class. Objects will be in reality instances of the persistent classes. Persistent classes will override every slot accessor in order to mark the object as changed. Discussion: It is certainly more complicated to implement this mechanism. There are a lot of issues to consider when we decide to have duplicated mirrored classes. We have not tried this route yet.

4 Modify the virtual machine in order to have a dirty bit on every object and get notified the first time an object changes so it can be marked. Discussion: It is not too difficult to implement and will most likely give the greatest benefits in terms of performance.

As for now, we have implemented option 1 since it is the most straightforward. This will allow us move forward with other aspects of the platform and have a broader picture of the situation before deciding to undertake a more complex implementation.

4.3 Serialization

This section explains how data is transferred between clients and server.

The easiest solution would be to just take the objects we want to send and use an ”off the shelf” serializer. The problem is that serializers usually take an entire object graph and produce a serialized format to be reconstructed exactly on the deserializing end.

What we want is to send just the necessary changes to upgrade a client to a particular version. To better illustrate this situation, let’s suppose that at the server, *object1* changed the content for the slot ’foo’, from *object2* to *object3*. We must communicate this situation to the client (which in this scenario already knows object3). A regular serializer would serialize object1 as well as object3, when in fact object3 was not even changed at all. We would like to improve this mechanism.

We use change records that note when a particular object has changed the associated values for a subset of its slots.

Going back to our example, all we need to send is a change object which indicates that the object with id ”*guidFor(object1)*” changed the value for its slot ’foo’ to the object with id ”*guidFor(object3)*”.

Of course, another scenario may occur where object3 is not known by the client. In that case, the change log sent to the client will also include an object definition describing the state of object3 and its guid.

Finally, there is a slightly more complex situation in which the new object is being created on the client’s end. In this case, an object definition will be sent to the server along with a temporary *local identifier* generated on the client. After registering the new object, the server will respond with a corresponding mapping from local to global identifiers.

Other Serialization Mechanisms

As mentioned before, we aim to make this platform extensible enough to allow different ways for storing the information on the repository and for serializing such information, depending on the type of the object.

At the moment we have implemented a basic mechanism for storing and serializing plain objects, as described above. The next major milestone is to be able to properly store, serialize and install program changes. We want to provide at least a basic default implementation for dealing with this case.

5 Newspeak

As mentioned before, the platform is being developed in Newspeak [BvdAB$^+$10], on both server and client ends. Newspeak offers a set of features which we see of value to accomplish the vision of *Object as Services*: atomic install, use of accessors, modularity [BvdAB$^+$10], reflection through mirrors [BU04], security by capability and actors. These are already total or partially integrated in Newspeak. As these are not the focus of this work they won’t be discussed here in detail.

Two of these features are specially useful:

- Atomic install[2]. This is the ability to apply a set of program changes atomically to a live program (rolling back in case of errors) and also to apply these changes without having to care about the right order of application.

  We expect to benefit from this feature at the moment of receiving and installing program updates, allowing us to do it without shutting down the application during the process.
- Use of accessors. Data is always accessed or modified via message sends (as in Self). This allows us to place logic before and after a write accessor is executed.

6 Status

We are developing a prototype in order to validate our ideas and also as a way for learning and discovering more about the problem.

[2]Gilad Bracha. Room 101: Atomic install. Blog post at: `https://gbracha.blogspot.com/2009/10/atomic-install.html`

Versions Server We have implemented a basic *in-memory* Versions Server. This component receives a synchronization payload from the client and notifies the corresponding Store. The store, in turn, creates a Store Version which will keep the sent information in new Object Versions. The information is currently stored by Object Versions using the same change object that is being sent from the client. We are not performing merges on the server, only linear versions. The *Anchor version* in a store is always identified whenever new changes occur.

Client On the client end, a *Store Client* is in charge of communicating with the server for sending and receiving updates. The store client uses a *Change Tracking Context* for detecting changes on the client side. The Change Tracking Context saves the state of the initial object's graph and will later traverse the current graph to detect changes. As mentioned before, this is an easy but not performant implementation.

*T*he store client also is charge of holding the *root object* as well as the current version number the client is at. When new objects are sent to the server, an association table from *local-to-global* identifiers will be sent back to the client and applied locally.

Change Model Regarding changes, we currently deal with basic object changes and primitive values. Dealing with code changes is our next step.

Communication between client and server is currently in memory (on the same process). We have yet to try this model between separate processes and separate devices.

7 Related work

We will now describe the current state on mobile platforms and how they relate to our current work. We will also discuss related work regarding serialization of objects, and existing work on *dynamic software update*.

7.1 Mobile Platforms

In the most common mobile application platforms, *iOs* and *Android*, data and program synchronization are usually handled independently using separate mechanisms. Data synchronization is handled by the application itself and it's usually implemented ad-hoc for every different application. It's a complex task, error prone and more importantly, it clutters the application's domain logic.

Program update, on the other hand, is accomplished by the operating system's host which typically relies on a software store (e.g. *PlayStore* or *AppleStore*). The OS receives notifications from the store and decides when to download a new version. After that, the active application will be shut down (usually without even notifying the user) and replaced by the new version. Finally, the application will be relaunched. There is neither live nor incremental update.

Shutting down the application can take place not only in case of updates, but also whenever the operating system needs resources (memory, CPU, etc). In Android, view objects are also destroyed and recreated whenever the user rotates the screen. These situations force the developer to take actions in order to maintain a pleasant usability of the application. The 'in memory' state before the tear down (e.g. the stack of opened windows with the corresponding user's selections) must be manually serialized and persisted in a long term memory. The inverse process will follow at the moment of view's reconstruction. To deal with this situation, there are certain callbacks that can be placed by the developer in order to be notified when a view object is about to be destroyed or recreated, so that the corresponding action can be executed. As stated before, this type of situation needs to be dealt with ad hoc in every application.

In order to keep the program's code in sync with the data, a migration system embedded in the application will execute the first time the new version runs. This module will be in charge of updating the underlying persistent store structure to be compatible with the new code. The specific actions for updating the store will be embedded in the application's code and coded by hand by the developer. That is to say that every application version will have the knowledge to update it's persistent store from one version to another incrementally.

7.2 Serialization

Ungar's work [Ung95] provides a detailed insight of what is entailed in moving (or transporting) objects from one environment to another, and the aspects that need to be considered (e.g. identity, mutability, abstract representations, initial values). Even though this work is focused on a prototype-based language most of the concepts also apply to a class-based language as well.

Since we have a version-oriented approach, new situations arise such as dealing with global and local ids, or deciding whether to send a complete object description or just a reference to an object that is known by the other end. Moreover, we decided to use a *changes* model instead of the *pieces* model described in the cited work.

7.3 Dynamic Software Update in Development

Lisp introduced REPLs and the ability to modify running programs. However REPLs did not have to deal with more complicated situations that appear in the process of debugging. For instance, dealing with existing call stacks or live data that must be preserved in the modifications.

Updating objects in the heap caused by class modifications was first present in Smalltalk. Old versions of objects were replaced by the new ones having the updated structure, using an operation called *become*. Smalltalk did not provide a migration mechanism to update instances (besides copying the existing data). In Smalltalk it is possible to update the representation of a class atomically, but other changes must be done sequentially.

- The Strongtalk system [BG93] was the first to provide atomic installation of reflective changes [BBG+02], consisting of an API that received a list of pairs of old and new mixins to be updated, as well as all the objects associated with the old mixins. Live instances of the old mixins were considered in the update process.
- In Newspeak, the chosen platform for our work, the atomic install process is similar to the one used Strongtalk. The logic strongly relies on primitives such as *#become:* and *#allInstances*. As in Self, Newspeak will allow these kind reflective tasks by the means of a mirror system [BU04] .
- Pharo Smalltalk [3] also has a mechanism for updating multiple classes atomically.
- In *Dart*, [MMTB17] shows an efficient implementation of atomic update which is embedded in the virtual machine and deals with aspects of asynchrony (since Dart's basic units are *isolates*, which are actor-like entities communicated only by asynchronous message passing). Isolates are the units of hot reloading.
- In Prototype-based languages this process becomes much harder since there is not a clear connection between an object's structure and its declaration. In *Javascript*, individual objects can be reshaped by the programmer at any time. However, most implementations assume that objects are unlikely to be reshaped after construction. Doing so will make subsequent message sends much slower.

7.4 Dynamic Update in Production

We will next discuss existing work on dynamic update of running applications in production.

- Java *class loaders* [LB98] can be used to support a specific form of software update. For this practice to work, every object must be accessed only via interfaces. New classes conforming to such interfaces can be loaded to replace the old ones. Since classes cannot be modified, existing instances will still belong to the old classes (instances cannot be reshaped). New instances for the new classes must be created and migrated from the old ones. This means that applications must be designed carefully in advance in order to support this design pattern.
- *Rubah* [PVH14] uses class loaders, bytecode rewriting and modifications to the class pointers of objects to support dynamic update of production Java applications. Multiple changes can be installed as a unit. It also allows developers to specify custom object migrations in Java. Unlike JVolve [SHM09], no changes to the underlying VM are needed. However, modifications to the application source code are required. Code must be modified to identify valid update points. This is not desirable since the amount of code to be modified will be proportional to the size

[3] Pharo Smalltalk `https://pharo.org`

of the system (hence polluting the source code). Rubah also performs lazy updates by the use of proxys, in such a way that certain changes will take place on demand.

- *Erlang* [Arm13] has support for module updates. It allows modification of existing code, but not data. Existing stack contexts will be preserved and new calls will run with the new code.

8 Conclusions

We have described the structure of a platform supporting *Orthogonal Synchronization*, combining the advantages of *Internet services* and traditional client applications.

Our proposed architecture consists of a *Versions Server* based on the notion of versioned stores. We showed a synchronization scheme which seeks to keep a client device always updated (program and user data) by merging with the anchor on every synchronization, and thus enabling version-free application deployment.

Based on the notion of *Program as Data*, we proposed that code and data be versioned together on the same store, sharing the benefits of version operations like merging. This reinforces the idea that program and user data should evolve in *tandem* .

We have also described the *Client Application Infrastructure* in charge of the applications' life cycle, taking care of aspects like synchronization. This accompanies the presented idea of *Orthogonal Synchronization*.

References

[Arm13] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2013.

[BBG+02] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in Strongtalk. In *ECOOP '02 Workshop on Inheritance*, June 2002.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 215–230, New York, NY, USA, 1993. ACM.

[Bra] Gilad Bracha. The Newspeak Programming Language. `http://newspeaklanguage.org`.

[Bra06] Gilad Bracha. Objects as Software Services. `http://bracha.org/objectsAsSoftwareServices.pdf`, January 2006.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA'04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM.

[BvdAB+10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *ECOOP 2010*, 2010.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *In Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98), volume 33, number 10 of ACM SIGPLAN Notices*, pages 36–44. ACM Press, 1998.

[MMTB17] Ryan Macnak, John McCutchan, Todd Turnidge, and Gilad Bracha. Hot reloading and debugging in the dart virtual machine. 2017.

[PVH14] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Dsu for java on a stock jvm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 103–119, New York, NY, USA, 2014. ACM.

[SHM09] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A vm-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM.

[Ung95] David Ungar. Annotating objects for transport to other worlds. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 73–87, New York, NY, USA, 1995. ACM.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.