Fidelity-Aware Replication for Mobile Devices

Kaushik Veeraraghavan*
Microsoft Research Silicon Valley
Mountain View, CA 94043
kaushikv@eecs.umich.edu

Venugopalan Ramasubramanian
Microsoft Research Silicon Valley
Mountain View, CA 94043
rama@microsoft.com

Thomas L. Rodeheffer
Microsoft Research Silicon Valley
Mountain View, CA 94043
tomr@microsoft.com

Douglas B. Terry
Microsoft Research Silicon Valley
Mountain View, CA 94043
terry@microsoft.com

Ted Wobber
Microsoft Research Silicon Valley
Mountain View, CA 94043
wobber@microsoft.com

ABSTRACT

Mobile devices often store data in reduced resolutions or custom formats in order to accommodate resource constraints and tailor-made software. The Polyjuz framework enables sharing and synchronization of data across a collection of personal devices that use formats of different fidelity. Layered transparently between the application and an off-the-shelf replication platform, Polyjuz bridges the isolated worlds of different data formats. With Polyjuz, data items created or updated on high-fidelity devices—such as laptops and desktops—are automatically replicated onto low-fidelity, mobile devices. Similarly, data items updated on low-fidelity devices are reintegrated with their high-fidelity counterparts, when the application permits it. Polyjuz performs these fidelity reductions and reintegrations as devices exchange data in a peer-to-peer manner, ultimately extending the eventual-consistency guarantee of the underlying replication platform to the multi-fidelity universe.

In this paper, we present the design and implementation of Polyjuz and demonstrate its benefits for a fidelity-aware contacts management application.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

fidelity, replication, transcoding, weak consistency

1. INTRODUCTION

As personal portable devices proliferate, automated tools for sharing and keeping data up-to-date on a collection of devices are gaining widespread use. A practical problem, however, greatly hinders replication on mobile devices. In order to accommodate memory and bandwidth constraints and to work with custom application software, mobile devices often store data in formats of reduced fidelity. For example, some cell phones restrict the amount of information stored per contact to a small, fixed number of phone numbers and an address whereas a general personal information management (PIM) application such as Microsoft Outlook running on a desktop or laptop supports a potentially unlimited number of phone numbers, addresses and other pieces of information. In this case, a person's cell phone stores low-fidelity contacts while her full-featured desktop maintains high-fidelity versions of those same contacts.

Several commercial platforms [20, 22] as well as academic systems [1, 6, 7, 11, 12, 14, 16] can replicate homogeneous data across weakly connected devices, allowing users to update data even when disconnected and guaranteeing eventual consistency. Supporting fidelity-aware replication entails the following additional requirements:

- A device's native fidelity level should be accommodated, and fidelity should be reduced when updates are transferred from a high-fidelity device to a low-fidelity device. Storing items at a higher fidelity than what the device needs is wasteful.
- Low-fidelity devices should also be allowed to make updates, which should be reintegrated with the corresponding items on high-fidelity devices when possible. Overall, updates should flow seamlessly across different fidelity representations of an item.
- Application semantics should be preserved while transferring updates across heterogeneous representations. For example, a contact manager might require that updates made to a related set of fields, such as an address, are always applied atomically; that is, all fields are updated together or none at all.

We have designed and built the Polyjuz framework to support replication among weakly connected devices that store data at differing fidelities. Polyjuz sits between the application and a conventional replication platform that synchronizes data items of a single fidelity. Layering enables Polyjuz to focus solely on fidelity-related functionalities such as reducing fidelity and reintegrating updates while allowing users to employ existing software for data sharing and synchronization.

Polyjuz is based on the key design principle of separating different fidelity worlds. Polyjuz maintains a separate collection of data items at each fidelity level, which is replicated by the underlying platform within that fidelity world. This separation enables a laptop and a desktop to synchronize high-fidelity data items using one replication platform, while two mobile devices synchronize low-fidelity data items using a different platform. A high-fidelity device, like a laptop, that interacts with low-fidelity devices not only stores the collection for its native fidelity level but also stores lower-fidelity collections.

*Kaushik is a Ph.D. candidate at University of Michigan. He was an intern at Microsoft Research Silicon Valley during this work.

MobiSys'09, June 22–25, 2009, Kraków, Poland.

Polyjuz acts as the bridge between the worlds of different fidelity. For example, when a cell phone and a laptop synchronize their low-fidelity collections, Polyjuz transfers items between the low and high-fidelity collections on the laptop. It reduces the fidelity of items from the high-fidelity collection when copying them to the low-fidelity collection and, in the reverse direction, reintegrates updated items in the low-fidelity world with their counterparts in the high-fidelity world. The latter operation adheres to application-specific semantic requirements. For example, when dealing with personal contacts, reintegration could simply involve copying the fields in the low-fidelity representation to the high-fidelity representation. If required, the atomicity of an update involving multiple fields of an item can be preserved during reintegration.

Essentially, Polyjuz implements an eventually consistent replication mechanism on top of the separate fidelity worlds. It assigns its own version numbers for updates and maintains its own version vectors, distinct from those of the underlying replication platforms. To support fidelity, it introduces the notion of *tagged versions*, where the version number of an item is tagged with a label indicating its fidelity. Tagged versions and tagged version vectors enable Polyjuz to apply updates made in one fidelity world in other fidelity worlds correctly and to ensure that eventual consistency is achieved across the entire system.

We have implemented the Polyjuz framework in C# and built a sample application for replicating Windows Contacts. Our contact management application supports two levels of fidelity and automatically performs fidelity reductions and update reintegrations. An experimental evaluation of this application shows that Polyjuz achieves eventual consistency with a modest overhead in the presence of updates to both low and high-fidelity versions.

The rest of this paper follows this organization: The next section provides a motivating scenario highlighting the principal problems in replicating multiple-fidelity data. Sections 3, 4 and 5 present the design and implementation of the Polyjuz framework and explain how Polyjuz performs fidelity-aware replication. Section 6 then presents the contacts application built on top of Polyjuz followed by an evaluation in Section 7. Finally, Section 8 discusses related work in this area before Section 9 concludes with a summary of our contributions.

2. MOTIVATION

The following scenario, depicted in Figure 1, illustrates the needs of fidelity-aware replication.

Bob and Chuck, who work for the same company, are at a trade show scouting for new talent. At the trade show, they each talk to potential candidates and collect their names, email addresses, and phone numbers, which they record on their company-provided cell phones. During breaks, Bob and Chuck exchange the candidates' contact information directly between the cell phones using Bluetooth since there is no mobile Internet connectivity at the trade show. At the end of the day, they upload the contacts to their laptops and to a server in their office.

Back at the office, their colleagues examine the potential candidates, collect their resumes, and construct a complete portfolio for each interesting candidate. Later that week, Bob and Chuck will individually meet each chosen candidate for a casual interview at a coffee shop near the candidate's work place. To prepare for the interviews, Bob and Chuck each fetch the candidates' complete portfolios onto their laptops and also carry the candidates' contact information, now updated with work addresses, on their cell phones.

Because the candidates' recruitment information exists on different devices and servers, Bob and Chuck and their colleagues would greatly benefit from an automated system that enables a) data sharing, b) synchronization of updates, and c) data operations even when disconnected. If the data format were identical on all devices, suitable replication platforms are already available.

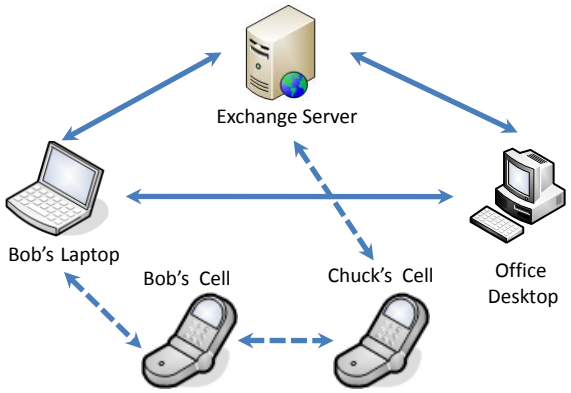

Figure 1: Example fidelity-aware replication scenario: The solid lines indicate high-fidelity data exchanges while the dotted lines indicate low-fidelity data exchanges.

However, in the example scenario, the cell phones restrict the contact format to a name, address, phone number, and email, whereas the laptops, servers, and desktops can store an extensible representation for a contact, including a resume and other attached documents. Replication of data with multiple representations at different fidelity levels poses new challenges as outlined below.

2.1 Cross-fidelity replication

The system needs to recognize the native fidelity level of each device and transfer items at that fidelity level. Devices may not be capable of storing items at any other fidelity level. Besides, storing or transferring potentially larger-sized, higher-fidelity items is wasteful of storage, bandwidth, and computation power—precious resources on mobile devices.

The above requirement entails that the system perform the necessary transformations before transferring items to the destination device during synchronization. For example, when replicating data from a high-fidelity device like a laptop to a low-fidelity device like a cell phone, data needs to be transformed from a high-fidelity to a low-fidelity format. Furthermore, the system needs to repeat these transformations every time an item is updated in the high-fidelity world and needs to be replicated in the low-fidelity world.

2.2 Update reintegration

Further problems arise if the low-fidelity devices are allowed to make updates. For example, in the above scenario, a candidate that Bob meets may give him a new phone number during the interview, which Bob updates on his cell phone. In this case, when Bob synchronizes his cell phone with his laptop, the laptop needs to take the updated low-fidelity item and merge it with the older, high-fidelity version, making sure to retain the candidate's resume and other extra information in the high-fidelity version. We call this operation a *reintegration*.

The reintegration process is complex not only due to application semantics but also because it must account for the weak consistency nature of the underlying replication platform. A high-fidelity device performing reintegration needs to ensure that it is reintegrating correct versions of an item. It needs to detect conflicting updates; traditional mechanisms such as *version vectors* [10] for detecting update conflicts can help but must be adapted to deal with multiple fidelities.

A difficult reintegration scenario occurs when a high-fidelity device receives an update made at another high-fidelity device through a low-fidelity intermediary. For instance, the cell phone could hold a low-fidelity copy of a data item that it obtained from the server, which it then transfers to the laptop before the laptop has a chance to talk directly to the server. The laptop should accept this updated item, recognize that this item is of low fidelity, and subsequently replace it with a high-fidelity version from the server during a future synchronization. This scenario can become even more difficult if the cell phone updates the item before transferring it to the laptop.

2.3 Adherence to application semantics

Finally, reintegrations must respect application semantics. For some applications, a reintegration may just copy the fields in the low-fidelity representation to the older, high-fidelity item. However, simply copying over fields may lead to a violation of atomicity. For instance, suppose one of Bob's colleagues in the office updates the address of a candidate who has moved to a new city but Bob's cell phone can only store the city and state fields of the address. The cell phone might send its low-fidelity version of the candidate portfolio, including the updated city and state information, to Bob's laptop, which might reintegrate only the city and state fields into the old address, resulting in an incorrect address.

Application semantics are thus important, and a fidelity-aware system should ideally be flexible in supporting diverse application needs. It should be able to ensure atomic updates of items if the application requires it while supporting more eager, opportunistic updates when desired.

The rest of this paper presents Polyjuz and explains how it provides fidelity-aware replication while meeting the above challenges. We focus on applications whose data format is a set of fields—as in contacts, calender entries, and e-mail—and updates involve changes to the data associated with one or more fields. Fidelity awareness, however, is also crucial for other types of data, such as pictures, media, and documents. Polyjuz can provide limited support for fidelity-based adaptation of such data—for example, updating comments and tags. However, it cannot currently support common complex operations, such as a red-eye reduction.

3. DESIGN

3.1 System model

We follow a system model that is common for weakly consistent replication systems. The system strives to replicate a *collection* of *items* of a single data type (supporting composite collections of multiple data types is an easy extension) on a set of devices called *replicas*. This set of replicas in the system may grow organically—we temporarily assume that this set is fixed and known in advance. A replica can create, update, or delete an item at any time without coordinating with other replicas. We use the term *update* liberally to mean any of these operations. The replication system usually keeps metadata to identify *concurrent updates* or *conflicts*, which occur when two replicas update an item without synchronization, creating a divergent history for the item.

A replica *synchronizes* with another in an opportunistic manner and receives updated items that it has not seen before from the remote replica. We don't assume any predetermined pattern for when and with which remote partner a replica might synchronize. However, the synchronization patterns must lead to a connected topology in order to achieve convergence. Figure 1 illustrates one such synchronization topology.

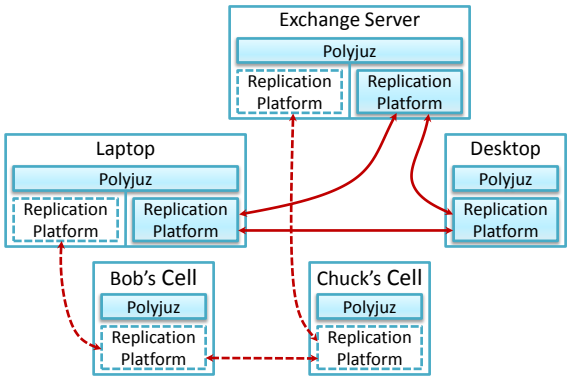

Figure 2: Polyjuz architecture for the example scenario in Figure 1. The solid lines indicate high-fidelity collections and data exchanges while the dotted lines indicate the same for low fidelity.

For fidelity-aware replication, we assume that a data item consists of a set of *fields*. A representation of a given *fidelity level* consists of a subset of those fields. We call the representation containing all the fields the *full-fidelity* representation of the item. Each replica specifies a fidelity level it supports and exposes to the application. A replica may store items of fidelity lower than its fidelity level, but never higher. We assume that the fidelity level of a replica is fixed and known in advance and that there is at least one full-fidelity replica in the system.

3.2 Architecture

The key principle we follow in the design of Polyjuz is **separation**. At a high level, our architecture creates separate worlds of replication for each fidelity level. Each world independently replicates a collection of items of a common fidelity with the help of an off-the-shelf replication platform. On replicas where the worlds meet, multiple collections might exist, one for each fidelity level that the replica needs to support. For example, a full-fidelity laptop that serves as a synchronization partner for a low-fidelity cell phone and a high-fidelity desktop will have two collections, but will only expose the high-fidelity collection to the application.

Polyjuz then unifies the separate fidelity worlds at a layer above the replication platforms. Essentially, Polyjuz is another weakly consistent replication system implemented on top of the underlying platforms. It is, however, fidelity aware, ensuring that the operations needed to copy data between different fidelity worlds—namely, fidelity reductions and update reintegrations—are performed correctly. It provides an eventual consistency guarantee that holds across the unified replication system by building on the consistency guarantees of the underlying replication platforms.

Figure 2 illustrates this architecture for the scenario depicted in Figure 1. The desktop has a high-fidelity collection, the cell phones have low-fidelity collections, the laptop and the server each have a high-fidelity and a low-fidelity collection. The high-fidelity collections synchronize with each other (shown in solid lines) independent of the low-fidelity collections (dashed lines). The Polyjuz layer transfers items across the high and the low-fidelity collections in the laptop and the server.

We chose the layered design for several pragmatic reasons: First, it keeps the Polyjuz layer simple and specific to fidelity-aware operations, allowing the reuse of synchronization protocols, knowledge representation schemes, and transport mechanisms. Second,

it enables Polyjuz to inherit the benefits of the underlying replication layer. Some replication systems [9, 12] are optimized for low bandwidth consumption by exchanging concise knowledge of previously-seen updates during synchronizations while a few others [1, 14] support partial replication. Polyjuz inherits such features. Finally, a layered design facilitates adoption by allowing users to continue to use their favorite replication platforms.

On the other hand, layering has obvious down sides. It increases storage overhead by keeping multiple representations of items on replicas that support multiple fidelity levels as well as two layers of replication metadata. And, as discussed later, it can also increase bandwidth overhead by creating spurious conflicts when idempotent fidelity operations are performed independently on different replicas.

An alternative, integrated design, where fidelity awareness is directly built into a single replication platform, will avoid these extra overheads. The core mechanism introduced in this section can support fidelity in an integrated replication system equally well. However, we stick with the layered design for the previously mentioned reasons and offer optimizations to reduce the additional overheads our design entails in Section 4.

3.3 Overview of weakly consistent replication

In this section, we provide an overview of the basic design principles of weakly consistent replication as a background for Polyjuz. Implementations of weakly consistent replication systems may vary in significant ways, but they generally share the following important characteristics.

Weakly consistent replication systems use version numbers to track updates. A *version number* is a two-tuple consisting of a *replica identifier* and an *update count*. A replica assigns a unique version number to each update (or a create or a delete) of an item containing its own identifier and an update count it maintains. The update count grows monotonically.

Additionally, each version has a *version vector* used to detect concurrent updates or conflicts. A version vector is a vector of update counts, one per replica. An update count u for replica r implies that the version incorporates any update of the item performed at r with an update count less than or equal to u. This implication leads to a trivial check for conflicts: version v_1 of item i is concurrent with version v_2 of item i if and only if $v_1[r_1] > v_2[r_1]$ and $v_1[r_2] < v_2[r_2]$ for some replicas r_1 and r_2. In other words, the version vector defines a partial order over the possible update history of an item. If the version vectors of two versions are ordered, then one supersedes the other, otherwise they are concurrent. We use the same symbol, v for example, to denote both a version of an item and its version vector for convenience.

The replication system also notifies the application about conflicts. The application can then resolve the conflict in an automated manner [15, 18] or in turn expose it to the user. Conflict resolution results in an updated version of the item with a new version number and a new version vector that combines the version vectors of the conflicting versions and the new version number. Combination means that for each replica r the update count in the new version vector will be the maximum of the corresponding update counts in the version vectors of the conflicting versions.

The version vector information is also used during a synchronization to decide which updates are missing at a replica. In an unoptimized synchronization protocol, a replica will send the version vectors of all its items to the remote replica. If the remote replica has an item whose version number is not contained in the received version vector, the version is a missing update, which the remote replica then sends. In practice, replicas often keep a concise

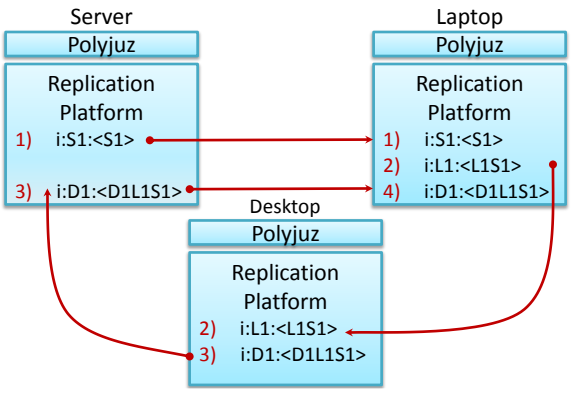

Figure 3: Example metadata changes during weakly consistent replication for a scenario with three replicas of equal fidelity.

knowledge of the updates they have seen, for instance, the version vectors of all their items combined into a single version vector [1, 9, 12, 14], and only send this compressed knowledge during a synchronization.

These basic mechanisms ensure that the system achieves *eventual consistency*. That is, after some finite time, all replicas in the system reach an identical state, appearing to have applied all non-conflicting updates to an item in the same order.

Figure 3 illustrates the operations of a weakly consistent replication system for an example scenario with three replicas—the server, laptop, and desktop depicted in Figure 1. In step 1, the server S first creates an item i with version number $S1$ and version vector <$S1$>. We use the triplet *item_id:version_number:version_vector* to represent the metadata of an item. The laptop L then receives this version from the server during a synchronization. In step 2, the laptop performs an update changing the item's metadata to i:$L1$:<$L1S1$> and sends the updated version to the desktop D on a subsequent synchronization. In step 3, the desktop performs an update leading to the version i:$D1$:<$D1L1S1$> which it sends to the server S to replace the older version. Finally, in step 4, the laptop synchronizes with the server and receives the most recent version of i. The system reaches a consistent state at this point.

3.4 Fidelity-aware replication in Polyjuz

We introduce fidelity awareness to weakly consistent replication through the novel mechanism of *fidelity tags*. A fidelity tag is a short label for a fidelity level. We define a partial order over fidelity tags. The full-fidelity level is the root of this partial order and *dominates* every other fidelity level. Tags of other fidelity levels may just be a flat tier under this root or might define a more intricate partial order relationship with multiple tiers. A simple way to define tags for fidelity levels that are subsets of the fields in the full-fidelity data is through a bit vector, where a bit is set for each field present in a representation. An alternative way is to define a short label for each fidelity level and specify the ordering relationships explicitly as a separate map.

We extend the traditional definition of a version number to a three tuple consisting of the replica identifier, update count, and a fidelity tag. This *tagged version number* defines the fidelity level at which the item is represented. Similarly, we also extend the definition of the traditional version vector to a *tagged version vector*, which is a vector of tagged version numbers, one per each replica.

We can now define two kinds of partial orders on the tagged version vector: the *tagged partial order*, $\geq_t$, defines supersession over

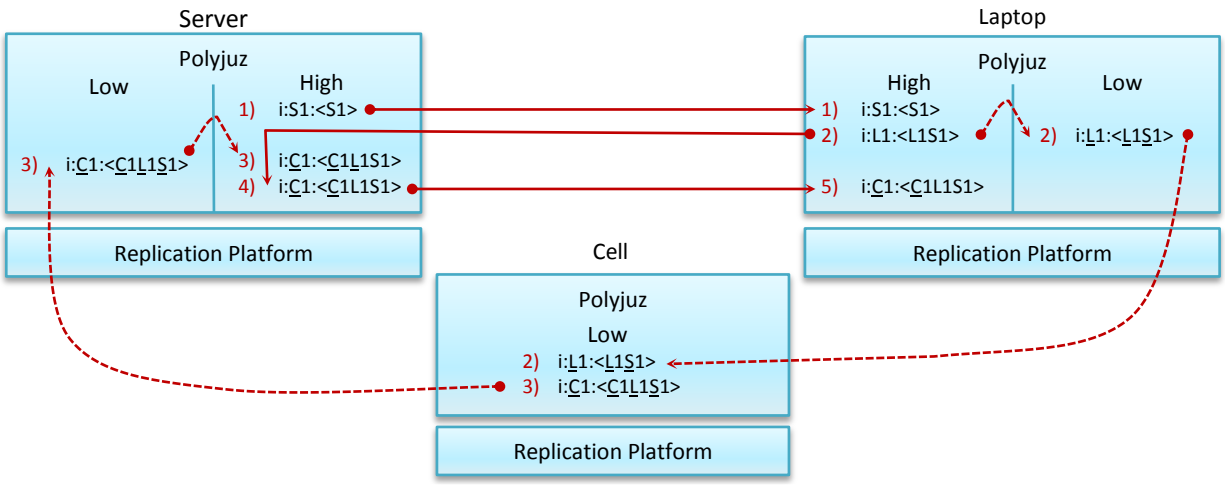

Figure 4: Example metadata changes during two updates, a fidelity reduction, and two update reintegrations in Polyjuz for a scenario with two high-fidelity replicas and one low-fidelity replica. An underline of a replica identifier indicates low-fidelity version. The system converges to a consistent state at the end of these interactions.

tagged version vectors: that is, a tagged version vector v_1 supersedes another tagged version vector v_2 if and only if $(v_1[r] > v_2[r])$ or $(v_1[r] = v_2[r]$ and $v_1[r].tag \geq v_2[r].tag)$ for each replica r. Here, $v[r]$ is the update count of the version vector element for replica r, and $v[r].tag$ is the corresponding fidelity tag.

The second partial order, *untagged partial order*, is the traditional partial order over version vectors, which does not take into account fidelity tags. That is, $v_1 \geq v_2$, if and only if, $v_1[r] \geq v_2[r]$ for each replica r.

Polyjuz implements fidelity-aware replication using a tagged version number and a tagged version vector as *fidelity metadata* for each version. Polyjuz encapsulates this fidelity-aware metadata and hides it from the underlying replication protocol, which replicates the Polyjuz metadata along with the rest of the item, unbeknownst. Section 4 describes how this encapsulation can be done in practice.

Polyjuz performs the traditional operations of weakly consistent replication systems except synchronization, which happens through the underlying replication platform. To handle an update, Polyjuz assigns to it a new version number tagged with that replica's fidelity level and updates the tagged version vector to include the new tagged version number. It also allows applications to register automated conflict-resolvers and notifies an application about concurrent updates.

3.5 Fidelity transformations

The central fidelity-aware operation in Polyjuz is the transformation of items between different fidelity worlds. Polyjuz copies items between fidelity worlds on replicas that support multiple fidelity levels. This transformation can be triggered in multiple ways. The underlying replication platform could notify Polyjuz of new updates, invoking the necessary transformations. Or, Polyjuz could periodically inspect the state of the collections and perform transformations in the background.

3.5.1 High-to-low transformation

The fidelity metadata on an item indicates when a transformation is required from the high-fidelity to the low-fidelity world (and vice versa). Polyjuz copies an item i from a high-fidelity collection to a low-fidelity collection, performing a fidelity reduction, under one of the following three conditions:

1. A version of i is present in the high-fidelity collection but no versions of i are in the low-fidelity collection. This is the case of first appearance of an item in the low-fidelity world.

2. The version in the high-fidelity collection, v_h, has a different version number than the version v_l in the low-fidelity collection, and the version vector of v_l does not supersede the version vector of v_h under the tagged-partial-order relationship. This is analogous to the condition for a remote replica to send an update to its synchronization partner.

3. The version in the high-fidelity collection, v_h, has the same version number as the version v_l in the low-fidelity collection but a higher fidelity tag. This condition specifies a supersession for the tagged version number.

Polyjuz performs the following operations to copy the item i from a high-fidelity collection to a low-fidelity collection. If the version of the item in the high-fidelity collection, v_h, is already at a fidelity level lower or equal to the fidelity level of the low-fidelity collection, Polyjuz simply copies the item over. Otherwise, the transformation happens in three steps: 1) Polyjuz creates a new representation of the item matching the fidelity level of the low-fidelity collection with the help of some application-specific logic to parse the data format. 2) It sets the item's fidelity tag to the minimum of the fidelity level of the low-fidelity collection and the original fidelity tag of the item. The same change is also made to the fidelity tag of every element of the item's version vector. And, 3) it copies the fidelity-reduced representation to the low-fidelity collection.

Polyjuz ensures that the above transformation is not performed in duplicate. All it needs to do for that is check if the tagged version number of the transformed item would match the tagged version number of the item in the low-fidelity collection or not.

Figure 4 illustrates the above fidelity-reduction operation for the example scenario consisting of the high-fidelity server, the high-fidelity laptop, and one of the low-fidelity cell phones depicted in Figure 1. In step 1, the server S first creates an item i with version number $S1$ and version vector <$S1$> in the high-fidelity collection. The laptop L then receives this version from the server in its high-fidelity collection during a synchronization that happened in the

underlying replication platform. In step 2, the laptop L performs an update changing the item's metadata to i:$L1$:<$L1S1$>.

A subsequent synchronization from the low-fidelity cell phone C triggers a fidelity reduction in L. The version i:$L1$:<$L1S1$> in the high-fidelity collection at L gets transformed to the version i:$\underline{L}1$:<$\underline{L}1\underline{S}1$> in the low-fidelity collection. We use a line below the replica identifier to indicate a version tagged as low-fidelity—no line denotes a high-fidelity version. Note that all elements in the version vector are tagged as low. Ultimately, the cell phone C receives the low-fidelity version i:$\underline{L}1$:<$\underline{L}1\underline{S}1$>.

3.5.2 *Low-to-high transformation*

Polyjuz copies an item i from a low-fidelity collection to a high-fidelity collection when one of the following conditions is satisfied:

1. A version of the item is present in the low-fidelity collection, but no versions of the item are in the high-fidelity collection. This is the case of first appearance of an item in the high-fidelity world. It occurs when the item is created on a low-fidelity replica or when a fidelity-reduced version is received from a low-fidelity replica first.

2. The version v_l in the low-fidelity collection has a different tagged version number than the version v_h in the high-fidelity collection, and the version vector of v_h does not supersede the version vector of v_l under the tagged partial order relationship.

Polyjuz simply copies the selected items to the high-fidelity collection without making any changes to data or metadata. This might result in multiple versions—usually a high-fidelity version and an updated low-fidelity version—of an item in the high-fidelity collection. For example, in Figure 4, in step 3, the cell phone C makes a low-fidelity update to item i resulting in the version i:$\underline{C}1$:<$\underline{C}1\underline{L}1\underline{S}1$>. This version then gets replicated to the low-fidelity collection on the server S. The Polyjuz framework on the server detects this updated version in the low-fidelity collection and copies it over to the high-fidelity collection, resulting in two versions as shown in step 3a of Figure 5. The actual update reintegration happens in a separate process described next.

3.6 Update reintegration

The underlying replication platform sees the multiple versions as concurrent updates to the same item in the high-fidelity collection. Polyjuz registers an automated conflict resolver with the underlying replication platform to handle these conflicts. It performs the following actions for two conflicting versions v_1 and v_2 reported by the underlying platform depending upon the relationship between their tagged version vectors.

Case 1 ($\mathbf{v_1 =_t v_2}$) : This indicates a *spurious conflict*. It occurs when different replicas perform an idempotent fidelity reduction or update integration independently. The resulting versions have the same data and fidelity metadata but still appear as concurrent updates to the underlying platform. Polyjuz resolves the spurious conflict trivially by keeping one of the versions.

Case 2 ($\mathbf{v_1 <> v_2}$) : A genuine concurrent update has occurred. The conflicting versions are retained and the application is notified of the conflicting versions. Note that the comparison here is with respect to the untagged partial order.

Case 3 ($\mathbf{v_1 >_t v_2}$ or $\mathbf{v_2 >_t v_1}$) : A simple supersession happens. Polyjuz resolves this in favor of the superseding version—the older version disappears from the collection. Note that the common update reintegration scenario, where a low-fidelity replica takes an item from a high-fidelity replica, makes an update, and passes it back to the high-fidelity replica, does not actually produce this case but the following one (see Figure 4).

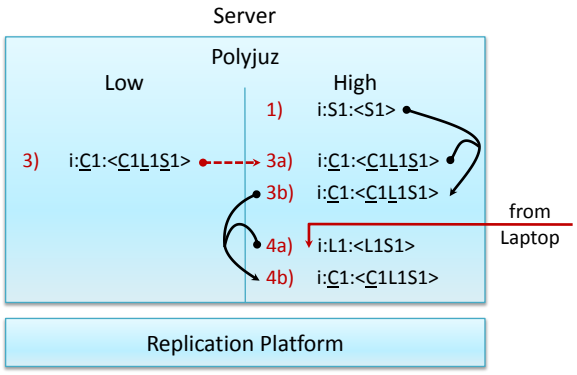

Figure 5: Update reintegrations on server S split into two substeps each. In step 3a, the updated version is copied from the low-fidelity collection to the high-fidelity collection. In step 3b, the update is reintegrated with the older version through an automated conflict resolution process invoked by the underlying replication platform. Subsequently, the older, high-fidelity version fetched from the laptop in step 4a is reintegrated with the latest, low-fidelity version in step 4b in a similar manner.

Case 4 ($\mathbf{v_1 <>_t v_2}$) : This case occurs when an updated version is present with a mismatched fidelity level, indicating an opportunity for update reintegration. If $v_1 > v_2$ (vice versa for $v_2 > v_1$), that is the version with vector v_1 is more recent, then Polyjuz creates a new, update-integrated version v of the item as follows: 1) It copies v_2 and updates only those fields defined by the fidelity tag of v_1 with the values in v_1, 2) sets the new item's tagged version number to be the same as v_1 and 3) sets the new item's tagged version vector to be a combination of the tagged version vectors v_1 and v_2. The version vectors are combined element by element, for each element the result of the merge is the superior tagged version number, that is, $v[r] = \max(v_1[r], v_2[r])$ and $v[r].tag = v_1[r].tag$ if $v_1[r] > v_2[r]$ or $v_2[r].tag$ if $v_2[r] > v_1[r]$ or $v_1[r].tag \mid v_2[r].tag$ otherwise ($\mid$ is a bit-wise or operation).

Continuing the illustrative example in Figure 5, the underlying replication platform on the server notices the two concurrent updates to item i and notifies the Polyjuz layer. Polyjuz performs a reintegration according to Case 4 above. The resulting reintegrated version in step 3b (also the end of step 3 in Figure 4) has the fidelity metadata i:$\underline{C}1$:<$\underline{C}1\underline{L}1S1$>. Note that the element for replica S in the version vector has the high-fidelity tag now. This is correct because the reintegrated version includes, the initial high-fidelity create $S1$, the fidelity-reduced version $\underline{L}1$ of L's update, and the low-fidelity update $\underline{C}1$ made on the cell phone C.

Subsequently, in step 4a (Figure 5), the server synchronizes with the laptop L and receives the version $i : L1$, which the server only knew in the low-fidelity form before the synchronization. This results in another update integration (also Case 4) in step 4b, leading to the version i:$\underline{C}1$:<$\underline{C}1L1S1$> for item i. Finally, in step 5 of Figure 4, the laptop synchronizes with the server and receives this most recent version. Note that the system has converged to a consistent state at this point.

3.6.1 *Atomic updates*

In the above example, if the laptop's update included updates to fields not covered by the fidelity tag of the cell phone C, then the

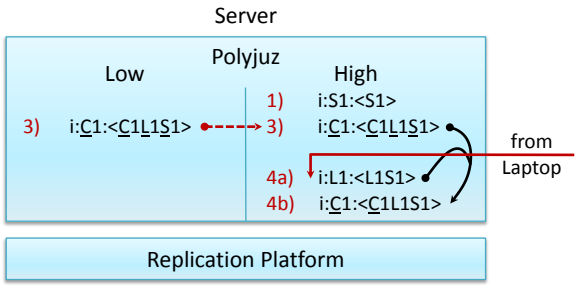

Figure 6: Update reintegration on server S with atomicity enforced. There are two un-integrated versions at the end of step 3 because an intermediate update is not available at the correct fidelity. The required high-fidelity version is received from the laptop in step 4a, and reintegration finally happens in step 4b.

update integration on server S in step 3 would have produced a version that only partially reflects the laptop's update. As we have mentioned before, this violation of atomicity of the laptop's update may not be acceptable to some applications.

If required, Polyjuz ensures that updates are reintegrated atomically by maintaining additional state. Each replica keep track of the native fidelity level of other replicas. It suppresses the update reintegration of two versions v_1 and v_2 (in Case 4) if an intermediate update by some replica r is not available at r's native fidelity level. This condition can be verified by computing the tagged version vector of the reintegrated version and checking if the fidelity tag for some replica r is lower than the native fidelity level of r. If this condition is met, the replica keeps separate versions of this item until it receives the intermediate version containing the update at the appropriate fidelity level.

For instance, in the example illustrated in Figure 4, at the end of step 3, the high-fidelity collection on the server has two non-integrated versions $S1$ and $\underline{C}1$ since the update $L1$ made by the laptop is not available at the required, high fidelity level. Figure 6 illustrates this intermediate state at the server S for atomic update reintegration. The final update reintegration happens in step 4b after the server receives the version $L1$ from the laptop L in step 4a.

In summary, the fidelity tag turns out to be an elegant mechanism to support fidelity-aware replication. It defines simple conditions that can be checked in order to detect opportunities for fidelity reduction and update reintegration. It is based on the same concepts of *supersession* and *conflicts* as in the version-vector-based replication systems, making it easy to reason about consistency.

4. ISSUES AND IMPLICATIONS

In this section, we discuss some practical issues that arise in applying the above design to the real world.

4.1 Overhead complexity

Polyjuz incurs additional storage, bandwidth, and computational overhead in order to support fidelity-aware replication. Separation of fidelity worlds requires extra space to store multiple representations of items on replicas that support more than one fidelity level. We expect that resource-constrained devices will only support one, low fidelity level and not incur this extra overhead. On other devices, a fidelity-aware storage component can vastly eliminate the additional storage required to maintain distinct copies of an item for each fidelity level. The storage component can store a single copy of the item at the highest fidelity level supported by the replica and generate a representation of the item at the required fidelity level on-the-fly.

Fidelity metadata maintained by Polyjuz also incurs storage and bandwidth overhead. We expect the size of a fidelity tag to be small and fixed in a bit vector representation, a bit per field in the data type. The per-item fidelity metadata consumes $O(R)$ bytes, where R is the number of replicas in the system. Thus, fidelity metadata adds a small bandwidth cost every time the item is transferred over the network assuming that the number of replicas is small. However, it does not add to the cost of exchanging knowledge during synchronizations (delegated to the underlying layer). The per-item storage cost of fidelity metadata at a replica r is $O(R.f_r)$, where f_r is the number of fidelity levels the replica supports. We expect f_r to be small in general and almost always one for resource-constrained, low-fidelity devices.

Another source of overhead in Polyjuz are spurious conflicts. Spurious conflicts occur in the underlying replication platform when fidelity reductions or update reintegrations for the same version are performed independently by more than one replica. The resulting versions are identical in content and fidelity metadata, but appear as distinct, conflicting versions to the underlying replication platform, and trigger unnecessary, bandwidth consuming data transfers. The number of spurious conflicts depends on the number of replicas that perform fidelity reductions and update integrations for an item.

There are several ways to alleviate the impact of spurious conflicts. First, fidelity operations can be restricted to a few selected replicas or even just one. Second, replicas that perform fidelity operations can be made to synchronize more often so that low-fidelity replicas see fewer conflicts. Resource-rich well-connected replicas would easily withstand the resulting overhead and make ideal candidates for performing fidelity operations. More importantly, the impact on the resource-constrained, low-fidelity devices will be alleviated. Third, if the underlying replication platform allows custom transport mechanisms, unnecessary data transfers can be suppressed by sending "diff"s or content hashes. Note that the diff would be empty for a spurious conflict, but this modified protocol would require an additional round of communication.

In addition to the above storage and bandwidth overhead, replicas supporting multiple fidelity collections also consume computation time to perform cross-fidelity transformations. An unoptimized approach that tries to integrate every pair of collections until no new transformable items are found can be expensive. We propose the following three step process to keep computational complexity manageable. First, Polyjuz transforms items from the low-fidelity collections to the full-fidelity collection. This restricts reintegration only to full-fidelity replicas. Second, it performs update reintegrations in the full-fidelity collection. And, finally, transforms items from the full-fidelity collection to the low-fidelity collections. This limits the per-item computational complexity to $O(f_r)$ for each round of this process at replica r.

4.2 Interactions with applications

So far, our design largely focussed on the application-independent mechanisms in Polyjuz. Application-specific interactions are necessary to ultimately transfer the benefits of fidelity-aware replication to the users. Polyjuz expects three types of interactions from an application-aware component we call the *application helper*. First, the helper assists in performing fidelity transformations. Since applications tend to have custom formats, the helper is responsible for parsing the data format, performing transformations, and re-encoding updates. Second, the helper defines and specifies fidelity levels suitable for the application. The helper might let users spec-

ify custom fidelity levels for new devices or pick from a predefined set. And, third, it is responsible for setting up fidelity-separated collections and invoking transformations between the correct collections.

4.3 Interactions with replication platforms

Polyjuz is intended to work with any platform that provides weakly consistent replication for arbitrary data items. It does not require any additional hooks into the replication platform although it can take advantage of any offered hooks. For example, Polyjuz adds fidelity metadata to data items, which it expects to be replicated along with the data item. It can achieve this on an opaque replication platform in one of two ways. First, it can create an encapsulated data item that includes the original data and the fidelity metadata and replicate the encapsulated data item. This incurs additional storage overhead. Alternatively, it can "stuff" the fidelity metadata into some unused portion of the data item, such as a comment field, and replicate the data item without creating encapsulated copies. Note that this alternative approach will fail if the underlying replication platform is format-aware and performs active reformatting during replication. Finally, if the replication platform allows a custom transport mechanism, the fidelity metadata can be attached to the data item on-the-fly (and detached at the receiving end) at the transport layer, transparent to both the application and replication platforms.

Another instance where hooks offered by the replication platform benefits Polyjuz is in the resolution of spurious conflicts. First, as mentioned earlier, an intelligent transport mechanism can suppress duplicate data transfers due to spurious conflicts. Additionally, if the replication platform also opens up the setting of item identifiers and version numbers then the spurious conflicts can be totally eliminated. Spurious conflicts do not occur if the Polyjuz layer assigns the same version number to an item as the underlying replication platform.

4.4 Consistency

Polyjuz attains an eventually consistent state in a similar manner as weakly consistent replication systems. When a replica completes synchronizing with a remote replica, its collection includes all updates present in the remote replica. The underlying replication platform provides this property. Similarly, when Polyjuz completes transformations between two collections on a replica supporting multiple fidelity levels, each of the two collections has all the updates present in the other. Thus, an update spreads in the fidelity world it was performed as well as in the other worlds. Moreover, since Polyjuz merges updates in an idempotent manner on each replica all replicas reach a convergent state, after which no change happens to an item's data or fidelity metadata until a further update.

The underlying platforms in turn replicate the reintegrated versions, resolve spurious conflicts, and reach convergence in their respective fidelity worlds.

5. IMPLEMENTATION

We have implemented Polyjuz in C#. Figure 7 illustrates the components that Polyjuz interacts with. Three key functions of our implementation merit discussion: replication platform interaction, maintenance of tagged version information, application-specific format transformations.

Our implementation relies on the Microsoft Sync Framework [22] for basic replication functionality. Microsoft Sync Framework is a replication platform that synchronizes data across multiple stores. Applications wishing to use the Sync Framework implement a *sync provider*, which manages storage for items in the collection and the

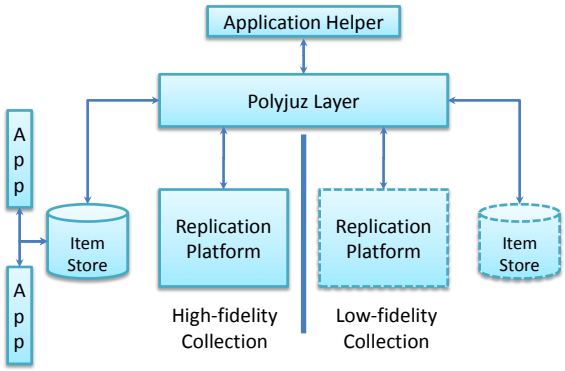

Figure 7: Polyjuz software architecture on a dual-fidelity replica.

replication metadata used to reconcile updates. It is also responsible for providing a transport mechanism to copy items between two replicas during a synchronization.

We implement Polyjuz as a custom sync provider. The Microsoft Sync Framework provides the necessary hooks to implement all the optimizations we outlined in the previous section in the sync provider. While we have not implemented all optimizations, we take advantage of the offered flexibility to perform on-the-fly encapsulation of fidelity metadata. During synchronization, the custom sync provider ensures that the fidelity metadata is also transferred along with data and stored at the other end. For every sent item, Polyjuz encapsulates both the item's data and fidelity metadata in a composite object on the fly, which it deserializes at the receiving end.

Polyjuz maintains fidelity metadata separately in a metadata store. It maintains tagged version vectors for every item in a replica's collection, at every fidelity level supported on the replica. The metadata is cached in memory but also written to the disk upon each change.

The data items are stored in a file-system directory that is known to Polyjuz at initialization. Each fidelity collection has a separate directory of its own. The application only accesses the directory of the highest-fidelity collection at a replica. We assume that the application directly creates, updates, or deletes files in the directory without interacting with Polyjuz.

Polyjuz then detects creates, updates, and deletes by inspecting the directories. The custom sync provider does this each time it synchronizes two collection for the directories of the synchronized collections. Having learnt about the updates during this inspection, the custom sync provider also checks the fidelity metadata (of other collections in the replica) to see if this update needs to be transferred to another collection or integrated with a previous update and performs the necessary fidelity reduction and update reintegration.

Finally, to generate fidelity reduced or reintegrated representations of an item, our implementation calls two application-provided routines. One routine is passed a high-fidelity item and returns the low-fidelity equivalent. The second routine takes a low-fidelity item along with the stored high-fidelity version of this same item, and returns a merged version.

6. CASE STUDY: POLYCONTACTS

We built PolyContacts, a fidelity-aware contacts management application so that a concise representation of an address book con-

tact can be transferred from the user's desktop to mobile device. The user is free to update the contact on his mobile device. When the mobile device synchronizes with the desktop, all updates to the low-fidelity version from the mobile device are automatically reintegrated with the desktop's high-fidelity version.

PolyContacts builds upon Windows Contacts, an address book application bundled with the Windows Vista operating system. Each entry in a Windows Contact address book is stored persistently in the file system as a unique well-formed XML file. Updating an address book entry is as simple as editing this XML file. Microsoft provides a programmatic interface to read a Windows Contacts address book into memory and perform operations like contact creation, update or manipulation.

PolyContacts uses the Windows Contacts API [21] to maintain a custom fidelity-aware address book rooted at any directory the user wishes. The user can create new contacts in this directory and manipulate it using the Windows Contacts application. PolyContacts implements the core functionality required of a fidelity-aware application: the Polyjuz fidelity transformation interface. This interface consists of two application-provided functions: fidelity reduction and reintegration.

6.1 Fidelity reduction

PolyContacts' fidelity reduction function takes as input a full address book contact entry and returns a lower-fidelity contact that only possesses the name, phone number and email address fields of the original contact. This function allows users to generate low-fidelity representations of their address book entries that can be transferred to their constrained cell phone.

In the scenario described in Figure 1, Bob might meet a prospective recruit and choose to create a new address book entry populated with the candidate's name, phone number and email address. Later, when Bob synchronizes his cell phone with his laptop, PolyContacts reads in the newly added address book entry and creates a new full-fidelity contact with the name, phone number and email address it just read in. PolyContacts leaves the other fields in the full-fidelity contact blank. When Bob returns to his office, he can populate the blank fields in the candidate's entry with other information gleaned from the candidate's resume such as the candidate's home and work address.

6.2 Reintegration

PolyContacts' reintegration function takes as input two address book contacts. This function creates a new reintegrated contact by extracting the low-fidelity fields from the first contact (i.e., name, phone number and email address) while retaining the remaining fields from the second contact.

In our sample scenario, when Bob meets the candidate for an interview at a coffee shop, the candidate might inform Bob of his recent move and updated phone number. Bob updates his cell-phone address book with the new number. Later, when Bob syncs his cell phone with his laptop, the PolyContact reintegration function creates a new contact with the updated phone number from the low-fidelity cell phone address book while retaining the candidate's home and work addresses from the high-fidelity address book entry on the laptop.

7. EVALUATION

Our evaluation answers the following questions:

- Does Polyjuz achieve eventual consistency in the presence of high-fidelity updates, low-fidelity updates, and high-fidelity followed by low-fidelity updates to an item?

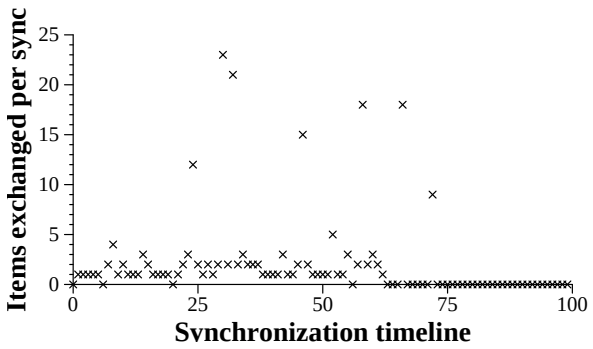

Figure 8: This scatter plot shows the timeline when sychronizing 100 contacts with 30 high-fidelity updates.

Replication	Low		High	Dual	
	bCell	cCell	Desk	Lap	Srv
Updates	-	-	9	10	11
Items sent+rcvd	42	44	30	64	60
Reductions	-	-	-	30	30
Reintegrations	-	-	-	-	-
Spurious conflicts	12	14	-	4	-

Table 1: Breakdown of operations for high-fidelity updates.

- How many fidelity-specific operations, that is reductions, reintegrations and spurious conflicts, result from various update patterns?

7.1 Methodology

As described in Section 5, we implemented the Polyjuz framework in C# with the Microsoft Sync Framework as our base replication layer. Our experimental evaluation of Polyjuz uses five replicas running on a single computer, a HP Workstation xw4600 running Windows Vista. These replicas represent our scenario described in Figure 1 where Bob's Laptop and the Exchange server support dual-fidelity (i.e. two worlds each) while the three other replicas support either high or low-fidelity (one world).

Before each experiment, we replicated 100 contacts amongst the 5 replicas. We then randomly selected replicas to perform updates using the PolyContacts application and allowed replicas to synchronize with each other. We report on their convergence trends and on the specific operations they performed. The experiments provide empirical evidence to show that Polyjuz is practical, provides eventual consistency, and incurs a modest overhead. The actual overhead costs and convergence times may vary depending on the application workload.

In all tables, "bCell" refers to Bob's cell phone, "cCell" refers to Chuck's cell phone, "Desk" refers to Bob's desktop, "Lap" refers to Bob's laptop and "Srv" refers to the company's exchange server.

7.2 High-fidelity updates

In our first experiment, we randomly select a high-fidelity replica and have it update one of its contacts. We intersperse 30 of these update operations with synchronizations. Figure 8 shows that the system converges after approximately 70 synchronizations. Since we perform one update every two synchronizations, the last update happens around the 60^{th} synchronization.

Table 1 provides a breakdown of the operations for 30 high-fidelity updates. From the table, we see that the desktop, laptop and

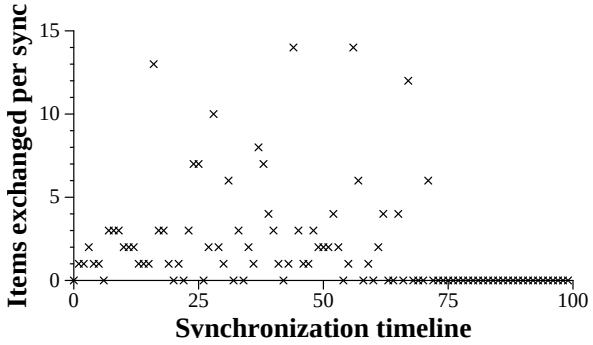

Figure 9: This scatter plot shows the timeline when sychronizing 100 contacts with 30 low-fidelity updates.

Replication	Low		High	Dual	
	bCell	cCell	Desk	Lap	Srv
Updates	19	11	-	-	-
Items sent+rcvd	31	38	30	62	79
Reductions	-	-	-	3	8
Reintegrations	-	-	-	27	22
Spurious conflicts	1	8	-	2	19

Table 2: Breakdown of operations for low-fidelity updates.

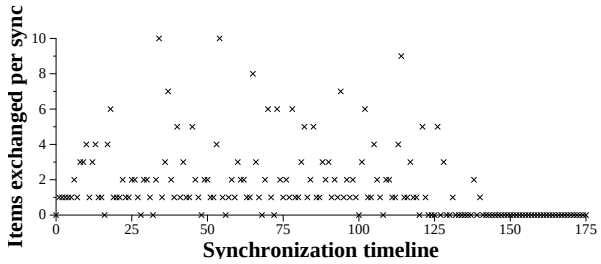

Figure 10: This scatter plot shows the timeline when sychronizing 100 contacts with 30 high-fidelity updates followed by low-fidelity updates to the same item.

Replication	Low		High	Dual	
	bCell	cCell	Desk	Lap	Srv
Updates	15	15	-	15	15
Items sent+rcvd	45	45	56	135	131
Reductions	-	-	-	44	42
Reintegrations	-	-	1	15	17
Spurious conflicts	1	2	-	15	16

Table 3: Breakdown of operations for high-followed-by-low-fidelity updates.

server each carried out 9, 10 and 11 updates respectively. We also observe that the laptop and the server each performed 30 fidelity reductions as they each received an updated high-fidelity version and performed a fidelity-reduction to generate a low-fidelity representation that could be synchronized with Bob's or Chuck's cell phone.

We see that Bob's cell phone sent or received 42 items, and it detected 12 as spurious conflicts. Such a conflict occurs because Bob's cell phone receives a low-fidelity representation authored by Bob's laptop, while Chuck's cell phone receives a low-fidelity representation authored by the server. Since Microsoft Sync Framework is unaware of fidelities, it signals a conflict when Bob's cell phone synchronizes with Chuck's cell phone, as an updated contact on Bob's cell phone possesses a different version number than on Chuck's. However, Polyjuz running on Bob's cell phone examines the fidelity metadata associated with the conflicting versions and ignores the conflict. Note that if the laptop and the server had been synchronizing more frequently, they would have resolved the spurious conflicts between themselves, and the cell phones would not have seen them.

Observe that, for each replica, subtracting away the number of spurious conflicts from the number of items sent or received gives 30, which is the number of updates introduced into the system. Similarly, adding up all spurious conflicts also gives 30 which corresponds to the number of duplicate fidelity reductions performed by the laptop and server.

7.3 Low-fidelity updates

In our second experiment, we randomly select a low-fidelity replica, that is Bob or Chuck's cell phone, and have it update one of its contacts. As before, we intersperse 30 of these update operations with synchronizations. Figure 9 shows that the system once again converges after approximately 70 synchronizations.

Table 2 provides a breakdown of the operations for 30 low-fidelity updates. Bob's cell phone makes 19 updates while Chuck's makes 11, which add up to 30 updates. The interesting numbers in this table are in the reduction and reintegration rows. When a low-fidelity replica updates an item, its updates have to be reintegrated into the high-fidelity representations. From Table 2, we see that Bob's laptop and server perform 27 and 22 of these reintegrations respectively.

Updates might travel through various intermediaries before arriving at a replica. For instance, Chuck's cell phone could update an item and synchronize it with the server. The server would perform a reintegration to introduce it to the high-fidelity world when the server next synchronizes with Bob's laptop. Bob's laptop would perform a fidelity reduction to introduce this updated contact to the laptop's low-fidelity world. Since this scenario requires many more synchronizations, it is rarer—hence, we only see 3 and 8 reductions on Bob's laptop and the server respectively. Observe that adding up the number of reduction and reintegrations performed on each dual-fidelity replica equals 30.

7.4 High-fidelity updates followed by low-fidelity updates

Our third experiment was designed to evaluate the difficult update reintegration scenario described in Figure 4. Here, we select a dual-world replica i.e. Bob's laptop or the server to perform a high-fidelity update on a contact. The dual-world replica then syncs with another high-fidelity replica to propagate this updated contact in the high-fidelity world. The updating replica then performs a fidelity reduction on the updated contact and synchronizes with a low-fidelity replica (i.e. Bob or Chuck's cell phone). The low-fidelity replica then again updates this low-fidelity contact and finally propagates the twice-updated contact in the low-fidelity world.

In this scenario, we performed 30 such dual-updates. From Figure 10, we see that the system converges after about 130 synchronizations, which is as expected as our last update occurs around the 120^{th} synchronization.

Table 3 provides a breakdown of the operations for 30 high-followed-by-low-fidelity updates. Each of the dual-world replicas performs 15 updates, and each low-fidelity device performs the same number of updates.

This experiment is much more complex than the previous two as it generates two-times more updates and also introduces updated items simultaneously in both high and low-fidelity worlds, causing more items to be exchanged in each synchronization and also increasing the number of reductions and reintegrations. The number of items exchanged during synchronizations, as well as the number of reductions and reintegrations, depends on the order in which replicas are selected to update items and to synchronize.

8. RELATED WORK

Odyssey [8] and dynamic distillation [5] were amongst the first to demonstrate that clients of replicated systems can save bandwidth and power and improve performance by incorporating mobile adaptation and fidelity support as a first class design principle. Other systems have then demonstrated content adaptation for specific applications. For example, Puppeteer [3] provides adaptation for PowerPoint presentations and web (HTML) pages. It takes the approach of decomposing a document into a hierarchy of components, each of which may be reduced in fidelity, for example by omitting subcomponents or degrading images. PageTailor [2] provides an adaptation system whereby webpages can be automatically customized for viewing on a PDA.

Some recent file systems provide a more generic, application-independent support for dealing with multiple representations. EnsemBlue [11] integrates custom electronic devices with general-purpose computers through device-specific plugins that deal with custom formats and interfaces. These plugins run on the general-purpose computers that the CEDs connect to, perform necessary format transformations, and store state on the device itself to enable similar operations on other computers the device may connect to. quFiles [19] is a different file system that maintains multiple representations of a file internally while presenting a single, context-aware representation to the applications on the device, accounting for its screen size, network connectivity, current battery status, etc.

Polyjuz compliments the above systems through fidelity-aware peer-to-peer replication. EnsemBlue plugins can use fidelity metadata introduced by PolyJuz for update reintegration. In turn, Polyjuz could benefit from the mechanisms introduced by EnsemBlue and quFiles. For example, plugins can help Polyjuz to execute its replication logic outside the CEDs, on other computers serving as proxies, if the CEDs do not permit custom code.

Few systems have been developed that support updates to fidelity-reduced content. CoFi [4] takes the approach of decomposing a document into a hierarchy of components, similar to Puppeteer, and supports editing of fidelity-reduced components. It shows how to modify the state transition diagrams of replication systems to support updates of fidelity-reduced data. It does not, however, deal with the methods to consistently merge updates made at different fidelity levels and avoids going into the semantics of update operations that can or cannot be supported.

MoxieProxy [13], on the other hand, provides a methodology and middleware architecture for reconciling updates to fidelity-reduced data. It discusses the type of operations on transcoded data that can be reintegrated with their full-fidelity counterparts, and provides examples of application-defined *transforms* that convert an operation on transcoded data into an equivalent operation on the original data for a few complex applications, including images, speech-to-text, and pdf-to-text. Update reintegration enabled by these transforms then happen at a central server that the clients talk to. In contrast, Polyjuz enables fidelity-aware replication in a serverless system, where replicas synchronize directly with each other.

Polyjuz is layered on top of systems that replicate items between intermittently connected clients. Broadly speaking, these systems can be classified into full replication and partial replication systems [17]. Full replication systems such as Coda [7], Ficus [6], Bayou [12] and WinFS [9] require that each replica stores all the items in a collection. A simple approach to provide fidelity-awareness on top of these systems is to replicate data at full fidelity on all replicas and generate representations at the desired fidelity level on each replica. This approach has the obvious drawback of additional storage and network overhead for maintaining and replicating full-fidelity items on all replicas.

Partial replication systems, such as Cimbiosys [14], Perspective [16], and PRACTI [1], enable replicas to select the set of items they store. This feature facilitates an alternative approach to support fidelity. We can break an item into components, and individually replicate each component as a distinct item. Partial replication helps a replica to pick the set of components (fields) in its fidelity level and replicate the selected components of an item. This scheme has the following two shortcomings: First, maintaining versioning metadata at a fine, component-level granularity multiplies the overhead of the synchronization protocol. Second, it makes it harder to enforce additional semantics such as atomicity of updates to multiple components in an item and to detect concurrent, conflicting updates made to different components of the same item.

9. CONCLUSIONS

This paper presented the Polyjuz framework for fidelity-aware replication. Data representations of multiple fidelity are common in mobile devices, posing problems for conventional, fidelity-unaware tools that strive to share and synchronize data across a multitude of devices. The Polyjuz framework forms a compatibility layer on top of such existing weakly consistent replication tools and allows them to seamlessly exchange updates across worlds of different data representations. Furthermore, it preserves the eventual consistency guarantee these tools provide and extends the same guarantee to hold across representational boundaries. In this paper, we outlined a design to support applications with data formats that separate into multiple fields—address books, calendar entries, e-mail, etc.—and demonstrated a concrete application built on Polyjuz for managing personal contacts.

Acknowledgments

We thank Rama Kotla, Cathy Marshall, and Iqbal Mohomed for inspiring discussions and feedback on earlier revisions of this paper. We are also grateful to our shepherd Steve Gribble and other reviewers for insightful comments and feedback. Finally, we thank Lev Novik and the Microsoft Sync Framework product team for bringing the problem of fidelity to our attention and helping us with their replication framework.

10. REFERENCES

[1] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[2] N. Bila, T. Ronda, I. Mohomed, K. N. Truong, and E. de Lara. PageTailor: Reusable end-user customization for the mobile web. In *Proc. of the ACM Conference on Mobile*

Systems, Applications and Services (MobiSys), San Juan, Puerto Rico, June 2007.

[3] E. de Lara, R. Kumar, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proc. of the USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, Mar. 2001.

[4] E. de Lara, R. Kumar, D. S. Wallach, and W. Zwaenepoel. Collaboration and multimedia authoring on mobile devices. In *Proc. of the ACM Conference on Mobile Systems, Applications and Services (MobiSys)*, San Francisco, CA, May 2003.

[5] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proc. of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Oct. 1996.

[6] R. G. Guy. Ficus: A very large scale reliable distributed file system. Technical Report CSD-910018, Computer Science Department, University of California, Los Angeles, June 1991.

[7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[8] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile, application-aware adaptation for mobility. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, Oct. 1997.

[9] L. Novik, I. Hudis, D. B. Terry, S. Anand, V. Jhaveri, A. Shah, and Y. Wu. Peer-to-peer replication in WinFS. Technical Report MSR-TR-2006-78, Microsoft Research, June 2006.

[10] D. S. Parker, Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. A. Edwards, S. Kiser, and C. S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, 1983.

[11] D. Peek and J. Flinn. EnsemBlue: integrating distributed storage and consumer electronics. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[12] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, Oct. 1997.

[13] T. Phan, G. Zorpas, and R. Bagrodia. Middleware support for reconciling client updates and data transcoding. In *Proc. of the ACM Conference on Mobile Systems, Applications and Services (MobiSys)*, Boston, MA, June 2004.

[14] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Apr. 2009.

[15] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *Proc. of the USENIX Summer Technical Conference*, Boston, MA, June 1994.

[16] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Feb. 2009.

[17] D. B. Terry. *Replicated Data Management for Mobile Computing*. Morgan & Claypool Publishers, 2008.

[18] 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 *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[19] K. Veeraraghavan, E. B. Nightingale, J. Flinn, and B. Noble. quFiles: A unifying abstraction for mobile data management. In *Proc. of the ACM Workshop on Mobile Computing Systems and Applications (HotMobile)*, Napa Valley, CA, Feb. 2008.

[20] Live mesh. `http://www.mesh.com`.

[21] Microsoft coding4fun developer kit. `http://www.microsoft.com/express/samples/C4FDevKit/`.

[22] Microsoft sync framework. `http://msdn.microsoft.com/en-us/sync/default.aspx`.