# Fidelity-Aware Replication for Mobile Devices

Venugopalan Ramasubramanian, Kaushik Veeraraghavan, Krishna P.N. Puttaswamy, Thomas L. Rodeheffer, Douglas B. Terry, and Ted Wobber

**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 possible. 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 multifidelity universe. In this paper, we present the design and implementation of Polyjuz and demonstrate its benefits for fidelity-aware contacts management and picture sharing applications.

**Index Terms**—Distributed applications, distributed systems, weak consistency, fidelity, replication, transcoding.

## 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 data replication across 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 [21], [23] as well as academic systems [1], [6], [7], [12], [13], [15], [17] can replicate 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 higher fidelity device to a lower fidelity device. Storing items at a higher fidelity than what the device needs is wasteful.
- Lower fidelity devices should be allowed to make updates, which should be reintegrated with the corresponding items on higher 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 that interacts with low-fidelity devices not only stores the collection for its native fidelity level but also stores all lower fidelity collections.

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

- *V. Ramasubramanian, T.L. Rodeheffer, D.B. Terry, and T. Wobber are with Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043. E-mail: {rama, tomr, terry, wobber}@microsoft.com.*
- *K. Veeraraghavan is with the Department of Computer Science and Engineering, Univeristy of Michigan, 2260 Hayward St., Room 4929, Ann Arbor, MI 48109. E-mail: kaushikv@umich.edu.*
- *K.P.N. Puttaswamy is with the Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106. E-mail: krishnap@cs.ucsb.edu.*

*Manuscript received 26 Oct. 2009; revised 11 Feb. 2010; accepted 25 Apr. 2010; published online 25 June 2010.*
*For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org and reference IEEECS Log Number TMCSI-2009-10-0458. Digital Object Identifier no. 10.1109/TMC.2010.118.*

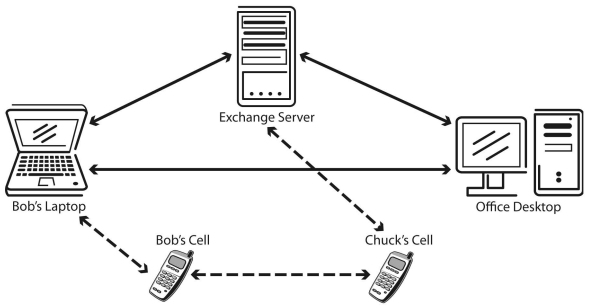


Fig. 1. Example fidelity-aware replication scenario: The solid lines indicate high-fidelity data exchanges while the dotted lines indicate low-fidelity data exchanges.

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 two sample applications. The first is a contact management application that 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 second is a picture sharing application that enables sharing images with devices of different fidelity and synchronizing updates to image metadata on any shared device.

The rest of this paper follows this organization: Section 2 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 presents the contacts management and picture sharing applications built on top of Polyjuz followed by an evaluation in Section 7. Section 8 discusses related work in this area and Section 9 concludes with a summary of our contributions.

## 2 Motivation

The following scenario, depicted in Fig. 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, e-mail 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 1) data sharing, 2) synchronization of updates, and 3) data operations even when disconnected. If the data format were identical on all devices, suitable replication platforms are already available.

However, in the example scenario, the cell phones restrict the contact format to a name, address, phone number, and e-mail, 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 performs 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 need 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*.

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 only stores the street, city, and state fields of the address omitting the zip code. The cell phone might send its low-fidelity version of the candidate portfolio, including the updated street, city, and state fields, to Bob's laptop, which might reintegrate these updated fields into the old address while retaining the old zip code, 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, calendar 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. Updates to such data might involve complex operations—for example, red-eye reduction on a picture or paragraph reordering in a document. Our picture sharing application, PolyPix, demonstrates that Polyjuz can provide limited support for fidelity-based adaptation of images. Specifically, PolyPix supports updates to image metadata (e.g., authors, location, or tags) on any device while only permitting full-fidelity devices to perform complex update operations on the data portion of an item.

## 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*. 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 coordination, 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 do not 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. Fig. 1 illustrates one such synchronization topology.

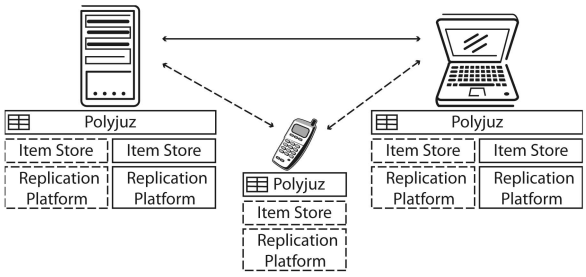


Fig. 2. Polyjuz architecture for two dual-fidelity replicas and a low-fidelity replica. The solid lines indicate high-fidelity collections and data exchanges while the dotted lines indicate the same for low fidelity.

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. The same set of items have a separate, duplicate representation, at the appropriate fidelity level, in each world. Within a world, the collection of items are replicated independently 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 full-fidelity desktop will have two collections, but will only expose the full-fidelity collection to the application.

Polyjuz 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.

Fig. 2 illustrates this architecture for a simple setup where the company's server and the user's laptop are dual-fidelity replicas, i.e., they each support both 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 low-fidelity collections on both the desktop and the laptop.

We chose a 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 [10], [13] are optimized for low bandwidth consumption by exchanging concise knowledge of previously seen updates during synchronizations while a few others [1], [15] 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 downsides. 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

Polyjuz uses the following basic design principles of weakly consistent replication systems and supports other replication platforms built with the same principles for its underlying layer.

Each item has a version number 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 [11]. 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 notifies the application about conflicts. The application can then resolve the conflict in an automated manner [8], [16], [19] 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.

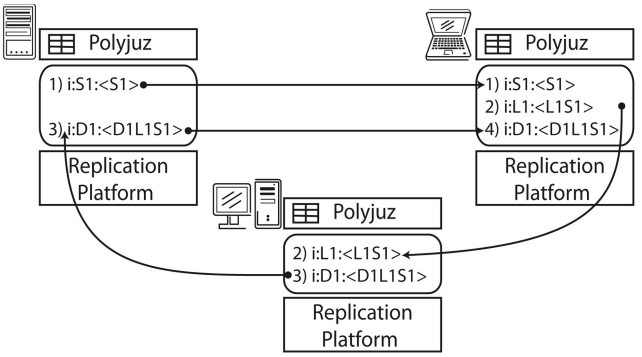


Fig. 3. Example metadata changes during weakly consistent replication for a scenario with three replicas of equal fidelity.

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 *knowledge* of the updates they have seen, for instance, the version vectors of all their items combined into a single-version vector [1], [10], [13], [15], and only send this compressed knowledge during a synchronization.

These basic mechanisms ensure that the system achieves *eventual consistency*. That is, if updates cease, all replicas in the system reach an identical state, appearing to have applied all nonconflicting updates to an item in the same order.

Fig. 3 illustrates the operations of a weakly consistent replication system for an example scenario with three replicas—the server, laptop, and desktop, depicted in Fig. 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 Contact Management Scenario

Fig. 4 describes a sample scenario that illustrates the difficulty in maintaining an address book on three devices of varying fidelity. As in Fig. 2, the server and the laptop are full-fidelity devices while the cell phone is a low-fidelity device. In step 1, the server adds a new contact for Jon Doe to the company's address book. The laptop receives this new contact during a synchronization. In step 2, the user proceeds to update Jon Doe's e-mail address from *jon@acm.org* to *doe@acm.org* on his laptop. This updated contact is sent to the

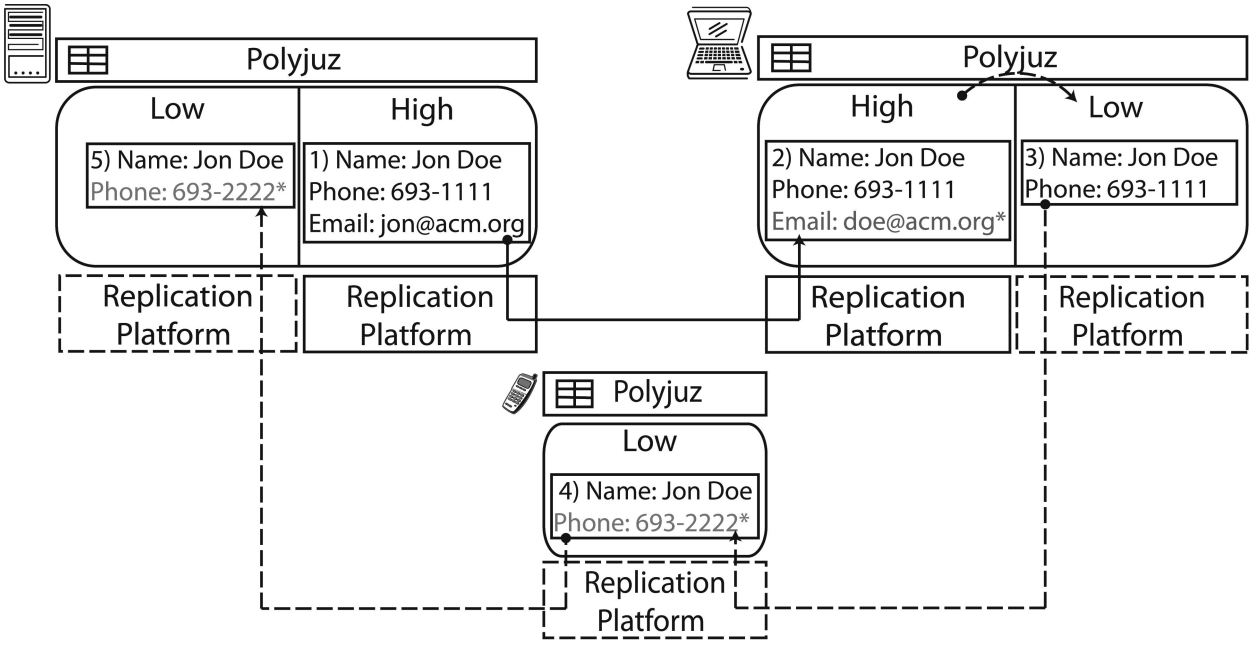


Fig. 4. Sample scenario that illustrates the difficulty in maintaining an address book on three devices of varying fidelity. We indicate updates with an "$*$".

cell phone on a subsequent synchronization. However, since the cell phone's address book is constrained to storing only a contact's name and phone fields, the laptop generates a low-fidelity representation of Jon Doe's contact to ship to the cell phone. In step 4, the user updates Jon Doe's phone number from 693-1111 to 693-2222 on the low-fidelity cell phone. Later, the cell phone synchronizes with the server and sends over an updated contact for Jon Doe with a new phone number. In step 5, the server reconciles the cell phone's updates to the low-fidelity Jon Doe contact with its high-fidelity representation.

Ideally, we would like the server to discern that the cell phone has only updated Jon Doe's phone number and hence, the reintegrated contact should contain the updated phone number while retaining all the other fields (such as, name, e-mail address, etc.) as is. Similarly, when the server next synchronizes with the laptop, we would like the server to retain Jon Doe's updated phone number (the cell phone's update) while replacing Jon Doe's e-mail address with the laptop's version. Finally, the server should push Jon Doe's complete contact, with both the updated phone number and e-mail address to the laptop so all devices have a consistent view over Jon Doe's contact.

### 3.5 Fidelity-Aware Replication in Polyjuz

We introduce fidelity awareness to weakly consistent replication through the novel use 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 *tagged version number* 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 replica.

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

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

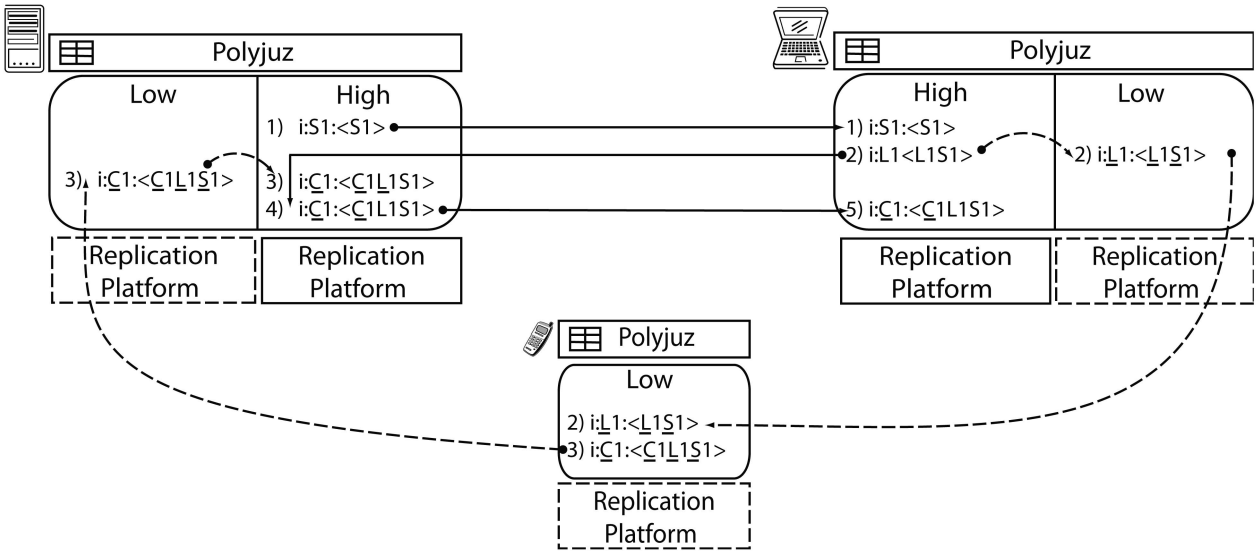


Fig. 5. Example metadata changes during two updates, a fidelity reduction, and two update reintegrations in Polyjuz for a scenario with two high-fidelity replicas and one low-fidelity replica. An underline of a replica identifier indicates low-fidelity version. The system converges to a consistent state at the end of these interactions.

register automated conflict resolvers and notifies an application about concurrent updates.

### 3.6 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.

#### 3.6.1 High-to-Low Transformation

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

1. A version of $i$ is present in the higher fidelity collection but no versions of $i$ are in the lower fidelity collection. This is the case of first appearance of an item in the lower fidelity world.
2. The version in the higher fidelity collection, $v_h$, has a different version number than the version $v_l$ in the lower 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 higher fidelity collection, $v_h$, has the same version number as the version $v_l$ in the lower 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 higher fidelity collection to a lower fidelity collection. If the version of the item in the higher fidelity collection, $v_h$, is already at a fidelity level lesser than or equal to the fidelity level of the lower 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 lower fidelity collection with the help of some application-specific procedure to parse the data format. 2) It sets the item's fidelity tag to the minimum of the fidelity level of the lower 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 lower fidelity collection.

Fig. 5 illustrates the above fidelity-reduction operation for the contact management scenario depicted in Fig. 4. In step 1, the server $S$ first creates Jon Doe's contact, henceforth referred to as 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 $L$ updates Jon Doe's e-mail address, 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. In our examples, we use a line below the replica identifier to indicate a version tagged as low-fidelity—no line denotes a full-fidelity version. Ultimately, the cell phone $C$ receives the low-fidelity version $i{:}\underline{L}1{:}<\underline{L}1\underline{S}1>$.

#### 3.6.2 Low-to-High Transformation

Polyjuz copies an item $i$ from a lower fidelity collection to a higher fidelity collection when one of the following conditions is satisfied:

1. A version of the item is present in the lower fidelity collection, but no versions of the item are in the higher fidelity collection. This is the case of first appearance of an item in the higher fidelity world. It occurs when the item is created on a lower fidelity replica or when a fidelity-reduced version is received from a lower fidelity replica first.
2. The version $v_l$ in the lower fidelity collection has a different tagged version number than the version $v_h$ in the higher 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 higher fidelity collection without making any changes to data or metadata. This might result in multiple versions—a higher fidelity version and an updated lower fidelity version—of an item in the high-fidelity collection. For example, in Fig. 5, in step 3, the cell phone $C$ makes a low-fidelity update to item $i$ (updated phone number) 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 Fig. 6. The actual update reintegration happens in a separate process described next.

### 3.7 Update Reintegration

The underlying replication platform sees the multiple versions as concurrent updates to the same item in the higher 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 lower fidelity replica takes an item from a higher fidelity replica, makes an update, and passes it back to the higher fidelity replica, does not actually produce this case but the following one (see Fig. 5).

**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 bitwise OR operation).

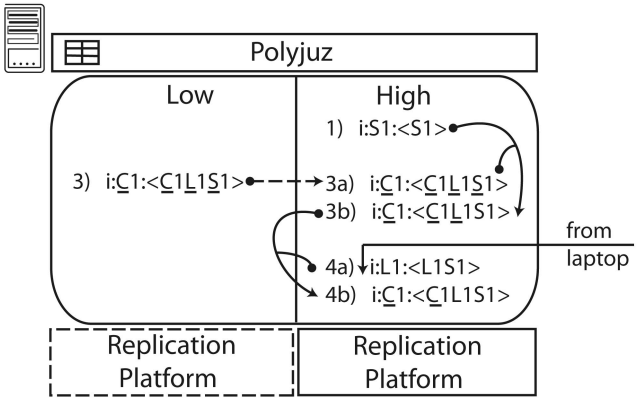


Fig. 6. Update reintegrations on server $S$ split into two substeps each. In step 3a, the updated version is copied from the low-fidelity collection to the high-fidelity collection. In step 3b, the update is reintegrated with the older version through an automated conflict resolution process invoked 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.

Continuing the illustrative example in Fig. 6, 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 Fig. 5) 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 now has the high-fidelity tag. This correctly identifies that 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 (Fig. 6), 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 Fig. 5, 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.

#### 3.7.1 Atomic Updates

Polyjuz can ensure that updates are reintegrated atomically by maintaining additional state. Each replica keeps 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

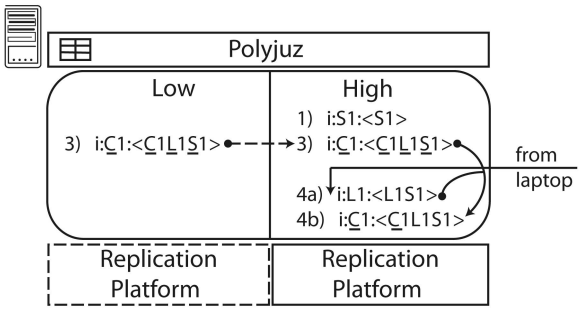


Fig. 7. Update reintegration on server $S$ with atomicity enforced. There are two unintegrated versions at the end of step 3 because an intermediate update is not available at the correct fidelity. The required high-fidelity version is received from the laptop in step 4a, and reintegration finally happens in step 4b.

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 Fig. 5, at the end of step 3, the high-fidelity collection on the server has two nonintegrated versions $S1$ and $\underline{C}1$ since the update $L1$ made by the laptop is not available at the required, high-fidelity level. Fig. 7 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.

## 4 Implementation Guidelines

The previous section presented the design of the Polyjuz layer. In this section, we discuss some practical issues that arise in applying the Polyjuz design to generic replication platforms and applications. A specific implementation of Polyjuz for Microsoft Sync Framework and two applications are outlined in Sections 5 and 6.

### 4.1 Layering

Polyjuz can be layered transparently atop any weakly consistent replication platform that supports disconnected operation and automated conflict resolution. Polyjuz is transparent to applications as it provides the same interface as the underlying replication platform. When an application updates a data item, the intermediary Polyjuz layer encapsulates the updated data item with its fidelity metadata to create a composite item. Polyjuz is transparent to the underlying replication platform as it simply notifies the platform of the update as the regular application would, and provides the composite item as the object to synchronize. The underlying replication platform replicates this composite item to the remote device. The Polyjuz layer on the remote device extracts the fidelity metadata and returns the composite item to its original state so the remote application only sees the raw data item it expects.

Polyjuz offers three options for data encapsulation. First, Polyjuz can store the encapsulated and raw forms of each data item separately—the encapsulated form is visible to the replication platform while the raw form is visible to the application. Polyjuz keeps the raw and encapsulated versions synchronized. Second, Polyjuz packs the fidelity metadata into an unused portion of the data item, such as the *comments* field in a JPEG image. While this option reduces the number of duplicate copies of the data item, it might inadvertently expose fidelity metadata to a user. Third, Polyjuz can perform the encapsulation on the fly at the transport layer, if the replication platform supports a custom transport mechanism. Our prototype implementation uses the third option.

Our goal of transparency ensures that applications can utilize Polyjuz without any modification. However, since fidelity is an application-specific notion, Polyjuz requires that each application provide it with a component that can parse data items and define fidelity levels suitable for the application's target devices. Polyjuz invokes this application-aware component to perform fidelity reduction and update integration. Similarly, Polyjuz does not require any modifications to the replication platform apart from the ability to register Polyjuz's custom fidelity-aware conflict resolver.

### 4.2 Optimizations

Polyjuz incurs additional storage and bandwidth overhead in order to support fidelity-aware replication. This section presents an analysis of these overheads and discusses a few optimizations to alleviate them.

**Storage.** The 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 fidelity-aware storage component need only store a single copy of the item, at the highest fidelity level supported by the replica, and generate a reduced-fidelity representation when required.

Fidelity metadata maintained by Polyjuz also incurs storage overhead. We expect the size of a fidelity tag to be small and fixed in a bit-vector representation, one 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. This results in a per-item storage cost of $O(R \times f_r)$ at replica $r$, 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.

**Bandwidth.** The fidelity metadata adds a bandwidth cost ($O(R)$) every time an item is transferred over the network. This cost will be modest if the number of replicas is small. Moreover, the cost of knowledge exchange during synchronizations (delegated to the underlying replication layer) remains unaffected.

**Spurious conflicts.** On the other hand, these are a key source of bandwidth overhead in Polyjuz. They occur in the underlying replication platform when two or more replicas independently perform fidelity reductions or update integrations for the same version of an item. The resulting versions are identical in content and fidelity metadata, but appear as distinct versions to the underlying replication

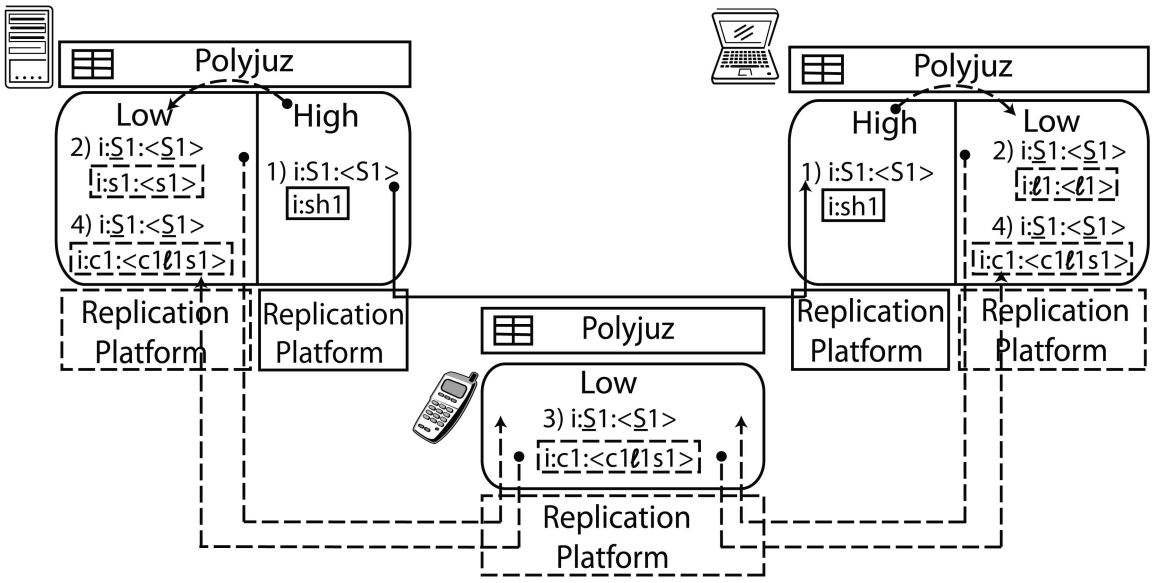


Fig. 8. A scenario illustrating how spurious conflicts occur because of Polyjuz's layered design. The version vectors within the boxes are illustrative of the underlying replication platform's metadata.

platform. Therefore, the replication platform replicates each conflicting version separately. Similarly, when Polyjuz automatically resolves a spurious conflict as presented in Section 3, the replication platform replicates the resulting conflict-resolved version as well.

Fig. 8 illustrates the bandwidth overhead of spurious conflicts with an example. In step 1, the server $S$ creates item $i$ with version number $S1$ and version vector $<S1>$ in the high-fidelity collection and synchronizes it with the laptop $L$ through the underlying replication platform. We differentiate Polyjuz's version vector from the underlying replication platform's version vector $i{:}sh1$ which is illustrated within a box. In step 2, both the server $S$ and the laptop $L$ produce a fidelity-reduced version of item $i$ with the version number $\underline{S}1$ and version vector $<\underline{S}1>$ and store this in their low-fidelity collections. To the replication platform in the low-fidelity world, they appear to be two concurrently introduced versions of item $i$ with version vectors $i{:}s1{:}<s1>$ and $i{:}l1{:}<l1>$, respectively, on the server and laptop. In step 3, when the cell phone $C$ synchronizes with the laptop and the server, it receives both fidelity-reduced versions. Since their version vectors in the underlying replication platform don't match, a conflict is flagged. The Polyjuz layer in $C$ resolves this conflict automatically, resulting in a new version of the item in the underlying replication platform $i{:}c1{:}<c1l1s1>$. When the cell phone next synchronizes with the server and the laptop in step 4, the conflict-resolved version is replicated. Note that the spurious conflict results in four data transfers involving the cell phone. Of the four, only one data transfer was required while the rest duplicated data already present on the remote devices.

The number of *spurious transfers* that occur depends on the number of replicas performing fidelity transformations of an item and the pattern of synchronization in the system. Three schemes can alleviate the impact of spurious transfers:

1. Fidelity transformations can be restricted to a single replica. Since only one replica performs fidelity reductions and update integrations, spurious conflicts never occur. This option might be undesirable as this full-fidelity replica could become a bottleneck in the flow of updates between the high and low-fidelity worlds.
2. Lower fidelity collections of multifidelity replicas can be forced to synchronize whenever the high-fidelity collections synchronize. In the above example, this means that the low-fidelity collections on the server and the laptop should also synchronize over the network subsequent to the synchronization of the high-fidelity collections. In general, it is superfluous for the lower fidelity collections of multifidelity replicas to synchronize because the high-fidelity synchronization already transfers the necessary data, which can then be fidelity-reduced locally. However, this local fidelity reduction leads to a spurious conflict in the underlying replication framework. Forcing the lower fidelity collections to synchronize after the corresponding high-fidelity synchronization enables them to reconcile the spurious conflict between themselves. While forced synchronization does not eliminate spurious transfers, the resulting overhead is incurred by the well-provisioned multi-fidelity replicas instead of resource-constrained low-fidelity replicas.
3. If the underlying replication platform allows custom transport mechanisms, the duplicate transfer of data can be suppressed through a two-phase data transfer protocol. In the first phase, only the fidelity metadata part of the item is transferred over the network. If the fidelity metadata indicate that the item is indeed a genuine update, the second phase subsequently fetches the data component. While this method does not eliminate spurious conflicts, it avoids the overhead of transferring data unnecessarily over the network.

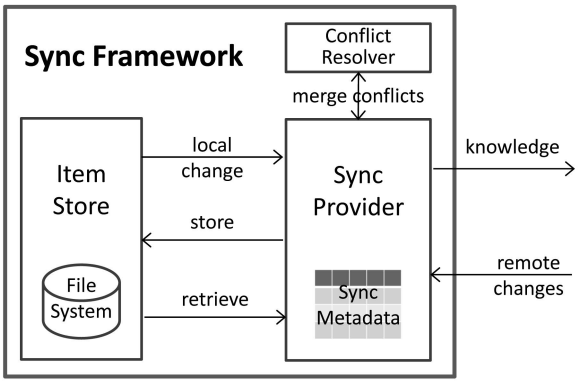


Fig. 9. The architecture of Microsoft Sync Framework software on each replica.

### 4.3 Consistency

Polyjuz guarantees that the multifidelity system attains eventual consistency. This guarantee stems from the following three properties. First, we assume that the underlying replication platform ensures eventual consistency within each fidelity-specific collection. That is, the underlying platform propagates updates to all replicas within the fidelity world in which the update originated. Second, the Polyjuz layer maintains strongly consistent copies of items across collections of different fidelity on a single host. That is, it ensures that an update for an item, either generated by the local application or learned via replication in a fidelity world, is applied to all representations of the item supported by that replica, with appropriate fidelity transformations. Third, fidelity transformations are deterministic as Polyjuz layers on different replicas perform transformations in the exact same way, resulting in eventually consistent data and fidelity metadata across replicas. These three properties ensure that the overall system converges to a consistent and correct state.

## 5 Implementation

We have implemented Polyjuz in C# using the Microsoft Sync Framework [23] as the underlying replication platform. This section describes how Polyjuz interacts with the Sync Framework, manages fidelity metadata, and supports applications. We first describe the software architecture of the Sync Framework.

The core component of the Microsoft Sync Framework is called the Sync Provider. The Sync Provider manages synchronization metadata for replicated items and implements a synchronization protocol for exchanging updates with other Sync Providers. An additional component called the Item Store manages storage of replicated items, which may reside in the file system, in a database, or in memory. The Item Store provides a store and retrieve interface to the Sync Provider, as well as notifying the Sync Provider when an application makes an update. Applications access and update items directly (for instance, through the file system). In addition, the Sync Provider interacts with an application-specific Conflict Resolver to handle conflicts automatically. Fig. 9 illustrates the software architecture of the Microsoft Sync Framework.

In the context of the Microsoft Sync Framework, Polyjuz acts as a mediator between the Sync Provider and the Item Store. It appears as an item store to the Sync Provider and as a sync provider to the Item Store. Fig. 10 illustrates this software architecture on a dual-fidelity replica. The replica has two sets of Sync Framework components for the high and low-fidelity worlds. The respective Sync Providers synchronize with other Sync Providers within their own fidelity worlds, and the respective Item Stores are rooted at different file system directories (or database tables). The Polyjuz component is shared between the two worlds, appearing as a fidelity-aware Item Store to each of the Sync Providers.

The Polyjuz Item Store stores a data item and its fidelity metadata separately; but presents a compound, encapsulated item to the Sync Provider, performing encapsulation on the fly. The components of an encapsulated item—the original item and its fidelity metadata are stored in the Item Store and a Metadata Store, respectively. Separate per-item fidelity metadata, namely a tagged version number and version vector, are maintained for each fidelity level supported by the device. Polyjuz stores the fidelity metadata on persistent media but also caches it in memory for efficient access.

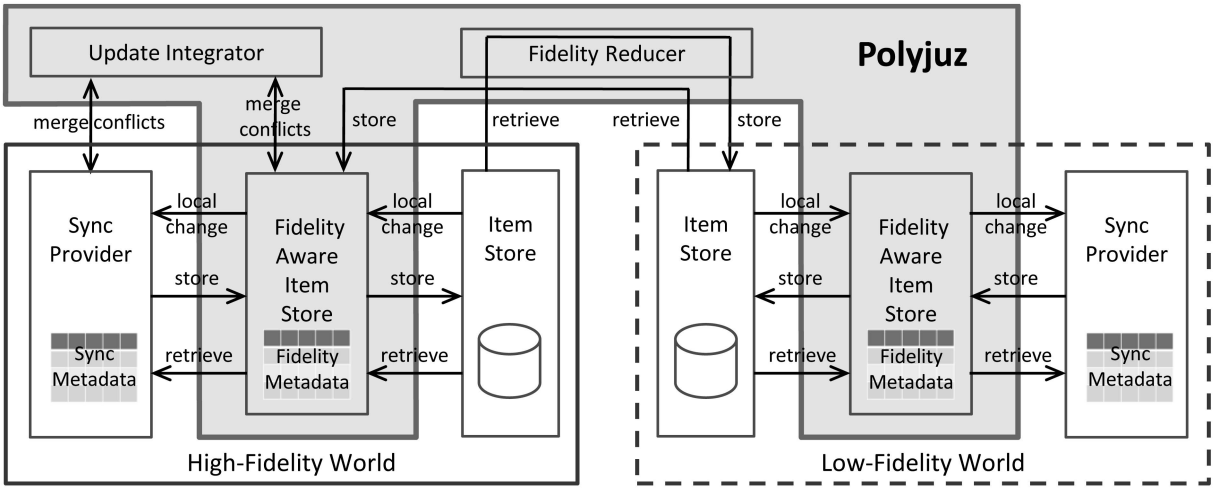


Fig. 10. Software architecture of Polyjuz on a dual-fidelity replica.

Two other Polyjuz components are responsible for fidelity reduction and update integration. The Fidelity Reducer acts as a bridge between the high and low-fidelity Item Stores. The high-fidelity Item Store notifies the Fidelity Reducer of local updates, which reduces the fidelity of the updated item and stores it in the low-fidelity Item Store. The Update Integrator, on the other hand, is registered as a Conflict Resolver to the high-fidelity Sync Provider. It integrates updates by merging conflicts detected by the Sync Provider.

Building applications with Polyjuz is easy. An application developer needs to provide two application-specific components to Polyjuz. The first is an Item Store that is capable of handling the application's data. This could be a generic item store, which can read and write files for example, or an application-specific item store, which marshals and unmarshals application data in a special manner. This component is the same as what the developer would build for any application that manages replicated data using the Microsoft Sync Framework.

The Fidelity Reducer and Update Integrator components can be customized through the use of application-specific callbacks. These routines help Polyjuz to parse application-specific data formats, reconstruct correctly formatted data, and implement application-specific merge semantics. For example, the Fidelity Reducer can be tailored to produce a low-fidelity version of a high-fidelity item without requiring Polyjuz to understand the details of the data object being transformed.

Users interact with a Polyjuz-enabled replication system just as they would without the Polyjuz enhancements. That is, they directly insert, update, and delete items through the file system (or database). We assume that users access the items at the highest fidelity supported by the replica. The corresponding lower fidelity representations appear only to users of lower fidelity replicas.

## 6 Case Studies

We built two applications to explore how Polyjuz works in practice. The first application, PolyContacts, replicates address book entries between devices of different fidelity. The second, PolyPix, provides fidelity-aware image replication.

### 6.1 PolyContacts

PolyContacts is a fidelity-aware contacts management application that implements a concise representation of an address book contact to be transferred to the user's mobile devices from other high-fidelity devices such as desktops and laptops. The user is free to update the contact on his mobile device. When the mobile device synchronizes with the desktop (or laptop), 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 XML file. Updating an address book entry is as simple as editing this XML document. PolyContacts uses a programmatic interface, the Windows Contacts API [22], 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 creates a shadow replica of this directory, hidden from the user, and uses it as the root directory for the low-fidelity replication subsystem. It presents these directories to Polyjuz through an application-specific Item Store. In addition, PolyContacts implements customized fidelity reduction and integration functions as follows:

**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 e-mail 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 Fig. 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 e-mail 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 e-mail 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.

**Update integration.** 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 e-mail 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.

### 6.2 PolyPix

In addition to PolyContacts, we built PolyPix, a fidelity-aware image replication system that enables high-fidelity devices, such as desktops and laptops, to share images with lower fidelity devices, such as cell phones and photo sharing Web services.

#### 6.2.1 PolyPix Design

Our primary goal when designing PolyPix was to ensure that no device wastes storage space or bandwidth storing a higher resolution image than what it can display. Hence, PolyPix adapts images to match the resolution supported by the target device—for instance, a high-fidelity image from a desktop is scaled to a lower resolution before being replicated on a low-fidelity cell phone.

PolyPix permits two types of updates to a shared image. First, any replica can update an image's metadata (e.g., change an image rating, or tag). PolyPix replicates image metadata consistently across all interested replicas allowing

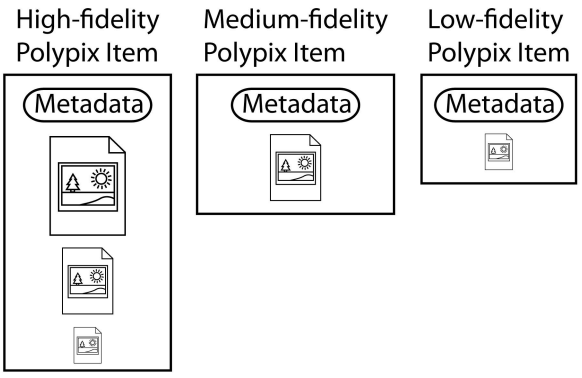


Fig. 11. Each device stores the image metadata and the PolyPix components appropriate for its fidelity level. Full-fidelity devices store all available components.

an update on a low-fidelity cell phone or a photo sharing Web service to be reintegrated with the full-fidelity image on the user's desktop. Second, only full-fidelity replicas can perform content-based updates (e.g., red-eye reduction or crop) on a full-fidelity image. PolyPix treats an image updated in this manner as a new image that can be propagated to other full-fidelity devices as is, or transformed to generate a fidelity-reduced representation that can be pushed out to lower fidelity replicas as a newer version of an existing image. PolyPix does not currently support content updates to a reduced-fidelity image. We have considered retaining updated reduced-fidelity images in the system and displaying them to the user as a conflict during reintegration, but dot not currently support this.

#### 6.2.2 PolyPix Implementation

PolyPix performs fidelity-aware image replication by treating an image as a data structure with multiple components. Apart from the image content itself, most image file formats (e.g., JPEG, PNG, and GIF) include an image header that stores image metadata such as the owner, comment, image title, creation time, and modification time. If the header is not present in a given image, it is possible to programmatically add a header (of a specified size) to the image file.

PolyPix creates a new *resolution* component for each fidelity at which an image is replicated. For example, in a replication system with a desktop, a cell phone, and a photo sharing Web service, PolyPix might create a medium-resolution-image component and a low-resolution-image component in addition to the original full-fidelity image component. A full-fidelity device (the desktop in our example system) stores the metadata (i.e., image header) and all available resolutions of the image. Lower fidelity devices only store the image metadata and those components that they can display. For example, the photo sharing Web service stores medium fidelity PolyPix items which consists of the image metadata and the medium-resolution component, while the cell phone stores low-fidelity PolyPix items composed of the metadata and the low-resolution image component. Fig. 11 illustrates this example.

We implemented PolyPix as a new fidelity-aware Item Store. For image manipulation, we used the image metadata editing APIs bundled with the .NET Framework Class Library as well as the command line tools for resolution-scaling from ImageMagick (version 6.5.6). Like PolyContacts, PolyPix replicates all the images under a given directory and creates shadow replicas for any low-fidelity subsystems. In addition, PolyPix implements the stubs that assist in fidelity reductions and update integration.

**Fidelity reduction.** PolyPix's fidelity reduction function takes as input a full-fidelity image with a valid header. It invokes the ImageMagick tool to scale down the image to a resolution supported by the target device. The function then copies over the image header from the full-fidelity image to the newly created reduced-fidelity image.

**Update integration.** PolyPix's update integration function takes as input a full-fidelity image and a fidelity-reduced image, and returns an image with the header of the fidelity-reduced image but the content of the full-fidelity image.

## 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?
- How much overhead does Polyjuz entail in the form of fidelity reductions, update integrations, and spurious conflicts?
- What optimizations help in reducing the overhead due to spurious conflicts and updates?

### 7.1 Methodology

We ran experiments using the C# implementation of Polyjuz with Microsoft Sync Framework as the underlying replication platform (see Section 5). Our experimental evaluation had five replicas representing the scenario in Fig. 1, but running on a single computer. Bob's Laptop and the Exchange server support dual-fidelity (i.e., two worlds each) while the three other replicas (Bob's and Chuck's cell phones and Bob's Desktop) support either high or low fidelity (one world).

Before each experiment, we replicated 100 contacts among the five replicas. We then randomly selected replicas to perform updates using the PolyContacts application followed by synchronizations. We report on their convergence trends and on the number of 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 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. Fig. 12 shows that the system converges after approximately 50 synchronizations. Since we perform one synchronization after every update, the last update happens around the 30th synchronization.

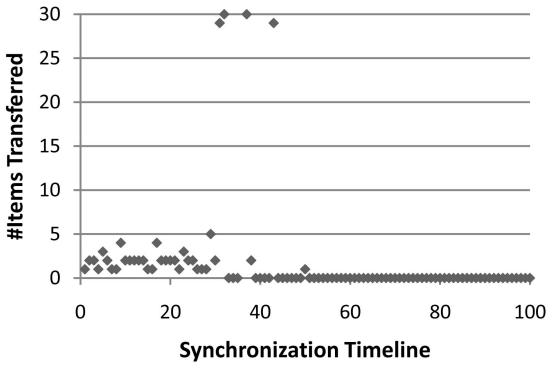


Fig. 12. This scatter plot shows the number of items transferred by the Sync Framework when synchronizing 100 contacts with 30 high-fidelity updates.

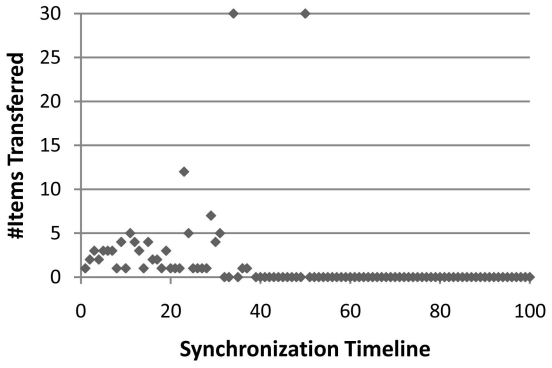


Fig. 13. This scatter plot shows the number of items transferred by the Sync Framework when synchronizing 100 contacts with 30 low-fidelity updates.

Table 1 provides a breakdown of the operations for 30 high-fidelity updates. From the table, we see that the desktop, laptop, and server each carried out 14, 4, and 12 updates, respectively. We also observe that the laptop and the server, respectively, performed 29 and 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. The laptop did not perform fidelity reduction for one update as it received it for the first time from Bob's cell phone in the fidelity-reduced format instead of from the server or the desktop in the high-fidelity format, as it did for all other updates.

We see that Bob's cell phone received 59 items as sync transfers, of which about 29 resulted in a spurious conflict or update. The spurious 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. Furthermore, a spurious conflict once resolved shows up as an update in the Sync Framework and eventually results in a spurious transfer to the other replica. Polyjuz examines the fidelity metadata associated with these spurious versions and ignores the conflict.

Observe that, for each replica, subtracting the number of spurious transfers from the number of items received gives the number of updates introduced into the system by other replicas (except for the laptop, for which it is one higher due to the additional fidelity-reduced update it received from Bob's cell phone for one item). Similarly, adding up all spurious transfers also corresponds to twice the number of fidelity reductions performed by the laptop and server. The two spurious transfers per fidelity reduction corresponds to a spurious conflict and a subsequent spurious update.

### 7.3 Low-Fidelity Updates

In our second experiment, we randomly select a low-fidelity replica, (Bob or Chuck's cell phone) and have it update a contact. As before, we intersperse 30 update operations with synchronizations. Fig. 13 shows that the system again converges after approximately 50 synchronizations.

Table 2 provides a breakdown of the operations for 30 low-fidelity updates. Bob's cell phone makes 11 updates while Chuck's makes 19, adding up to 30. We see that Bob's laptop and server perform 30 update integrations each as a result of these updates. These update integrations, in turn, lead to 30 spurious conflict resolutions at the desktop and subsequently to 30 spurious updates each at the laptop and server.

### 7.4 High-Fidelity Updates Followed by Low-Fidelity Updates

Our third experiment was designed to evaluate the difficult update reintegration scenario described in Fig. 5. 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

TABLE 1
Breakdown of Operations for High-Fidelity Updates

| Replication | Low | | High | Dual | |
|---|---|---|---|---|---|
| | bCell | cCell | Desk | Lap | Srv |
| Local updates | - | - | 14 | 4 | 12 |
| Reductions | - | - | - | 29 | 30 |
| Integrations | - | - | - | - | - |
| Items transferred to | 59 | 59 | 17 | 56 | 48 |
| Spurious transfers | 29 | 29 | 1 | 29 | 30 |

TABLE 2
Breakdown of Operations for Low-Fidelity Updates

| Replication | Low | | High | Dual | |
|---|---|---|---|---|---|
| | bCell | cCell | Desk | Lap | Srv |
| Local updates | 11 | 19 | - | - | - |
| Reductions | - | - | - | - | - |
| Integrations | - | - | - | 30 | 30 |
| Items transferred to | 19 | 11 | 60 | 60 | 60 |
| Spurious transfers | - | - | 30 | 30 | 30 |

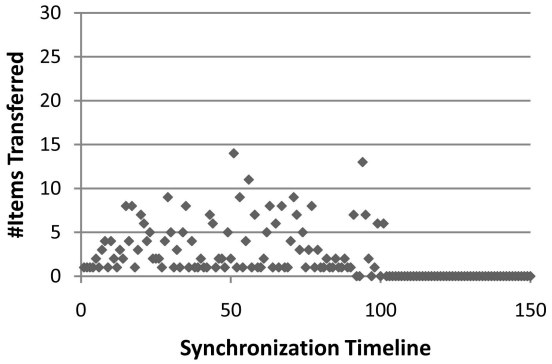


Fig. 14. This scatter plot shows the number of items transferred by the Sync Framework when synchronizing 100 contacts with 30 high-fidelity updates followed by low-fidelity updates to the same item.

finally propagates the twice-updated contact in the low-fidelity world.

In this scenario, we performed 30 such dual-updates. From Fig. 14, we see that the system converges after about 100 synchronizations—our last update occurs around the 60th synchronization.

Table 3 provides a breakdown of the operations for 30 high-followed-by-low-fidelity updates. The dual-world replicas, Laptop and Server, performed 11 and 19 updates, respectively, and a corresponding low-fidelity device also performed 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. We refrain from explaining the other numbers as they depend on the order in which replicas are selected to update items and to synchronize.

### 7.5 Reducing Spurious Transfers

In this section, we evaluate the three optimizations outlined in Section 4.2 for alleviating the overhead due to spurious transfers. For each optimization, we perform the high-followed-by-low-updates experiment described in Section 7.4.

We demonstrate the effectiveness of each optimization by measuring four components: *metadata transfers* show the total number of fidelity metadata exchanges among the replicas, *contact transfers* show the total number of contacts transferred among the replicas, *spurious transfers low* indicate the number of spurious transfers that occur between replicas in the low-fidelity world, and *spurious transfers high* denote the number of spurious transfers between replicas in the high-fidelity world.

TABLE 3
Breakdown of Operations for
High-Followed-by-Low-Fidelity Updates

| Replication | Low | | High | Dual | |
|---|---|---|---|---|---|
| | bCell | cCell | Desk | Lap | Srv |
| Local updates | 11 | 19 | - | 11 | 19 |
| Reductions | - | - | - | 42 | 33 |
| Integrations | - | - | - | 7 | 27 |
| Items transferred to | 67 | 67 | 54 | 75 | 73 |
| Spurious transfers | 36 | 36 | 3 | 37 | 32 |

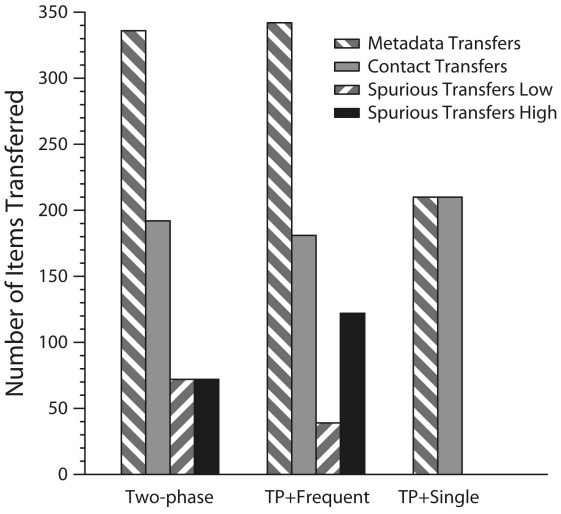


Fig. 15. This graph shows how effective our optimizations are in reducing the overhead of spurious transfers.

The first optimization we evaluate is the two-phase data transfer protocol which first transfers the fidelity metadata of an updated contact to the remote replica. Only if the remote replica is unaware of the update is the actual contact transferred. Our results are shown in the leftmost set of bars of Fig. 15. We see that it takes 336 fidelity metadata transfers for the replicas to converge following the 60 updates. Observe that the "Items transferred to" row of Table 3 adds up to 336. Our two-phase optimization allows the remote replica to only request genuine updates reducing the number of contacts transferred to 192. Thus, there are 72 spurious metadata transfers that occur on the low and the high-fidelity replicas, respectively. Again, observe that the "Spurious transfers" row of Table 3 adds up to 144. Clearly, the two-phase protocol is effective in eliminating spurious transfers of contact data.

The second optimization we evaluate forces the low-fidelity collections of dual-fidelity replicas to synchronize whenever their corresponding high-fidelity collections synchronize. We retain our first optimization, i.e., the two-phase data transfer so we can see the relative improvement of the second optimization. The middle (TP + Frequent) set of bars of Fig. 15 presents our results. This optimization decreases the number of spurious transfers on low-fidelity replicas considerably while increasing the number of spurious transfers on high-fidelity replicas by an equivalent measure. As with the previous experiment, the number of spurious transfers low (39), spurious transfers high (122), and contact transfers (181) add up to the number of metadata transfers (342).

The final optimization we evaluate restricts fidelity operations to a single replica. The rightmost set of bars (TP + Single) of Fig. 15 presents our results. As expected, the number of metadata transfers equal the number of contact transfers indicating that there were no spurious conflicts or transfers in this experiment. While this optimization seems attractive, it comes at the cost of reduced flexibility. The

nonavailability or nonreachability of the single, fidelity-transforming replica can considerably delay the flow of updates between the high and low-fidelity worlds.

## 8 Related Work

Odyssey [9] and dynamic distillation [5] were among the first to demonstrate that clients of replicated systems can save bandwidth, power and improve performance by incorporating mobile adaptation and fidelity support as a first-class design principle. Several systems, such as Puppeteer and PageTailor, have demonstrated content adaptation for specific applications. Puppeteer [4] adapts PowerPoint presentations and Web (HTML) pages by decomposing a document into a hierarchy of components, each of which maybe reduced in fidelity, say by omitting subcomponents or degrading images. PageTailor [2] automatically adapts Webpages for viewing on a PDA.

Some recent file systems provide a more generic, application-independent support for dealing with multiple representations. EnsemBlue [12] integrates consumer electronic devices with general-purpose computers through device-specific plug-ins that deal with custom formats and interfaces. These plug-ins run on the general-purpose computers to which the devices connect, perform necessary format transformations, and store state on the device itself to enable similar operations on other computers. Another file system, quFiles [20], maintains multiple representations of a file internally while presenting a single, context-aware representation to the applications on the device, accounting for the device's screen size, network connectivity, current battery status, etc.

Polyjuz complements the above systems through fidelity-aware peer-to-peer replication. EnsemBlue plug-ins 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, plug-ins can help Polyjuz execute its replication logic on computers serving as proxies for devices that do not permit custom code.

Few systems have been developed that support updates to fidelity-reduced content. CoFi [3] 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.

MoxieProxy [14] provides a methodology and middleware architecture for reconciling updates to fidelity-reduced data. Certain types of operations on transcoded data can be reintegrated with their full-fidelity counterparts, and application-defined *transforms* can convert an operation on transcoded data into an equivalent operation on the original data in a few cases including image, speech-to-text, and pdf-to-text conversions. Update reintegration enabled by these transforms then happen at a central server. 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 [18]. Replication systems, such as Coda [7], Ficus [6], Bayou [13] and WinFS [10], replicate collections consisting of entire databases or file volumes. However, each replica may choose to only cache a subset of the items in the collection due to limited storage or other constraints. A simple approach to provide fidelity awareness on top of these systems is to replicate the collection's items at full fidelity on all replicas and generate additional 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 when lower fidelity items suffice.

Replication systems, such as Cimbiosys [15], Perspective [17], and PRACTI [1], enable replicas to select the set of items they store using queries over the metadata or contents of items. This feature facilitates an alternative approach to support fidelity: an item can be broken into subcomponents (fields), a subset of which is partially replicated onto a device based on the device's fidelity level. 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, complications arise when enforcing additional semantics such as atomicity of updates to multiple components in an item and detecting concurrent, conflicting updates made to different components of the same item.

## 9 Conclusions

Polyjuz is a 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. This paper presented a design to support applications with data formats that separate into multiple fields—address books, calendar entries, and e-mail—and demonstrated a concrete application built on Polyjuz for managing personal contacts. We also demonstrate that Polyjuz can provide limited support for fidelity-based adaptation of multimedia data such as updating comments and tags in a photo header.

## Acknowledgments

The authors thank Rama Kotla, Cathy Marshall, Iqbal Mohomed, and Patrick Stuedi for inspiring discussions and feedback on earlier revisions of this paper. They also thank Lev Novik and the Microsoft Sync Framework product team for bringing the problem of fidelity to their attention and helping them use their replication framework.

## References

[1] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "PRACTI Replication," *Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 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," *Proc. ACM MobiSys*, June 2007.

[3] E. de Lara, R. Kumar, D.S. Wallach, and W. Zwaenepoel, "Collaboration and Multimedia Authoring on Mobile Devices," *Proc. ACM MobiSys,* May 2003.
[4] E. de Lara, D.S. Wallach, and W. Zwaenepoel, "Puppeteer: Component-Based Adaptation for Mobile Computing," *Proc. USENIX Symp. Internet Technologies and Systems (USITS),* Mar. 2001.
[5] A. Fox, S.D. Gribble, E.A. Brewer, and E. Amir, "Adapting to Network and Client Variability via On-Demand Dynamic Distillation," *Proc. ACM Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* Oct. 1996.
[6] R.G. Guy, "Ficus: A Very Large Scale Reliable Distributed File System," Technical Report CSD-910018, Computer Science Dept., Univ. of California, June 1991.
[7] J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Trans. Computer Systems,* vol. 10, no. 1, pp. 3-25, Feb. 1992.
[8] P. Kumar and M. Satyanarayanan, "Flexible and Safe Resolution of File Conflicts," *Proc. USENIX Winter Technical Conf.,* Jan. 1995.
[9] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker, "Agile, Application-Aware Adaptation for Mobility," *Proc. ACM Symp. Operating Systems Principles (SOSP),* Oct. 1997.
[10] 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.
[11] 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 Trans. Software Eng.,* vol. 9, no. 3, pp. 240-247, May 1983.
[12] D. Peek and J. Flinn, "EnsemBlue: Integrating Distributed Storage and Consumer Electronics," *Proc. USENIX Symp. Operating Systems Design and Implementation (OSDI),* Nov. 2006.
[13] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers, "Flexible Update Propagation for Weakly Consistent Replication," *Proc. ACM Symp. Operating Systems Principles (SOSP),* Oct. 1997.
[14] T. Phan, G. Zorpas, and R. Bagrodia, "Middleware Support for Reconciling Client Updates and Data Transcoding," *Proc. ACM MobiSys,* June 2004.
[15] 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," *Proc. USENIX Conf. Networked Systems Design and Implementation (NSDI),* Apr. 2009.
[16] P.L. Reiher, J.S. Heidemann, D. Ratner, G. Skinner, and G.J. Popek, "Resolving File Conflicts in the Ficus File System," *Proc. USENIX Summer Technical Conf.,* June 1994.
[17] B. Salmon, S.W. Schlosser, L.F. Cranor, and G.R. Ganger, "Perspective: Semantic Data Management for the Home," *Proc. USENIX Conf. File and Storage Technologies (FAST),* Feb. 2009.
[18] D.B. Terry, *Replicated Data Management for Mobile Computing.* Morgan & Claypool, 2008.
[19] 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," *Proc. ACM Symp. Operating Systems Principles (SOSP),* Dec. 1995.
[20] K. Veeraraghavan, J. Flinn, E.B. Nightingale, and B. Noble, "quFiles: The Right File at the Right Time," *Proc. Eighth USENIX Conf. File and Storage Technologies (FAST),* Feb. 2010.
[21] "Live Mesh," http://www.mesh.com, 2010.
[22] "Microsoft Coding4Fun Developer Kit," http://www.microsoft.com/express/samples/C4FDevKit, 2010.
[23] "Microsoft Sync Framework," http://msdn.microsoft.com/en-us/sync/default.aspx, 2010.

**Venugopalan Ramasubramanian** received the BTech degree in computer science and engineering from the Indian Institute of Technology, Chennai, and the PhD degree in computer science from Cornell University, Ithaca, New York, in 2007. He is a researcher at Microsoft Research Silicon Valley. His research interests broadly fall in the fields of distributed systems and networking.

**Kaushik Veeraraghavan** received the BS degree in computer engineering from the University of Maryland, College Park, in 2003. He is a PhD candidate in computer science and engineering at the University of Michigan, Ann Arbor. His research interests are in operating systems, distributed systems, and mobility.

**Krishna P.N. Puttaswamy** received the BS degree in computer science from the National Institute of Technology, Karnataka, India, in 2003. He is a PhD candidate in the Department of Computer Science at the University of California, Santa Barbara. His research interests are in the areas of security and privacy, and large-scale distributed systems.

**Thomas L. Rodeheffer** received the PhD degree from Carnegie Mellon University in 1985. After, he joined the Digital Equipment Corporations Systems Research Center in Palo Alto, California, where he worked for many years in the areas of operating systems and communicating processes. He enjoys working at the boundary between hardware and software, constructing working prototypes, and employing formal methods to deal with the complexity of these systems. He joined Microsoft Research in 2002.

**Douglas B. Terry** received the PhD degree in computer science from the University of California at Berkeley. He is a principal researcher at the Microsoft Research Silicon Valley Lab. His research focuses on distributed systems issues, such as information management, fault tolerance, and mobility. He serves as the chair of the ACM Special Interest Group on Operating Systems (SIGOPS). Prior to joining Microsoft, he was the cofounder and CTO of Cogenia, chief scientist of the Computer Science Laboratory at Xerox PARC, and an adjunct professor in the Computer Science Division at the University of California at Berkeley, where he regularly teaches a graduate course on distributed systems. He is a fellow of the ACM.

**Ted Wobber** is a principal researcher at the Microsoft Research Silicon Valley Lab and an ACM distinguished scientist. His research interests include operating systems, distributed systems, and security. He is currently working on distributed systems and security issues such as those addressed by the Community Information Management project. He is also investigating the architecture and performance of systems using nonvolatile memory, such as solid state disks.