RC 21774 (Log#98049) (08/11/2000)
Computer Science/Mathematics

IBM Research Report

Design and Implementation of the MNCRS Java Framework for Mobile Data Synchronization

Norman H. Cohen

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

This page intentionally left blank.

Table of Contents

1 Introduction

The Mobile Network Computing Reference Specification, or MNCRS [Mon98] is an extension of the Open Group's technical standard for network computing clients [Ope98] to address issues unique to mobile devices, including primarily disconnected operation, power management, communication over slow, expensive, and unreliable links, and support for a wide variety of devices and network connections. The MNCRS was developed by a consortium that was established in June 1997 and came to include Alcatel, Apple Computer, Bellcore, Ericsson Mobile, Fujitsu, Hitachi, Hugh Symons Group, IBM, Lotus Development, Matsushita, Mitsubishi Electric, Netscape Communications, Nokia Mobile Phones, Nortel, Oracle's Network Computer Inc. (NCI), Sharp, Sun Microsystems, and Toshiba. The specification, posted at `http://www.mncrs.org`, defines a common Java-based platform for communicating mobile devices. It includes hardware and software guidelines, as well as proposed standards for data synchronization, mobile communications, the boot sequence, adaptivity, power management, service discovery, security, smart card interfaces, and an application-programming interface (API) for electronic phone books.

This paper is concerned with the MNCRS framework for data synchronization. The heart of the framework is a persistent synchronizable store, or *sync store*, which contains Java objects retrievable by keys called *sync IDs*. A sync ID is also a Java object. There may be peer replicas of a sync store on several different devices, which may be disconnected from each other most of the time. A process called *synchronization* brings two replicas into identical states (assuming that no other changes were made to either replica during the synchronization). Synchronization may be initiated by an application, perhaps upon some action by the end user, or by a system utility that awakens specified times or upon specified events, such as reestablishment of a network connection. Each Java object in a sync store belongs to an application-defined class. This class is required to implement an interface that includes several methods that the framework may invoke during synchronization, including class-specific methods for resolving conflicts between concurrent updates.

The MNCRS Data Synchronization Working Group produced a document describing the architecture of the framework, a Java API, and a tutorial for application programmers. These products reflect a consensus achieved after long deliberations by participants with different goals, experiences, and approaches. Compromises were necessary to reach this consensus, leaving every participant generally satisfied, but no participant completely satisfied, with the framework. Indeed, there were cases in which there was broad consensus in the working group that a problem existed, but no consensus about how to solve it, and the problem was knowingly left unsolved pending future revisions of the framework. Some of the participating companies developed their own implementations of the framework. We shall describe the framework agreed upon by the working group, some of the issues and alternatives debated by the working group in reaching this agreement, and one implementation of the framework, by a team at the IBM Thomas J. Watson Research Center. The framework and the implementation described here reflect the work of many people; the opinions presented here are the author's own.

This monograph describes version 1.1 of the MNCRS data-synchronization framework, published in March 1999. Section 2 gives examples of sync-store-based applications. Section 3

explains the framework's fundamental assumptions about the nature of sync stores and the effect of synchronization. Section 4 presents the architecture of the framework and the role of its major components, some of which are defined in the framework itself, some of which are defined implementors of the framework, and some of which are provided by application programs. Section 5 describes how applications using sync stores are written. Sections 6 and 7 discuss the IBM Research implementation of the MNCRS data-synchronization framework. Section 8 presents alternative approaches that were considered during the design of the framework, but not adapted. Section 9 reevaluates some of the design goals that shaped the framework, and describes follow-on projects.

2 Application Examples

A wide range of applications can be built on top of the MNCRS data-synchronization framework. This section provides a few illustrative examples.

2.1 A Personal Information Manager Database

A database of personal information comprises several sync stores, for example a sync store containing appointment-calendar entries and a sync store containing address-book entries. There are two replicas of each sync store, one in a handheld device and one in a personal computer. There is a separate program used to edit and view the data in each sync store on the personal computer. These programs take advantage of the keyboard, mouse, and powerful graphical interfaces available on the personal computer. There is a single program on the handheld device, tailored to the limited input and output mechanisms available on that device, that can be used to edit or modify any of the sync stores on that device. Pushing a button on the handheld device causes all the sync stores on that device to be synchronized with the corresponding replicas on the personal computer.

2.2 A Shared Address Book

A sync store containing contact information for customer prospects resides on a central server and has several additional replicas in mobile devices used by members of the sales force. A program on the mobile device allows a salesperson to view the database of prospects, to edit existing entries, to add new entries, and to exchange information with the central server. When commanded to exchange information with the central server, the program establishes a connection to the server and synchronizes its replica with the replica on the server.

2.3 A Decentralized Discussion Database

There are replicas of a sync store containing *newsgroup articles* in some unknown number of mobile devices. The authors of articles compose those articles on their mobile devices, specifying an expiration date for each article. When two users of such mobile devices meet, they may choose to synchronize their replicas. In this way, articles propagate through the community of users. The sync store has no complete central replica, but as a user synchronizes with more other users, his replica becomes a closer approximation to the set of all articles that have been written and have not yet expired.

2.4 Mobile Access to an Enterprise Database

A company uses a central server to maintain a large database. The company has mobile workers who each maintain a subset of the data in the database on their mobile devices, in the form of Java objects. Each mobile worker has a sync store on his mobile device containing all the objects in his subset of the database. Each such sync store has a replica in a gateway machine tied by local-area network to the server. Whenever data is modified in the server database, a server program propagates those changes to all the gateway replicas whose subsets include that data, and this information is propagated to the corresponding mobile replicas during synchronization.

Similarly, mobile workers may make modifications and additions to the sync stores on their local devices, and these are propagated to the corresponding gateway-based replicas during synchronization. When the contents of a gateway-based sync store changes, an attempt is made to update the server database accordingly. If this attempt fails, an error-message object is added to the sync store in the gateway, and propagated to the corresponding mobile replica at the next synchronization.

3 The Synchronization Model

A sync store is a collection of Java objects identified by sync IDs. Two sync stores may be synchronized. A complete synchronization causes the two sync stores to have identical contents. That is, the two sync stores map the same set of sync IDs to objects, and map equal sync IDs to objects having the same contents.

Two sync stores that are intended to be synchronized with each other are called *replicas*. Unlike Bayou, which requires a replica to be created as a copy of a specified existing data store [Pet97], the MNCRS data-synchronization framework allows any two arbitrary, independently created sync stores to be synchronized with each other. Such a synchronization merges the contents of the two stores. More typically, a replica will be created as a new, empty sync store, and initialized by synchronizing it with some existing replica.

Replicas are peers. There is no notion of a primary replica. Some replicas may reside on servers, but they are indistinguishable to the data-synchronization framework from replicas residing on client devices. Thus the framework allows two client devices to synchronize with each other, using precisely the same mechanisms that are used when a client synchronizes with a server.

Server data stores often hold far more data than can be replicated on a memory-constrained mobile client device. The user of a client device may be interested in only a subset of the objects in the server data store. Since synchronization replicates a sync store in its entirety, the MNCRS data-synchronization framework accommodates such scenarios with multiple sync stores on a server, each replicating a sync store on a different client device. Figure 1 illustrates two ways in which this can be done. In Figure 1(a), each client sync store has a corresponding mirror sync store on the server. A server application called an *update monitor* watches for changes in each of these mirror sync stores and in the master server data store. When it observes a change in the master data store, the update monitor applies the corresponding change to all applicable mirror sync stores. When it observes a change in some mirror sync store, the update monitor applies the change to the master data store (which may in turn cause it to be applied to other mirror sync stores). In Figure 1(b), sync stores on the server are implemented using a special implementation of the framework that stores their contents in some larger, shared persistent store. A change to an object in one sync store is immediately reflected in other sync stores that also contain that object.

3.1 Versions, Conflicts, and Reconciliation

If the object identified by a given sync ID is updated in more than one replica without any intervening synchronization, the updates *conflict*. During synchronization, a sync store may receive an update for some object that conflicts with some other update to the object already reflected in the receiving store. The receiving sync store *reconciles* the conflict by invoking a method of the object in question. The method examines the state of the object in the sync store and the state to which the received update would set the object, and sets the object to a state reflecting the appropriate resolution of the conflict. The class of the object, and thus the reconciliation method, are provided by the application. The reconciliation method reflects the semantics of the class. Reconciliation strategies include, for example, choosing one of the conflicting updates based on its age or the identity of the user who initiated it, or merging the two

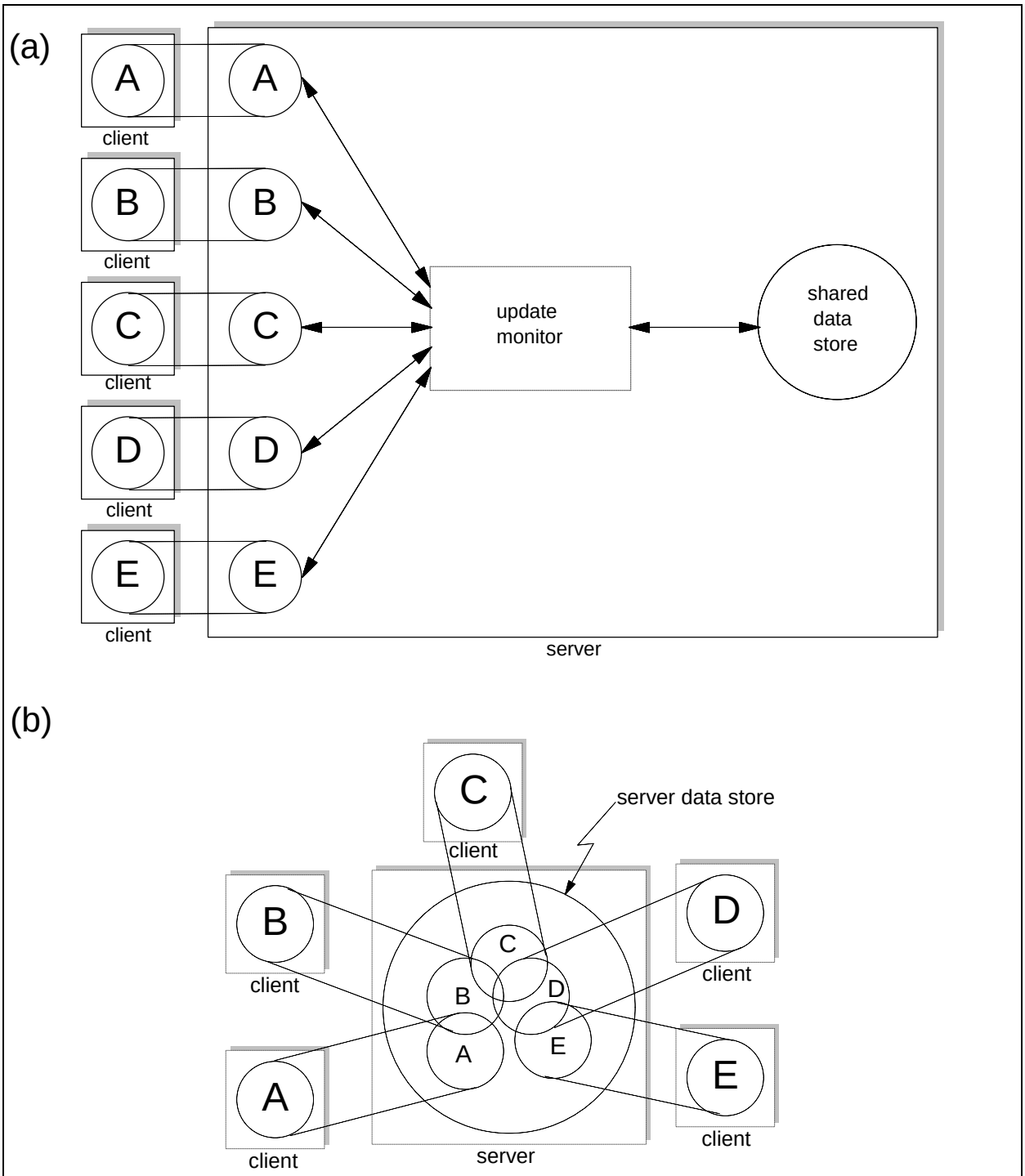

Figure 1. Replicating subsets of shared server data on client devices. (a) Maintaining a server-side mirror of each client sync store. The update monitor is an application that watches for updates in a mirror store and applies them to the master data store, and watches for updates in the master data store and applies them to the appropriate mirror sync stores on the server. (b) Implementing the server sync stores as subsets of some larger persistent store.

updates by taking maxima of corresponding numeric fields, or unions of corresponding set-valued fields.

During a synchronization with one replica, a sync store will pass on updates it received earlier from other replicas. Therefore, an update can be received from a sync store other than the one at which it was first applied. A sync store may receive updates to the same object from two different replicas during two different synchronizations, but these updates do not necessarily conflict. One update may have been applied at a sync store where the other update had already been known, in which case the intent of the later update was to supersede the earlier one. Consider the scenario in Figure 2. In Step 1, *x*, *y*, and *z* are given values in replica *A*. In Steps 2 and 3, the contents of replica *A* are propagated to replicas *B* and *C*. In Steps 4 and 5, *x* and *z* are updated at replica *B* while *y* and *z* are concurrently updated at replica *C*. In Step 6, the contents of replica *B* are propagated to replica *D*. In Step 7, the contents of replica *C*, including different values for each of *x*, *y*, and *z*, are propagated to replica *D*. At this point, replica *D* should ignore replica *C*'s version for *x* (in fact, the synchronization algorithm can ensure that it will never be sent), replace its value for *y* with *C*'s value (Y2), and invoke the method *z.reconcile*(Z3) to resolve the conflict between its current value for *z* and the value received from replica *C*.

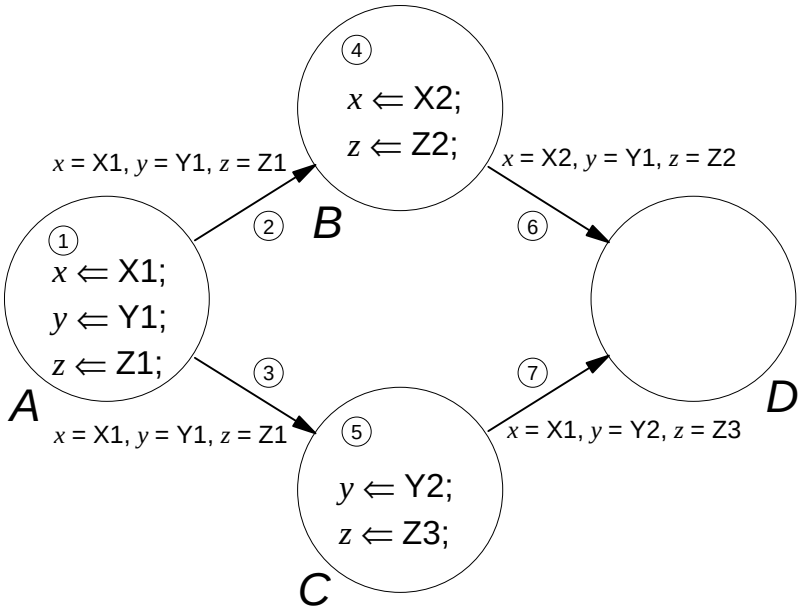

Figure 2. The propagation of superseding and conflicting updates. *A*, *B*, *C*, and *D* are replicas, and *x*, *y*, and *z* are objects stored in those replicas. Updates are applied at *A* and sent to *B* and *C* during synchronization, where further updates are applied. Then *B* and *C* both synchronize with *D*. The update to *x* that *D* receives from *B* supersedes the update to *x* that *D* receives from *C*. The update to *y* that *D* receives from *C* supersedes the update to *y* that *D* receives from *B*. The updates to *z* that *D* receives from *B* and *C* conflict.

Conceptually, every object in a store has an *object-state instance*, reflecting not only the state of the object, but a history of update actions. (Two objects may have identical states but

distinct object-state instances, resulting from different histories. Multiple applications of the same state transformation, even to identical states, are distinct update actions.) An object-state instance, *i*, *reflects* a set of updates. This set, *ReflectedBy*(*i*), is defined as follows:

- If *u* is the update creating a new object and *i* is the resulting object-state instance, *ReflectedBy*(*i*) = {*u*}.
- If an update is applied to object-state instance *i*, resulting in object-state instance *j*, *ReflectedBy*(*j*) = *ReflectedBy*(*i*)∪{*u*}.
- If *u* is an update applied to reconcile conflicting object-state instances *i* and *j*, resulting in object-state instance *k*, *ReflectedBy*(*k*) = *ReflectedBy*(*i*)∪*ReflectedBy*(*j*)∪{*u*}.

Two object-state instances conflict if and only if each reflects at least one update not reflected by the other. Object-state instance *i* supersedes object-state instance *j* if and only if *ReflectedBy*(*i*) is a proper superset of *ReflectedBy*(*j*).

Every set of update actions has a *version* associated with it, used to determine when one object-state instance conflicts with or supersedes another. There is a one-to-one correspondence between versions and sets of update actions. The version corresponding to set *S* is called *VersionForSet*(*S*). If *i* is an object-state-instance, it is sometimes convenient to speak of *VersionForSet*(*ReflectedBy*(*i*)) as the version associated with *i*, or with the update action that results in *i* (the most recent update action in *ReflectedBy*(*i*)). A sync store retains the version of the most recent update to each object it stores, and sends the version along with the update during synchronization. Versions are partially ordered by a relation *later than* obeying the following rule:

VersionForSet(S_2) later than *VersionForSet*(S_1) if and only if $S_1 \subsetneqq S_2$

Thus, an update supersedes another update to the same object if and only if its version is later than the version of the other. Two updates conflict if and only if they have versions neither of which is later than the other. In Figure 2, the update to *x* at *B* has a version later than that of the update to *x* at *A*, and the update to *y* at *C* has a version later than that of the update to *y* at *A*. The updates to *z* at *B* and *C* each have versions later than that of the update to *z* at *B*, but incomparable with each other by the *later than* relation.

Among the properties of the *later than* relation are the following:

- Whenever an application updates an object in a sync store, the update is assigned a version that is later than the version previously associated with the object.
- The reconciliation of a conflict between two updates is itself an update, with a version that is later than the versions of the two conflicting updates.
- For versions *A*, *B*, and *C*, if *A* is later than *B* and *B* is later than *C*, then *A* is later than *C*.
- Given the update sets u_1, ..., u_n, the set of versions S = {*VersionForSet*(u_1), ..., *VersionForSet*(u_n)} of update-set versions has an *earliest later bound*, a version *b* such that for all versions *v* in *S*, *b* is later than or equal to *v*, and any other version later than or equal to all versions in *S* is later than *b*; specifically, b = *VersionForSet*($u_1 \cup ... \cup u_n$).

3.2 Selection and Ordering of Transmitted Updates

During synchronization, the sender ought to send only those updates that are new to the receiver. To determine which updates to send, the sender first needs to obtain a succinct description of which updates are already reflected in the receiver. This description takes the form of a *summary version*, a version associated with the store as a whole and equal to *VersionForSet*(R), where R is the set of all updates that have been applied to the store. Each time a new update u is applied to the store, the summary version is replaced with earliest later bound of that update's version and the previous summary version. As noted at the end of Section 3.1, the earliest later bound of *VersionForSet*(R) and *VersionForSet*($\{u\}$) is *VersionForSet*($R \cup \{u\}$).

The sender transmits those current updates with versions later than or conflicting with the receiver's summary version. (An update to a given object at a given store is *current* if it is the most recent update applied to that object at that store.) Since none of the updates already reflected by the receiving store has a version later than or in conflict with the receiving store's summary version, but each of the transmitted updates does, none of the transmitted updates has already been reflected by the receiving store. Conversely, if a current update has not been reflected by a receiving store, that update's version is later than or in conflict with the receiving store's summary version, ensuring that any current update that has been applied to the sending store, but not to the receiving store, is in fact selected for transmission. (Suppose u is the update in question, R is the set of updates reflected by the receiving store, and the summary version, *VersionForSet*(R), is later than *VersionForSet*($\{u\}$). Then, by the definition of the *later than* relation, $\{u\} \subsetneq R$, but this contradicts the assumption that u has not yet been reflected in the receiving store.)

The framework requires that updates selected for transmission be transmitted in the sending store's *introduction order*—the order in which they were introduced to the sending store, either by an application running locally or by a previous synchronization session. Updates are applied to the receiving store in the order in which they were sent. (Thereafter, the received updates, or the reconciliations they trigger, now come last in the receiving store's introduction order.) It follows from induction on the number of updates that have been transmitted or applied throughout the distributed system that if any update originally performed at some replica A is reflected in replica B, then any updates performed earlier at replica A are also reflected in replica B. Following [Pet97], we call this the *prefix property*.

The prefix property allows sets of update actions, or equivalently their corresponding versions, to be represented succinctly as *version vectors* [Par83]. As Section 7.4 will explain in greater detail, a version vector specifies, for every replica, an integer indicating the last update originating at that replica that is a member of the set.

The prefix property also ensures that if the transmission of updates is interrupted, so that only those selected updates preceding a certain point in the introduction order are received, the distributed system remains in a normal state. As long as the receiving store advances its summary version each time it applies a remote update, it will always be the case that an update reflected by the sender is reflected by the receiver if and only if the receiver's summary version is not later than the version of the update. No special recovery measures are necessary after an interrupted transmission: A subsequent transmission can be undertaken in the usual manner, by selecting

those updates with versions later than or conflicting with the receiver's summary version. None of the updates that was successfully applied before the interruption needs to be retransmitted.

3.3 Synchronization Phases

A synchronization consists of some number of *phases*, each of which sends updates in one direction. Conflicting updates are detected and reconciled at the receiving sync store, and the result of the reconciliation is reflected in the state of the receiving store. A synchronization phase that completes without errors leaves the receiving store at least as up-to-date as the sending store was at the start of the phase.

A complete synchronization, leaving two sync stores *A* and *B* with identical contents, can be achieved by a phase sending updates from *A* to *B* followed by a phase sending updates from *B* to *A*. The results of reconciling conflicts at *B* during the first phase are sent back to *A* during the second phase (along with any nonconflicting updates that were performed at *B* before synchronization started). A set of *n* sync stores can be completely synchronized by arranging the sync stores in a ring and performing a sequence of 2*n*-2 one-phase synchronizations that propagate updates from one sync store in the ring to the next, until each replica has received updates from all other replicas, as shown in Figure 3. One-phase synchronizations are also useful in applications where information is known to flow in one direction, for example if a sync store contains a price list that is updated each day at a server, and read but never updated on client devices.

Synchronizations of more than two phases can be used to delegate responsibility for conflict reconciliation, as in the following scenario, depicted in Figure 4: In the first phase, a sync store residing on a client sends updates to a sync store residing on the server. If the server sync store detects conflicting updates to a particular object, it "reconciles" the conflict by placing the object in a special state that indicates that conflicting updates have occurred, and that contains descriptions of the conflicting updates. During the second phase, the object in this special state is sent back to the client device and updated in the client sync store. The client sync store responds to an update placing an object in such a state by initiating a dialog with the user to determine the appropriate way to resolve the conflict, and putting the object in a normal state that reflects the user's wishes. During the third phase, the object in its normal state is sent back to the server sync store, leaving the two sync stores with the same contents.

In each of these examples, the phases of a synchronization occur in sequence. However, nothing in the framework precludes a synchronization in which phases proceed concurrently.

A synchronization is initiated by one of its participants, called the *requester*. The other participant in the synchronization is called the *responder*. The roles of requester and responder are orthogonal to the roles of sender and receiver: In a "push-pull" synchronization, the requester will send updates to the responder during the first phase and receive updates from the responder during the second phase, and conflicts will be reconciled by the responder. In a "pull-push" synchronization, the requester will ask the responder to send it updates during the first phase, and will send its own updates back to the responder in the second phase, with conflicts resolved by the requester. A device capable of acting as a responder is called a *synchronization server*. Depending on the underlying transport, a synchronization server may, for example, listen at a

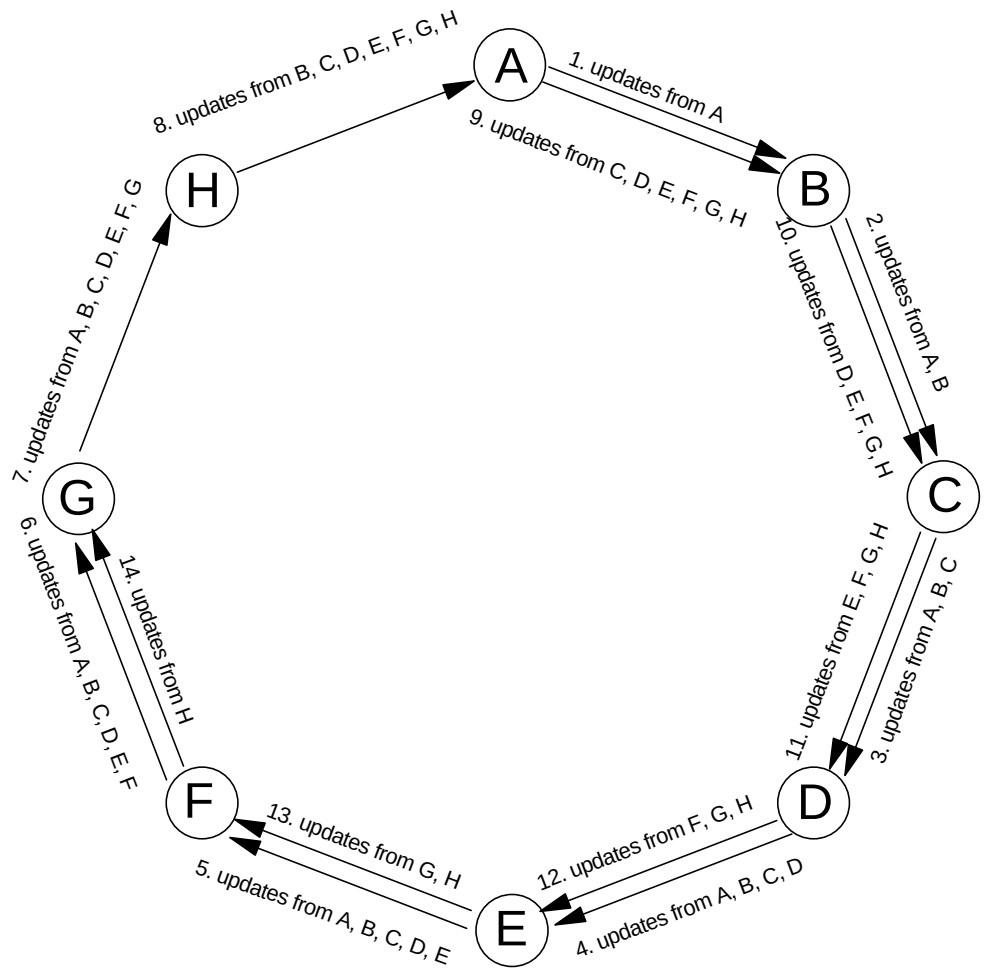

Figure 3. The propagation of updates around a ring of sync stores. We can synchronize n=8 sync stores through a sequence of $2n$-2=14 one-way propagations, leaving all the sync stores in identical states if there are no intervening updates. Once an update is propagated from one sync store to another, it will not be propagated again the next time that sender propagates updates to that receiver.

well-known TCP/IP port for an incoming synchronization request, check an in-box for SMTP messages requesting synchronization, or check message queues for queued synchronization requests. Every sync store is identified by a uniform resource locator (URL). An application running on the requester initiates synchronization by specifying the URL of the remote replica. This URL identifies the protocol to be used, the responder's host name, and a name distinguishing the remote replica from all other sync stores on the responding host.

3.4 Consistency Properties

A complete MNCRS synchronization establishes *mutual consistency* between two replicas of a sync store. That is, the contents of the two replicas correspond exactly. Repeated

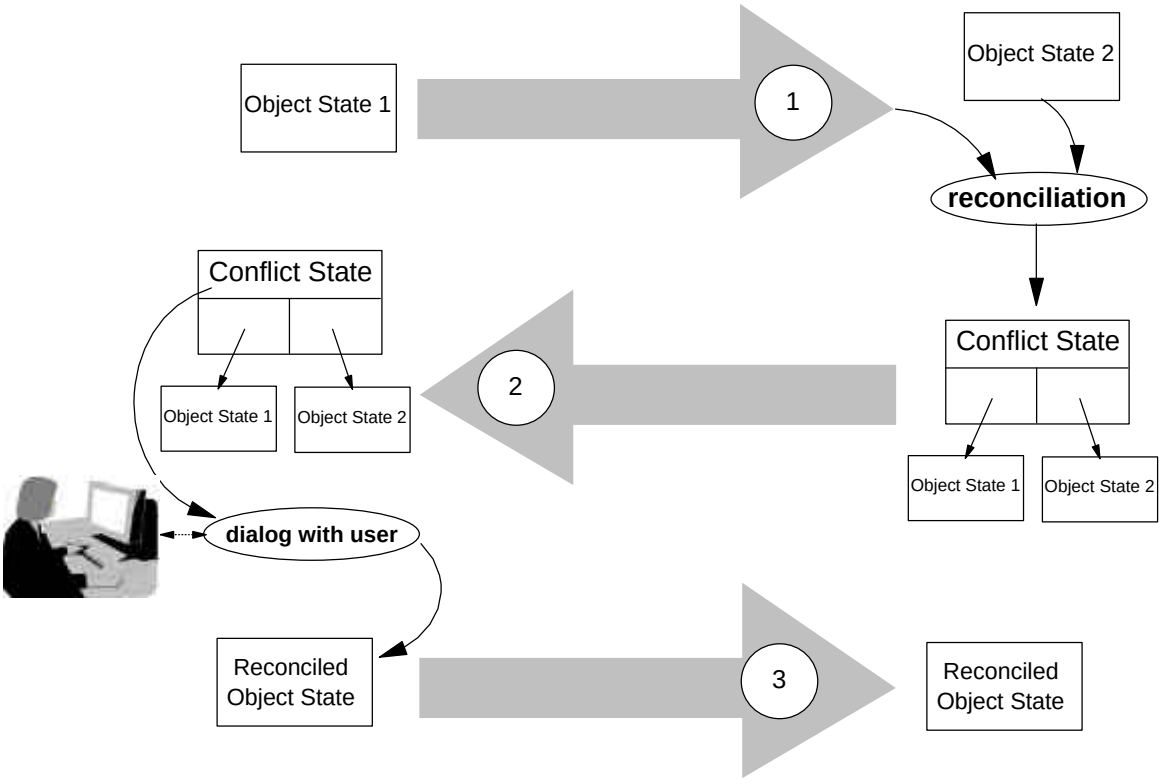

Figure 8. Delegating responsibility for conflict resolution with a three-phase synchronization. As far as the data-synchronization framework is concerned, conflict detection and reconciliation take place during Phase 1; the application-specified reconciliation is to place the object in a new state that indicates a conflict and includes references to the two object states that were found to be in conflict. During Phase 2, application code invoked each time an object is modified checks for objects that have been placed in the conflict state; each time such an object is found, the application code starts a dialog with a user who manually reconciles the two conflicting object states; the application code places the object in this reconciled state. All objects modified in this way during Phase 2 are propagated back to the other replica in Phase 3.

synchronization, propagating updates to all replicas in the network, achieves eventual consistency among these replicas.

However, Davidson, Garcia-Molina, and Skeen [Dav85] observe that mutual consistency is neither a necessary nor a sufficient condition for disconnected replicas to be in what we would intuitively consider a "correct" state: Consider replicated copies of a checking-account balance, each inaccessible to the other. If the two copies are initially identical and a withdrawal transaction for $100 is executed at each replica, each copy will hold the same incorrect value afterward. If $100 is withdrawn from only one replica, the two copies will have different values, but the distributed database may be in a correct state if suitable precautions (for example, pessimistic locking) are in place to protect against conflicting withdrawals, and if the system can recognize the replica at which the withdrawal took place as more up-to-date.

According to [Dav85], a more intuitive notion of "correctness" is based on serializable execution of atomic *transactions*, possibly subject to integrity constraints. For a replicated data

base, a common criterion for correctness is *one-copy serializability*, meaning that the concurrent execution of transactions on replicated data is equivalent to some serial execution of the same transactions on nonreplicated data. Nonetheless, among the approaches described in [Dav85] are two that depart from one-copy serializability. The first approach, *weak consistency*, allows for nonserializable read-only transactions. (For example, consider two mutually inaccessible replicas containing items *a* and *b*. At one replica, *a* is updated and then a read-only transaction is executed whose outcome depends on the values of both *a* and *b*. At the other replica, *b* is updated and then another read-only transaction whose outcome depends on *a* and *b* is executed. There is no serialization of the four transactions that is consistent with the outcomes of the two read-only transactions.) The second approach, *data patch*, uses *integration programs*, analogous to MNCRS reconciliation methods, to resolve inconsistencies between disconnected replicas in a manner consistent with any external effects that have already been observed by users. (These already observed effects might not correspond to any serial execution.)

However, the MNCRS data-synchronization framework has no machinery to enforce stronger forms of consistency. A complete synchronization phase will preserve causality, but an interrupted synchronization session can leave the contents of the store in a state inconsistent with the order in which updates were performed: The object states resulting from current updates are transmitted in introduction order, but once an update to an object is superseded, the state resulting from that update is not retained. If an object has been updated twice since the last synchronization, once before the last successfully received update and once afterward, only the state resulting from the second update to that object will be recorded in the sending sync store, so no indication of the first update will be transmitted. We call this the *missing-update anomaly*. An example is depicted in Figure 5.

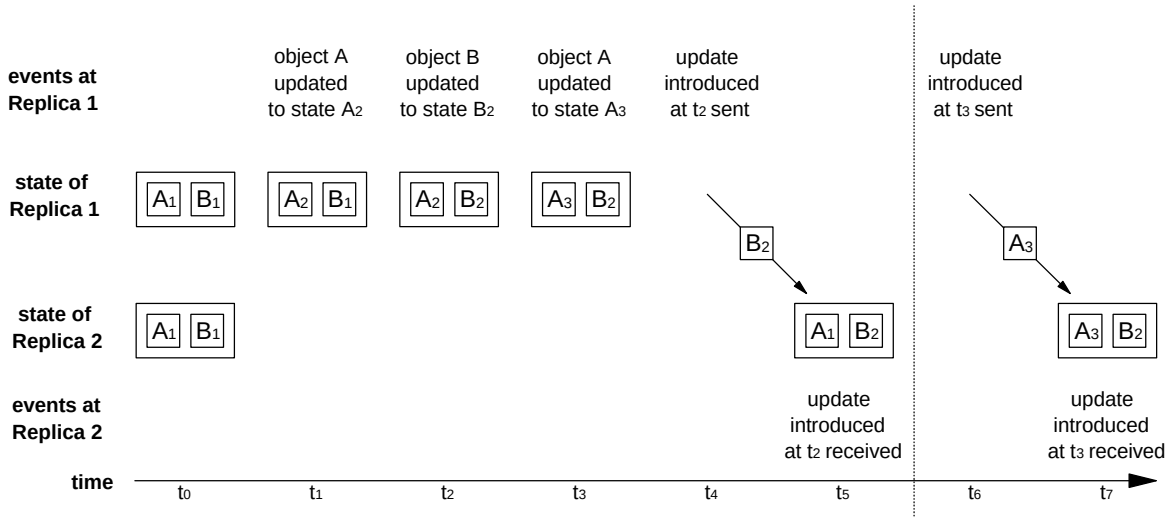

Figure 5. An example of the missing-update anomaly. When synchronization begins at time t_4, the update introduced at t_1 is no longer the current update to object *A*, so the first current update in introduction order is the update to object *B* at time t_2. If the synchronization phase completes normally, Replica 2 attains the same state as Replica 1 at time t_7. However, if the transmission of updates is interrupted after t_5 but before t_7, the state of Replica 2 will reflect the update to *B* at time t_3, but not the update to *A* made earlier, at time t_2.

An implementation can protect against the missing-update anomaly by buffering incoming updates and not applying any of them until all updates have been received. However, the framework allows implementations to apply updates one at a time, as they are received, to accommodate practical constraints of mobile devices. First, buffering may be impractical on memory-constrained devices. Second, since the receiving store's summary version at the start of the next transmission will not reflect updates that were received and buffered, but never applied, those updates will have to be retransmitted; all work achieved before the interruption will be lost. Since wireless communication links are often unreliable, it is important to be able to make incremental progress even if the link is lost before synchronization is complete. Section 9.1.4 will propose an alternative approach for coping with the missing-update anomaly.

The design of the framework anticipates transactional extensions. It specifies that during synchronization, synchronizers exchange objects implementing an interface named `SyncUpdate`, and that each such object specifies a set of updates to be applied atomically. The framework provides no methods for grouping operations into a single `SyncUpdate` object, but individual implementations of the framework may provide such methods as nonstandard extensions.

4 Architecture of the Framework

The fundamental components of the MNCRS data-synchronization framework are sync stores and *synchronizers*. A sync store is a persistent store containing Java objects that implement an interface named `Reconcilable`. Each `Reconcilable` object in a sync store is identified by an object implementing an interface named `SyncId`. A synchronizer is a component that obtains updates from a local sync store, exchanges updates with a synchronizer on another device, and applies remote updates. A synchronizer is an active object, with its own thread. It may be activated by an application program or by a daemon. Examples of daemons are one that initiates synchronization when the establishment of a communications link is detected, one that initiates synchronization at certain intervals or times of the day, and one that initiates synchronization when the user of a mobile device pushes a "Synchronize" button on the device. In contrast to a synchronizer, a sync store is a passive object, acted upon by an application program or by a synchronizer that is synchronizing it with a remote replica. See Figure 6.

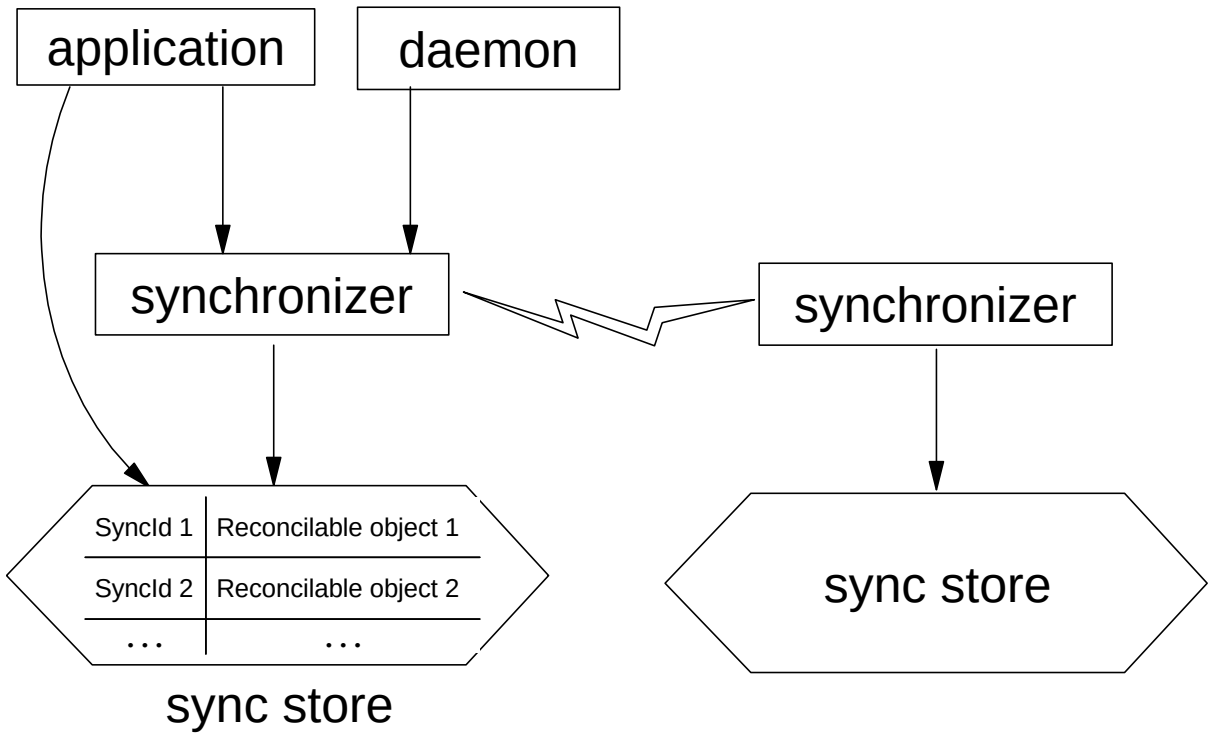

Figure 6. The architecture of the MNCRS data-synchronization framework. A sync store is a passive object providing a persistent object store. A synchronizer is an active object that manipulates sync stores and communicates with remote synchronizers. Application programs may manipulate sync stores and activate synchronizers. Synchronizers may also be activated by daemons.

The MNCRS data-synchronization framework does not include classes implementing sync stores. Rather, the framework includes two interfaces, `SyncStore` and `SyncStoreUpdater`, that declare methods for manipulating a sync store. The `SyncStore` interface declares those methods used by an application program. `SyncStoreUpdater` extends `SyncStore` with methods used by a synchronizer. The vendor of an MNCRS device is expected to provide a class that implements

SyncStoreUpdater (and hence SyncStore) in a manner appropriate to the device, exploiting knowledge of the device's file system or other persistent storage, for example. Classes implementing SyncStoreUpdater might also be provided by independent middleware vendors. Particular classes might implement value-added extensions of the SyncStoreUpdater interface, with additional features, such as the bundling of updates into transactions, sophisticated indexing and query facilities, or the implementation of a sync store as a subset of a larger persistent store, as depicted in Figure 1(b). Of course an application using such extensions is not portable to other implementations of the framework.

Similarly, rather than classes implementing synchronizers, the framework provides an interface named Synchronizer. Different classes implementing this interface handle different underlying transports, such as sockets, messaging services, or e-mail, and different protocols for the dialog that takes place during synchronization. Classes implementing Synchronizer might be provided by manufacturers of MNCRS devices, manufacturers of communications equipment, service providers, middleware providers, consortia, and standards bodies, for example.

The framework includes a class named StoreManager responsible for the administration of sync stores on the local device. This class keeps track of the sync stores that exist on the device and their locations in the device's persistent storage. The class has static methods for creating new sync stores or opening existing sync stores so that their methods may be invoked, for listing all sync stores on the device, and for removing a sync store from the device.

The data-synchronization framework comprises three categories of Java types:

- **Fixed classes.** These include both concrete and abstract classes. The framework specifies only the APIs, not implementations. An implementation of the framework includes definitions for each of these classes. The StoreManager class falls into this category.
- **Interfaces for pluggable infrastructure components.** The SyncStoreUpdater and Synchronizer interfaces fall into this category. An implementation of the framework includes at least one implementation of each of these interfaces, but the implementation of the framework should also be able to work with independently developed classes implementing these interfaces.
- **Interfaces and abstract superclasses for pluggable application components.** The Reconcilable and SyncId interfaces are examples of interfaces in this category. Concrete classes implementing these interfaces are provided by application writers. The framework includes abstract classes partially implementing certain of these interfaces with default behaviors. An application writer has the option of implementing a pluggable component by extending one of these abstract classes and inheriting the implementations of certain methods. An implementation of the framework includes implementation of the concrete methods of these classes.

In addition, an implementation will include internal types that are not specified in the framework.

The framework was carefully designed to allow the implementation of the StoreManager class, pluggable implementations of sync stores, and pluggable implementations of synchronizers to be developed independently. The framework includes a cluster of classes and interfaces whose implementations can be viewed as part of the implementation of sync stores, and another cluster

of classes and interfaces whose implementations can be viewed as part of the implementation of synchronizers. The framework specifications enable efficient implementations in one cluster to be built without *a priori* knowledge of implementations in the other cluster. (An implementor of the framework could also choose to tightly integrate implementations in both clusters, trading off the ability to interchange components for premium performance.) Furthermore, the framework specifications enable the `StoreManager` class to be implemented without any *a priori* knowledge of sync-store implementations, so that new, possibly independently developed, sync-store implementations can be added to the device later. Conversely, it is possible to implement sync stores without any *a priori* knowledge of the implementation of the `StoreManager` class, so that independent sync-store implementors can develop sync-store implementations that can be plugged into any implementation of the framework. Of course both clusters can be implemented without any knowledge of how an application implements the `Reconcilable` and `SyncId` interfaces, and an application can be written without any knowledge of how classes and interfaces in the framework are implemented.

The implementation of the `StoreManager` class is necessarily device-specific, because it must understand the device's persistent-storage medium. Unfortunately, implementations of sync stores must also read from and write to persistent storage. Had the framework included an interface for a *persistent-storage manager*, it would have been possible for implementations of sync stores to access persistent storage through the methods of this interface, avoiding dependence on any particular form of persistent storage. An implementation of the framework would include one or more platform-specific implementations of the persistent-storage-manager interface, for particular file systems, database management systems, nonvolatile memories, or other persistent-storage mechanisms. Indeed, our implementation of the framework includes such an interface, to decouple implementation of sync stores from particular persistent-storage mechanisms, making the sync-store implementation reusable on any device. Implementations of the framework by other consortium members include similar interfaces, with small differences. Unfortunately, the MNCRS data-synchronization working group was unable to agree on a common, standard definition for the persistent-store-manager interface, so no such interface is defined in version 1.1 of the data-synchronization framework.

Figure 7 shows the relationships among the interfaces and classes defined by the framework, by framework implementations, and by application programs. The remainder of this section discusses in greater detail the interfaces and classes associated with applications, sync stores, and synchronizers.

4.1 Interfaces and Classes Associated with Applications

An object to be stored in a sync store belongs to a class that is part of an application, for example a class representing employees, customers, or medical records. The application programmer declares this class to implement the `Reconcilable` interface, which allows it to be passed to the `SyncStore` method for inserting new objects in a store. This declaration obligates the programmer to implement the methods declared in the `Reconcilable` interface.

Among the methods of the `Reconcilable` interface are a method to reconcile update-update conflicts, a method to reconcile update-delete conflicts, and a method to set the contents of an existing object to those of another object of the same class. The framework

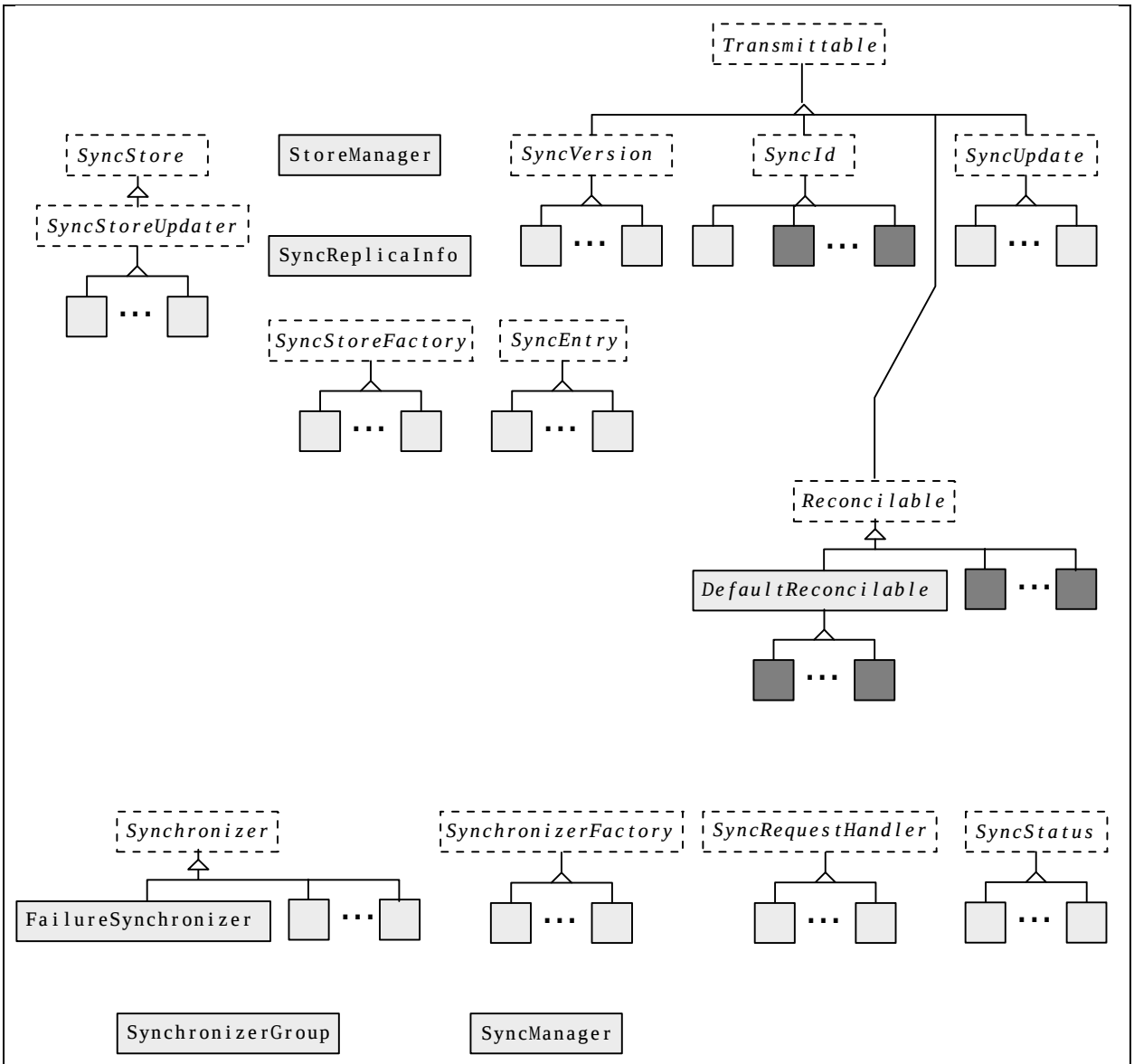

Figure 7. Interfaces and classes in the MNCRS data-synchronization framework. Boxes with dashed line and italic letters represent interfaces defined in the framework. Boxes with solid lines and italic letters represent abstract classes named in the framework. Boxes with solid lines and roman letters represent concrete classes defined in the framework. Solid boxes with no text represent implementation classes not defined in the framework. Boxes shaded with dots represent classes that are part of the framework implementation. Boxes shaded with diagonal lines represent classes that are part of an application.

invokes these methods as necessary during synchronization. Section 5 will discuss these methods in greater detail.

In an earlier version of the framework, the `Reconcilable` interface extended the `Serializable` interface of the standard Java library package `java.io`. Thus the framework could invoke the `java.io.ObjectOutputStream` method `writeObject` to write a byte-stream representation of a `Reconcilable` object for transmission during synchronization or storage in the local persistent store, and the `java.io.InputStream` method `readObject` to reconstruct a

Reconcilable object from a byte-stream representation received during synchronization or read from the local persistent store. A programmer could control the byte-stream representation by giving the Reconcilable class private writeObject and readObject methods with object-stream parameters, or by declaring the class to implement the interface java.io.Externalizable and overriding its writeExternal and readExternal methods. However, a programmer willing to accept the default serialization behavior would obtain that behavior automatically, by doing nothing.

The mandatory use of Java serialization was later abandoned because of concerns about efficiency and flexibility. The primary efficiency concern was that the classes that must be loaded to perform serialization (which, in turn depends on the Java reflection API) are too large for some memory-constrained mobile devices. There was also concern about execution time and the verbosity of the default byte-stream representations (a vital concern when the byte-stream representation is to be transmitted over a slow or expensive communications link). The flexibility concern was that some applications would require distinct byte-stream representations for transmission during synchronization and for persistent storage. Some applications require Reconcilable objects with both synchronized and unsynchronized fields, the unsynchronized fields containing data of only local significance that is stored persistently but not exchanged during synchronization. Some applications might require a particular representation during synchronization to facilitate interoperability with other synchronization systems. Some applications might require a particular representation in persistent storage for a sync store implemented on top of an existing persistent store with contents in a predetermined format.

An earlier draft of the framework included two interfaces, named Transmittable and Persistable, to address these problems. The Transmittable interface declared two methods to be invoked when transporting objects. These methods, named writeRemote and readRemote, were modeled after the writeExternal and readExternal methods of java.io.Externalizable. The Persistable interface declared two analogous methods to be invoked when accessing the persistent store, writePersistent and readPersistent. Types for objects transmitted during synchronization were declared to extend or implement Transmittable and types for objects stored persistently were declared to extend or implement Persistable. (Typically, a type falling into one of these categories would also fall into the other, so that the type would implement both of these interfaces.) The Reconcilable interface was declared to extend both these interfaces, thus obligating the programmer to implement writeRemote, readRemote, writePersistent, and readPersistent methods for all Reconcilable classes. To avoid burdening the programmer willing to accept Java's default serialized representations for both transmission and persistent storage, a new abstract class, DefaultReconcilable, was included in the framework. DefaultReconcilable is a partial implementation of the Reconcilable interface, implementing writeRemote, readRemote, writePersistent, and readPersistent by invoking default serialization, but leaving the other methods of the Reconcilable interface abstract. An application class that extends DefaultReconcilable can inherit the serialization-based implementation of the four byte-stream-representation methods.

Objections were later raised to this approach on the grounds that the stream-oriented methods writePersistent and readPersistent, while appropriate if the underlying persistent store is a file system, are inappropriate if the store is an object or relational database. For such databases, it would make more sense to pass an entire object to a writing method than to write a

sequence of bytes to a stream. In fact, the way in which an object of a particular application class should be written to a particular kind of persistent store depends in part on the application class and in part on the persistent store. Unable to agree on a solution to this problem, the MNCRS data-synchronization working group decided to remove the `Persistable` interface from the framework. In version 1.1 of the framework, all types that had previously extended or implemented `Persistable` extend or implement `Transmittable` instead. Sync-store implementations for which stream-oriented persistent-store operations make sense use the `Transmittable` methods for writing and reading persistent storage as well. However, framework version 1.1 provides no standard way for an application to control persistent-storage representations on other sync-store implementations. A solution to this untenable situation is proposed in Section 8.1.

The `SyncStore` interface includes methods that generate sync ID values automatically. These values are objects of some class, internal to the sync-store implementation, that implements the `SyncId` interface. An application can also provide its own classes implementing the `SyncId` interface, and insert objects in a sync-store with specified sync IDs that correspond to natural application keys. Sync IDs are both stored persistently and transmitted during synchronization, so the `SyncId` interface is declared to extend `Transmittable`. In addition, the `SyncId` interface overrides the `hashCode` and `equals` methods of class `java.lang.Object` with abstract methods. Thus, an application writer implementing the `SyncId` interface must implement `writeRemote`, `readRemote`, `hashCode`, and `equals` methods. The `equals` method should report that two objects of the class are equal if and only if they represent the same key, so that a given sync ID will identify the same sync entry each time it is reconstructed from its byte-stream representation. Two objects that are reported equal should have the same hash code. The framework does not provide a `DefaultSyncId` class analogous to `DefaultReconcilable`, but such a class would be convenient for application programmers willing to use default serialization for sync IDs.

4.2 Interfaces and Classes Associated with Sync Stores

Section 3.1 described the versions associated with each update to a `Reconcilable` object. A version is represented by an object of some class, internal to the implementation of sync stores, that implements an interface named `SyncVersion`. The `SyncVersion` interface declares methods to compare two versions and determine whether one is later than the other. The framework uses `SyncVersion` objects for internal bookkeeping and for communication between sync stores and synchronizers; they are not seen or manipulated directly by the application programmer.

Updates themselves are represented by objects of some class, internal to the implementation of sync stores, that implements an interface named `SyncUpdate`. The `SyncUpdate` interface is empty, but is used to specify certain methods in the `SyncStoreUpdater` interface that are invoked by synchronizers. These include a method to extract pending updates from a sync store so that they can be sent to a remote synchronizer and a method requesting a sync store to apply to itself an update that has been received from a remote synchronizer. The application programmer does not see or manipulate `SyncUpdate` objects directly.

A sync store does not map a sync ID directly to a `Reconcilable` object, but rather to a *sync entry*. A sync ID and a `Reconcilable` object are associated with each sync entry. A sync entry is represented by an object of some class, internal to the implementation of sync stores, that

implements an interface named SyncEntry. The SyncEntry interface declares methods to retrieve the associated sync ID and Reconcilable object.

A classic problem in replicated databases, pointed out by Fischer and Michael [Fis82], is that the presence of an item in Replica *A* and its absence from Replica *B* can mean either that the item was recently inserted in *A* and news of the insertion has not yet reached *B*, or that the item was recently deleted from *B* and news of the deletion has not yet reached *A*. Ratner, Popek, and Reiher [Rat96] call this *the create/delete ambiguity*. The MNCRS data-synchronization framework resolves the create/delete ambiguity by retaining the sync entry for an object when the object is deleted from the sync store, and recording the deletion in the sync entry. The SyncEntry interface includes a method indicating whether a given sync entry corresponds to a deletion. Of course deletion sync entries cannot be allowed to accumulate forever, especially on a memory-constrained mobile device, so a deletion sync entry should be removed once news of the deletion has reached every replica. Section 7.5 discusses the difficult problem of determining when a given deletion sync entry may be removed from the sync store.

As Section 3.1 explained, a sync store retains the version associated with the most recent update to each object it contains. This version is most naturally stored in a sync entry. However, the SyncEntry interface does not include methods to set or retrieve this version. SyncEntry objects are accessible to the application programmer, but SyncVersion objects should not be. Therefore, any such methods, and perhaps additional methods such as one marking a sync entry as a deletion entry, are declared in the class, internal to the sync-store implementation, that implements the SyncEntry interface.

The SyncStore interface provides a method to retrieve the sync entry associated with a given sync ID and a method to retrieve an iterator over all sync entries in the sync store, including deletion entries. Iterating over all objects in the sync store entails iterating over all sync entries, testing whether each one is a deletion entry, and retrieving the Reconcilable object associated with each nondeletion entry. In retrospect, the application writer's view of the framework is unnecessarily complicated by the inclusion of deletion sync entries in the iterator, and the provision of a method to test whether a given sync entry corresponds to a deletion. There is no reason an application programmer needs to be aware of the existence of deletion entries. In fact, it would have been possible to keep the application writer oblivious to *all* sync entries, as was done with versions, thus simplifying the framework. Since the SyncStore interface already provides a method to retrieve the Reconcilable object associated with a given sync ID, the capabilities provided by iterating through sync entries could have been provided instead by a method returning an iterator over all sync IDs corresponding to objects in the sync store.

The StoreManager class uses the *abstract factory* design pattern [Gam95] to create new sync stores. The framework includes an interface named SyncStoreFactory. This interface declares a single method, which attempts to create a SyncStore object consistent with a set of attributes specified in the method call. Each class implementing SyncStoreUpdater is accompanied by a corresponding class implementing SyncStoreFactory. A system property defined on the command line invoking the Java virtual machine specifies the names of all classes implementing SyncStoreFactory that are installed on the device. To create a new sync store, the StoreManager method open invokes an instance of each of these classes in turn, until one

factory succeeds in constructing an object implementing the SyncStore interface. If none of the factories succeeds, the open method throws an exception.

4.3 Interfaces and Classes Associated with Synchronizers

A synchronizer is constructed to perform a single synchronization between a particular local sync store and a particular remote replica. The local sync store is specified by an object implementing the SyncStore interface and the remote replica is specified by an object of a class named SyncReplicaInfo. A SyncReplicaInfo object specifies the URL of a remote replica and the default schedule of phases to be performed when synchronizing with that replica. An application can construct a SyncReplicaInfo object with a given URL and default phase schedule, and can change the default phase schedule.

A synchronizer can be activated to perform a synchronization consisting only of a sending phase, a synchronization consisting only of a receiving phase, or a synchronization following the phase schedule of its SyncReplicaInfo object. In addition, an active synchronizer can be requested to abort its synchronization as soon as possible. (Depending on the protocol, aborting a synchronization might not have any effect on the remote processing of updates that have already been sent.) The methods to start a synchronizer return immediately, allowing the synchronization to proceed concurrently with the calling thread. Likewise, the method requesting a synchronizer to abort its synchronization returns immediately. The Synchronizer interface also has a method, waitUntilDone, that blocks until the synchronization has ended, either successfully or unsuccessfully, or optionally until a specified time-out period has elapsed.

Sometimes there is a need for a program—particularly a daemon responsible for synchronizing all the sync stores on a device—to start, stop, or wait for a *group* of synchronizers together. A class named SynchronizerGroup, representing a group of synchronizers, provides methods to activate each synchronizer in the group according to that synchronizer's default phase schedule, to request all synchronizers in the group to stop, and to wait until all synchronizers in the group have stopped or a specified amount of time has elapsed.

An application program does not construct a Synchronizer or SynchronizerGroup object directly, but uses one of the following facilities:

- The SyncStore interface provides the application with methods that synchronize a given sync store with one or more replicas and return once the synchronization has succeeded, failed, or, optionally, exceeded a time limit. Behind the scenes, these methods create, activate, and wait for Synchronizer or SynchronizerGroup objects.
- Static methods of a class named SyncManager provide applications with more intricate control over synchronization, such as the ability to perform other activities while synchronization proceeds concurrently. These SyncManager methods return SynchronizerGroup objects whose synchronizers are constructed to synchronize particular sync stores with particular replicas. The caller can then manipulate the SynchronizerGroup object directly.

For even finer control, SynchronizerGroup has a method returning an array of the synchronizers in the group, allowing synchronizers to be controlled individually.

The SyncStore synchronization methods and the SyncManager class construct synchronizers using an abstract-factory pattern similar to that used by StoreManager to construct sync stores. An interface named SynchronizerFactory has a method that attempts to create a synchronizer appropriate for synchronizing a specified sync store with a specified remote replica, returning a nonnull Synchronizer result if it succeeds and returning null if it fails. For each class *C* implementing the Synchronizer interface, there some class implementing the SynchronizerFactory interface and constructing objects of class *C*. A system property specifies the names of all classes implementing SynchronizerFactory that are installed on the device. A new synchronizer is constructed by invoking an instance of each of these classes in turn, until one succeeds. If none succeed, an object of a class named FailureSynchronizer, which is defined in the framework and implements the Synchronizer interface, is returned. Any attempt to activate a FailureSynchronizer object immediately fails, and sets the corresponding SyncStatus object to reflect this failure.

On a synchronization server, an object called a *synchronization request handler* continuously listens for an incoming synchronization request, obtains a synchronizer to handle each request, and invokes the synchronizer. A synchronization request handler belongs to some class implementing an interface named SyncRequestHandler. This interface provides methods to start listening for incoming requests and to stop listening. Different implementations of the SyncRequestHandler interface handle different kinds of transport. For example, one implementation might listen at a well-known TCP/IP port for a socket connection request, a second might monitor a message queue for incoming messages, and a third might periodically check an e-mail in-box.

The processing of a synchronization protocol may be shared by a synchronizer factory, a synchronization request handler, and synchronizers. On the requesting side, a synchronization factory may make the initial attempt to contact the responder. If successful, it may engage in preliminary handshaking to determine whether the responder is capable of communicating with the kind of synchronizer this factory creates. If so, the synchronizer factory will construct such a synchronizer and activate it to process the remainder of the protocol. On the responding side, a synchronization request handler may engage in preliminary handshaking to indicate whether or not the kind of synchronizer it constructs is capable of handling the incoming request. If so, the synchronization request handler will construct such a synchronizer and activate it to handle the remainder of the protocol. The appropriate division of labor depends on the protocol and on the design of the three cooperating classes implementing SynchronizerFactory, SyncRequestHandler, and Synchronizer. These three classes will typically be implemented together. The point in the protocol at which the requester's synchronizer takes over from the synchronizer factory that created it may or may not be the point at which the responder’s synchronizer takes over from the synchronization request handler that created it.

An interface named SyncStatus provides methods for querying the progress that a synchronization has made. There is a SyncStatus object associated with every synchronizer at the time the synchronizer is activated, and a method in the Synchronizer interface returning a reference to this object, or null if the synchronizer has not yet been activated. The SyncStatus interface does not include any methods for setting the status. A class implementing the interface must provide such methods. A class implementing the Synchronizer interface constructs a SyncStatus object of a known class, so it can invoke that object’s status-setting methods.

4.4 Package Structure of the Framework

The MNCRS data-synchronization framework consists of three packages, `org.mncrs.datasync`, `org.mncrs.datasync.sync`, and `org.mncrs.datasync.event`. As Section 5 will explain more fully, the framework uses the JavaBeans event model [Ham97] to inform the application of certain occurrences. A number of types related to events, including event-object classes, listener interfaces, and event-adapter classes, are packaged together in `org.mncrs.datasync.event`. The package `org.mncrs.datasync` contains the remaining types needed to write application programs, including `SyncStore`, `Reconcilable`, `Transmittable`, `DefaultReconcilable`, `SyncId`, `SyncEntry`, `StoreManager`, `SyncReplicaInfo`, `SyncStatus`, and several exception classes. The package `org.mncrs.datasync.sync` contains those types that an application programmer should not know about, but writers of pluggable framework components may have to know about, including `SyncStoreUpdater`, `SyncVersion`, `SyncUpdate`, `SyncStoreFactory`, `SynchronizerFactory`, `SyncRequestHandler`, and `FailureSynchronizer`. Three types, `Synchronizer`, `SynchronizerGroup`, and `SyncManager`, are not needed by an application writer willing to use the straightforward methods of the `SyncStore` interface to perform synchronization, but may be needed by more sophisticated application writers. These types have been placed in `org.mncrs.datasync.sync` to keep the `org.mncrs.datasync` package as simple as possible for elementary applications. Those types that are part of the framework implementation, but not named in the framework, belong to other packages, determined by the implementation order.

5 The Application Programming Model

5.1 Querying and Modifying the State of a Sync Store

A sync store is an associative store. `Reconcilable` objects can be placed in the store in association with some sync ID, and retrieved using that sync ID. The `SyncStore` interface includes methods named `put`, `get`, `delete`, and `size`. The `put` method places objects in a sync store and the `get` method retrieves the object associated with a given sync ID. The `delete` method removes an object from the sync store. The `size` method returns the number of objects currently in the sync store.

The `put` method establishes an association between a particular sync ID and a particular `Reconcilable` object. This association can only be broken by calling `delete`. Until then, as long as the sync store remains open, the contents of the object associated with a given sync ID may change from time to time, but the object associated with that sync ID will always be the same object. (If objects of different classes, say `Ellipse` and `Rectangle`, are to be associated with the same sync ID at different times, an extra level of indirection is required. A "referring object" with a member of some common supertype, say `Shape`, can be placed in the sync store. A given sync ID will always be associated with the same referring object, but the referring object may contain a reference to an `Ellipse` object now and a reference to a `Rectangle` object later.) There are two versions of the `put` method. One takes a sync ID and a `Reconcilable` object and associates the specified sync ID with the specified object. The other takes only a `Reconcilable` object, automatically generates a new, unique sync ID belonging to some class defined by the sync-store implementation, associates the sync ID with that object, and returns the generated sync ID. (The `SyncStore` interface has another method named, `generateId`, that returns a unique sync ID each time it is called.)

The `get` method takes a sync ID and returns the corresponding `Reconcilable` object, or `null` if there is no object associated with that sync ID. (There is also a method named `contains` that takes a sync ID and returns a boolean result indicating whether there is any object associated with that sync ID.) A call on `get` returns the same object reference that was passed as an argument to `put`. For example, suppose an application has a class `Employee` implementing the `Reconcilable` interface and providing a method `setName` that modifies the contents of an `Employee` object. Suppose further that `store` belongs to a class implementing the `SyncStore` interface, `id` belongs to a class implementing the `SyncId` interface, and `emp` is an object of class `Employee`. The statements

```
store.put(id, emp);
emp.setName("Clark Kent");
```

have the effect of modifying the contents of the object in the sync store, as do the statements

```
Employee e = store.get(id);
e.setName("Lois Lane");
```

The contents of the object referenced by `emp` may also be modified by a synchronizer.

Whenever an application modifies the contents of an object in a sync store, it must call a method in the SyncStore interface to inform the sync store of the change. This is normally done by calling a method named markAsUpdated, passing the modified object's sync ID as a parameter. To avoid the risk that an application will inadvertently modify an object in the sync store without calling markAsUpdated, an application writer can declare all the fields of a class implementing Reconcilable to be private and include a call on markAsUpdated in all of the class's setter methods.

An MNCRS-compliant device has some form of persistent memory in which data can be saved when the device is powered off. The contents of a sync store are saved in this persistent memory when a sync store is closed, and restored when an existing sync store is opened. The SyncStore interface provides a method named flush that can be called to update the copy of a sync store in persistent storage with the current state of the sync store. If a system failure occurs (other than the destruction of the persistent storage itself), then the state of the sync store as of the last flush will be restored the next time the sync store is opened. Some applications might offer the end-user a mechanism for issuing a "Save" command, and call flush every time this command is issued. Other applications might call flush after each major update (e.g. after all the changes involved in adding a single appointment, together with its subsidiary data, to an appointment-book sync store). Other applications might call flush after every *n* changes, or every *n* minutes, for some suitable value of *n*. (The flush method is not an appropriate mechanism for committing transactions, because implementations are permitted to write updates to persistent storage even before the method is called. A device that stores Java objects in nonvolatile memory might not have a separate persistent store. The flush method need not do anything on such a device, since the specified object is already in persistent storage.)

A call on markAsUpdated imposes two obligations on the sync store:

- The object must be marked as containing a newer variant, so that the new object state will replace any remote copy of the old object state during synchronization.
- The copy of the object in persistent storage must be updated, no later than the next call on flush, to reflect the updates.

On occasion, a Reconcilable object has fields whose values are only meaningful in the context of the local sync store; replicas of the sync store may have different values for these fields. Changes to such fields should not be propagated during synchronization. Nonetheless, changes to such fields should be written to the local persistent store, so that they will be preserved when the sync store is closed and later reopened. When such a field is changed, the application program calls a SyncStore method named markForFlushing instead of markAsUpdated. This call imposes an obligation on the sync store to update the copy of the object in persistent storage, but not to mark the object has having a newer variant. More rarely, a Reconcilable object may have fields that have no local significance, but are used to propagate some transitory information to another sync store. When such a field is modified, a SyncStore method named markForSynchronization can be called instead of markAsUpdated. This call marks the object as holding a new variant, so that the new object state will be propagated to the remote replica during synchronization, but it does not impose any obligation on the sync store to update the persistent copy of the object.

The SyncStore interface declares a method, named getSyncIdOf, that performs an inverted lookup. This method takes a reference to a Reconcilable object and checks whether that object is stored in the sync store. If it is, the associated sync ID is returned; otherwise null is returned. This method does not examine the contents of objects; it checks whether the object whose reference is passed as a parameter, not some object with equal contents, is in the sync store.

The SyncStore interface also has a method named getEntry, which takes a sync ID and returns the corresponding sync entry (or null if there is no such entry), and a method named entries, which returns an iterator over all entries in the store. The SyncEntry interface has methods named isDeleted, indicating whether a sync entry is for a deleted object, getId, returning the sync ID associated with the sync entry, and get, returning a sync entry's Reconcilable object (or null if that object has been deleted). When getEntry does not return null, the expression

```
store.getEntry(id).get()
```

(invoking the get method of the SyncEntry interface) is equivalent to

```
store.get(id)
```

(invoking the get method of the SyncStore interface).

5.2 Application Classes for Stored Objects

The objects in a sync store belong to application-defined classes that implement the Reconcilable interface. The Reconcilable interface inherits the writeRemote and readRemote methods from the Transmittable interface described in Section 4.1, and declares three methods of its own, named setTo, reconcile, and reconcileWithDelete. As Section 4.1 explained, an application programmer can extend the DefaultReconcilable class to inherit default implementations of writeRemote and readRemote, using Java serialization.

An object's setTo method sets the contents of that object to the contents of another Reconcilable object passed as a parameter. The programmer may presume that the parameter is an object of the same class, and simply write a sequence of assignment statements, one for each field of the class, assigning each field of the parameter to the corresponding field of the target object. During synchronization, when the variant of an object in one replica is newer than the corresponding variant in the other replica, the setTo method of the object containing the older variant is automatically invoked with a copy of the newer variant passed as a parameter.

The reconcile method is invoked during synchronization when an object is found to have been updated concurrently in the two replicas being synchronized. The method determines a new state for the object that appropriately reflects the two updates, either by selecting one of the updates or by merging the two updates in an application-defined manner. The method is invoked on the object in the local replica, and a copy of the object in the remote replica is passed as a parameter. The method leaves the object in the local replica ("this") holding the new state and returns an integer code indicating whether the new state is identical to the old local state, identical to the old remote state, or identical to neither. The sync-store implementation may use this code

as a hint that the variant of the local object in the persistent store is still valid, and need not be rewritten; or that the result of the reconciliation is already present in the remote replica, and need not be transmitted back to that replica. It is always safe for the `reconcile` method to indicate that the new state of the object is identical to neither of the old states, in which case neither of these optimizations will take place.

The `reconcileWithDelete` method performs a function similar to the `reconcile` method, except that it is called to resolve a conflict between a deletion from one replica and an update in another replica. The method has a boolean parameter, `localDeleted`, indicating whether the deletion occurred in the local replica or the remote replica. When the update occurs in the local replica and the deletion is from the remote replica, the `reconcileWithDelete` method of the object in the local replica is called, with the parameter `localDeleted` set to `false`. When the deletion is from the local replica and the update occurs in the remote replica, a local copy of the remote, updated object is constructed. The `reconcileWithDelete` method of this new object is called, with `localDeleted` set to `true`. Like `reconcile`, `reconcileWithDelete` returns an integer code indicating whether the object should be kept in the old local state or the old remote state (one of which is the deleted state and one of which is a modified state), or in some new modified state. If the deletion is to prevail, the method examines `localDeleted` and returns a code indicating that the object is to be kept in the local state if `localDeleted` is true, or in the remote state if `localDeleted` is false. If the update is to prevail, the method does just the opposite. If some new modified state is to prevail, the method sets its object ("`this`") to that modified state and returns the corresponding code.

There are no restrictions on the behavior of a `reconcile` or `reconcileWithDelete` method, so MNCRS conflict resolution is quite flexible. For example, an application could invoke a dialog with the end user to resolve the conflict. Alternatively, an application could place an object in a special state that references copies of the two conflicting object states and marks the object to be in conflict. (The other methods of the object would have to specify some appropriate behavior for an object in this state.) Then the conflict might later be resolved manually, or resolution could be deferred until the conflict object reaches a device where enough information is available to resolve the conflict automatically, as in Figure 4. Alternatively, a class implementing the `Reconcilable` interface could include a field containing a "resolver" object with a method invoked by the class's `reconcile` method. Such a scheme provides Bayou's flexibility to apply different reconciliation algorithms to different objects of the same type.

Nonetheless, certain reconciliation behaviors make more sense than others. In particular, a reconciliation function should be *idempotent*, having the same effect regardless of the number of times a given remote object state is reconciled with a given local object. Functions like union of corresponding set-valued fields and maximum of corresponding numeric fields have this property, but functions like averaging and summation of corresponding numeric fields do not. When reconciliation functions are not idempotent, the state to which an object's replicas eventually converge depends on the pattern of synchronization; the greater the number of paths by which news of an update reaches a given store, the greater that update's influence on the eventual state, as illustrated in Figure 9.

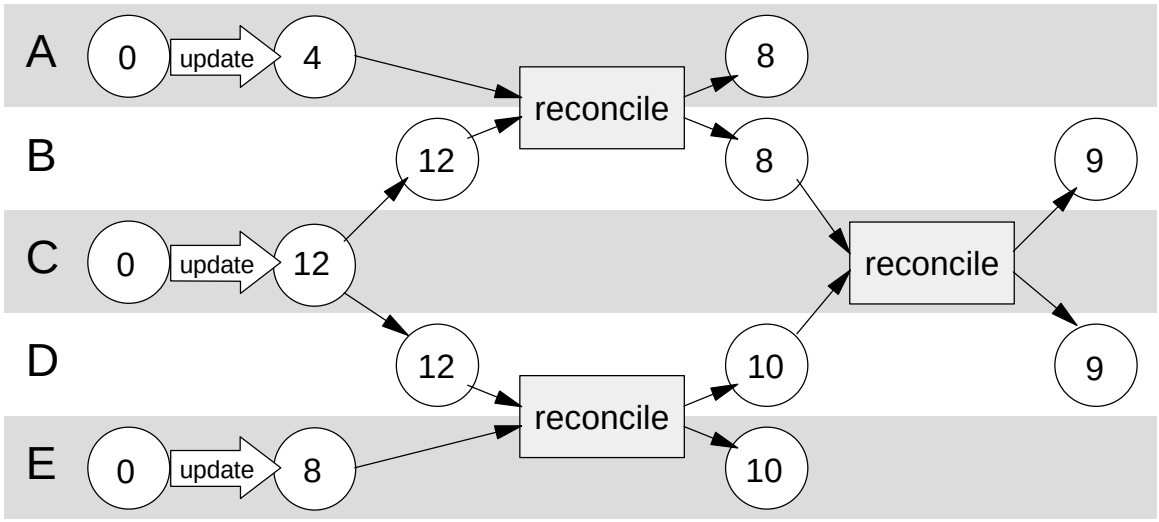

Figure 9. The effect of nonidempotent reconciliation. Initially, corresponding objects in replicas A, C, and E each contain the value 0 in a particular field, and reconciliation consists of averaging the contents of this field. The objects in replicas A, C, and E are updated to hold the values 4, 12, and 8, respectively, and synchronization propagates the object in replica C to replicas B and D. Replica A synchronizes with replica B, replica D synchronizes with replica E, and then replica B synchronizes with replica D. Even though the average of the values set by the three updates is 8, the final reconciliation (which is later than all other updates and will eventually propagate to all five replicas) is 9, because the update at replica C contributed to the average twice.

5.3 Managing Local Sync Stores

To avoid confusion in the use of the term *sync store*, it useful to distinguish between an object of a class implementing the `SyncStore` interface and the persistent collection of data that can be accessed through such an object. We shall refer to the former as a *`SyncStore` object* and to the latter as a *sync-store data collection*. A sync-store data collection is named by a URL that includes a host name and a collection name. Every sync-store data collection on a given host has a distinct collection name.

The class `StoreManager` keeps track of all the sync-store collections on the local device. names. This class has static methods to construct a `SyncStore` object for a given sync-store data collection, to delete a sync-store data collection from the device, and to obtain the collection names of all sync-store data collections on the device.

The method that constructs and returns a `SyncStore` object for a given sync-store data collection is named `open`. A call on `open` may specify a sync-store collection that already exists on the local device, or it may request that a new, empty sync-store data collection be created. Optionally, the call on `open` may request that a newly created sync-store data collection have specified properties, such as supporting a particular extension of the `SyncStore` interface or a particular class of `SyncUpdate` objects; or verify that an existing sync-store data collection has such properties. A new collection can be populated by an application or by synchronizing it with

a remote replica. The SyncStore object returned by open has a method named getName, which returns the collection name that was passed to open.

Each call on open generates and returns a reference to a distinct SyncStore object. However, several SyncStore objects may correspond to the same sync-store data collection, as shown in Figure 10. If the sync-store data collection is modified through a method of one SyncStore object, the modification can be observed using a method of another SyncStore object.

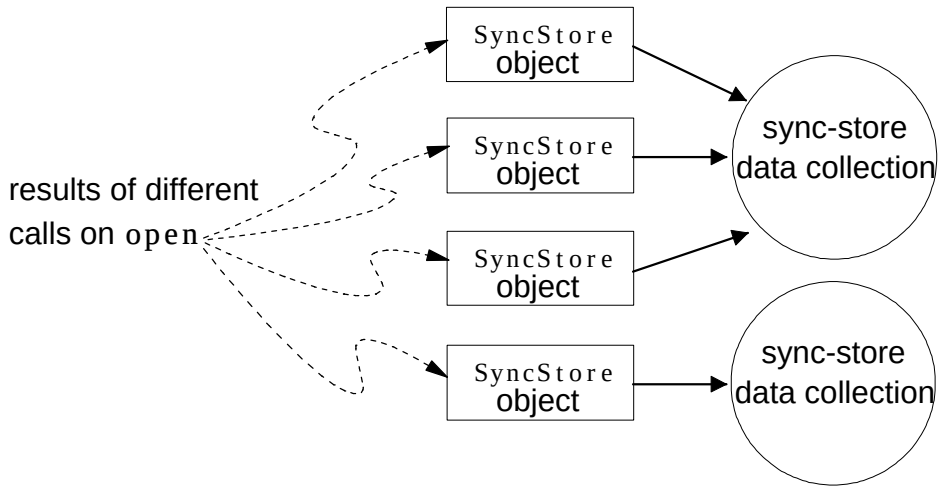

Figure 10. Sharing of sync-store data collections through multiple SyncStore objects. Each call on the StoreManager method open returns a reference to a new SyncStore object. Different SyncStore objects may refer to the same data collection or to different data collections.

A SyncStore object has a method named close, intended to be called when an application no longer needs the object. (Any call on a method of a closed SyncStore object throws an exception.) If other SyncStore objects for the same data collection remain open, the Reconcilable object references obtained or inserted through the closed SyncStore object may still be shared by the holders of the other SyncStore objects. If no such SyncStore objects remain open, then the in-memory representation of the sync-store data collection may be discarded, in which case the next SyncStore object constructed for that collection will yield references to *new* Reconcilable objects, freshly reconstructed from their representations in the persistent store. In either case, it is generally prudent for an application closing a SyncStore object to discard all Reconcilable references it obtained through that object.

5.4 Events and Listeners

Using the JavaBeans event model [Ham97], an application can register objects that *listen* for certain *events*. These objects, called *listeners*, can be used to track the progress of synchronization or changes to a sync-store data collection by other applications or by synchronizers. Events that may be listened for include the insertion, modification, or deletion of an object in the sync store, the opening or closing of a sync store, the flushing of a sync store into persistent storage, and progress in synchronization. For example, an application might listen for

insertion events during synchronization to accumulate a list of all newly inserted objects, and listen for completion of the receiving phase of synchronization to insert these objects in a data structure; or it might update a graphical display of the current contents of a sync-store data collection every time it changes.

For each kind of event, the data-synchronization framework provides a *listener interface*, extending the interface `java.util.EventListener` and declaring one or more abstract methods, each invoked in a different circumstance. Each listener-interface method has a single parameter, called an *event object*, that belongs to a subclass of `java.util.EventObject` and describes the event. An event object has a method that returns some object designated as the *source* of the event, and may have other methods as well. Each object that may act as a source for a certain kind of event has methods to register and deregister listeners for that kind of event. When an event of a certain kind occurs, an event object describing the occurrence is created. Then the appropriate method of each of the listeners currently registered to listen for that kind of event is invoked, with the event object passed as a parameter. The listener methods are invoked synchronously, in some arbitrary order. The data-synchronization framework catches and ignores any exception thrown by a listener method. To listen for a particular kind of event and handle it in a particular way, an application programmer writes a class that implements the listener interface for that kind of event, overriding the abstract methods of the listener interface with methods that perform the desired event-handling actions, and registers an instance of the class as a listener.

A synchronizer can be the source of a *sync-status event*, which marks a milestone in a synchronization phase. A listener for a sync-status event has seven distinct methods, invoked in the following circumstances:

- the start of a sending phase of a synchronization
- the start of a receiving phase of a synchronization
- the completion of some portion of a phase
- the successful completion of a sending phase
- the failure of a sending phase
- the successful completion of a receiving phase
- the failure of a receiving phase

A class implementing the `Synchronizer` interface determines its own policy for the points at which a sync-status event is fired to report completion of some portion of a phase, perhaps based on the number of updates or the number of bytes transmitted so far. (A phase is considered to have failed if it times out.) The event object passed to each of the seven listener methods has a method that returns a *copy* of the synchronizer's `SyncStatus` object (see Section 4.3), providing a snapshot of the synchronization status at the time of the event. Every `Synchronizer` object has methods for registering and unregistering listeners for the sync-status events it triggers. In addition, to avoid the need for an application programmer to keep track of `Synchronizer` objects, the class `SyncManager` provides *static* methods to register and deregister listeners for all sync-status events triggered by *any* `Synchronizer` object. (In retrospect, a listener providing only three methods, one called at the beginning of a phase, one called after completion of portions

of a phase, and one called at the end of a phase, would have been easier for application programmers to use. Parameters to these methods could have distinguished between send and receiving phases and between normal and abnormal termination.)

A `SyncStore` object can be the source of a *sync-object event*, which occurs when a `Reconcilable` object is inserted, marked as updated, or deleted. A listener for a sync-object event has a distinct method for each of these three cases. The event object passed to these listener methods has methods that return the sync ID of the affected object and the object itself.

A `SyncStore` object can also be the source of a *sync-store event*, which occurs when the sync store is opened, flushed, or closed. A sync-store event involves the sync-store data collection as a whole rather than one particular `Reconcilable` object. A listener for a sync-store event has three distinct methods for opening, flushing, and closing events.

All `SyncStore` objects for a given sync-store data collection (see Figure 10) share a single registry of sync-object-event listeners and a single registry of sync-store-event listeners. Each sync-object or sync-store event affecting the sync-store collection is reported to all registered listeners. The source of the event is the `SyncStore` object through which it was triggered. By examining the source of an event, an application can distinguish the events triggered through its `SyncStore` object from those triggered through other `SyncStore` objects. The sync-object-event and sync-store-event listeners registered through a particular `SyncStore` object are automatically removed when that `SyncStore` object is closed.

Event objects for the three different kinds of events belong to classes named `SyncStatusEvent`, `SyncObjectEvent`, and `SyncStoreEvent`. These are each subclasses of an abstract class named `SyncEvent`, which is in turn a subclass of `java.util.EventObject`.

Listeners for the three different kinds of events belong to application-provided classes that implement interfaces named `SyncStatusListener`, `SyncObjectListener`, and `SyncStoreListener`. A class implementing one of these interfaces must provide definitions for all of its methods, even those methods for which the desired action is to do nothing. To make it more convenient to write listeners in which many of the methods have no associated action, the data-synchronization framework provides an *adapter class* for each listener interface, implementing every method of the listener interface with a null body. Rather than implementing a listener interface directly, an application programmer can extend an adapter class, overriding only those methods for which a nonnull body is desired.

5.5 Invoking and Monitoring Synchronization

The framework supports two programming styles for managing synchronization, synchronous and asynchronous. The synchronous style entails calling a method that does not return until synchronization has completed, and then examining the synchronizer's final status to determine whether synchronization completed successfully or unsuccessfully. The asynchronous style entails registering listeners for sync-status events, then calling a method that initiates a

synchronization thread, then returns, so that the calling thread can continue in parallel with the synchronization thread.

As Section 4.3 explained, the remote replica with which synchronization is to take place is specified by an object of class SyncReplicaInfo. Associated with each sync-store data collection is a registry of SyncReplicaInfo objects for the data collection's known replicas. A SyncStore object has methods for adding SyncReplicaInfo objects to its data collection's registry and removing them. The framework has methods to synchronize a sync store with all its known replicas, as well as methods to synchronize a sync store with a particular replica (which need not be in the registry).

The SyncStore interface provides two methods for synchronous synchronization. One attempts to synchronize with a replica specified by a SyncReplicaInfo parameter and returns a SyncStatus result describing the outcome. The other attempts to synchronize with all registered replicas, and returns an array of SyncStatus objects, one for each attempted synchronization. Each method has a parameter specifying a time limit after which all synchronization attempts will fail by timing out, with zero indicating that no limit is imposed.

The SynchronizerGroup class described in Section 4.3 provides more flexibility, but is more complicated to use. The class is especially useful for writing daemons that synchronize all the sync stores on a device. The SyncManager class has static methods that construct and return synchronizer groups with the following contents:

- a single synchronizer, to synchronize a specified sync store with a specified replica
- synchronizers to synchronize a specified sync store with each of its registered replicas
- synchronizers to synchronize each of several specified sync stores with a corresponding specified replica
- synchronizers to synchronize each of several specified sync stores with all of their registered replicas

The SynchronizerGroup class has a method named start to activate each of its synchronizers and return. Each synchronizer begins executing the default phase schedule specified in its SyncReplicaInfo object. Used by itself, the start method supports the asynchronous programming style. To support the synchronous programming style, a SynchronizerGroup object also has a method named waitUntilDone, which blocks the calling thread until either all synchronizers in the group have completed or a specified time limit expires, and then returns an array of SyncStatus objects, one for each synchronizer in the group; a time limit of zero indicates that the calling thread is to wait indefinitely for the synchronizers to complete. Synchronizers are not aborted upon expiration of the time limit, but another SynchronizerGroup method, named stop, can then be called to stop all the synchronizers in the group.

Even more intricate control can be exercised over synchronization by calling a SynchronizerGroup method that returns an array of Synchronizer objects, one for each member of the group, and then using methods of the Synchronizer interface to manipulate individual synchronizers. These include start, waitUntilDone, and stop methods analogous to the SynchronizerGroup methods of the same name, but applying to an individual synchronizer rather than to a collection of synchronizers. In addition to the start method, which initiates

synchronization according to the default phase schedule of the synchronizer's SyncReplicaInfo object, there are short-cut methods to initiate a synchronization consisting of a single sending phase or one consisting of a single receiving phase. The Synchronizer interface also has methods to retrieve the SyncStore and SyncReplicaInfo objects for, respectively, the local and remote replicas being synchronized. Another method returns the SyncStatus object associated with a synchronizer that has been activated, or null if the synchronizer has not been activated. There are methods to add and remove listeners that will be notified of sync-status events generated by this synchronizer only.

The SyncStatus interface has a method returning a reference to its synchronizer, a method reporting the number of phases the synchronizer was constructed to synchronize and methods that, given an index into the phase schedule, return the following information about a given phase:

- whether the phase is a sending phase or a receiving phase
- whether the phase is not yet started, is in progress, has completed successfully, or has failed
- the completed portion of the phase, as computed by the synchronizer (reported as an integer in the range 0 to 100, representing a percentage)
- a message describing the status of the phase, consisting of numeric codes in a fixed format optionally followed by free-form text

Experience has shown that the SyncStatus interface is missing an important method: It is disconcertingly awkward to determine from a SyncStatus object whether a synchronization as a whole has succeeded, failed, or not yet completed. A synchronizer that executes its phases in sequence may abort the synchronization after one phase fails, leaving subsequent phases marked as not yet started. However, a synchronizer that executes its phases in parallel may continue executing other phases after one phase has failed, and a phase marked as not yet started may become active even after another phase has failed. Nonetheless, there is a roundabout way to determine whether a synchronizer has completed: The Synchronizer interface has a method for determining whether a synchronizer is currently active. This method returns false both before the synchronizer has been activated and after it has terminated. These two cases can be distinguished by the fact that the Synchronizer method returning a synchronizer's status returns null before the synchronizer has been activated. Once it is established that a synchronizer has terminated, its SyncStatus object can be used to check phase-by-phase that each phase has completed successfully.

The MNCRS data-synchronization working group struggled with the appropriate behavior for methods invoking a group of synchronizers if one of those synchronizers should throw an exception. For some applications, it might be appropriate for the method to abort all other synchronizations and throw an exception to its caller. For other applications, it might be appropriate to let the remaining synchronizations proceed normally, leaving the method's caller with the responsibility for checking the final status of each synchronizer in the group. The working group considered distinguishing between errors arising during the creation of synchronizers, e.g. by the SyncManager methods, and errors arising within a synchronizer itself: A method activating multiple synchronizations would first attempt to construct all the needed

synchronizers; if any of these attempts failed, none of the synchronizers would be activated and the method would throw an exception; otherwise each synchronizer would be activated and allowed to proceed independently, recording its success or failure in its SyncStatus object. This approach was rejected because it overly complicated the application programmer's task of checking the outcome of a synchronization attempt, and because the dividing line between the handshaking performed to construct a synchronizer and the communication performed by the synchronizer itself varies from one protocol to another.

To reduce the number of ways in which synchronization errors might manifest themselves, it was decided that the construction of a synchronizer group always succeeds. None of the SyncManager methods returning a SynchronizerGroup result ever throws an exception, except perhaps as a result of preliminary checks that its arguments are of the required form. Since a synchronizer is constructed by invoking each kind of synchronizer factory in turn until one returns a nonnull value, there must be an error-proof way of constructing a synchronizer even when all synchronizer factories return null. The FailureSynchronizer class was added to the framework to address this need. The FailureSynchronizer constructor, invoked when all synchronizer factories return null, never throws an exception. Any attempt to activate a FailureSynchronizer object immediately fails, and sets the corresponding SyncStatus object to reflect this failure.

An earlier version of the framework allowed synchronizers to be reused for multiple synchronizations, but this invited race conditions in the examination of a synchronizer's SyncStatus object. To ensure that a synchronizer is used at most once, any attempt to activate a synchronizer that has already been activated must be intercepted. So that the methods activating a synchronization will not throw exceptions, the framework stipulates that these methods have no effect when invoked on an already-activated synchronizer.

5.6 Dealing with Concurrency

A sync-store data collection can be opened for either *exclusive access* or *shared access*. While it is open for exclusive access, any other attempt to open the sync-store data collection throws an exception. (Similarly, an exception is thrown upon an attempt to open a sync-store data collection for exclusive access while it is already open for shared access.) In contrast, a sync-store data collection open for shared access may have several open SyncStore objects at the same time. On a synchronization server, a synchronization request handler that receives a request to synchronize with a particular sync-store data collection must open that data collection before it can construct a synchronizer to satisfy the request. Thus an application that opens a sync-store data collection for exclusive access blocks the servicing of incoming requests to synchronize with that data collection.

An application that opens a sync-store data collection for shared access must be careful to account for the possibility of concurrent access by multiple threads. The data collection may opened by more than one application, or by an application and a synchronization request handler, or it may be opened several times by the same synchronization request handler to service multiple incoming synchronization requests (with each resulting SyncStore object passed to a synchronizer executing its own thread). An application with more than one thread accessing the

same SyncStore object must be careful to account for concurrency even if it has exclusive access to the sync-store data collection.

Since all users of a sync-store data collection share references to the same Reconcilable objects (inserted in the sync store through calls on the SyncStore method put and retrieved through calls on the SyncStore method get), race conditions can arise between two updates to the same object, or between an update and an examination of the same object. It is also possible for one thread to delete an object from the data collection while another thread continues to update or examine it as if it were still part of the data collection.

Race conditions can be avoided by using Java's synchronized blocks to lock objects while they are being manipulated. There is a lock associated with every Java object. The execution of a synchronized block is associated with a particular object, and a thread about to enter the block is forced to wait until it has obtained that object's lock. The lock is relinquished at the end of the block. (If a method consists only of a synchronized block for the method's object, this, the synchronized modifier can equivalently be placed on the method itself instead of on the block.)

By declaring all fields of a class implementing Reconcilable to be private, the author of the class retains complete control over access to those fields. If there is a sequence of operations on an object that ought to be executed indivisibly—that is, without interleaved examination or modification of that object by another thread—the entire sequence of actions should be placed in a synchronized block. The data-synchronization framework allows a synchronizer to examine or update a Reconcilable object or its sync entry only within a synchronized block for that object. (The deletion of an object is an update to its sync entry.) If an application method updates a Reconcilable object, the requisite call on markAsUpdated, markForFlushing, or markForSynchronization should be enclosed in the same synchronized block as the update, to ensure that a thread performing synchronization does not examine the object after it is been modified by an application, but before it has been marked as containing a newer variant. It is possible for an object to be deleted from a sync-store data collection between the time an application obtains a reference to it by calling get and the time the application makes some later access to the object. If the correctness of the later access relies on the object being in the sync store, the application should perform a test, within the same synchronized block as the access, to confirm that the object is indeed still there. Multiple objects can be updated indivisibly by nesting synchronized blocks for each of the objects, thus ensuring that all of the objects are locked while the operation is in progress. However, care must be taken to avoid deadlock, for example by ensuring that any time a particular pair of objects is to be locked, the locks are obtained in the same order.

5.7 Trusting the Application Writer

Like the MNCRS data-synchronization framework, Bayou and Coda run application code to resolve conflicts. Both Bayou and Coda take the view that this code is not to be trusted, and take measures to protect the system and its data from malicious or erroneous application code. Bayou *merge procedures*, which are roughly analogous to reconcile methods, are not allowed to have any side-effects other than writing the database [Dem94]. This restriction protects a device hosting against arbitrary actions by a merge procedure, which is, in effect, a mobile agent

of the application writer. A fundamental tenet of Coda is that server machines—shared assets requiring strong protection —are trusted, and client workstations—used for a variety of individual purposes by experts who develop and install much of their own software—are untrusted ([Kum93], [Kum95]). Conflicts are resolved by programs called *application-specific resolvers*, which are executed on client machines with user privileges, thus protecting servers against Trojan horse resolvers invoked by apparently innocuous file references. To guard against resolvers that run forever, or that incorrectly report they have successfully resolved a conflict when they have not (which would cause the Coda client code to reinvoke the resolver forever), there are configurable limits on how long a resolver may run and how much time must pass before it is run again.

In contrast, following the network computing model, an MNCRS platform is intended as a limited-function device whose software is provided by some central source rather than by the end user. Therefore, the MNCRS data-synchronization framework is far more trusting of application code than Coda or Bayou. For example, an application is expected to inform a sync store when it changes the contents of a `Reconcilable` object in that store. The application methods invoked to resolve conflicts are expected to do no harm, to terminate, and to return an integer code accurately describing whether the reconciliation is to keep the local state of the object, the remote state of the object, or some other state. Similarly, application implementations of the `Transmittable` interface are expected to work correctly. Applications are expected to discard `Reconcilable` references obtained through a given `SyncStore` object when that object is closed, as described at the end of Section 5.3, and to guard against race conditions as described in Section 5.6.

This page intentionally left blank.

6 Implementation of Synchronizers

We now turn our attention from the MNCRS data-synchronization framework itself to the implementation of that framework developed at IBM Research.

Our implementation initially included one class implementing the `Synchronizer` interface. This class, designed for testing the framework implementation over a strongly connected LAN, performs synchronization over TCP/IP sockets, and makes no provision for error recovery. Section 6.3 discusses this form of synchronization, called *simple socket synchronization*, in greater detail. Later, as part of the Mobile Data Synchronization Service project [But00] (discussed further in Section 9), we implemented a second synchronizer, which performs synchronization over a reliable messaging service. Section 6.4 discusses the implementation of this synchronizer. We begin by describing aspects of the implementation common to both kinds of synchronizers: Section 6.1 discusses our implementation of the `SyncStatus` interface and Section 6.2 describes the creation and invocation of `SynchronizerFactory` objects. Section 6.5 describes other kinds of synchronizers that have been contemplated, but which we have not implemented.

6.1 Implementation of Synchronization-Status Objects

From the time it is activated, every object of a class implementing the `Synchronizer` interface has associated with it an object of a class implementing the `SyncStatus` interface. Although it is possible for each `Synchronizer` implementation class to use a different `SyncStatus` implementation class that it understands (or even for a single class to implement both the `Synchronizer` and `SyncStatus` interfaces), both of our `Synchronizer` implementation classes use a `SyncStatus` implementation class named `SyncStatusImpl`. This class extends the `SyncStatus` interface with methods called by a synchronizer to modify the status. The `SyncStatusImpl` constructor takes a phase schedule as a parameter and constructs an object in which each phase is marked as not yet started. Four distinct methods allow a synchronizer to modify the status to reflect the start of a specified phase, the completion of some percentage of a specified phase, the successful termination of a specified phase, or the unsuccessful termination of a specified phase for a reason indicated by some failure code. For the convenience of the synchronizer writer, this status-updating interface is oriented towards the milestones a synchronizer encounters during synchronization rather than the components of a `SyncStatusImpl` object that must be updated to reflect those milestones.

Each time a synchronizer generates a sync-status event, a clone of its `SyncStatus` object must be placed in the event object, so that the event object will continue to reflect the same snapshot of the status even as the synchronizer updates its `SyncStatus` object. Since sync-status events are generated inside a potentially critical synchronizer loop whenever the synchronizer has progress to report, the `SyncStatusImpl` class is designed to make cloning as inexpensive as possible. As illustrated in Figure 11, a `SyncStatusImpl` object contains a reference to its synchronizer and a reference to an array with one element for each scheduled phase. Each array element contains a reference to an object holding the status information for one phase. Cloning consists of copying the synchronizer reference, copying the array, and substituting a reference to the array copy for the reference to the original array. Once created, an object holding the status information for one phase is immutable. A `SyncStatusImpl` object is updated by creating a new

object holding the updated information for the affected phase, and replacing the reference to the old phase-status object in that SyncStatusImpl object's array with a reference to the new phase-status object. The arrays of other SyncStatusImpl objects referring to the old phase-status object remain unchanged.

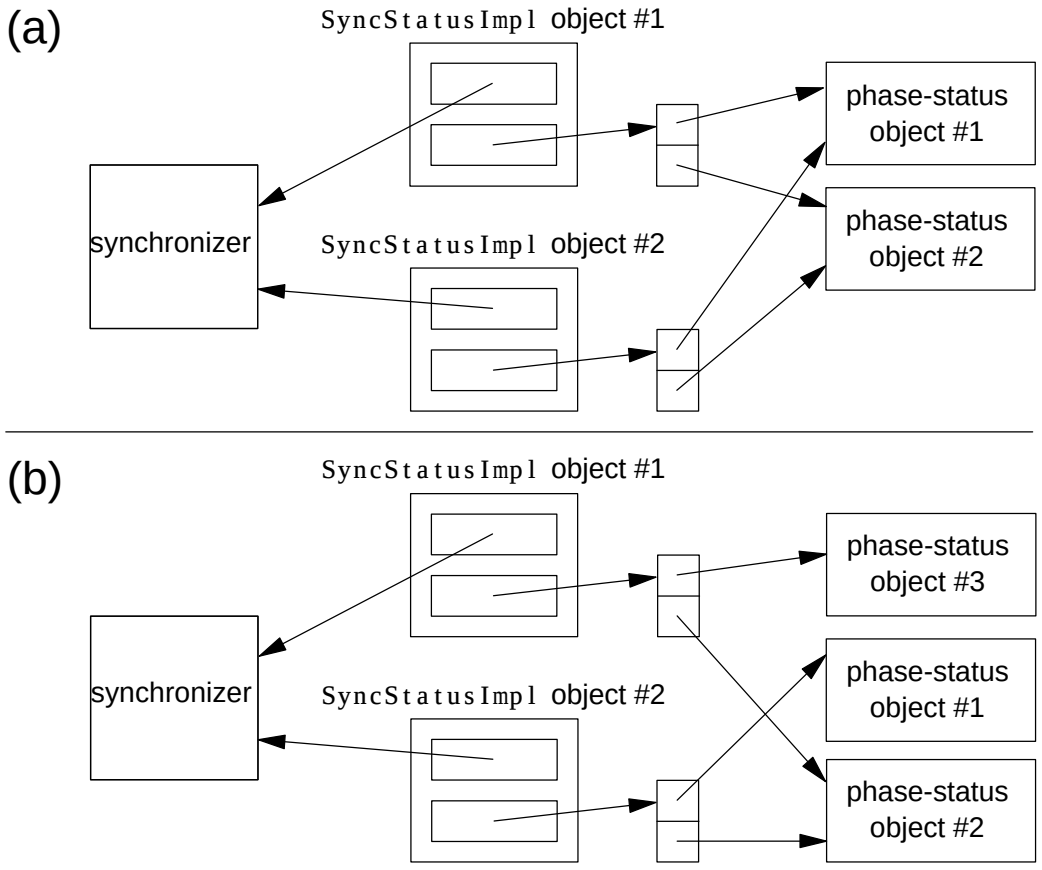

Figure 11. Cloning and modifying SyncStatusImpl objects. In (a), SyncStatusImpl object #2 is a clone of SyncStatusImpl object #1. The clone has a reference to the same synchronizer, and a reference to a distinct array containing references to the same phase-status objects, as the original. In (b), the status of the first phase of SyncStatusImpl object #1 has been modified. This is reflected by creating a new phase-status object, #3, containing the revised phase information, and replacing the first element of the array with a reference to the new phase-status object. Existing phase-status objects, and the arrays of other SyncStatusImpl objects, remain unchanged, so SyncStatusImpl object #2 continues to reflect the same status as it did before SyncStatusImpl object #1 was updated.

A string containing a textual description of the current status is passed as a parameter to each of the four status-updating methods. The SyncStatus method getPhaseMessage returns a string consisting of a fixed-format prefix followed by this textual description. The prefix reflects other information in a phase-status object. To avoid the expensive string manipulation required to construct the phase-message string each time a status-updating method is called, the status-updating methods simply save a reference to the string passed as a parameter, and

construct the full phase message only on demand. The first time getPhaseMessage is called to obtain the phase message for a particular phase-status object, the full phase message is constructed, and saved in the phase-status object for any future calls on getPhaseMessage.

6.2 Creation and Invocation of Synchronizer Factories

In our reference implementation, a static initializer for the SyncManager class reads the system property that lists installed classes implementing the SynchronizerFactory interface. It then constructs an array containing one instance of each class. The SynchronizerFactory interface has one method, getSynchronizer, which takes an open SyncStore object and a SyncReplicaInfo object as parameters and returns a synchronizer to synchronize the given local sync store with the given remote replica. The SyncManager methods returning synchronizer groups construct each group member by iterating over the array of factory objects, invoking the getSynchronizer method of each array element, until a nonnull Synchronizer reference is obtained or the end of the array is reached. If the end of the array is reached, a FailureSynchronizer object is constructed placed in the synchronizer group.

6.3 Simple Socket Synchronization

The class implementing Synchronizer to perform simple socket synchronization, named SynchronizerImpl, is part of a package named simpleSocketSynchronization. That package also includes corresponding implementations of the framework's SynchronizerFactory and SyncRequestHandler interfaces, named SynchronizerFactoryImpl and SyncRequestHandlerImpl. Because the data-management aspects of synchronization are handled by the sync store, the implementation of the synchronizer is straightforward.

The protocol for performing a simple socket synchronization is illustrated in Figures 12 and 13. A synchronization server constructs an SyncRequestHandlerImpl object and instructs it to begin listening for connection requests at a well-known port. The getSynchronizer method of SynchronizerFactoryImpl extracts the remote-replica URL from its SyncReplicaInfo parameter and requests a connection at the well-known port of the host named by the URL. If the connection request fails, because the URL is malformed or specifies an unknown host, or because of some other communication error, getSynchronizer constructs and returns a FailureSynchronizer object. Otherwise, the synchronization request handler listening at that port accepts the connection request and sets up a bidirectional socket connection. Using this socket, getSynchronizer sends the data-collection name from the URL and the default phase schedule from the SyncReplicaInfo parameter to the synchronization request handler. The phase schedule is encoded so that the most common schedules—send/receive, receive/send, send, and receive—are each specified by one byte, and any other schedule of up to eight phases is specified by two bytes. The synchronization request handler attempts to open the named data collection on its platform, and sends back a one-byte code indicating success or failure. In case of failure, the code specifies the cause of the failure (typically that the named store does not exist, or that it is already open for exclusive access) and the synchronization request handler goes on to listen for future connection requests. The getSynchronizer method examines this one-byte

code. If the code indicates failure, or if some communication error occurred during the interchange, `getSynchronizer` constructs and returns a `FailureSynchronizer` object.

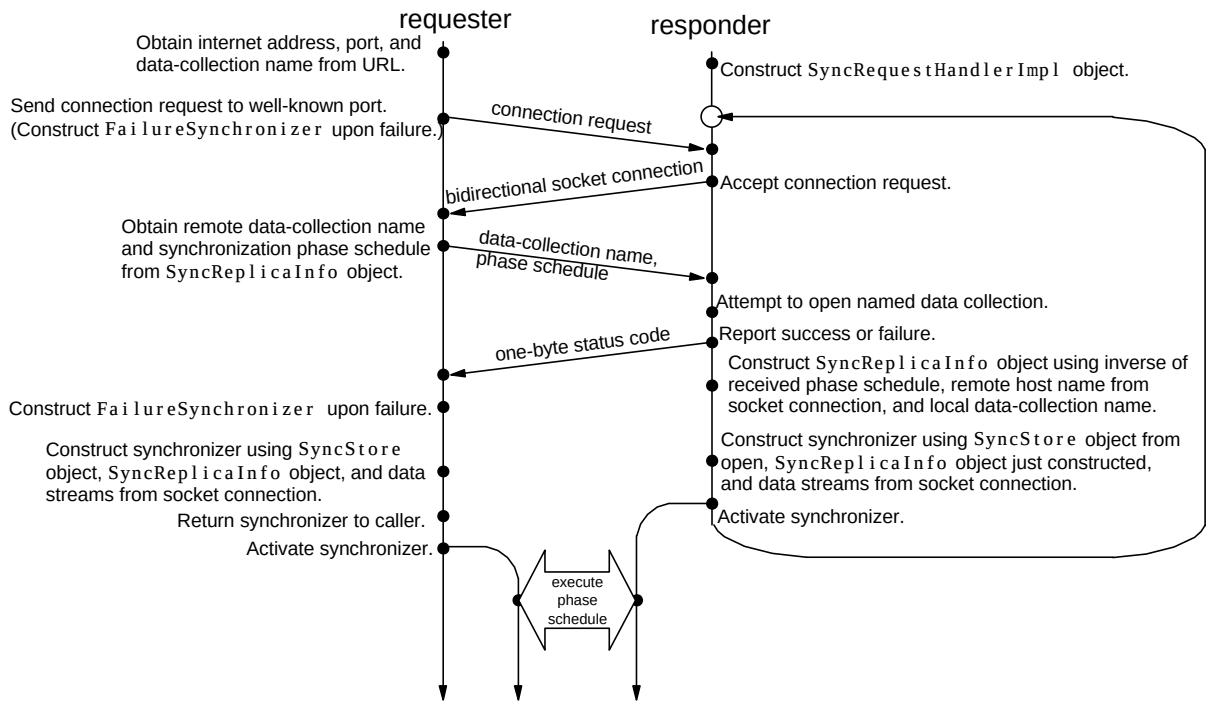

Figure 12. The protocol for initiating a simple socket synchronization. A synchronization server creates a `SyncRequestHandlerImpl` object to listen for connection requests at a specified port. The thread associated with this object executes an iteration of the loop at the right of the diagram each time a connection request is received. To initiate synchronization with a synchronization server named by the URL in a given `SyncReplicaInfo` object, a requester executes the thread on the left of the diagram. The requesting and responding threads each construct a synchronizer with a specified `SyncStore` object, `SyncReplicaInfo` object, input data stream, and output data stream. When activated, each synchronizer spawns a thread that executes a sequence of sending and receiving phases specified by the `SyncReplicaInfo` object with which it was constructed. Figure 13 depicts the protocol for a single phase.

If the sync store on the synchronization server is opened successfully, both the factory and the synchronization request handler construct `SynchronizerImpl` objects. The parameters of the `SynchronizerImpl` constructor include a `SyncStoreUpdater` object for the local sync store, a `SyncReplicaInfo` object for the remote replica, and input and output data streams for the socket connection. The factory's `getSynchronizer` method obtains the first argument by casting its own `SyncStore` parameter to `SyncStoreUpdater`, and returns the result of the constructor. (The synchronizer returned by the factory will be activated later, when one of its methods, or the `start` method of the synchronizer group containing it, is called.) The synchronization request handler casts the `SyncStore` object it just opened to `SyncStoreUpdater`, constructs a phase schedule by inverting the sending and receiving phases of the phase schedule sent by the requester, and uses this phase schedule to construct a `SyncReplicaInfo` object describing the requester's sync store. The URL for this `SyncReplicaInfo` is synthesized by getting a `java.net.InetAddress` object from the socket connection, calling the `InetAddress` method

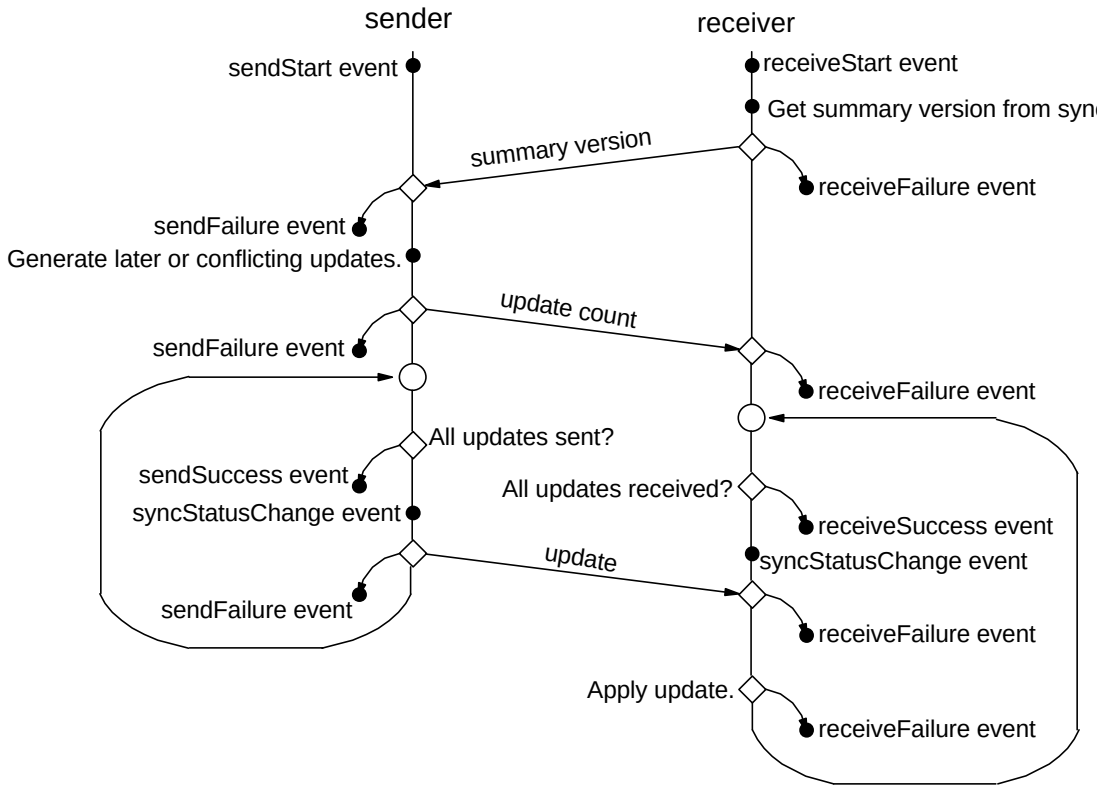

Figure 13. The protocol for performing a single phase of a simple socket synchronization. The correspondence between the roles of sender and receiver depicted in this diagram and the roles of requester and responder depicted in Figure 12 varies from phase to phase. This diagram shows the points at which sync-status events are triggered. An exception may be thrown by an attempt to send a message, receive a message, or apply an update, in which case the phase fails.

`getHostName` to determine the requester's host name, and appending the local data-collection name received from the requester (even though this is the name of the synchronization server's data collection, and the name of the requester's replica may be different). After constructing its `SynchronizerImpl` object, the synchronization request handler registers with it a sync-status-event listener that closes the local sync store if the synchronization is complete. Finally, the synchronization request handler activates the synchronizer and goes on to listen for future synchronization requests.

The `start` method of `SynchronizerImpl` creates a `SyncStatus` object corresponding to the default phase schedule and starts a thread that repeatedly executes either a sending phase or a receiving phase, according to the schedule. Once both synchronizers are activated, they execute complementary phase schedules, so that one synchronizer is sending through its output data stream while the other is receiving through its input data stream. In each phase, the receiving synchronizer invokes a `SyncStoreUpdater` method to determine the current *summary version* of its store. This is the earliest version later than or equal to all the versions associated with objects in the store. The receiving synchronizer sends the summary version to the sending synchronizer. The sending synchronizer invokes a `SyncStoreUpdater` method to extract a collection of `SyncUpdate` objects, one for each `Reconcilable` object in its store whose version is later than or

in conflict with the receiving store's summary version. Since the SyncUpdate interface extends the Transmittable interface, it has writeRemote and readRemote methods. The sending synchronizer transmits the number of SyncUpdate objects in its collection, then calls the writeRemote method for each of these objects to write the object's byte-stream representation to the synchronizer's output stream. The receiving synchronizer reads the update count, and then invokes the SyncUpdate method readRemote the corresponding number of times to read the byte-stream representations from its input stream and reconstruct the SyncUpdate objects. As the receiving synchronizer reconstructs each SyncUpdate object, it calls a SyncStoreUpdater method to apply the update to the receiving store.

Each synchronizer updates its SyncStatus object and then triggers a sync-status event at several points in each phase. An event marking the start of the phase is triggered before the receiver's summary sync version is transmitted or received. An event marking progress is triggered just before each individual SyncUpdate object is transmitted or received; the portion of the phase completed so far is computed by dividing the number of updates that have already been processed by the total number of updates. An event marking successful completion of synchronization is triggered after the last SyncUpdate object is transmitted or received. If an exception occurs during some phase, the synchronizer is marked as inactive, the synchronizer's SyncStatus object is updated, an event marking failure of the phase is triggered, and the synchronizer's thread terminates. Such an exception can result from a communications failure, an error in one of the application-provided methods, or, amazingly, from an error in the implementation itself; in addition, the Synchronizer method stop invokes the java.lang.Thread method stop for the thread performing synchronization, which causes the exception ThreadDeath to be thrown within that thread. The application-provided methods invoked during synchronization include the writeRemote method for a Reconcilable object contained in a SyncUpdate object, called by the implementation's writeRemote method for the SyncUpdate object; the corresponding readRemote method; the setTo method called when an update is applied to the receiving store, to replace the contents of an object; and the reconcile or reconcileWithDelete method called when a conflicting update is applied to the receiving store.

6.4 Message-Queue Synchronization

Our second synchronizer implementation uses IBM's MQ Series Everyplace product, a lightweight message-queuing service with a small footprint suitable for memory-constrained mobile devices, to provide reliable communication. The class implementing Synchronizer is named MQSynchronizerImpl, and the corresponding class implementing SynchronizerFactory is named MQSynchronizerFactoryImpl. The synchronizer is intended for use in an environment where there is a central server running an MQ Series Everyplace *queue manager*. In this environment, it is unnecessary to exchange summary versions to determine which updates should be transmitted. A client can keep track of the latest version it previously sent to or received from the server, and send only updates later than or conflicting with that version. Similarly, the server can track this latest-version information for each registered client, and send a particular client only those updates later than or conflicting with the latest version previously sent to or received from that client. Our implementation of the SyncReplicaInfo class has methods beyond those specified in the MNCRS framework, to save and retrieve the latest-version information for a

given remote replica. The URL identifying a sync-store data collection on the server includes the names of a *server input queue* and a *server output queue*.

The `getSynchronizer` method of `MQSynchronizerFactoryImpl` creates a *stub queue manager* on a client device, based on the input queue name in the URL, and passes a reference to this object to the `MQSynchronizerImpl` constructor. (The method returns a `FailureSynchronizer` object instead of an `MQSynchronizerImpl` object if the `SyncReplicaInfo` object passed as a parameter has a malformed URL, if certain required system properties are not defined, or if MQ Series Everyplace fails to create a stub queue manager.) When an `MQSynchronizerImpl` object is activated, it starts a thread that executes the scheduled phases. If the synchronizer's `stop` method is invoked, the thread's `stop` method is invoked, causing `ThreadDeath` to be thrown within the thread; the thread handles this exception by updating the synchronizer's `SyncStatus` object, triggering an event marking the failure of the phase, and terminating.

The sending phase updates the synchronizer's status and generates a sync-status event to reflect the start of the phase, then invokes a `SyncStoreUpdater` method to extract `SyncUpdate` objects for each `Reconcilable` object with a version later than or in conflict with the last-exchanged version in the remote replica's `SyncReplicaInfo` object. It creates a byte-stream representation of the entire sequence of updates and invokes a method of the stub queue manager to enqueue a message on the server input queue named in the remote replica's URL. This message contains the byte-stream representation of the sequence of updates, the presumed data-collection name of the remote replica, authentication information, and reports of any errors encountered in a preceding receiving phase. Finally, the status is updated and a sync-status event is generated to reflect the end of the sending phase. (No sync-status events reporting partial progress are generated.)

The receiving phase updates the synchronizer's status and generates a sync-status event to reflect the start of the phase, invokes a method of the synchronizer's stub queue manager to dequeue the message from a server output queue named in the remote replica's URL, then reconstructs the collection of `SyncUpdate` objects from the byte sequence contained in the message, then calls a `SyncStoreUpdater` method once for each update in the collection, to apply the update to the receiving store, generating a sync-status event to report partial progress as each update is applied. Finally, the receiving phase updates the status and generates a sync-status event to reflect the successful completion of the phase. However, an event reflecting the failure of the receiving phase is generated if the attempt to dequeue a message throws an exception, if a dequeued message is not of the expected form, or contains an incorrect data-collection name or invalid authentication information, if the message indicates that the previous phase in the other direction resulted in a serious error, if an exception resulted from the attempt to apply some update, or if some unanticipated exception occurred.

6.5 Other Synchronization Protocols

We conclude our discussion of synchronizer implementations by mentioning other kinds of synchronizers that have been discussed within the MNCRS data-synchronization working group and elsewhere, but which we have not yet implemented. Recall that several different synchronizer factories can be installed on a given device and listed in a system property. The first listed factory

that is able to construct a suitable synchronizer for a given local sync store and remote replica does so, and the synchronization proceeds using that synchronizer's protocol.

The working group had hoped to agree on a standard protocol, defined in terms of the exchange of Java objects, that all MNCRS platforms would support, perhaps in addition to other open or proprietary protocols. While proprietary protocols might provide superior performance, the standard protocol would guarantee that any two MNCRS platforms would be capable of synchronizing with each other. Unfortunately, the working group did not complete this work. The protocol definition was to include a standard class implementing the `SyncVersion` interface, used to define one or more standard classes implementing the `SyncUpdate` interface, both used to define a standard class `SyncMessage` containing, among other information, a summary version, a sequence of updates, or both. A synchronization phase would consist of the transmission of a `SyncMessage` object containing a summary sync version in one direction, followed by the transmission of a `SyncMessage` object containing a sequence of updates in the other direction. For a sending phase to be followed by a receiving phase, the sequence of updates for the sending phase and the summary version for the subsequent receiving phase could be combined in one `SyncMessage` object. `SyncMessage` objects would be transmitted reliably through a wireless implementation of the Java Message Service API [Hap98] that the MNCRS working group on mobile communication recommended be supported on every MNCRS platform.

A synchronization protocol can be built on top of HTTP. The principal attractions of HTTP are that it is ubiquitous and that it can pass through firewalls. Synchronization messages might be encoded in XML or in binary form in the body of an HTTP `POST` request, or a more extensive vocabulary of requests, such as that used in the WebDAV distributed authoring and versioning extensions to HTTP ([Whi98], [Gol99]), might be used.

Even the Simple Mail Transfer Protocol (SMTP) can be used for synchronization. In essence, synchronizers would e-mail their updates to each other. Like HTTP, SMTP traverses firewalls, but possibly in a less timely fashion. However, in an environment with sporadic and unreliable connections, it may be attractive to communicate by accumulating outgoing mail and forwarding it in bursts when a connection is available. The vision of an SMTP-based synchronization disciplined the working group to design a data-synchronization framework that would accommodate synchronization through asynchronous message passing as well as through rigorously scripted dialogs.

Another attractive basis for a synchronizer is the Tuplink system [Emb98] built at IBM's Tokyo Research Laboratory. The Tuplink project implemented a lightweight, fault-tolerant communications buffer for small mobile devices based on the *tuple space* ([Gel85], [Wyc98], [Jav99]) abstraction. The tuple-space message-passing model is that a sender deposits tuples in a shared data store and a receiver extracts them. In the actual Tuplink implementation, there is no shared store, just separate buffers on the sending and receiving devices. When a communications link is available, Tuplink platforms exchange newly deposited tuples, tolerating communications failures by keeping track of which exchanges were successful and which must be retried the next time a link is available. Tuplink itself is a data-synchronization system in microcosm. However, by restricting access to a given tuple space to only two parties, by allowing the insertion or removal, but not the in-place modification, of a tuple (thus precluding the need to detect or reconcile conflicts), and by dealing with small chunks of data at a low level, Tuplink greatly

simplifies its synchronization problem, facilitating fast data exchange and an extremely small memory footprint.

With a spectrum of protocols available, ranging from highly optimized but narrowly applicable protocols to nonoptimal but universally applicable protocols, it is natural to consider a *metaprotocol* by which two synchronizer factories exchange information about the capabilities of their devices, the speed and reliability of the communications link, and the preferences of users, then negotiate a protocol to use for the synchronization proper. The negotiation might select the fastest protocol supported by both devices, the protocol that works best with a given sync-store implementation, the protocol that works best over a fast and reliable connection, or the protocol that works best with a weak connection, for example. The two synchronizer factories would then construct synchronizers to execute the agreed-upon protocol.

This page intentionally left blank.

7 Implementation of Sync Stores

7.1 Sync-Store Data Structures

Each time a given data collection is opened, a new object implementing the `SyncStore` interface is created, as Figure 10 illustrates. In our implementation, a class named `SyncStoreHandle` implements the `SyncStoreUpdater` interface, and hence also implements the `SyncStore` interface, which `SyncStoreUpdater` extends. For each data collection with open `SyncStoreHandle` objects, there is one object of a class named `SyncStoreState`, holding the shared state associated with the data collection. All `SyncStoreHandle` objects for a given data collection contain a reference to the same `SyncStoreState` object.

The principal components of a `SyncStoreState` object are:

- a *forward* hash table mapping sync IDs to sync entries
- a *reverse* hash table mapping `Reconcilable` objects to sync IDs
- an ordered log of updates
- a 64-bit globally unique identifier called a *replica ID*
- a summary version
- an update counter, acting as a virtual local clock
- registries for sync-object-event and sync-store-event listeners
- a registry of remote replicas
- a reference to a *persistent-store manager*

The forward hash table is used by the `SyncStore` methods `put`, `get`, `delete`, `markAsUpdated`, `markForFlushing`, and `markForSynchronization` to insert, retrieve, delete, or record a modification to a `Reconcilable` object with a given sync ID. The reverse hash table is used by the `SyncStore` method `getSyncIdOf` to look up the sync ID of a given `Reconcilable` object. The log of updates is an object of class `java.util.Vector`. Conceptually, for each sync ID known to the sync store, the log contains an *update-info object* describing the most recent update corresponding to that sync ID; in reality, to avoid shifting the contents of the log each time an update is superseded by a later update for the same sync ID, an update-info object is never removed from the log vector, but is marked as superseded by setting a flag inside the object. The sync-entry objects found through the forward hash table each contain a reference to the corresponding update-info object in the log vector, as well as flag indicating whether the persistent store contains up-to-date information for the corresponding sync ID. Every update-info object contains a sync ID, a `SyncVersion` value, and a *persistent store key* that can be used to retrieve the state of the corresponding `Reconcilable` object from persistent storage or to delete it from persistent storage. An update-info object that does not describe a deletion also contains a reference to a `Reconcilable` object. See Figure 14.

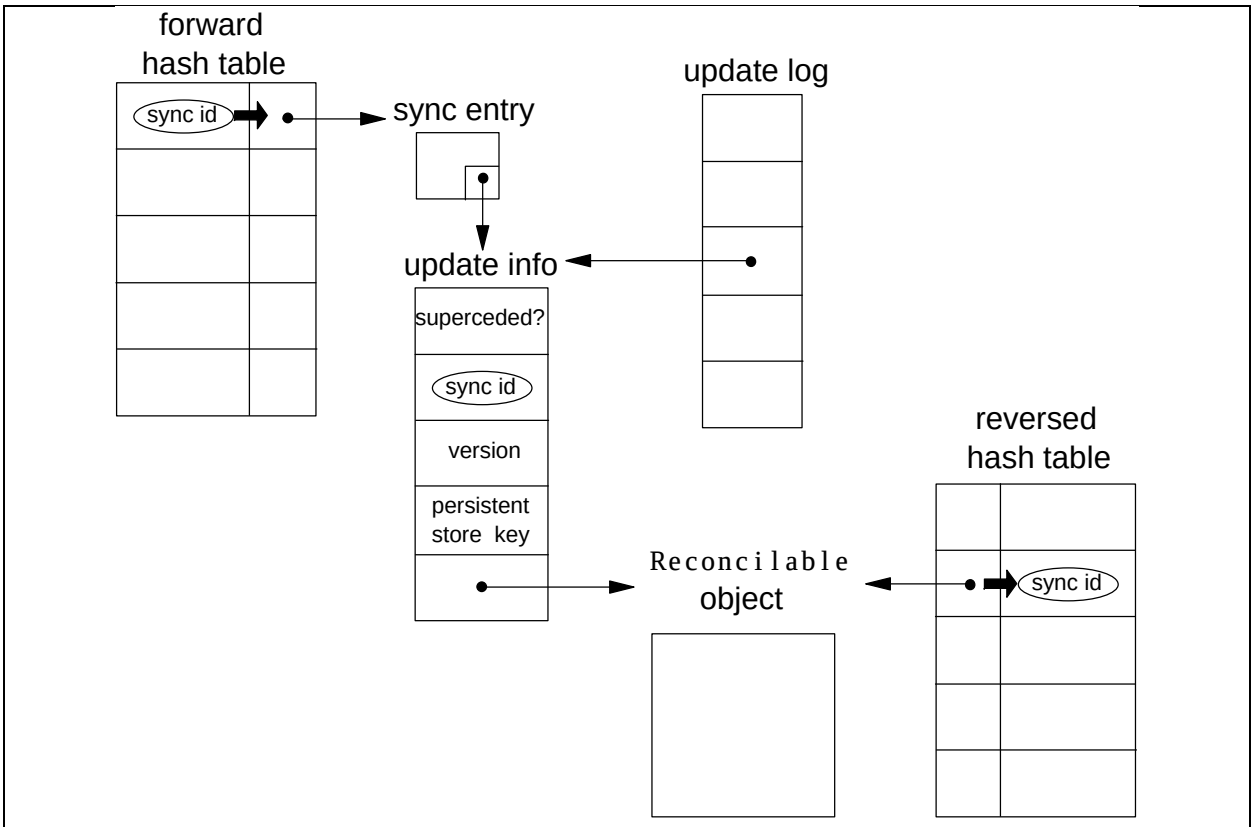

Figure 14. SyncStoreState data structures. A hash table maps sync IDs to sync entries and an inverted hash table maps Reconcilable references to sync IDs. An update log contains an ordered sequence of update-info objects, including an update-info object describing the most recent update for each sync ID in the hash table. An update-info object includes a sync ID, a version, a persistent-store key, and, except in the case of deletions, a reference to the Reconcilable object. Each sync entry includes a reference to the corresponding update-info object.

7.2 Opening and Closing Sync Stores

Our implementation of the StoreManager class maintains a persistent record of the names of data collections stored persistently on the device. (A system property identifies the directory in which these names are stored. The persistent representation of each data collection resides in a subdirectory of this directory whose path name corresponds to the hierarchically structured data-collection name.) The StoreManager class keeps a record of these data-collection names solely to implement the framework method that returns these names to the caller. The StoreManager method open simply validates that the data-collection name passed to it is well-formed, iterates over the installed SyncStoreFactory objects until a factory returning a nonnull result is found, and adds the name of a successfully opened new data collection to its persistent record. The parameters to the SyncStoreFactory construction method include any attributes that the application passes to open, to constrain the choice of sync-store implementations; an *exclusive flag* indicating whether the store should be opened for exclusive access; and a *creation flag* indicating the desired behavior—either throwing an exception or creating a new, empty data collection—if the named data collection does not exist.

Our implementation includes one class implementing the SyncStoreFactory interface. This class, SyncStoreFactoryImpl, constructs and returns SyncStoreHandle objects. A data collection is *active* when there is at least one open SyncStoreHandle object for it; there is one SyncStoreState object for every active data collection. Through a static hash table, the SyncStoreFactoryImpl class tracks the set of currently active data collections and their SyncStoreState objects. The constructor for SyncStoreHandle takes a SyncStoreState reference as a parameter. To open a currently inactive or nonexistent data collection, the factory first invokes a SyncStoreState constructor, then passes the resulting SyncStoreState reference to the SyncStoreHandle constructor and returns the resulting SyncStoreHandle reference. To open an already active data collection, the factory finds the data collection's SyncStoreState object in the hash table, passes it to the SyncStoreHandle constructor, and returns the resulting SyncStoreHandle reference. The SyncStoreHandle constructor calls a method of its SyncStoreState object to register itself with that object, allowing the SyncStoreState object to track the number of open handles it has.

Figure 15 depicts the logic for opening a data collection. When asked to open a sync store, SyncStoreFactoryImpl first checks that any attributes passed to it are consistent with the SyncStoreHandle implementation, then determines the path name for the persistent representation of the named data collection. A subdirectory with this path exists if and only if the named data collection already has a representation in persistent storage. If the data collection already has a persistent representation, the factory enters a synchronized block where, with exclusive access to the factory's hash table, the executing thread determines whether the data collection is already active. If the data collection is already active, the factory extracts the corresponding SyncStoreState object from the hash table and ensures that exclusive access was not requested and that the data collection is not already open for exclusive access. If the data collection is not already active, the factory constructs a new SyncStoreState object from the persistent representation of the data collection. If the data collection does not already have a persistent representation, the factory verifies that the create flag is set, creates the subdirectory corresponding to the data-collection name, and constructs a SyncStoreState object for a new, empty data collection. The SyncStoreState object obtained in any of these cases is then used to construct the SyncStoreHandle object that is returned.

The static hash table declared in class SyncStoreFactoryImpl controls access to the persistent representation of a data collection from within the Java virtual machine (JVM) that loaded the class. However, to preserve consistency between a SyncStoreState object in a given virtual machine and the corresponding persistent representation, it is necessary to ensure that multiple virtual machines, each loading the SyncStoreFactoryImpl class and maintaining its own static hash table, do not activate the same data collection simultaneously. Separate virtual machines can only communicate through shared system resources, such as the file system or socket connections to a lock-granting server running on the device. We experimented with both of these approaches and found the use of the file system to be about 35 times faster. In contrast to the Unix convention of creating a zero-byte file to indicate that some resource is locked, we create a zero-byte file to indicate that a resource is *unlocked*. The delete method of class java.io.File attempts to delete a file and returns a boolean result indicating whether or not the attempt succeeded, providing us with the equivalent of a test-and-set primitive. Whenever the factory is about to activate an existing data collection, it first attempts to obtain a *JVM lock* for

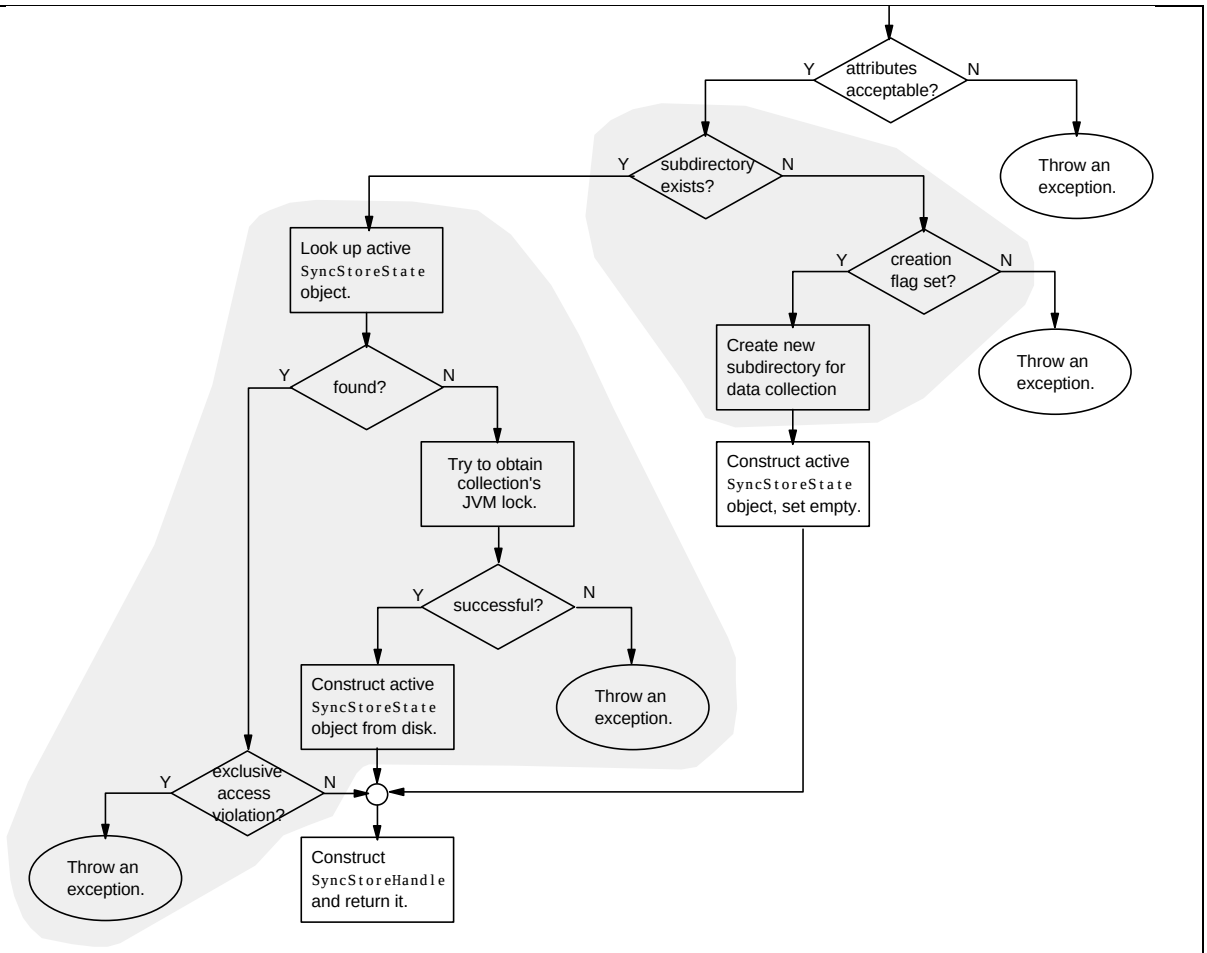

Figure 15. Construction of a new `SyncStoreHandle` object. The shaded areas represent critical regions. The one in the lower left, enforced by a `synchronized` block, ensures that at most one thread at a time within a given virtual machine accesses the static hash table of that virtual machine's `SyncStoreFactoryImpl` class. The one in the upper right, enforced by the system-wide factory lock, ensures that at most one virtual machine at a time accesses the persistent-storage directory structure. The JVM lock reserves use of a particular data collection for a particular virtual machine.

that collection by deleting a zero-byte file named "`$jvm_lock`" from the directory containing the persistent representation of the data collection. If this attempt fails, the factory throws an exception, because the data collection is presumed to be in use by another virtual machine. The file is recreated when the data collection is deactivated. When a new, empty data collection is created, no JVM-lock file is placed in its subdirectory. Thus the data collection comes into existence already locked. The JVM-lock file is created for the first time when the newly created data collection is first deactivated.

A similar file-based lock, called the *system-wide factory lock*, is used to prevent race conditions in which two threads, possibly in different virtual machines, simultaneously notice that a data collection with a given name does not exist, and try to create it. The statements that test for the existence of the data collection's subdirectory, and create it if it does not already exist, are executed in a critical region enforced across all virtual machines accessing a given persistent-storage directory subtree. A single zero-byte file, residing in the root directory of this

subtree, is deleted before the test is performed, and recreated after either the data collection's subdirectory is found to exist, or is found not to exist and is created. Since this lock is held only briefly and contention is expected to be rare, a thread that fails to obtain this lock simply spins and tries again after sleeping for a short period of time.

An application closes a sync store by invoking the `SyncStore` method `close`. The `SyncStoreHandle` implementation of this method calls a method of its `SyncStoreState` object to deregister itself; this method call decrements the `SyncStoreState` object's count of open handles and returns the new count. If the count has gone to zero, the update-log file is compacted by rewriting it with all superseded updates removed, then a method of the `SyncStoreFactoryImpl` class is called to deactivate the data collection by removing the corresponding entry from the `SyncStoreFactoryImpl` hash table of active `SyncStoreState` objects and releasing the JVM lock (i.e., recreating the corresponding zero-byte file). Regardless of the count, a sync-store event is generated with the `SyncStoreHandle` object as the source and, after all notifications have been performed, all sync-store-event and sync-object-event listeners added through this `SyncStoreHandle` object are removed. (The deregistration of the `SyncStoreHandle` object, the test of the new open handle count, and the possible deactivation all take place within a `synchronized` block for the same object that controls concurrent access to the hash table upon the opening of a sync store.)

An application can remove the persistent representation of an inactive data collection by passing its data-collection name to a static method of the `StoreManager` class. This method opens the data collection for exclusive access, casts the resulting `SyncStore` object to `SyncStoreUpdater`, invokes the `SyncStoreUpdater` method `destroy`, and prunes the `StoreManager` persistent record of data-collection names. The `destroy` method invokes a `SyncStoreFactoryImpl` method to remove the entire directory subtree corresponding to the deleted data collection.

7.3 Sync-Store Operations

7.3.1 Construction of Sync-Store States

By examining the creation flag passed to it, the `SyncStoreState` constructor determines whether a new, empty data collection is to be created, or whether the state of an existing data collection is to be constructed from its image in persistent storage. A `SyncStoreState` object for a new, empty data collection is constructed simply by setting each component of the object to an appropriate initial value, then writing a persistent image of the initial state.

The representation of an existing data collection in persistent storage has three parts:

- a *state file* holding a byte-stream representation of a *state-variables object*, which contains the replica registry and singleton variables such as the replica ID, summary version, and update counter
- a *log file* containing a sequence of byte-stream representations for each update-info object in the update log, without any information about the contents of `Reconcilable` objects

- a mapping, implemented by the persistent-store manager, from persistent-store keys to the contents of corresponding `Reconcilable` objects

The first step in creating a `SyncStoreState` object for an existing data collection is to create a state-variables object and fill it in from the byte-stream representation in persistent storage. The second step is to read through the sequence of update-info byte-stream representations. For each item in the sequence, a new update-info object is created and initialized from its byte-stream representation, the object is appended to the in-memory update-log vector, a new sync entry referencing the object is created, and a new hash-table entry, mapping the sync ID found in the update-info object to the sync entry, is inserted in the forward hash table.

The contents of a `Reconcilable` object are not read from persistent storage when a `SyncStoreState` object is created, but only when they are needed. The `Reconcilable` reference in each newly created update-info object is left null, and the reverse hash table is left empty. When the `get` method of a sync entry or a `SyncStore` object is called, and the corresponding update-info object is found to have a null `Reconcilable` reference, the persistent-store key in the update-info object is passed to the persistent-store manager, which returns a reference to a new `Reconcilable` object with the appropriate contents. This reference is inserted in the update-info object and a new entry is added to the reverse hash table, mapping the reference to the sync ID found in the update-info object.

7.3.2 Associating Updates with Sync IDs

A primitive common to the implementation of both the application-invoked and synchronizer-invoked operations on sync stores is the association of a new update-info object with a given sync ID. The primitive is invoked whenever the state associated with a given sync id changes. First, the sync ID is looked up in the forward hash table to find the associated sync entry, if any. If an existing sync entry is found, the update-info object previously referenced by that sync entry is marked as superseded; otherwise, a new sync entry is created and inserted into the forward hash table. The sync entry is set to reference the new update-info object and the new update-info object is appended to the update log. Figure 16 illustrates the effect of this operation in the case where there was a previously existing sync entry. Update-info objects are appended to the update log in introduction order (see Section 3.2).

7.3.3 Operations Invoked by Applications

The `get` method of the `SyncStore` interface uses the sync ID passed to it to find the corresponding sync entry in the forward hash table, then calls the sync entry's `get` method to retrieve the corresponding `Reconcilable` object reference. The `get` method of the `SyncEntry` interface returns the `Reconcilable` reference in its sync entry's update-info object, obtaining it first from persistent storage, as described at the end of Section 7.3.1, if the reference had been null.

When one of the `SyncStore` methods `put`, `markForSynchronization`, or `delete` is called, a new update-info object, describing the change, is created and associated with the sync ID passed to the method. (The corresponding sync entry is created in the case of `put`, and already exists in the case of `markForSynchronization` and `delete`.) In addition, the `put` method inserts

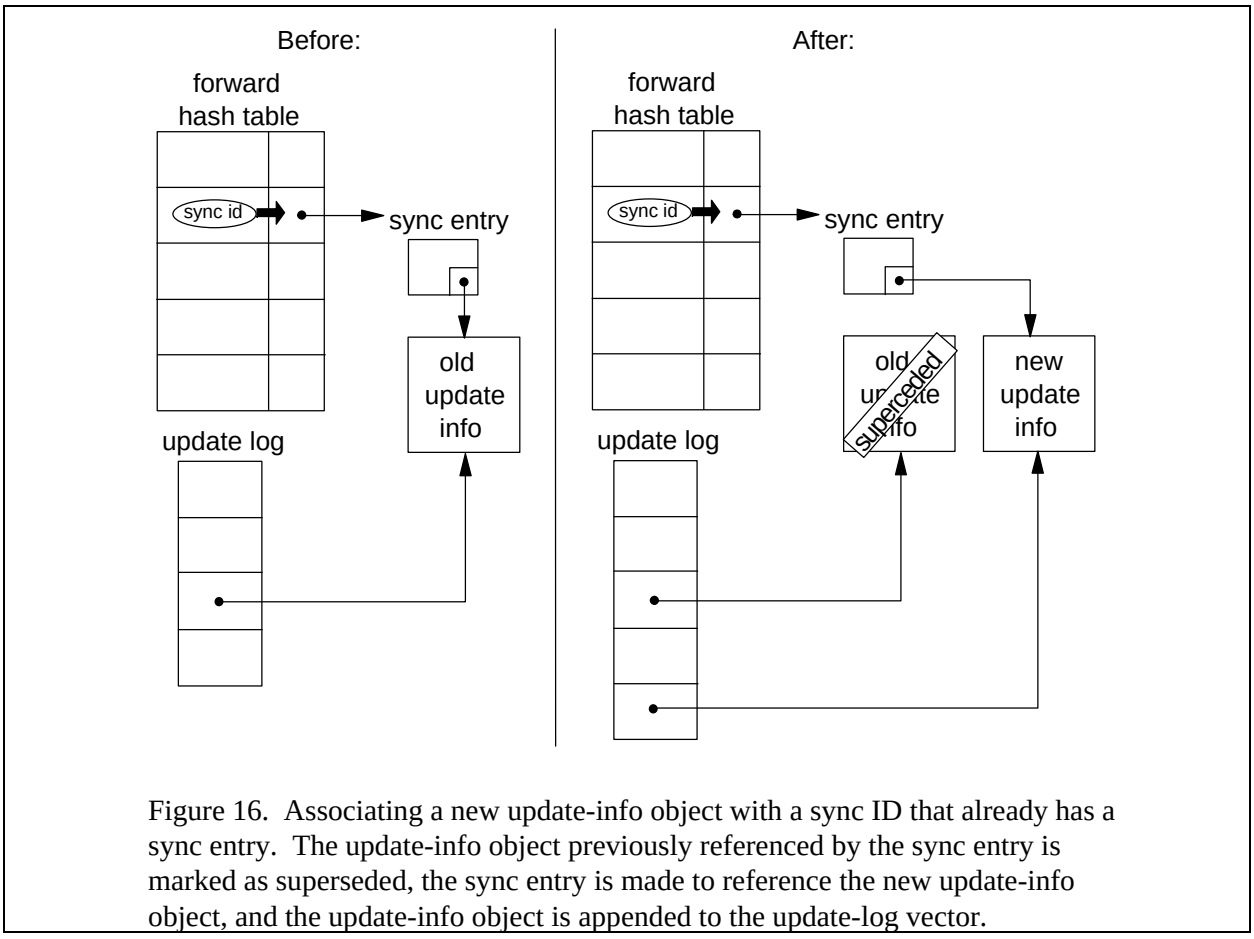

Figure 16. Associating a new update-info object with a sync ID that already has a sync entry. The update-info object previously referenced by the sync entry is marked as superseded, the sync entry is made to reference the new update-info object, and the update-info object is appended to the update-log vector.

an entry in the reverse hash table, mapping the newly inserted `Reconcilable` object to the sync ID; the `put` and `delete` methods mark the sync entry as needing to be flushed. A sync-object event is generated. The `SyncVersion` value in the new update-info object is computed by incrementing the update counter and modifying a copy of the previous version to reflect the new update-counter value. In the case of `put`, the previous version is one indicating no updates by any replica; in the case of `markForSynchronization` and `delete`, the previous version is taken from the update-info object previously associated with the sync ID.

The `markForFlushing` method uses the sync ID passed to it to find the corresponding sync entry in the forward hash table, marks the sync entry as needing to be flushed, and generates a sync-object event. The `markAsUpdated` method combines the actions of the `markForSynchronization` and `markForFlushing` methods. The `flush` method, using a marker that indicates the last update-info object in the in-memory log vector whose byte-stream representation was written to the log file, appends the byte-stream representations of the new, still unwritten, update-info items to the log file, and updates the marker; writes the byte-stream representation of the state-variables object to the state file; and iterates over all sync entries in the forward hash table, invoking the persistent-store manager (with the key found in the sync entry's update-info object) to save the state of any `Reconcilable` object whose sync entry is flagged as needing flushing. However, when the last `SyncStoreHandle` object for a `SyncStoreState` is closed, the log file is rewritten in its entirety, removing all superseded updates.

The SyncStore method getSyncIdOf takes a Reconcilable reference as a parameter and returns the sync ID, if any, with which that reference is associated, or null if the sync store does not currently contain that reference. (The put method checks that a Reconcilable reference being inserted in the sync store is not already there, ensuring that a Reconcilable reference corresponds to at most one sync ID.) If the Reconcilable reference that an application passes to getSyncIdOf is already in the sync store, then either the application must have had access to the reference and passed it to a call on put, or the reference must have been generated by the sync store and returned to the application by a call on the SyncStore or SyncEntry method get. Even though the reverse hash table starts out empty, an entry is made in it every time put is called, and the first time get is called for a given sync ID. Thus the getSyncIdOf method can always obtain the required result simply by looking up the Reconcilable reference in the reverse hash table.

7.3.4 Operations Invoked by Synchronizers

Synchronizers invoke the SyncStoreUpdater methods applyUpdate during a receiving phase and generateUpdates during a sending phase. The applyUpdate method takes a SyncUpdate reference as a parameter and the generateUpdates methods returns an iterator over SyncUpdate references. Our implementation has one abstract class, SyncUpdateImpl, implementing the framework's SyncUpdate interface, and three concrete subclasses directly or indirectly extending SyncUpdateImpl:

- ObjectContentsSyncUpdate. An object of this class represents either an insertion or a modification of an existing object, depending on whether an object with the same sync ID already exists in the receiving sync store. The object contains a sync ID, a version, and a copy of a Reconcilable object.
- DeletionSyncUpdate. An object of this class represents a deletion. The object contains a sync ID and a version.
- VersionSyncUpdate. An object of this class represents an update to the version, but not the contents, of a Reconcilable object. The object contains a sync ID and the new version.

Section 7.3.4.1 discusses the application of updates and Section 7.3.4.2 discusses the generation of updates, particularly those of class VersionSyncUpdate.

7.3.4.1 Applying Updates

As Figure 17 illustrates, the SyncStoreUpdater method applyUpdate invokes an abstract method of the SyncUpdateImpl object passed to it, and this invocation dispatches, based on the class of that object, to a specialized SyncStoreUpdater method that applies that kind of update. As Figure 18 indicates, the behavior of the specialized method depends on whether there is currently a sync entry with the same sync ID as the update, and if so, whether the version associated with that sync entry is earlier than, later than, equal to, or conflicting with the version associated with the update.

Let us first consider the case in which there is no sync entry with the same sync ID. The method applying an object-contents update inserts the Reconcilable object from the update into the local store, in the manner of the put method, but using the version contained in the update.

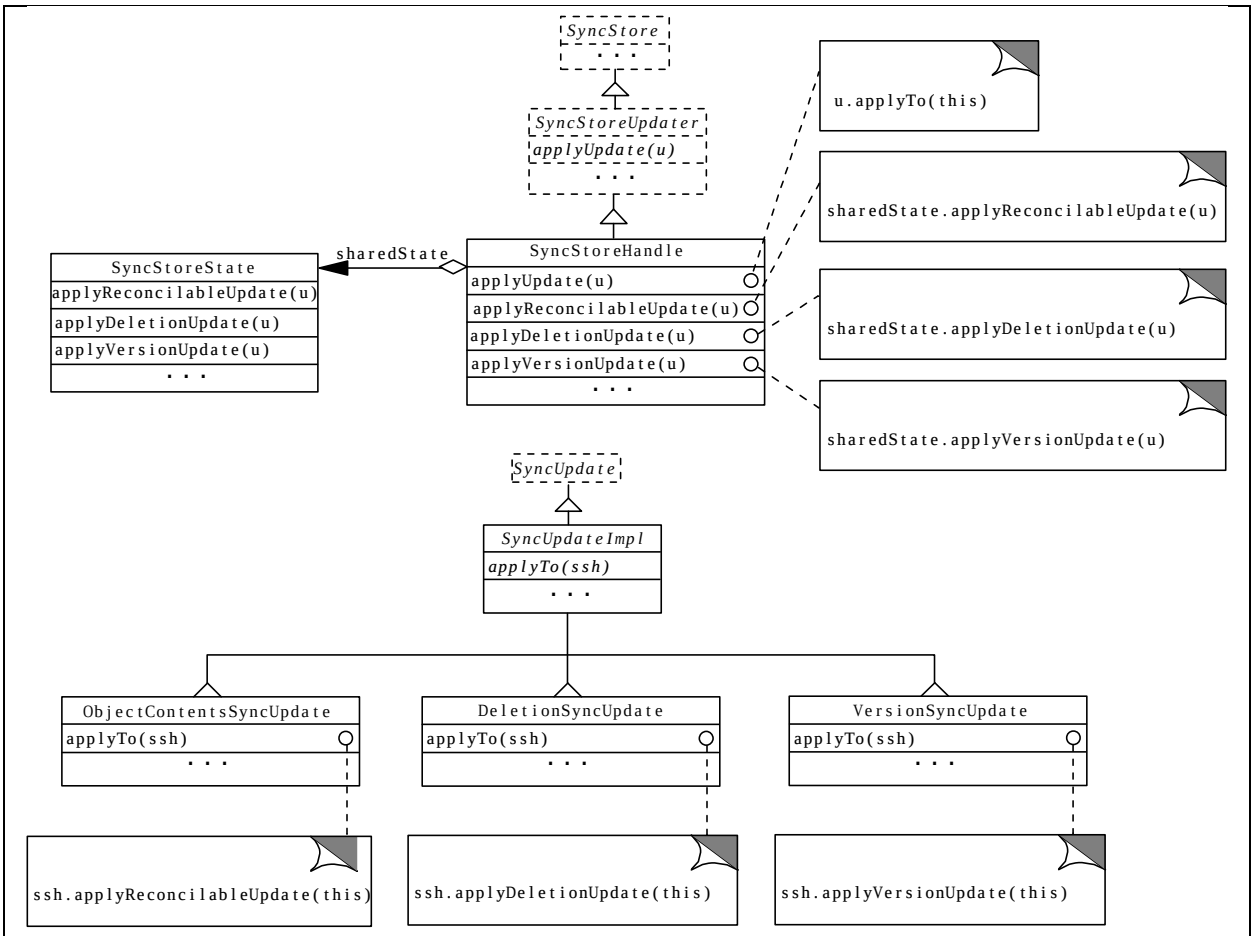

Figure 17. Applying updates. The `SyncUpdate` object passed to the `SyncStoreHandle` method `applyUpdate` is cast to `SyncUpdateImpl`, and its `applyTo` method is called with the `SyncStoreHandle` object as an argument. Depending on the class of the object, `applyTo` invokes one of the `SyncStoreHandle` methods `applyReconcilableUpdate`, `applyDeletionUpdate`, or `applyVersionUpdate`. Each of these three methods invokes a corresponding method of the `SyncStoreState` class.

The method applying a deletion update inserts a deletion sync entry into the local store, again taking the version from the update; this case arises when an object is created in a remote replica and then deleted before the creation is propagated to the local store. (The method applying a version update will be invoked only in the following unusual circumstance:

- The local store initially contains a deletion sync entry with the same sync ID.
- The remote store reconciles a conflict with this deletion by choosing to keep the deletion.
- After a determination that the deletion has been propagated to all replicas, the corresponding sync entries are removed from all replicas, as part of a scheme that will be described in Section 7.5.
- The remote replica propagates the result of the reconciliation back to the local replica as a version update containing the version of the reconciliation.

state of update-info object		kind of arriving update object		
		object-contents	**deletion**	**version**
none (no local sync entry)		Insert object from remote update.	Insert sync entry for deletion.	Insert sync entry for deletion.
local version later than or equal to remote version	**undeleted**	Ignore remote update.	Ignore remote update.	Ignore remote update.
	deleted	Ignore remote update.	Ignore remote update.	Ignore remote update.
local version earlier than remote version	**undeleted**	Replace local object contents with contents in remote update, replace version with remote version.	Change sync entry to a deletion sync entry, replace version with remote version.	Replace version with remote version.
	deleted	Reinsert object, replace version with remote version.	Replace version with remote version.	Replace version with remote version.
local and remote versions conflict	**undeleted**	Invoke `reconcile` method of local object, merge local and remote versions.	Invoke `reconcileWithDelete` method of local object, merge local and remote versions.	Merge local and remote versions.
	deleted	Invoke `reconcileWithDelete` method of object copy in remote update, merge local and remote versions.	Merge local and remote versions.	Merge local and remote versions.

Figure 18. Rules for applying remote updates to the local sync store. For each kind of update, the action to be taken depends on whether or not the local store already contains a sync entry for the sync ID contained in the update. If there is a sync entry, the action to be taken also depends on how the

The appropriate action for the local store is to insert a deletion sync entry into the local store, taking the version from the update.)

If there is already a sync entry in the local sync store, but its version is later than or equal to the version in the arriving update object, the update represents “old news”: Either the update was already received earlier at this sync store, or it was received and later superseded at another replica, and the superseding update was already received at this sync store. In either case, the appropriate action is to ignore the update. Our synchronizer implementations avoid sending updates that were old news at the start of the synchronization, so this case arises only if multiple synchronizations are in progress at the same time, and the application of an update from one remote synchronizer turns some update that has been generated, but not yet sent, by the other remote synchronizer into old news. Since our sync-store and synchronizer implementations are independent of each other, our sync-store implementation is designed to work even with synchronizer implementations that generate old-news updates on a regular basis.

If there is already a sync entry in the local sync store, and its version is earlier than the version in the arriving update object, the local `Reconcilable` object should be set to the state specified in the update and the sync entry's version should be advanced to the version specified in the update. Sometimes, this will entail replacing an object-contents update-info object with a deletion update-info object or (if a deletion previously present in the local store was reconciled in some remote replica with a conflicting update by keeping the object, and the update now being applied is the result of that reconciliation) replacing a deletion update-info object with an object-contents update-info object. If the update object to be applied is a version update object, the corresponding update-info object is assumed to specify the correct state already, and only the version is updated. (The local update-info object may be an object-contents update-info object, and the update may be the result of an update-update conflict reconciliation that preserved the contents of the `Reconcilable` object; or the update-info object may be a deletion update-info object, the update may be the result of an update-deletion conflict reconciliation that kept the deletion.)

Finally, if there is already a sync entry in the local sync store, and its version conflicts with the version in the arriving update object, the conflict is resolved and the two versions are merged to produce a new version, later than both, for the result of the resolution. The merged version is computed by incrementing the local sync store's version counter, constructing the earliest version later than both the local and remote versions, and advancing this version to reflect the new version-counter value. This version characterizes the result of the reconciliation as a new, locally executed, modification, later than the two conflicting modifications that it replaces. (Even if the conflict is reconciled by preserving the effect of one of the two conflicting updates, the version is updated to reflect that both of the conflicting updates were seen, and that the current state of the object has taken both conflicting updates into account.) If the sync entry references an object-contents update-info object and the update is an object-contents update, the conflict is reconciled by calling the `reconcile` method of the `Reconcilable` object in the local store, passing the `Reconcilable` object in the remote update as a parameter. If the sync entry references an object-contents update-info object and the update is a deletion update, the conflict is reconciled by calling the `reconcileWithDelete` method of the `Reconcilable` object in the local store. If the sync entry references a deletion update-info object and the update is an object-contents update, the conflict is reconciled by calling the `reconcileWithDelete` method of the `Reconcilable` object in the remote update. (A `boolean` parameter to `reconcileWithDelete` distinguishes between the two ways in which the method can be called.) If the sync entry references a deletion update-info object and the update is a deletion update, the conflict is a delete-delete conflict, which is automatically reconciled by keeping the object deleted (but merging the versions of the two conflicting deletions). If the arriving update is a version update, then a remote update preserving a `Reconcilable` object in some state *x* conflicts with an update, already reflected in the local store, placing that object in some other state *y*. Since the update, or sequence of updates, that transformed the object to state *y* began with the object in state *x*, we presume that the same sequence of updates would have occurred even if the update preserving the object in state *x* had occurred earlier, at the same replica, instead of as a conflicting update. Therefore, we resolve the conflict by leaving the object state unchanged (perhaps in a deleted state), in effect applying the version update and then the updates with which it conflicts; versions are merged as for any reconciliation.

7.3.4.2 Generating Updates

The `generateUpdates` method simply iterates over the nonsuperseded entries in the update-log vector, constructing a collection that contains a corresponding `SyncUpdate` object for each update-info object with an applicable version, and returns an iterator over that collection, in introduction order. (Introduction order was defined in Section 3.2. The update log lists update-info objects in introduction order.)

The version resulting from the reconciliation of a conflict is always later than the version of the update received from the remote replica. Thus the results of the reconciliation will always be propagated back to the remote replica the next time that the local replica sends updates to it (unless the remote replica receives the results of the reconciliation from some intermediary replica first). If the integer code returned by a call on `reconcile` asserts that the local `Reconcilable` object was set to the state of the remote `Reconcilable` object, it suffices for the local replica to send a `VersionSyncUpdate` object, containing the new version but not the `Reconcilable` object contents, back to the remote replica. When synchronizing with some third replica, however, it may be necessary to send a full `ObjectContentsSyncUpdate` object.

Generating the appropriate kind of update is not a simple matter. The sync store generates updates in response to a call on its `generateUpdates` method by the synchronizer. The synchronizer has the context to determine that a call on `generateUpdates` takes place in the sending phase of a synchronization whose earlier receiving phase triggered the reconciliation. Similarly, the synchronizer is aware of the identity of the remote replica on whose behalf `generateUpdates` is being called. However, the MNCRS data-synchronization framework does not provide a means to convey this information to the sync store. (In particular, the `generateUpdates` method does not have parameters through which this information can be passed. In retrospect, the framework ought to have included such a parameter.) Since our goal was to develop a synchronizer implementation that works with other, independently developed, sync-store implementations, and to develop a sync-store implementation that works with other, independently developed synchronizer implementations, our synchronizer and sync-store implementations communicate with each other only through the features of the framework.

Our solution is to exploit the parameter to `generateUpdates` that specifies the lower bound on the versions for which updates should be generated (the *starting version*). A synchronizer always invokes `generateUpdates` with a starting version that is earlier than or equal to the summary version of the receiving sync store. (Otherwise, the synchronization could bypass current updates in the sending store not yet reflected in the receiving store, violating the introduction-order requirement described in Section 3.2.) It follows that the receiving sync store has already seen any (nonsuperseded) updates with versions earlier than or equal to the starting version passed to the sending sync store's `generateUpdates` method.

When the integer code returned by a call on `reconcile` asserts that the local `Reconcilable` object was set to the state of the remote `Reconcilable` object, the update-info object we log for the reconciliation is a *version update-info object*, a special form containing not only a sync ID, new version, persistent-store key, and `Reconcilable` reference, but also the previous version. When `generateUpdates` encounters an update-info object of this form in the update log, with a current version later than or conflicting with the starting version, it compares the previous version in the update-info object with the parameter specifying the lower-bound

version. If the previous version in the update-info object is earlier than or equal to the starting version, the receiving sync store has already received the object state corresponding to the previous version, so a `VersionSyncUpdate` object is generated. Otherwise, an `ObjectContentsSyncUpdate` object is generated.

The decision about what kind of update object to generate is based not on the correspondence between synchronization phases or on the identity of the recipient, but on whether or not the recipient has yet received the old object state. Consider the following sequence of synchronization phases:

1. Replica *A* sends updates to Replica *B*, which receives object *x* for the first time.
2. Replica *A* sends updates to Replica *C*, which already has a conflicting update for object *x*. Replica *C* reconciles the conflict by keeping the object state it received from *A*.
3. Replica *C* sends updates to Replica *A*.

Replica *C* can send a `VersionSyncUpdate` object for *x* in step 3, since Replica *A* has already seen the same object state, and indeed our version-based approach generates such an object. An approach based on the phases of a synchronization or on the identity of the replica would generate the larger `ObjectContentsSyncUpdate` object.

7.3.5 The Role of Sync-Store Handles

Most `SyncStoreHandle` methods simply verify that the sync store is not closed, then invoke a corresponding method of the shared `SyncStoreState` object, or of a replica-registry or listener-registry object associated with that `SyncStoreState` object. `SyncStoreState` methods accessing the hash tables, update log, summary version, or update counter are all declared `synchronized`, to prevent race conditions. However, the framework requires certain `SyncStoreHandle` methods to notify listeners of sync-object or sync-store events. These notifications always take place in the `SyncStoreHandle` method, after the corresponding `SyncStoreState` method has returned. Thus the shared `SyncStoreState` object does not remain locked as applications' listener methods are executed.

Different calls on `applyUpdate` result in different kinds of events, and some result in no event at all. Whether an event should be generated, and if so, what kind, can only be determined inside the synchronized `SyncStoreState` methods that `applyUpdate` indirectly invokes. These methods each return a reference to an object describing the kind of event `applyUpdate` should generate, or `null` if no event is to be generated. See Figure 19.

7.4 Implementation of Versions

Section 3.2 explained that by requiring updates to be transmitted in introduction order, the MNCRS data-synchronization framework *allows* versions to be represented as version vectors [Par83]. An early draft of the framework specified the behavior of `SyncVersion` methods directly in terms of a version-vector model. In this model, a `SyncVersion` object maps globally unique replica identifiers to values of the local update counters of the identified replicas. The mapping is defined for all possible replica identifiers, including those that have never been generated, or have never been seen by a given sync store; such replica identifiers are mapped to

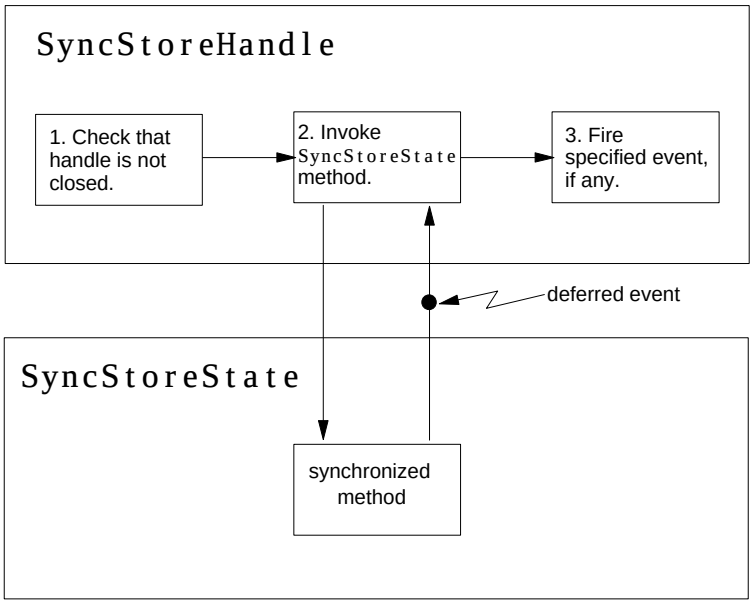

Figure 19. A typical SyncStoreHandle method. The method checks that the handle is not closed, then invokes a synchronized method that manipulates the corresponding SyncStoreState method in a critical region and returns a *deferred event*, indicating which event, if any, should be fired by the SyncStoreHandle method upon return from the synchronized method. Deferred events avoid the invocation of event listeners from within a synchronized block.

zero by default. The version vector corresponding to the insertion of a newly created object in replica r maps r to the just-incremented update counter of replica r, and maps all other replica identifiers to zero. Let $v[i]$ be the value to which version vector v maps replica identifier i. Versions v_1 and v_2 are merged upon resolution of a conflict by constructing a new version v_m such that $v_m[i] = \max(v_1[i], v_2[i])$ for each replica identifier i. A version v is advanced to reflect an update or reconciliation on replica i by constructing a new version v_a such that $v_a[i]$ is the just-advanced update counter on replica i and $v_a[j] = v[j]$ for $j \neq i$. Two version vectors v_1 and v_2 are equal if and only if $v_1[i] = v_2[i]$ for every replica identifier i. Version vector v_1 is earlier than v_2 (and v_2 is later than v_1) if and only if $v_1 \neq v_2$ and $v_1[i] \leq v_2[i]$ for every replica identifier i. Two version vectors v_1 and v_2 conflict if and only if there exist replica identifiers i and j such that $v_1[i] < v_2[i]$ and $v_1[j] > v_2[j]$. (Rather than update-counter readings, the original formulation of version vectors in [Par83] uses successive integers for a given component of a given object's version vector, so that the version-vector component is equal to the number of times that object was updated at the corresponding site.)

Associating a version vector with each object in a store consumes a considerable amount of storage. Furthermore, the amount of space required to represent a given object's version vector grows with the number of replicas that have ever updated that object. Sync-store implementations tailored to environments with restricted synchronization topologies can track versions and detect conflicts with more space-efficient data structures. For example, less information needs to be stored if there is a central replica, and all other replicas synchronize only

with the central replica; even less has to be stored if there are two replicas that only synchronize with each other. Therefore, later drafts of the MNCRS data-synchronization framework incorporated a more abstract specification of versions, stipulating only that the SyncVersion interface provides methods to compare SyncVersion values. The intent was to allow a wide variety of version implementations. Version vectors remain the natural implementation for fully general sync stores supporting arbitrary peer-to-peer synchronization.

7.4.1 Defining a Universal Version Abstraction

Our implementation effort took on a more ambitious goal: to implement versions in a variety of ways, appropriate for different synchronization topologies, but to define a common set of operations meaningful for all of these implementations, so that a single sync-store implementation could execute the same algorithms for all version implementations. The implementation includes an abstract class SyncVersionImpl, implementing the framework's SyncVersion interface and providing operations for reading and writing byte-stream representations of sync versions. Each byte-stream representation is prefixed with a one-byte tag identifying the implementation class of a particular version object; the methods provided by SyncVersionImpl examine this tag and dispatch to reading and writing methods of the individual implementation classes. SyncVersionImpl has two subclasses: VersionVector, for general peer-to-peer synchronization topologies, and CentralizedSyncVersion, for central-replica topologies.

In addition to the comparison operations declared in the SyncVersion interface, we identified the following common methods needed by our sync-store implementation:

- a method to advance a version locally, i.e., to return a new version that is based on a given version, but also incorporates a given new value for the local update counter
- a method to merge two versions, i.e., to construct the earliest version later than or equal to two given versions
- a method combining the actions of the previous two methods, i.e. merging two given versions and then advancing the result locally to incorporate a given local update-counter value
- a method returning the version earlier than all other versions
- a method returning a pure local version, corresponding to a local update with a given update-counter value and no remote updates
- methods to translate between various representations of the same version value

Each of these methods returns a SyncVersionImpl result, allowing us to apply the Abstract Factory pattern of [Gam95] once again. Rather than declaring these methods in SyncVersionImpl, we declare them in an abstract class named SyncVersionFactory. (Indeed, the method to return the earliest possible version and the method to return a pure local version are not naturally associated with an existing version object, so they would be naturally declared in SyncVersionImpl as static methods; but static methods can not be overridden in different ways by different subclasses of SyncVersionImpl.) There are three concrete subclasses of SyncVersionFactory—one that generates VersionVector objects, one that generates

CentralizedSyncVersion objects on a central replica, and one that generates CentralizedSyncVersion objects on subordinate replicas. A parameter to the SyncStoreState constructor specifies the *network role* of the replica—either a peer, a central replica, or a subordinate replica in a central-replica topology. Based on this parameter, the SyncStoreState constructor creates a version-factory object of the appropriate class, and all of the SyncStoreState object's manipulations of versions use this version-factory object.

7.4.2 Implementation of Version Vectors

Abstractly, a VersionVector object maps replica identifiers to update-counter values, and its methods follow the mathematical rules given in the first paragraph of Section 7.4. This mapping is implemented as a vector of pairs each consisting of a replica identifier and an update-counter value. There is one pair for each replica with a nonzero update-counter value. The pairs are sorted by replica identifier to facilitate component-by-component comparisons and the computation of component-by-component maxima.

7.4.3 Implementation of Centralized Versions

The implementation of the CentralizedSyncVersion class is more complex. Our scheme for maintaining versions in a central-replica topology is best explained by a series of transformations from a version-vector-based scheme. Suppose that we modify the central-replica sync-store implementation so that each new update received from another replica is marked as updated by the central replica as soon as it arrives. We say that the central replica *endorses* the arriving update. (Endorsement is equivalent to the central replica modifying each object for which it receives a new update by overwriting the object with itself.)

Since an endorsement does not change the contents of a Reconcilable object, it is recorded in the central replica's update log as a version update-info object. Suppose the endorsed update originated on subordinate replica *s*. When the central replica generates all updates with versions later than or conflicting with the summary version of *s*, a version update object will be generated from the version update-info object; for all other subordinate replicas, an object-contents update will be generated. Thus, when updates are transmitted to *s*, the version for the affected object is advanced to the new version that the central replica generated during the endorsement. When updates are transmitted to a subordinate replica other than *s*, both this new version and the object contents received from *s* are transmitted.

Let s_A and s_B be the replica identifiers of two subordinate replicas, and let c be the replica identifier of the central replica. Let v_A be the version vector for a given object at s_A, and v_c be the version vector for the same object at the central replica. Using the notation introduced at the beginning of Section 7.4, suppose $v_A[s_B] < v_c[s_B]$. This indicates that an update to the given object from s_B has reached the central replica, but the central replica has not yet propagated the update to s_A. As soon as the update reaches the central replica, it is endorsed, which has the effect of increasing the value of $v_c[c]$. Since the endorsement has not yet reached s_A, it must be the case that $v_c[c] > v_A[c]$. Thus

$$v_A[s_B] < v_c[s_B] \text{ only if } v_c[c] > v_A[c]. \quad (1)$$

Since $v_A[s_A]$ and $v_c[s_A]$ indicate the value of the update counter at s_A at the time of the last update to the given object at s_A, and since an update at s_A is recorded at s_A before it is recorded at any other replica, it is always the case that

$$v_A[s_A] \geq v_c[s_A]. \tag{2}$$

Since an update at any replica r other than s_A becomes known at s_A only after it becomes known at the central replica, it is always the case that

$$v_A[r] \leq v_c[r] \text{ for } r \neq s_A. \tag{3}$$

If v is a version vector and i and j are replica identifiers, let **proj**(v, i, j), the *projection* of v onto i and j, be the version vector that maps i to $v[i]$, j to $v[j]$, and all other replica identifiers to zero. It follows from (1), (2), and (3) that to determine whether v_A is earlier than, later than, equal to, or in conflict with v_c, it suffices to compare **proj**(v_A, s_A, c) with **proj**(v_c, s_A, c). Consider the four possible relationships between v_A and v_c:

- If v_A is earlier than v_c, then there is some replica i such that $v_A[i] < v_c[i]$, and for each replica j such that $i \neq j$, $v_A[j] \leq v_c[j]$. It follows from (2) that $v_A[s_A] = v_c[s_A]$, and that $i \neq s_A$. Letting i play the role of s_B in (1), it follows that $v_A[c] < v_c[c]$. Since $v_A[s_A] = v_c[s_A]$ and $v_A[c] < v_c[c]$, **proj**(v_A, s_A, c) is earlier than **proj**(v_c, s_A, c).
- If v_A is later than v_c, then there is some replica i such that $v_A[i] > v_c[i]$, and for each replica j such that $i \neq j$, $v_A[j] \geq v_c[j]$. It follows from (3) that $i = s_A$, so $v_A[s_A] > v_c[s_A]$. Since $i = s_A$, $v_A[j] \geq v_c[j]$ for $j \neq s_A$, but $v_A[j] \leq v_c[j]$ for $j \neq s_A$ by (3), so $v_A[j] = v_c[j]$ for $j \neq s_A$. In particular, $v_A[c] = v_c[c]$. Since $v_A[s_A] > v_c[s_A]$ and $v_A[c] = v_c[c]$, **proj**(v_A, s_A, c) is later than **proj**(v_c, s_A, c).
- If $v_A = v_c$, then $v_A[s_A] = v_c[s_A]$ and $v_A[c] = v_c[c]$. Therefore **proj**(v_A, s_A, c) = **proj**(v_c, s_A, c).
- If v_A conflicts with v_c, then there must be a replica i such that $v_A[i] > v_c[i]$ and a replica j such that $v_A[j] < v_c[j]$. By (3), i can only be s_A, so $v_A[s_A] > v_c[s_A]$. If $j=c$, then $v_A[j] < v_c[j]$ implies $v_c[c] > v_A[c]$. Otherwise, j is some subordinate replica s_B, so it follows from (1) that $v_c[c] > v_A[c]$. Since $v_A[s_A] > v_c[s_A]$ and $v_A[c] < v_c[c]$, **proj**(v_A, s_A, c) conflicts with **proj**(v_c, s_A, c).

Thus it suffices for a subordinate replica to maintain a two-component version vector for each object, with one component corresponding to itself and the other component corresponding to the central replica. The central replica maintains a full version vector for each object, but performs all version-vector comparisons by comparing each subordinate-replica version vector with the corresponding projection of its full version vector.

Since the replica identifier for the central replica is fixed, there are three variable pieces of information in a two-replica projection of a version vector—the central replica's update-counter value, the subordinate replica's update-counter value, and the subordinate replica's replica identifier. However, all version vectors stored on a given subordinate replica specify *that* replica's identifier, so it is wasteful to store the subordinate-replica identifier with each version vector on that device. Rather, we adopt three distinct representations for `CentralizedSyncVersion` values, illustrated in Figure 20:

1. a full version vector, associated with each object in the central replica
2. a pair consisting of the central-replica update-counter value and the subordinate-replica update-counter value, associated with each object in a subordinate replica and with each update sent from the central replica to a subordinate replica
3. a triple consisting of the central-replica update-counter value, the subordinate-replica update-counter value, and the subordinate-replica identifier, associated with each update sent from a subordinate replica to the central replica

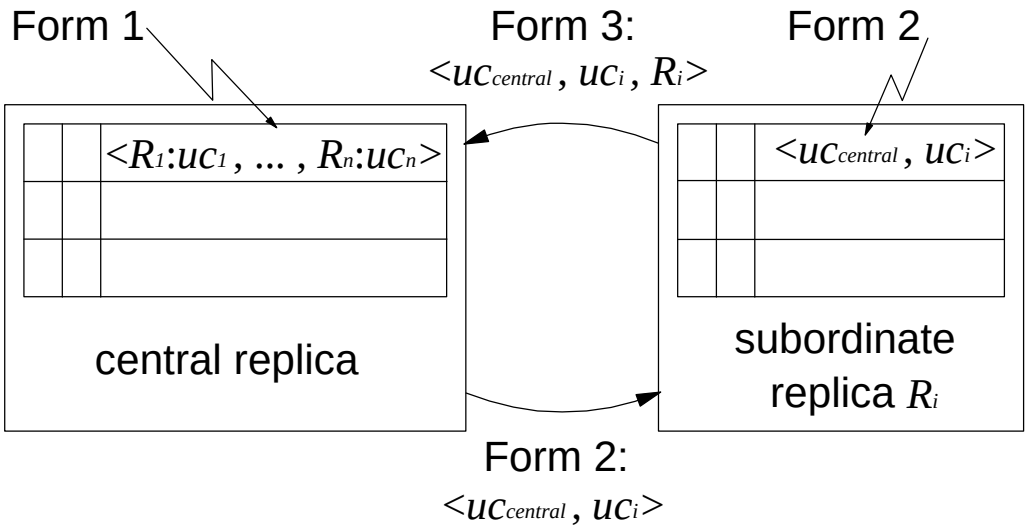

Figure 20. The three forms of `CentralizedSyncVersion` objects. R_i is a replica identifier, $1 \le i \le n$, and uc_i is a corresponding update-counter value. Form 1, a full version vector, resides in every sync entry of the central replica. Updates sent from the central replica to the subordinate replica are of form 2, which consists only of update-counter values for those two replicas, and this is the form in which versions are stored in subordinate-replica sync entries. Updates sent from a subordinate replica to the central replica are of form 3, which consists of update-counter values for those two replicas and the replica identifier of the subordinate replica.

`CentralizedSyncVersion` is an abstract class, with concrete subclasses for each of these forms, as shown in Figure 21. When an update is generated on a subordinate replica for transmission to the central replica, a method of the subordinate replica's version factory is invoked to create a `CentralizedSyncVersion` object of form 3 equivalent to a given `CentralizedSyncVersion` object of form 2. The replica identifier for the subordinate replica is passed as a parameter to the constructor for that replica's sync-version factory, and remembered by the sync-version factory for use in performing this transformation. When the `CentralizedSyncVersion` object of form 3 arrives at the central replica, a method of the central replica's version factory constructs an equivalent full version vector, in which components for replicas other than the central replica and the indicated subordinate replica are set to zero. The subordinate replica's summary version is also stored in form 2 on the subordinate replica, and translated to form 3 for transmission to the central replica. When the central replica generates updates for all update-info objects with versions later than or conflicting with some subordinate replica's summary version, a method of

the central replica's version factory examines that summary version to identify the subordinate replica for which the update is destined, and computes the appropriate projection of the update-info object's full version vector, to generate an update with a version of form 2.

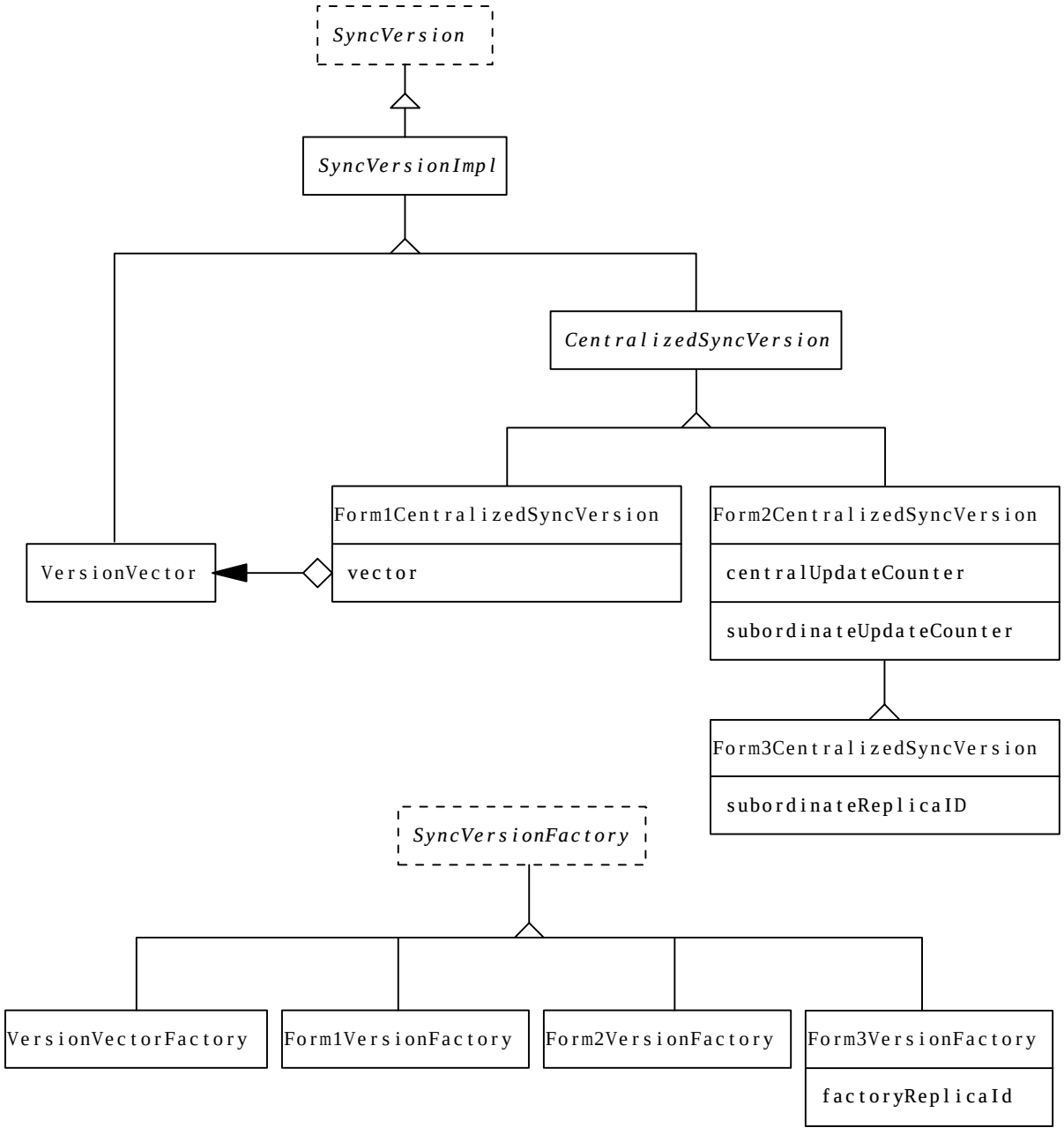

Figure 21. Interfaces and classes for versions and version factories.

For a pair of replicas that synchronize only with each other, all versions can be represented in form 2, since the identity of both replicas remains fixed. There is no need for endorsement of updates.

7.4.4 Obstacles to More Efficient Version Management

Form 2 and form 3 of centralized sync versions are, in essence, optimized representations of version vectors. In abandoning the version-vector-based specification of the `SyncVersion` interface for a more general specification, the MNCRS data-synchronization working group had envisioned more varied representations of versions. In the central-replica topology, it appears at first glance that a single integer, corresponding to the central replica's update counter, should suffice to record the version of each object on the central replica, while an integer corresponding to a central-replica version plus a one-bit flag indicating whether an object had been updated on the subordinate replica should suffice to record the version of each object in a subordinate replica. In the paired-replica topology, it appears at first glance that a one-bit flag should suffice to record the version of each object. (There would be two possible version values—*modified* and *unmodified*.)

In fact, these highly efficient version representations work only if synchronization protocols are constrained to use synchronous request/response dialogs; but as Section 6.5 explained, the MNCRS data-synchronization framework is designed to accommodate asynchronous protocols as well. A one-bit flag indicating that the state of an object has been modified must be cleared at the same time that an update is generated for that object, as part of an atomic operation, so that any subsequent modifications will result in new updates the next time that `generateUpdates` is called. However, if two replicas concurrently execute conflicting updates to the same object, then concurrently initiate synchronization phases sending updates to each other, the conflict will go unnoticed, as shown in Figure 22.

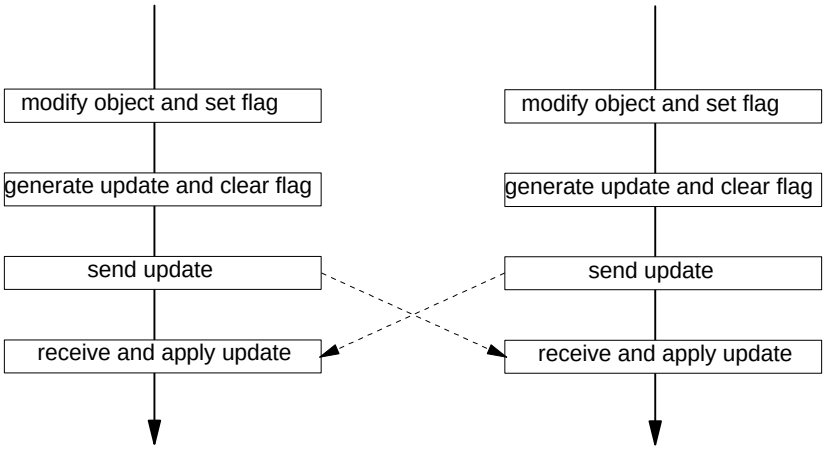

Figure 22. Incompatibility of modification flags and asynchronous exchange of updates. Two paired replicas concurrently update the same object, then concurrently transmit updates. Although the updates are conflicting, neither replica notices the conflict, because each resets its own modification flag for the object before the other replica's updates are received and applied.

By requiring a sync-store implementation to accommodate asynchronous exchange of updates, the MNCRS data-synchronization framework precludes the use of space-efficient version representations. In light of this unexpected consequence, the importance of asynchronous

exchange of updates should be carefully evaluated. The manipulation of these space-efficient versions is quite different from the manipulation of version vectors, so it would be difficult to formulate a single update-application algorithm and a single update-generation algorithm that works with both full version vectors and space-efficient versions, with all differences in behavior encapsulated in the implementations of the `SyncVersion` methods. (Indeed, we did not quite achieve that goal in our implementation even using full version vectors at the central replica: Because the endorsement of arriving updates occurs only at a central replica, the update-application logic includes an explicit test of the sync store's network role to determine whether an incoming update should be endorsed.)

7.5 Deletion Tombstones

As Section 4.2 explained, when an object is deleted from a sync store, a deletion sync entry remains behind, to resolve create/delete ambiguities. In our implementation, a corresponding deletion update-info remains in the update-log vector and in the persistent image of the update log. The data structures left in place after a deletion are called *tombstones*.

Tombstones cannot be allowed to accumulate, especially on memory-constrained mobile devices. Once news of an object's deletion has reached every replica that was aware of the object's existence, the tombstones can be safely removed from all these replicas. The `SyncStoreUpdater` interface of the MNCRS data-synchronization framework includes a method, `trimHistory`, that is called to notify a sync store that all deletion tombstones with versions earlier than a specified version may be removed. The response to this notification is up to the sync-store implementation; our implementation immediately removes all applicable tombstones.

It is the responsibility of a synchronizer to communicate with other replicas, to determine that all deletions with versions earlier than a particular version have reached all replicas that had learned of the corresponding insertions, and to call the `trimHistory` method of the local sync store. However, the MNCRS data-synchronization framework does not specify the distributed algorithms or protocols that synchronizers should use to determine that `trimHistory` should be called.

In a central-replica or paired-replica topology, it is easy to determine when tombstones can be removed. A replica can remove a tombstone for a given deletion after observing that each replica with which it synchronizes has requested updates with versions later than the version of the deletion. (For a central replica, the requests of all subordinate replicas must be tracked; for a subordinate replica, only the requests of the central replica need be tracked; for a paired replica, only the requests of the other replica in the pair need be tracked.) Were it not for the need to accommodate asynchronous exchange of updates, matters would be even simpler: A subordinate replica or paired replica would be able to drop deletion sync entries as soon as updates were generated from them; a central replica could track the version of the most recent update it had sent to each subordinate replica, and remove a tombstone as soon as all these versions were later than the version of the deletion.

In a general peer topology, matters are not so simple. There is a two-phase distributed algorithm, analogous to those described by Sarin and Lynch [Sar87] and by Ratner, Reiher, and Popek [Rat97], in which one phase determines the latest version earlier than or equal to the

summary versions of all replicas, and another phase informs all replicas that it is safe to invoke `trimHistory` with this version. However, such algorithms are not well-suited to a network in which many nodes are weakly-connected mobile devices. High communication costs may make the message volume required by the algorithm untenable. More seriously, many nodes in the network may be unreachable for long periods of time, blocking the convergence of each phase. In the worst case, the user of a mobile device goes on a long vacation (and leaves the device behind, for the sake of family harmony), preventing all other nodes in the network from removing their deletion tombstones.

To make matters worse, the membership and topology of the network are dynamic, defined not by some recorded state, but by the act of synchronization. In particular, it is possible for a sync store to be synchronized with replicas other than those in its registry. There is no mechanism by which a sync store announces that it has joined the network, or that it is leaving. (Bayou supports *fluid replication* [Dem94], in which individual users can create new replicas without any central registration. However, the status of an MNCRS replica is closer to that of a Lotus Notes client [Kaw92], which is aware only of the replicas that have been registered directly with it, and with which it may synchronize directly. In contrast, the Ficus update-distribution algorithm described in [Rat96] appears to require that all peers be aware of each others' existence; indeed, [Guy90] states that each Ficus host is assigned a unique identifier "prior to system installation".) Suppose a replica that has not yet received a particular deletion, but has executed an update that conflicts with that deletion. Suppose further that the replica performing the update is presumed to have left the network because of a prolonged period in which it did not synchronize, allowing the deletion to be trimmed from other replicas; if the long-lost replica then initiates a synchronization, the deleted object will be incorrectly restored to the other replicas as a new insertion, rather than detected as a conflict.

One solution to the problem of inactive nodes is a time-out mechanism: If a particular replica is not heard from after a specified number of hours or days, it is considered to be out of date. Any attempt at synchronization with such a replica is rejected; the only recourse for the holder of an out-of-date replica is to destroy it, create a new empty replica, and populate it by synchronizing with some up-to-date replica. However, not being heard from, and being out of date, are in the eye of the beholder. The user of a device that has not recently synchronized may be on a *working* vacation on a remote island, disregarding family harmony and busily creating updates that are intended to be preserved, and propagated to the network, at the end of the vacation. Alternatively, the network may be partitioned into two groups (perhaps teams assigned to work on two separate projects), with frequent synchronization within each group, but with no synchronization between members of different groups for a long period.

Resorting to an expensive reinitialization of a sync store to correct for overly aggressive trimming is reminiscent of the approach taken by Bayou [Pet97], in which each store may drop the oldest committed entries in its write log. If it is determined during an anti-entropy session that a sending store has dropped entries that have not yet been seen by the receiving store, the state of the sending store must be copied in its entirety to the receiving store, a process far more expensive than the usual incremental propagation of recent, unseen updates. A Bayou store may choose its own criteria for trimming its log, for example conservatively estimating how long it will take a write to propagate through the network, or freeing space as needed.

The MNCRS data-synchronization framework provides mechanisms by which a synchronizer can implement a *speculative log-trimming* strategy similar to Bayou's. The `SyncStoreUpdater` interface has a method returning the latest version for which `trimHistory` has been called. A synchronizer can invoke `trimHistory` according to its own criteria, but compare the versions of all incoming updates with the latest trimmed version. If an object-contents update with a version earlier than the earliest trimmed version is received, and the sync store does not contain an object with the same sync id, there are two possibilities:

- The incoming update was superseded by a later deletion, which was then trimmed from the local sync store, so the update should be ignored.
- The incoming update reconciles a conflict between an update and a deletion that was trimmed from the local sync store, so the update should be applied to restore the object.

Since there is no way to distinguish these two possibilities, corrective action must be taken. For example, the receiving synchronization phase can be aborted, a sending phase can be initiated to ensure that the remote replica is at least as up to date as the local replica, and the local replica can be destroyed and recreated from scratch by another receiving phase. However, this recovery scheme fails if both replicas have been too aggressive in trimming their histories, and the remote replica aborts the sending phase for the same reason that the local replica aborted the initial receiving phase.

7.6 The Persistent Store

As Section 7.3.1 explained, the persistent representation of a sync-store data collection includes a mapping from persistent-store keys to the persistently stored contents of individual `Reconcilable` objects. In our initial implementation, each `Reconcilable` object's byte-stream representation was stored in a separate file, and the file's name was used as the persistent-storage key. File names were generated automatically in lexical order ("`A`" through "`Z`", "`AA`" through "`AZ`", "`BA`" through "`BZ`", ...), and the files resided in the subdirectory created by `SyncStoreFactoryImpl` (as described in Section 7.2) to hold the persistent representation of the data collection. An additional file in that subdirectory stored the next file name to be generated. We thus delegated to the underlying file system the details of allocating space in the persistent-storage medium.

This approach accelerated construction of our initial prototype, but we realized it would be inefficient because each file would consume a whole number of file-system allocation units—1,024 bytes each in our run-time environment—even to store a byte-stream representation only a few bytes long. Therefore, we defined two interfaces, `PersistentStoreKey` and `PersistentStoreManager`, to isolate our implementation of persistent-storage keys and the storage and retrieval of persistent object representations from the rest of our sync-store implementation. The `PersistentStoreKey` interface declares only overriding methods for the `hashCode` and `equals` methods of class `Object`. The principal methods of the `PersistentStoreManager` interface write the persistent representation of a given `Transmittable` object using a given `PersistentStoreKey` value and return the `Transmittable` object whose persistent representation was stored with a given `PersistentStoreKey` value. There are also methods to generate a new `PersistentStoreKey` value and to delete the

persistent-store entry associated with a given PersistentStoreKey value. Finally, there are methods that mediate between the persistent representation of the update log and its in-memory representation as a Vector object whose elements are update-info objects: One method, called when a new SyncStoreState is constructed, returns a new vector corresponding to the persistent log. Another, called when a sync-store is flushed, appends to the persistent log the updates in a specified suffix of the vector. Another, called when a sync-store state is deactivated, rewrites the entire persistent log from the contents of the vector, omitting superseded updates.

(Section 4.1 described the evolution of the MNCRS data-synchronization framework interfaces for classes whose objects are transmitted during synchronization or stored persistently. In the first stage, all such classes were required to implement the Serializable interface. In the second stage, all classes whose objects are transmitted during synchronization were required to implement the Transmittable interface and all classes whose objects are stored persistently were required to implement the Persistable interface. In the third stage, the Persistable interface was eliminated and all classes formerly implementing that interface were required to implement the Transmittable interface instead. The types of the objects written and read by the first two PersistentStoreManager methods underwent a parallel evolution, from Serializable to Persistable to Transmittable.)

We expect implementations of the PersistentStoreManager and PersistentStoreKey interfaces to be paired. For example, our prototype implementation included a class FilePerObjectPersistentStoreManager implementing PersistentStoreManager and a corresponding class FileNamePersistentStoreKey implementing PersistentStoreKey. We later wrote two other pairs of classes, each providing a more space-efficient implementation of the persistent store. The SyncStoreState constructor invokes a constructor for some class implementing PersistentStoreManager, and the persistent-storage implementation used depends only on which class's constructor is invoked; it would be a simple matter to make this choice dependent on a property specified when a given data collection is opened for the first time.

Our second persistent-storage implementation uses the boundary-tag/first-fit storage-allocation mechanism described in Section 2.5 of [Knu73] to allocate and deallocate blocks of bytes within a random-access file. Each block holds one object, and the file is expanded as necessary. The persistent-storage key is an integer index into a table of storage-block offsets, as shown in Figure 23. When a byte stream of a given length is to be written using a given persistent-storage key, the persistent-store manager first checks whether an offset is currently stored in the offset-table entry associated with that key. If not, a block of the required size is allocated, and the offset returned by the allocator is placed in the offset-table entry. If the offset-table entry already has an offset stored in it and the byte stream fits snugly in that block, it is stored there. (A byte stream *fits snugly* in a block if the size of the block is at least 1.00, but no more than 1.25, times the size of the byte stream.) Otherwise, the block is deallocated, a new block of the required size is allocated, and the offset returned by the allocator for the new block is placed in the offset-table entry. When a block is deallocated, it is merged with any adjacent unallocated blocks.

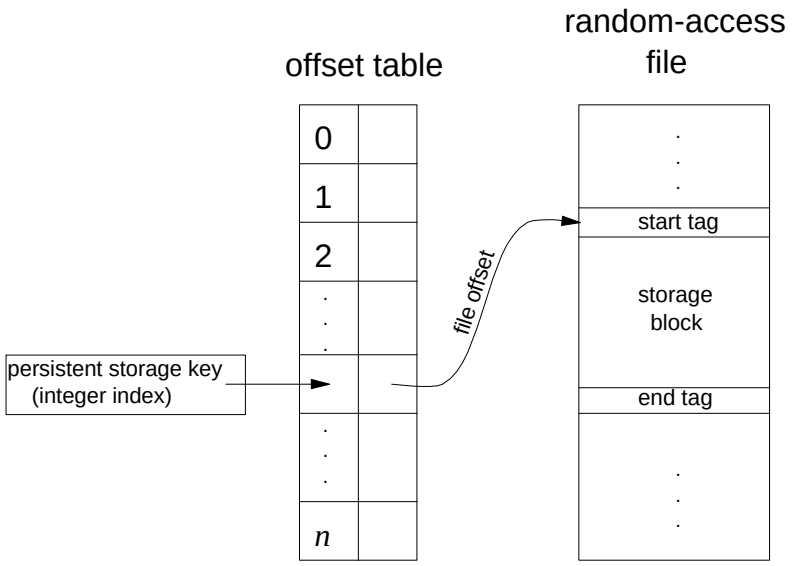

Figure 23. Persistent-storage management using boundary tags. A persistent-storage key is an index into an offset table. When the size of a persistent representation changes, the storage block may be deallocated and replaced with a newly allocated block elsewhere in the random-access file. When this happens, the corresponding offset-table entry changes, but the persistent-storage key remains the same.

In our third persistent-storage implementation, byte streams to be stored persistently are *appended* to a random-access file. The persistent-storage key is an integer index into a *mapping table* that records the length and offset of the most recent byte stream written for a given key, and also contains a flag that indicates whether the persistent-storage entry with that key has been deleted. The random-access file can be compacted in place by visiting table entries in offset order, sliding each undeleted byte stream up to the lowest unused position and adjusting the offset accordingly.

In a few contexts, for example, when reading the persistent copy of the registry of replicas to construct a `SyncStoreState` object, the framework reads the byte-stream representation of an object of a known class, by constructing an object of that class and invoking its `readRemote` method to set the object state. More often, however, the framework reads an object of some class that is not known *a priori*. For example, a receiving synchronizer reads the byte-stream representation of an object of type `SyncUpdate`, which may be of class `ObjectContentsSyncUpdate`, `DeletionSyncUpdate`, or `VersionSyncUpdate`; a persistent-store manager reads a byte-stream representation of a `Reconcilable` object, which belongs to some application class unknown to the framework. For objects whose class is not known *a priori*, a tag indicating the class of an object is always written to a byte stream before the representation of the object itself. For objects belonging to classes defined in the framework, for example objects belonging to subclasses of the abstract classes `SyncUpdateImpl` or `SyncVersionImpl`, the tag consists of a single byte; for objects belonging to application `Reconcilable` or `SyncId` classes, the tag consists of the UTF representation of the fully qualified class name.

Reading a tagged object representation entails reading the tag, constructing an object of the class indicated by the tag, and invoking that object's `readRemote` method. For framework classes, the one-byte tag serves as an index into a fixed array of objects of class `Class`, and the `newInstance` method of the indexed object is invoked. For application classes, the class name is passed as a parameter to the static method `Class.forName` to obtain an object of class `Class` whose `newInstance` method is invoked. For example, a simple socket synchronizer receiving a `SyncUpdate` object would first read a one-byte tag. If this tag indicates that the representation of an `ObjectContentsSyncUpdate` object follows in the byte stream, it would construct such an object and invoke its `readRemote` method. An `ObjectContentsSyncUpdate` object contains a `SyncId` reference, a `SyncVersion` reference, and a `Reconcilable` reference. Its `readRemote` method would first read the name of the application `SyncId` class, construct an object of the named class, and invoke its application-defined `readRemote` method. Next, it would read the one-byte tag for a `SyncVersion` object; if this were the tag for class `VersionVector`, it would construct a `VersionVector` object and invoke its implementation-defined `readRemote` method. Finally, it would read the name of the application `Reconcilable` class, construct an object of the named class, and invoke its application-defined `readRemote` method.

There are a number of exceptions, besides `IOException`, that can occur during the reading of a tagged object representation for reasons beyond the control of the framework. The `Class.forName` method will throw `ClassNotFoundException` if the class named in the input stream is not present anywhere in the local class path. If the named class is found, but the application programmer has failed to make it, or its zero-argument constructor, public, the `newInstance` method will throw `IllegalAccessException`. If the name found in the input stream refers to an interface rather than to a class, or if the application programmer has declared the class abstract, or if the construction of an object of the specified type fails for some other reason, the `newInstance` method will throw `InstantiationException`. If the definition of the class has changed, the call on `newInstance` may throw `NoSuchMethodError`.

If nothing special is done, these exceptions will manifest themselves in a stack trace, through several levels of calls on the `readRemote` methods for various implementation and application classes, with no clear indication of the underlying problem. Since the exceptions typically arise from application-programming errors, it is important to report these errors in a way that is readily understood by the application programmer, without forcing the application programmer to understand the framework implementation. Each of the framework's calls on an application-defined `readRemote` method is enclosed in a `try-catch` block that catches these exceptions and throws `IOException` instead, providing a message that names the class that could not be found or instantiated, and the original exception that had resulted from the attempt. The persistent-store manager, in turn, catches this exception and throws a different exception, `PersistentStoreException`, with a message conveying the same information. (`PersistentStoreException`, defined in our implementation of the framework, extends `SyncException`, defined in the framework and potentially thrown by any `SyncStore` operation that may need to access the persistent store.) Synchronizers catch the `IOException`, cause the current synchronization phase (and possibly any later-scheduled phases) to fail, and generate a failure event with a message conveying the same information as the `IOException` message. Application programmers reading messages from unhandled `PersistentStoreException`

occurrences, or examining the final status of failed synchronizations, will be directly informed about errors in the classes they wrote.

7.7 Replica Identifiers

Section 7.4 explained that version vectors map globally unique replica identifiers to values of local update counters. In our implementation, these replica identifiers are random 64-bit integers, generated when a data collection is first created. We are aware of other implementations that derive 64-bit replica identifiers from the data collection's URL. Both implementations are compatible with the MNCRS data-synchronization framework, which makes no mention of replica identifiers. However, there are subtle semantic differences.

Our approach allows a given replica to retain its identity when the subdirectory containing persistent representations is moved elsewhere on the same machine or to a new machine. It also allows multiple URLs—perhaps using different host names for the same machine, or different path names linked to the same subdirectory—to name the same data collection. Unfortunately, there is no way to prevent a replica from being cloned, with both copies retaining the same replica identifier and each copy being updated and synchronized independently. Since updates on both copies would update the same version-vector components, this misuse of our implementation would corrupt the sync store.

Similar problems arise if the same random number is generated for the identifier of more than one replica of the same data collection. Since there are over 1.8×10^{19} possible 64-bit identifiers, the probability of a clash seems minuscule compared to the probability of other events that can cause system failure. However, as illustrated by the well-known Birthday Paradox, the probability of a clash grows surprisingly quickly with the number of random selections. For a data store with 100 million replicas, there is a 0.03% probability of a clash, and for one billion replicas, there is a 2.67% probability of a clash. (See Figure 24.) Given the catastrophic consequences of a clash, these are significant probabilities. However, for one million replicas, the probability of a clash is only 0.000003%, for 100 thousand replicas, it is only 0.00000003%, and for ten thousand replicas, it is only one in 4 trillion. The probability of a clash can be reduced by increasing the length of a replica identifier, but for the scale of replication we envision, the resulting increase in the size of version-vector representations does not seem worthwhile.

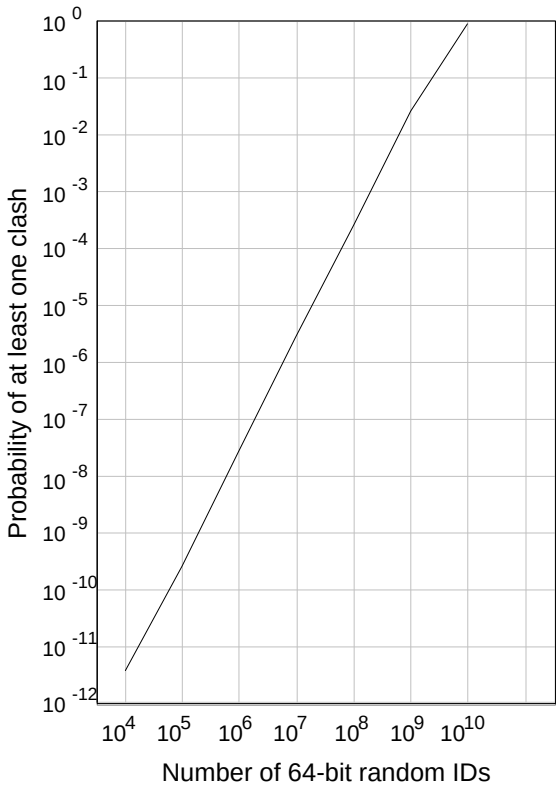

Figure 24. The probability p of at least one clash among r randomly selected 64-bit numbers. For $r < 10^9$, $p \approx r^2/10^{19}$. For $r > 10^{10}$, $p \approx 1$.

8 Roads Not Taken

In this section we discuss two issues that stymied the MNCRS data-synchronization working group—controlling the persistent representation of `Reconcilable` objects and synchronizing *transformations* to object states rather than synchronizing object states themselves. We explain the issues that created difficulty for the working group and propose our own solutions.

8.1 Transmittable and Persistent Representations of Application Objects

Section 4.1 described the `Persistable` interface that was present in an earlier version of the MNCRS data-synchronization framework, but removed because it constrained sync-store implementations to deliver representations of objects to the persistent-storage implementation a byte at a time through an output stream. The working group designing the framework was unable to design a standard counterpart to out implementation's `PersistentStoreManager` interface accommodating, for example, both persistent stores in which a byte-stream representation of an object is stored in a file and persistent stores in which an object is written to a relational-database record using the Java Reflection API to determine the structure of the object. Had such an interface been included in the framework, it would have made sync-store implementations easily portable to platforms with a wide variety of persistent stores. More important, *applications* would have been portable from sync stores using byte-stream-based persistent stores to sync stores using relational or object databases as persistent stores. In the absence of the `Persistable` interface, an application using a sync store with a database-based persistent store must provide sync-store-implementation-specific counterparts to the `Persistable` methods `writePersistent` and `readPersistent`, for the sync-store implementation to invoke when writing to or reading from the persistent store.

The working group's dilemma was how to satisfy two apparently contradictory needs simultaneously:

- There is an application-determined scheme for mapping the contents of a `Reconcilable` or `SyncId` object to data that is to be stored persistently, and for using data retrieved from persistent storage to reconstruct the contents of such an object. It should be possible for an application to specify this scheme independently of the persistent-storage implementation.
- There is a scheme determined by the implementation of a persistent store for storing and retrieving representations of objects. It should be possible for the persistent-store implementation to implement such a scheme without any knowledge of the objects being written by particular applications, and without using the heavy Java Reflection API.

One way to resolve this dilemma is with a standard intermediate representation. Application methods analogous to `writePersistent` and `readPersistent` would be responsible for mapping between the contents of `Reconcilable` or `SyncId` objects and this intermediate representation. Persistent-store implementations would be responsible for storing and retrieving instances of this intermediate representation. Then, given m application classes and n persistent-store implementations, it would not be necessary to write $m \cdot n$ distinct adapters, each adapting one application class to one persistent-store implementation, but only $m+n$ adapters, m of them each adapting one application class to the intermediate representation and n of them each

adapting the intermediate representation to one persistent-store implementation. More important than the number of adapters to be written is the separation of concerns: The author of a new application class need write only one adapter, and need not be concerned with individual persistent-store implementations; the author of a new persistent-store implementation need write only one adapter, and need not be concerned with individual application classes.

After the MNCRS data-synchronization working group completed its work, we encountered a similar dilemma in adapting the data-synchronization framework to the Mobile Data Synchronization Service (MDSS) project [But00]. In this case, the problem was adapting the application's *transmittable* object representation to the requirements of the synchronizer. The MDSS synchronizer transmits updates as part of a stylized XML document. This document contains both elements with synchronization-control information and elements describing the field-by-field contents of `Reconcilable` and `SyncId` objects. It would have been possible to implement a special-purpose parser for the XML document, which constructs a new application object upon encountering the element specifying the object's class, then calls the object's `readRemote` method to read and parse the XML text contained in that element describing the object's contents, and fills in the object accordingly. However, it was our goal to use a high-performance off-the-shelf XML parser and, more important, not to impose any burden on the `readRemote` method to parse XML text.

MDSS departed from the MNCRS framework by requiring each `Reconcilable` class in an MDSS application to provide `writeRemoteDOM` and `readRemoteDOM` methods, analogous to `writeRemote` and `readRemote`, but using tree representations of XML text (the Document Object Model [App98], or DOM) rather than byte streams to describe object contents. During a receiving phase, the MDSS synchronizer invokes the off-the-shelf parser once, to build a DOM representation of the arriving XML document. The synchronizer then traverses the DOM tree to find the classes of the `Reconcilable` objects to be constructed, constructs those objects, and invokes their `readRemoteDOM` methods, passing the appropriate subtrees of the DOM tree, to fill in those objects.

Had the MNCRS framework included an intermediate representation for transmittable application data, the MDSS project would have been able to provide an XML-based synchronizer within the framework. Instead of `writeRemoteDOM` and `readRemoteDOM` methods, application classes would provide standard methods to translate object contents to and from the intermediate representation; these would be invoked by MDSS synchronizer methods to translate the intermediate representation to and from DOM subtrees.

A suitable representation is an ordered sequence of name-value pairs, where each name is a string and each value is of one of the following forms:

- a value of one of the elementary Java types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`)
- an array of values in one of the elementary Java types
- an array of similar name-value pairs

Specifically, under our proposal, the contents of a `SyncId` or `Reconcilable` object is represented by an array of objects belonging to an abstract class `DataAttribute`, declared as follows:

```
public abstract class DataAttribute {

   public static final int BOOLEAN_TYPE = 0;
   public static final int CHAR_TYPE = 1;
      ...
   public static final int FLOAT_ARRAY_TYPE = 8;
   public static final int DOUBLE_ARRAY_TYPE = 9;
   public static final int DATA_ATTRIBUTE_ARRAY_TYPE = 16;

   public String getName();
   public int getType();
   public String getValue();
   public void read(DataInputStream dis);
   public void write(DataOutputStream dos);

}
```

The `getName` method returns the pair's name component, the `getType` method returns one of the 17 integer codes indicating the type of the pair's value component, the `getValue` method returns a `String` representation of the pair's value component, the `read` method sets the pair's value component by reading a byte-stream representation of the value from a given data input stream, and the `write` method writes a byte-stream representation of the pair's value component to a given data output stream. There are 17 concrete classes extending `DataAttribute`, one for each of the possible forms of a value. Each declares an additional method returning the value as the result of the appropriate Java elementary or array type. For example, the class corresponding to values of type `int` is declared as follows:

```
public class IntDataAttribute extends DataAttribute {
   private String name;
   private int value;
   public IntDataAttribute(String attrName, int attrValue)
      { name = attrName; value = attrValue; }
   public String getName() { return name; }
   public int getType() { return INT_TYPE; }
   public String getValue() { return Integer.toString(value); }
   public void read(DataInputStream dis) { value = dis.readInt(); }
   public void write(DataOutputStream dos) { dos.writeInt(value); }
   public int getIntValue() { return value; }
}
```

(`IntDataAttribute` extends `DataAttribute` with the `getIntValue` method.)

The `Transmittable` interface declares the following methods:

```
DataAttribute[] getTransmittableState();
void setTransmittableState(DataAttribute[] attributes);
```

If the `getTransmittableState` method of a newly constructed, default-initialized object is called, the elements of the result array should have default-initialized values, with types that reflect the logical structure of the object. The object's `setTransmittableState` method should

accept an array of DataAttribute objects with the same sequence of types. There is a separate Persistable interface with analogous getPersistentState and setPersistentState methods.

The Reconcilable and SyncId interfaces extend both Transmittable and Persistable. (For convenience, the framework could provide abstract DefaultReconcilable and DefaultSyncId classes implementing the Persistable methods in terms of the Transmittable methods.) An application writer implements the getTransmittableState and setTransmittableState methods in much the same way as the writeRemote and readRemote methods, except that getTransmittableState appends a DataAttribute object to the end of the result sequence instead of writing the binary representation of an elementary-type value to a data output stream, and setTransmittableState examines the next element in the DataAttribute[] parameter instead of reading the binary representation of an elementary-type value from a data input stream.

A sequence of DataAttribute objects is a convenient intermediate representation for a wide variety of synchronizers and persistent-store managers. In particular:

- A synchronizer that uses byte-stream representations of objects (in the style of the Transmittable interface of framework version 1.1) could work as follows: A byte-stream representation of an object is written by writing the object's class name, calling the object's getTransmittableState method to obtain an array of DataAttribute objects, and calling the write method of each of those objects in turn. This representation is read by first reading the string containing the class name, using the forName and newInstance methods of java.lang.Class to construct an object of that class, calling the getTransmittableState method to obtain an array of default-initialized DataAttribute objects, calling the read method of each DataAttribute object in turn to set their values, and calling the object's setTransmittableState method to fill in the object itself. For both reading and writing, the names in DataAttribute objects are ignored.
- A persistent-store manager that stores byte-stream representations of objects in files could work in the same way, but using the getPersistentState and setPersistentState methods instead.
- A synchronizer that uses XML documents could work as follows: A DOM subtree representing an object's contents is constructed by calling the object's getTransmittableState method to obtain an array of DataAttribute objects, and calling the getName and getValue methods of each of those objects in turn to obtain strings that can be passed to the DOM methods constructing the desired subtree. An object is filled in with contents specified by a DOM subtree by traversing the subtree nodes to obtain a sequence of elementary-type values and build an array of corresponding DataAttribute objects, then passing the array to the object's setTransmittableState method.
- A persistent-store manager that uses a relational database could work as follows: The persistent-store manager maintains a one-to-one correspondence between application class names and database tables; the class implementing PersistentStoreKey identifies both a table and a database key for a row of that table. A record reflecting an object's contents is written by using the name of the object's class to identify the corresponding table, calling the

object's `getPersistentState` method to obtain an array of `DataAttribute` objects, and calling the `getName` and `getValue` methods of each of those objects in turn to obtain strings that can be used to construct an SQL command (either through direct string manipulation or through JDBC calls). An application object is constructed from a database record by using the table name to determine the class name, using the `forName` and `newInstance` methods of `java.lang.Class` to construct an object of that class, calling the `getPersistentState` method to obtain an array of default-initialized `DataAttribute` objects, calling the `read` method of each `DataAttribute` object in turn to set its value, and calling the object's `setPersistentState` method to fill in the object itself. Each call on `read` obtains its input from a `ByteArrayOutputStream` that is constructed from the contents of the corresponding field of the database record.

The array of default-initialized `DataAttribute` objects returned by a call on the `getTransmittableState` (or `getPersistentState`) method of a newly constructed object acts, in a sense, as a program specifying the order in which values of various types are to be obtained; the application class determines the form of this "program" and the synchronizer (or persistent-store manager) "executes" it by calling the `read` method of each array element in turn. However, the "programming language" of `DataAttribute` sequences is clearly less expressive than Java itself. For example, the conditional reading in the following `readRemote` method is unattainable:

```
class TimeOfDay implements Reconcilable {

   private byte hours, minutes, seconds;
   private static final int HH=0, HHMM=1, HHMMSS=2;

   ...
   public void readRemote(InputStream is) {
      DataInputStream dis = new DataInputStream(is);
      byte format = is.readByte();
      switch(format) {
         case HH:
            hours = is.readByte();
            minutes = 0;
            seconds = 0;
            break;
         case HHMM:
            hours = is.readByte();
            minutes = is.readByte();
            seconds = 0;
            break;
         case HHMMSS:
            hours = is.readByte();
            minutes = is.readByte();
            seconds = is.readByte();
            break;
      }
   }

}
```

Such constructs are rare, and probably dispensable. However, variable-length homogeneous sequences are common and often unavoidable. `DataAttribute` implementation classes for arrays of elementary-type values address this need. An object of class `FloatArrayDataAttribute`, for example, acts as an "instruction" to obtain a variable-length sequence of `float` values. The synchronizer or persistent-store manager executing this instruction uses its own conventions for generating and interpreting representations of such sequences, for example by preceding the elements of the sequence with an integer giving its length. (Classes such as `FloatArrayDataAttribute` are also useful for sequences whose length is fixed, because a `FloatArrayDataAttribute` object containing a reference to an array of 100 `float` values, for example, requires much less space than an array of 100 references to `FloatDataAttribute` objects each containing a single `float` value.) A `DataAttribute` object for an array of `DataAttribute` objects allows interleaved sequences (such as *n* repetitions of an `int`, a `float`, and a `char`) or variable-length sequences of variable-length sequences.

`DataAttribute` objects for arrays of `DataAttribute` objects also allow an application to generate an intermediate representation that preserves hierarchical structure, but this is unlikely to be useful, because the application cannot assume that the synchronizer or persistent-store manager is capable of exploiting that hierarchical structure. Rather, the array of `DataAttribute` objects returned by `getTransmittableState` or `getPersistentState` should be viewed as the representation of an object's state as a flattened sequence of named elementary-type values, just as serialization or the `writeRemote` method generates a representation as a flattened sequence of byte values. By constructing the sequence at higher level of abstraction, we facilitate a clear separation of concerns: Application classes are responsible for the representation of an object's state as a flattened sequence of named, typed values (without prejudice as to whether the order of the elements or their names is more significant) and synchronizers and persistent-store managers are responsible for the sending and receiving or storing and retrieving of some representation of these sequences (without regard to the objects that the sequences represent). Some synchronizers and persistent-store managers will pay attention only to the order of `DataAttribute` objects, and ignore the names associated with them, while others will use the names but attach no significance to the order.

8.2 Differential Updates

Version 1.1 of the MNCRS data-synchronization framework synchronizes at the granularity of an entire object. The `markAsUpdated` method applies to objects rather than fields, and does not indicate the manner in which a field has changed. If an object is marked as updated, a copy of the entire updated object is transmitted during the next synchronization. If we were to ship a description of the *way* in which the object was updated, rather than the entire updated object, we would generally have less data to ship over the potentially slow and expensive connection. In addition, there are applications in which appropriate behavior depends on the transmission of transformations rather than the states resulting from those transformations. In particular, when an application increments or decrements a shared count, such as an inventory or account balance, the appropriate reconciliation of a conflict cannot be inferred from the values of the counts after conflicting increments or decrements, but rather from the amounts of the increments or decrements themselves. Early drafts of the framework included provisions for synchronizing based on transformations to objects, but these provisions were dropped when

complications came to light. In this section, we describe the initial proposal, explain the problems that arose, and offer a revised proposal that addresses these problems.

8.2.1 The Initial Proposal

In the initial proposal, an object in a sync store can belong to a class implementing either the Reconcilable interface we have already seen, or another interface named Diffable. Synchronization of Reconcilable objects is based on the exchange of whole objects and synchronization of Diffable objects is based on the exchange of transformations, or *differences*. The Reconcilable and Diffable interfaces extend a common supertype named Synchronizable, as shown in Figure 25. Many of the methods in the SyncStore interface are defined in terms of this supertype. For example, put takes a Synchronizable parameter and get returns a Synchronizable result.

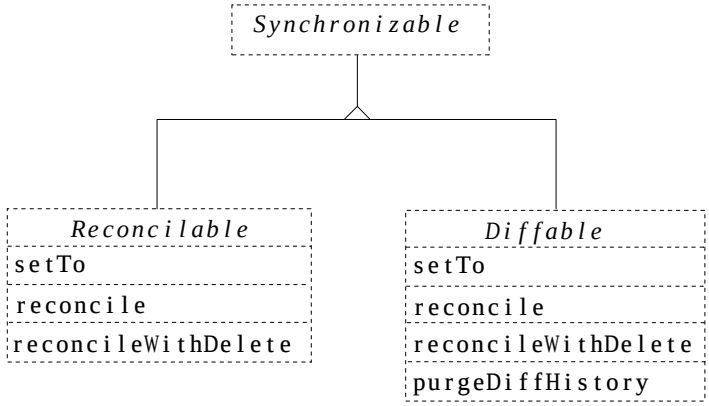

Figure 25. The initial proposal for a hierarchy including the Diffable interface. Reconcilable and Diffable are interfaces extending the Synchronizable interface. Objects in a sync store belong to application-defined classes implementing one of Reconcilable and Diffable.

Like Reconcilable, Diffable is meant to be implemented by an application-provided class, whose instances may be stored in a sync store. There is also an interface named Diff, containing no declarations, but meant to be implemented by an application-provided class whose objects represent transformations to objects of a corresponding Diffable class. One of the methods declared in the Diffable interface, and implemented by the application writer in a class implementing Diffable, is a method named applyDiff, which takes a parameter of type Diff and applies the transformation specified by the Diff object to this Diffable object. For example, an application might contain a class Calendar implementing Diffable, and classes AppointmentInsertion, AppointmentEdit, and AppointmentRemoval implementing Diff. The applyDiff method of Calendar would expect its Diff parameter to be either an AppointmentInsertion, AppointmentEdit, or AppointmentRemoval object, and it would apply the corresponding transformation to its Calendar object.

The sync store maintains both the current state of a Diffable object and a history of transformations that were applied to the object to place it in that state. The history includes a Diff object and a version for each transformation. The trimHistory method described in

Section 7.5, which removes deletion sync entries that have been propagated to all replicas, also removes transformations that have been propagated to all replicas.

The `SyncStore` interface has two overloaded `markAsUpdated` methods. The first is the method that was described in Section 5.1 and has a single parameter, of type `SyncId`. It is used to mark a `Reconcilable` object as updated, and throws an exception if the `Synchronizable` object identified by the specified sync ID does not implement the `Reconcilable` interface. The second has a parameter of type `SyncId` and a parameter of type `Diff`. It is used to mark a `Diffable` object as updated by a particular transformation (specified by the `Diff` parameter), which is appended to the object's history; the method throws an exception if the `Synchronizable` object identified by the sync ID does not implement the `Diffable` interface. (An application may modify a `Diffable` object in the sync store directly, construct a `Diff` object describing the modification, and call the second `markAsUpdated` method to inform the sync store of the modification; or it may first construct a `Diff` object describing the change to be made to a particular `Diffable` object, pass the `Diff` object as a parameter to the `Diffable` object's `applyDiff` method, and then pass the same `Diff` object to `markAsUpdated`.)

Three different kinds of `SyncUpdate` objects may be transmitted for a `Diffable` object during synchronization:

- An update corresponding to the *insertion* of the object into a sync store by a call on `put`. Such an update contains the state of the entire object. A sync store applies a remote update of this kind by inserting a local copy of the `Diffable` object.
- An update corresponding to a *transformation* of the `Diffable` object. Such an update contains a `Diff` object describing that transformation. To apply a remote update of this kind to a local copy of the `Diffable` object, a sync store compares the update's version to the version in the corresponding local sync entry. If the update version is older, the update has already been applied, and is ignored. If the update version is newer, the sync store calls the `Diffable` object's `applyDiff` method with the `Diff` object contained in the update. If the two versions are in conflict, the sync store invokes one of the `Diffable` object's methods to resolve the conflict.
- An update corresponding to the *deletion* of the `Diffable` object. To apply a remote update of this kind, a sync store first checks whether the object has already been deleted locally, in which case it simply updates the version in the corresponding local sync entry to be later than or equal to both the version of the remote update and the previous version in the sync entry. Otherwise, the sync store compares the update's version to the version in the corresponding sync entry. If the update version is earlier, the update has already been applied, and is ignored. (This can only happen if the local `Diffable` object's history includes an earlier reconciliation of a transformation with the same deletion, received from some other replica.) If the update version is newer, the sync store deletes the object locally. If the two versions are in conflict, the sync store invokes one of the `Diffable` object's methods to resolve the conflict.

Like the `Reconcilable` interface, the `Diffable` interface has methods named `reconcile` and `reconcileWithDelete`. However, the methods in the `Diffable` interface have different parameter types and different behavior. The `reconcile` method is invoked to resolve a conflict

between a remote transformation and one or more local transformations. Its parameters include the remote `Diff` object received in the update and an array of all local `Diff` objects for this `Diffable` object that conflict with the update. The method is expected to place its `Diffable` object in a state that reflects the resolution of the conflict and to return a `Diff` object describing this modification that will be propagated to other replicas. The `reconcileWithDelete` method is invoked to resolve a conflict between a *remote* deletion and one or more local transformations. Its parameters include an array of all `Diff` objects for the local `Diffable` object that conflict with the remote deletion. The method either returns an integer code indicating that the deletion is to prevail, or places its `Diffable` object in a state that reflects the resolution of the conflict and returns an integer code indicating that the object is to be reinstated in the remote replica.

A conflict between a remote transformation and a *local* deletion is problematic, because the entire object state is no longer present locally, and only the `Diff` object describing the transformation is received from the remote replica. In this case, the conflict is left temporarily unresolved. During the next synchronization phase that transmits updates from the local sync store to the remote replica (either as part of the current synchronization or as part of a later synchronization), the deletion will be transmitted to the remote replica, and the `reconcileWithDelete` method will be invoked there. If the update-delete conflict is resolved by retaining the object, the entire object will be transmitted from the remote replica back to the local sync store during the next synchronization phase in that direction, and it will be reinserted. (See Figure 26.)

8.2.2 Problems with the Initial Proposal

It is difficult for the application writer to implement the `reconcile` method of the `Diffable` interface to determine the desired reconciled state and to ensure that the two conflicting `Diffable` objects are placed in that state. After setting the local `Diffable` object to the desired reconciled state, the method must return a `Diff` object describing the way in which the local `Diffable` object was changed (or `null` if it was unchanged). When the `Diff` object is propagated to the remote sync store, another conflict is detected there, and `reconcile` is invoked again on the remote sync store, to determine how the transformation produced by the reconciliation at the first sync store should be reconciled with conflicting transformations at the second sync store to bring the two versions of the `Diffable` object into the same reconciled state. Achieving the desired behavior is complicated at best, and in some cases it is impossible.

For example, consider a `Diffable` object representing a command to a VCR to begin recording a program on a specified channel at a specified time. Initially, sync stores A and B both contain a command to begin recording on channel 2 at 8:00pm, as shown in Figure 27. At Sync Store A, a transformation is applied to change the channel to 4, resulting in a command to record channel 4 starting at 8:00pm. Meanwhile, at Sync Store B, a transformation is applied to change the starting time to 9:00pm, resulting in a command to record channel 2 starting at 9:00pm. The two stores synchronize and the `reconcile` method of the VCR-command object is invoked at A. The remote transformation changes the time to 9:00pm, the local transformation changes the channel to 4, and the object itself contains start time 8:00pm and channel 4.

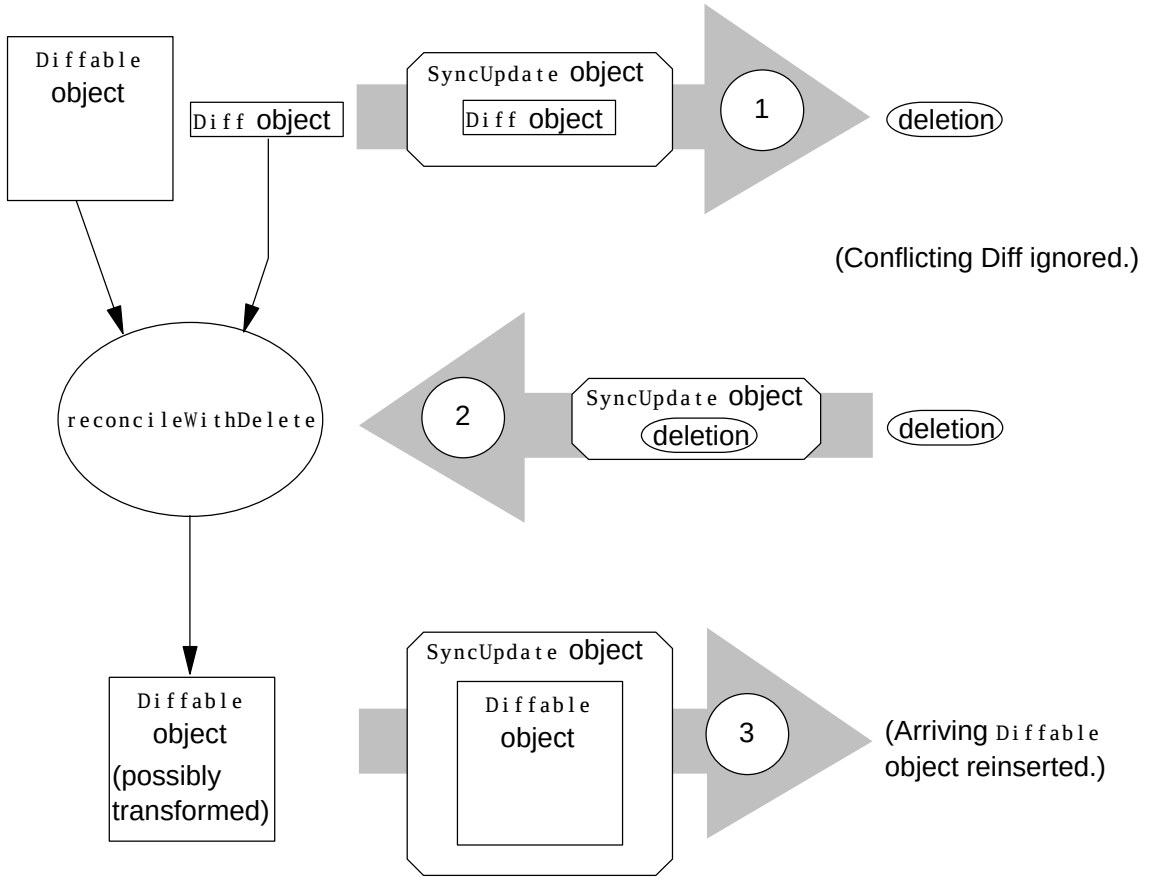

Figure 26. Resolving a conflict between a remote transformation and a local deletion. In the first synchronization phase, a `Diff` object arrives at a sync store from which the corresponding `Diffable` object has been deleted, and the `Diff` object is ignored. In the second phase, the deletion is propagated back to the replica containing the conflicting transformation, and `reconcileWithDelete` is invoked to resolve the conflict between a local transformation and a remote deletion. If `reconcileWithDelete` determines that the deletion should prevail, the object is deleted from that replica as well. Otherwise, in a third synchronization phase, the `Diffable` object, possibly altered by its `reconcileWithDelete` method to resolve the conflict, is transmitted in its entirety to the replica from which it had been deleted, where the object is reinserted.

There are three plausible reconciliation policies:

1. Keep the local user's command, to record the program on channel 4 at 8:00pm.
2. Keep the remote user's command, to record the program on channel 2 at 9:00pm.
3. Revert to the last agreed-upon command, to record the program on channel 2 at 8:00pm.

(It does *not* make sense to change the command contents to channel 4 at 9:00pm, which would result in the recording of a program in which nobody expressed any interest!) To implement Policy 1, the local `reconcile` method should leave the object unchanged and return `null`, but

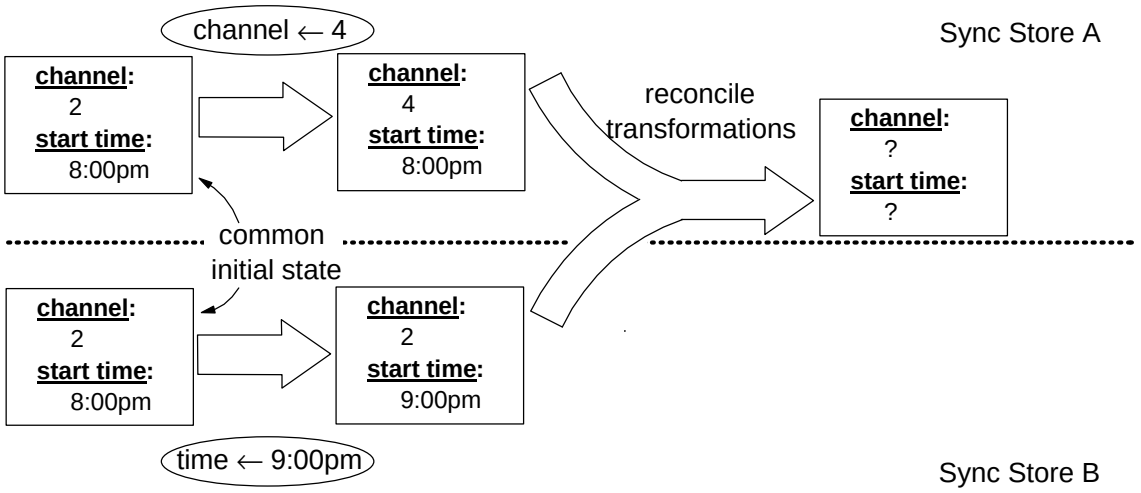

Figure 27. Lack of sufficient information to resolve conflicts among transformations. The only information available at Sync Store A is that the channel value was changed locally from some unknown value to 4, the current local start-time value is 8:00pm, and the start time was changed remotely from some unknown value to 9:00pm.

then the remote `reconcile` method will have no clue about the need to change its time from 9:00 to 8:00. Similarly, to implement Policy 2, the local `reconcile` method must change its channel back to 2, but no information indicating this is available: The channel to be restored is not present in the remotely generated transformation because the remote sync store never changed the channel, and it is not present locally because the old channel value has been overwritten. Similarly, neither sync store has enough information to implement Policy 3: Each has overwritten a piece of information about the old state. The information overwritten by each sync store is not recorded in the transformation generated by the other sync store, because the other sync store has left this information unchanged.

The task of the application writer is further complicated by the fact that the reconcile method of the `Diffable` interface is invoked once for each conflicting remote transformation, so the behavior of the method cannot depend on the collective effect of all conflicting remote transformations. For example, suppose a `Diffable` object represents a drawing that contains many graphical objects, and `Diff` objects represent graphical editing operations. Starting with a common object state, the drawing is modified locally by deleting a particular rectangle (adding one transformation to the local object's history); the drawing is modified remotely by first filling the interior of the rectangle (adding one transformation to the remote object's history) and then creating a copy of the rectangle elsewhere in the drawing (adding a second transformation to the remote object's history). A reasonable reconciliation is to keep the copy of the newly-filled rectangle, but to delete the original rectangle. The conflict results in two calls to `reconcile`, each passing one remote transformation. The first call will try to reconcile the local deletion of the rectangle with the remote filling of the rectangle. With no further context available, this reconciliation would likely keep the deletion, ignoring the filling. The second call will then try to reconcile the deletion of the rectangle with the copying of the rectangle, but in a state in which the rectangle is no longer present in the drawing! See Figure 28.

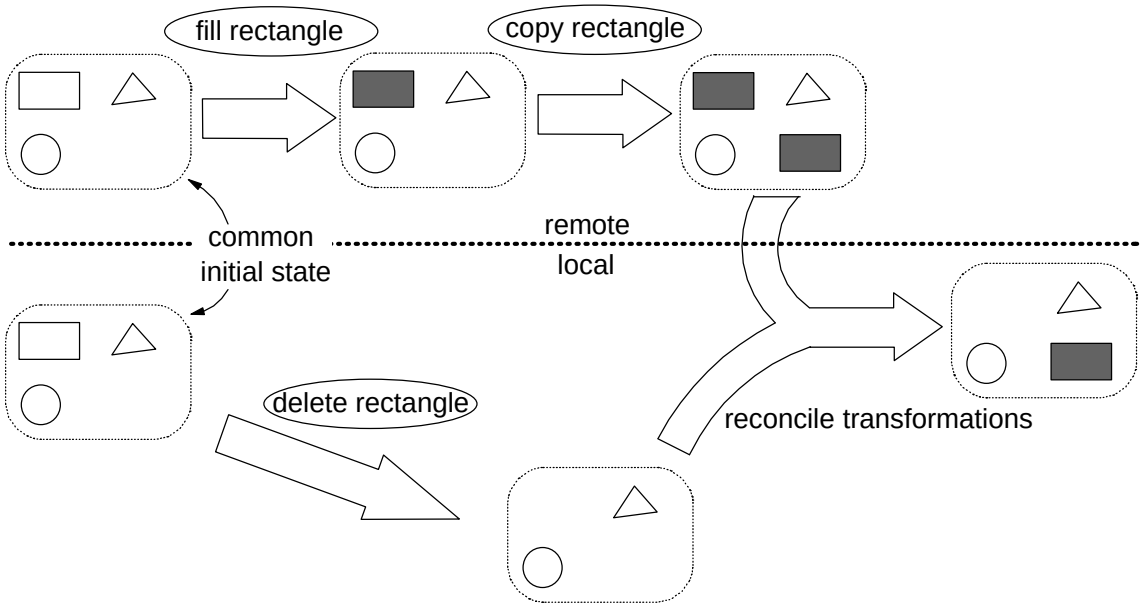

Figure 28. Difficulty of resolving remote transformations one at a time. The desired net effect is to remove the original rectangle but to keep the filled copy. However, when the local transformation deleting the rectangle is reconciled in isolation with the remote transformation filling the rectangle, the sensible reconciliation is to preserve the deletion.

The MNCRS data-synchronization working group considered a number of changes to the framework to resolve the difficulties that were discovered. However, the consensus of the group was that the proposed changes made the application writer's use of the interface even more complicated. The group decided to omit differential updates from version 1.1 of the framework. However, the `Synchronizable` interface was retained as a superinterface of `Reconcilable` to allow for the possible reintroduction of other extensions of `Synchronizable` in the future.

8.2.3 A Counterproposal

The proposed `Diffable` interface was difficult for application programmers to use because it tried to solve too general a problem. Meaningful reconciliation of differential updates is impossible unless the updates are constrained to have certain nice algebraic properties. Interfaces analogous to `Diffable`, but relying on such properties, would reduce the difficulty of specifying application-based reconciliation. We proposed two such interfaces, extensions of `Synchronizable` named `Cumulative` and `Revertible`, to the MNCRS data-synchronization working group, and described one possible implementation of a sync store accommodating these interfaces. However, in the interest of quickly completing a workable framework, the working group deferred consideration of these proposals to a future version of the framework.

Reconciliation is easy to handle if all transformations are commutative, i.e., if a set of transformations has the same effect regardless of the order in which they are applied. Common examples of commutative transformations include increasing some count by a specified amount or adding some element to an unordered set. If a set of transformations has the same effect regardless of the order in which they are applied, a transformation can be meaningfully applied regardless of the state of the `Synchronizable` object beforehand. Transformations with

conflicting versions are naturally reconciled by applying them *both* (in either order). In other words, every transformation to a given `Synchronizable` object, originating at any replica, is applied to that object. The object's state reflects the cumulative effect of all the transformations that have been applied to it. We proposed that objects of this kind implement an extension of the `Synchronizable` interface named `Cumulative`. The `Cumulative` interface has an `applyDiff` method, but no `reconcile` method, because there are no conflicts to reconcile! (Strictly speaking, all conflicts are reconciled automatically, by applying all the conflicting updates, in an order that does not matter.) The absence of a reconcile method simplifies the application programmer's task and enables sync-store implementations to apply remote updates more quickly. (The principle that transformations must commute has an interesting consequence for the behavior of deletions. If a deletion is just a special kind of transformation, and if transformations can be applied in any order, it should be possible to apply any transformation to a deleted object. The effect should be simply to leave the object deleted, as if the transformation had preceded the deletion. There is no need for a `reconcileWithDelete` method because, in effect, a deletion always prevails over other updates.)

Reconciliation is also reasonably straightforward if it is possible to revert to the state a `Synchronizable` object was in before the last *n* transformations in its history were applied. This is possible if each transformation has a corresponding inverse that can be applied to undo it. Alternatively, an application programmer with memory to spare can revert to previous object states by maintaining a log of object contents before each transformation. An application programmer willing to make the application dependent on a particular sync-store implementation might use a version-archiving service provided by that implementation. We proposed that objects of this kind implement an extension of the `Synchronizable` interface named `Revertible`. Reconciliation of `Revertible` objects works by, in effect, reverting to the most recent common version in the history of the local and remote objects and then applying the transformations required to go from that common ancestor state to the desired reconciled state. The framework provides one special class implementing `Diff`, named `RevertDiff`. A `RevertDiff` object specifies the modification to a `Revertible` object that would result from undoing all the transformations applied since a given sync version, specified by the `RevertDiff` object. The `RevertDiff` class has one method—

```
Diff[] annulledDiffs()
```

—returning an array of the transformations to be undone. The application-defined `applyDiff` method must be able to apply a `RevertDiff`. If it is relying on a special extended service of an archiving sync-store implementation, it can simply ask the sync store to roll back the state of the object by the specified number of versions. If the application knows how to compute an inverse for each application-defined `Diff` object, it can apply the inverses of the transformations in the array returned by `annulledDiffs`, starting with the most recent transformation. In many common cases, the application can be more direct. For example, if the only application-defined `Diff` class represents the addition of a specified amount to a remaining-inventory field, a `RevertDiff` can be applied by summing the amounts in the array returned by `annulledDiffs`, and subtracting that amount from the field. The `Revertible` interface has a `reconcile` method whose contract is to determine the desired reconciled state, to place this `Revertible` object in that state, and to return an array `Diff` objects that would transform the common ancestor state into the desired reconciled state. The `reconcile` method can determine the common ancestor

state by examining an array, passed as a parameter, that contains a `Diff` object for each transformation that has been applied to the local `Revertible` object since the `Revertible` object was in the common ancestor state. (If necessary, the `reconcile` method can actually restore the object to that state as an interim step in its execution, by manipulating the array of `Diff` objects in the same way that the `applyDiff` method treats the array returned by the `annulledDiffs` method. However, it will often be possible for the application to directly compute a transformation that *would* change the common ancestor state to the desired reconciled state, by somehow combining all the conflicting local and remote transformations.)

There are examples of `Diffable` classes for which the `reconcile` method is simple to implement, but we conjecture that this simplicity usually arises from `Cumulative` or `Revertible` properties that the application writer consciously or unconsciously exploits to write the `reconcile` method. Such classes can be rewritten to extend the `Cumulative` or `Revertible` interfaces. Indeed, *any* `Diffable` class can, at the cost of space, be made into a `Revertible` class, by modifying each `Diff` class that would not otherwise be invertible to specify not only the new information it is placing in the object, but the old information it is obliterating. Returning to the VCR-command example, if the `Diff` object specifying "change the channel to 4" instead specified "change the channel from 2 to 4", and the `Diff` object specifying "change the starting time to 9:00pm" instead specified "change the starting time from 8:00pm to 9:00pm", these `Diff` objects would have inverses, and the VCR-command class could be written as a `Revertible` class.

One can envision extending the `Synchronizable` interface in other ways as well, unrelated to differential updates. For example, the framework could provide an `Immutable` interface for objects that will be inserted in and removed from the sync store, but never modified while they reside in the store. This would save the application writer the work of writing `reconcile`, `reconcileWithDelete`, and `setTo` methods that would never be called. The writing of applications and the implementation of a sync store supporting differential updates might be simplified by treating Reconcilable updates as a special kind of differential transformation, whose effect is to replace the contents of the object with a specified complete new object state.

9 Conclusions

For a variety of business reasons, MNCRS consortium members' interest in a standard Java platform for mobile devices waned, and the activities of the consortium subsided after the release of MNCRS version 1.1 in March 1999. Some of the consortium's valuable work was handed off to the Internet Engineering Task Force, the Network Computer Management Group, and the Java Community Process. Consortium leaders proposed submitting the data-synchronization API to the Java Community Process for adoption as a Java extension, but no volunteer came forth to lead this effort.

Nonetheless, important lessons were learned from the specification and implementation of the framework, and we expect the framework to inspire and influence future data-synchronization research. There have already been two spinoffs of the MNCRS data-synchronization work at IBM: the COSMOS state-machine model of data synchronization and the Mobile Data Synchronization Service. These are discussed in Sections 9.2 and 9.3, respectively.

Had the MNCRS data-synchronization effort continued, the first priority of the working group would have been to specify the standard Java-object-based protocol described in Section 6.5. The working group seemed to be converging on a consensus approach, and we were not aware of any fundamental problems with the approach. Such a protocol, required to be supported by all implementations of the framework, would have ensured interoperability among all implementations.

The data-synchronization framework has no explicit security mechanisms, but it comfortably accommodates implementation-provided security mechanisms. For example, the implementation-defined properties passed to the `open` method of the `StoreManager` class (see Section 5.3) might include a password or certificate establishing a user's right to access the store. A security-oriented synchronizer might encrypt transmitted data or use secure transport such as SSL. A persistent-store manager might encrypt data stored on a mobile device to protect its privacy in the event that the device is lost or stolen.

The major unsolved problem preventing industrial use of the framework is that of determining when the `trimHistory` method should be called to remove deletion tombstones, and with what version. Section 7.5 explained the difficulty of this problem, especially in a network with a dynamic, unrecorded topology. It remains an important area for future research.

Early in its deliberations, the MNCRS data-synchronization working group adopted several fundamental principles, which were accepted as axioms and which constrained the design of the framework. In retrospect, in the light of our specification and implementation experience, some of these principles are questionable, because of their impacts on the conceptual simplicity of the framework and the performance of implementations. Section 9.1 revisits the most important assumptions.

9.1 Fundamental Assumptions Revisited

9.1.1 Synchronization as Replication

Synchronization maintains sync stores as replicas. We rely on the two approaches depicted in Figure 1 to enable different mobile devices to maintain different subsets of a central data store. This precludes a simple archiving function, in which an object is deleted from a store on a memory-constrained device without deleting it from the corresponding remote store. The overlapping subsets depicted in Figure 1(b) pose difficult semantic issues that must be resolved before such a scheme can be implemented. For example, if an object is deleted from a sync store that is a subset of some containing store, should the object be removed from the subset, but retained in the containing store? How should membership in a given subset be defined? If membership is determined based on a predicate that forms part of the data-store URL, what restrictions should be placed on the predicate? If the predicate is allowed to refer to aspects of the external environment, such as time and date, we gain the useful capability of objects that expire, but we must deal with the possibility of objects spontaneously joining and leaving a sync store without any synchronization or local updates to the store. How and when should these spontaneous membership changes be reported, and what is the effect on the event model? Can a modification to the contents of an object have the side effect of deleting the object from the store in which the modification was performed, and causing the object to be passively deleted from other stores as well?

9.1.2 Support for Peer-to-Peer Synchronization

The framework supports peer-to-peer synchronization. We presumed that if we solved the most general form of the synchronization problem, appropriate solutions for more restrictive topologies would fall out as a byproduct. As Section 7.4.4 explained, we were disillusioned by the difficulty of describing a wide variety of version-management schemes through a single `SyncVersion` abstraction. The simplest and most efficient implementation approaches for more restrictive topologies are fundamentally different from the approaches necessitated by peer-to-peer synchronization. [Rat97] points out that version vectors are not merely expensive; they do not scale well over time, because version vectors can grow arbitrarily long. Even a single exceptional update of an object at a site where it is not usually updated permanently burdens the object's version vector with a nonzero component for that site. The presence of a central server simplifies the management of deletion tombstones (which need not be maintained at all on clients, and can be removed from the central server after the server has sent the deletion to all clients). A central server also facilitates archiving of objects removed from a client store to recover storage.

We were hard pressed to come up with compelling applications for peer-to-peer synchronization. Like the developers of Ficus [Rat96] and Bayou [Dem94], we initially envisioned mobile workgroups with devices disconnected from any fixed network, but able to communicate with each other by infrared, radio, or even the exchange of diskettes. Peer-to-peer networks would also seem to be more robust in the face of network failures, because of the larger set of potential synchronization partners. However, emerging mobile devices tend to depend on common carriers, such as telephone lines, cellular infrastructure, or RF services like Palm**.**Net and Ricochet, to communicate. The common carrier can connect mobile devices with a central server as easily as with each other. *Ad hoc* networks that form when devices are brought into proximity

with each other tend to be asymmetric, with one device acting as a controller and other devices providing services such as printing or data entry. (Bayou takes a hybrid approach, allowing synchronization among any two stores, but treating one of the stores asymmetrically as the primary replica of the data.)

9.1.3 Support for Asynchronous Protocols

The framework supports asynchronous synchronization protocols. For example, updates might trickle to a mobile device through a pager throughout the day, and updates from the device might be sent in a burst once a day over a phone line; alternatively, if communication is only possible during rare, relatively short windows, a mobile device might synchronize by a full-duplex exchange of incoming and outgoing pending updates in concurrent bursts, followed by immediate disconnection. Section 7.4.4 explained how the need to accommodate asynchronous protocols complicates version management. Detection of, and recovery from, communication errors is also complicated. However, in some application environments, the flexibility and robustness attainable through asynchronous synchronization may be worth the price.

9.1.4 Order of Update Transmission

The framework dictates that updates be transmitted in introduction order. This restriction allows versions to be represented by version vectors, and facilitates incremental progress when the transmission of updates is interrupted. However, it would be useful in many circumstances to deliver updates in some order specified by applications or end users. For example, mail messages marked urgent, or sent by Very Important People, might be assigned a higher priority; very large objects might be assigned a low priority, so that other updates do not get blocked behind them. For synchronization over an unreliable link, it might be preferable to transmit higher-priority messages first, in case the link goes down before synchronization is complete. An end user with a limited amount of time available before catching a plane might prefer to receive as many updates as possible in five minutes, in priority order, before deliberately disconnecting.

The effect of prioritized delivery of updates can be achieved within version 1.1 of the MNCRS data-synchronization framework by placing objects with different synchronization priorities in different sync stores, and synchronizing those sync stores in priority order. The use of multiple sync stores is inconvenient for an application writer, because a reference to a stored object no longer consists of a sync ID, but a sync id plus the identity of the sync store containing the object. A revised framework could remove the bookkeeping burden from the application writer, providing a `put` method that assigns a priority to an object when it is inserted in the sync store and maintaining a distinct summary version for each priority level. Updates would be generated from highest priority to lowest, following introduction order within each priority level. In effect, synchronization would proceed *as if* objects of different priorities resided in different stores. Priority values could be included in `SyncUpdate` objects and propagated during synchronization.

There are certain limitations to each of these approaches. Both approaches assume a small number of discrete priority levels. Furthermore, both approaches require a priority to be assigned statically at the time an object is inserted into a sync store.

9.1.5 Eventual Consistency

The framework ensures that conflicts between writes to a single object are reconciled, and that complete synchronization results in mutually consistent replicas. Given a sufficient set of synchronization sessions without any intervening updates by applications to local replicas, all replicas eventually become mutually consistent.

The framework considers a conflict to have occurred if and only if there are two object-state instances for the same object, whose update histories each contain at least one update not contained in the other. The existence of conflict depends only on the mechanics of updates and synchronizations, and not on the semantics of the data; the detection of conflict is entirely the responsibility of the sync-store implementation, and not the application programmer. In contrast, the notion of conflict in Bayou is based on application semantics, and conflicts are detected by *dependency checks* [Ter95] supplied by the application. Application definitions of conflict can be quite sophisticated. [Kum95] gives the example of a write-write conflict between two versions of a file containing calendar information. If the conflicting updates create appointments that are two hours apart, and the two appointments are not in distant locations, the application can resolve the conflict by accepting both updates. As [Ter95] points out, a system that detects semantic conflicts can be programmed to detect mechanical conflicts. An application writer could (perhaps with the aid of helper classes provided by the framework) explicitly embed version vectors in application data objects, advance them as a part of the application's update and reconciliation operations, and examine them to detect conflicts. Clearly, there are applications for which the added flexibility of semantic conflict detection is useful. Furthermore, if applications requiring the detection of mechanical conflicts are rare, it makes sense for the storage burden of version vectors to be borne only by those applications requiring them. On the other hand, if such applications are common, it makes sense for the data-synchronization framework to relieve the programmer of responsibility for the bookkeeping.

The MNCRS data-synchronization working group deliberately chose not to enforce as strong a consistency condition as serializability. Fischer and Michael [Fis82] observe that there is an inherent conflict between serializability and availability in a distributed system, but that availability is a principal reason for deciding to replicate data in the first place. They assert that for applications such as appointment calendars, distributed e-mail in-boxes, and distributed file systems, availability is more important than serializability. They propose a synchronization model in which insert, delete, send and receive operations are *causally ordered*. Causal ordering is a weaker consistency condition than serializability, but it is precisely defined, and Fischer and Michael argue persuasively that it is intuitive and useful in practice.

Ladin, Liskov, Shrira, and Ghemawat [Lad92] extend the work of [Fis82], classifying update and query operations on replicated data as *causal*, *forced*, or *immediate*. The invocation of a causal update specifies the other operations on which it depends (and which therefore must be applied earlier), and the invocation of a causal query specifies the operations that must be executed before the query is processed. Causal operations at a node are blocked, if necessary, until all the updates that must precede that operation have been applied. Forced updates are totally ordered with respect to each other, and applied at each node in accordance with this order. The order is determined by one particular node, accessible to a majority of the nodes, designated as the *primary* node. Network partitions may cause different nodes to act as the primary at

different times. Immediate operations are executed with the same set of preceding operations at each node, using a three-phase protocol involving all nodes. Bayou *session guarantees* [Ter94] build on the notion of causal operations, associating updates with *sessions* so that dependencies can be computed automatically in accordance with user-controlled policies; and Bayou's *total order* for write operations [Pet97] achieves the effect of forced updates. However, immediate updates are inappropriate for a primarily disconnected network.

[Par83] notes that version vectors can be used to detect conflicts in updates to a single data item, but cannot detect nonserializability of a set of updates to different data items. That paper presents the example of two transactions, each of which read items *f* and *g*, one of which writes *f* and one of which writes *g*. These two transactions are in conflict in the sense that they cannot be serialized, but the version vectors for *f* and *g* do not indicate any conflict.

[Dav85] classifies strategies for ensuring correctness as *pessimistic* or *optimistic* and, orthogonally, as *syntactic* or *semantic*. Pessimistic strategies prevent conflicts, thus ensuring correctness, by limiting availability of data. Optimistic strategies allow replicas to be updated independently, but provide for the detection and resolution of conflicts that may result. Syntactic strategies define consistency purely in terms of one-copy serializability, based on the sets of items read and written by each transaction. Semantic strategies consider the contents of the data itself. It is widely agreed ([Guy90], [Kaw92], [Lu94], [Ter95]) that pessimistic approaches are inappropriate in a network with a significant number of primarily disconnected mobile devices. Pessimistic consistency algorithms generally presume that disconnection is a rare, transient state rather than a typical, long-lasting state. This presumption may be manifested in a number of different ways:

- A multiphase protocol may rely on timely communication with all nodes in the network.
- Writes, or reads and writes, may be blocked at a node that is not connected to a majority of the nodes, or to a designated primary node.
- Data may be protected from concurrent access by locks. A disconnected device can access such data only if it obtains the lock before it disconnects and releases the lock after it reconnects, making the data unavailable to the rest of the network for the entire period of disconnection, and requiring the user of a mobile device to hoard locks in anticipation of disconnected access.

As Section 3.4 explained, the cost of a strong consistency model is not restricted to reduced availability. A weaker model may be more appropriate for mobile replication because it frees up storage on a memory-constrained device, or because it allows incremental progress over unreliable links. Rather than preventing the missing-update anomaly depicted in Figure 5, a more practical compromise would be to alert the user to the presence of the anomaly. Just as Bayou data stores provide *full* and *committed* views of their data [Ter95], a sync store could provide a view of its data in which an object is flagged as being in a *suspicious state* if it is missing updates that occurred earlier than some update (to another object) that is reflected in the data store. The data-synchronization framework could provide the application with a means to determine which objects have suspicious states, and the application writer could choose to display the values of those objects to the end user in a special way. Suspicious states disappear by the next time the data store completes an uninterrupted synchronization with any replica that has received the

missing updates. Synchronization of a sync store containing suspicious states might be blocked to prevent propagation of suspicious states to other replicas; alternatively, objects in suspicious states might be propagated along with the suspicious-state flag.

Flagging of suspicious states could be achieved with only minor changes to our sync-store implementation. Upon marking an update-log entry as superseded, we would retain the log entry, but not the old object contents associated with the entry. During synchronization, as the update log was traversed in introduction order and updates were generated, `SyncUpdate` objects corresponding to superseded update-log entries would be generated at the appropriate points. These objects would belong to a new class, `SupersededSyncUpdate`. `SupersededSyncUpdate` objects would convey the sync ID of the updated `Reconcilable` object and the version of the update (but not the object contents, which have been discarded). A sync entry would contain two `SyncVersion` values: One would be advanced, as in the current implementation, each time an `ObjectContentsSyncUpdate`, `DeletionSyncUpdate`, or `VersionSyncUpdate` is applied, or an application marks the object as updated; the other would be advanced in each of these cases, but also when a `SupersededSyncUpdate` is applied. These two `SyncVersion` values would be equal except when the most recent remote update received for the corresponding `Reconcilable` object was a superseded update. This is precisely the case in which the object is in a suspicious state. It is safe to purge update-log entries for superseded updates when the superseding updates have been successfully transmitted to any potential recipient; the problem of determining when this condition holds is essentially the same as the problem discussed in Section 7.5, of determining when a deletion tombstone can be purged. In environments where communication failures are rare and storage is plentiful, a receiving synchronizer that batches all incoming updates before applying any of them can ignore `SupersededSyncUpdate` objects, sacrificing incremental progress to prevent missing-update anomalies; a sending synchronizer known to be communicating with such a receiving synchronizer need not transmit these objects.

9.1.6 Pluggable Components

The framework allows sync stores, synchronizers, and applications to be independently programmable. There is no doubt that we paid a performance penalty for the resulting information hiding. Nonetheless, this architecture facilitates the implementation of portable applications and the implementation of the framework on multiple platforms and communication media. It allows protocols to be chosen dynamically in response to application needs or currently available bandwidth. By restricting interdependencies among parts of the framework, it leaves flexibility for future extensions. The architecture does not preclude closely integrated implementations of multiple components.

9.1.7 Java-Centric Specifications

The framework is Java-centric. The Java orientation is manifested in the single-reference model of object storage and retrieval that was presented in Section 5.1. When a call on the `SyncStore` method `get` hands a reference to a stored object, the sync store loses the capability to count live references to the object. Neither the sync-store implementation nor the application program can safely free the object's storage. This is acceptable in a garbage-collected language like Java, but precludes a useful transliteration of the data-synchronization API to a language like C. Concern about the time and space efficiency of Java on small mobile clients was a major

reason that the MNCRS was not widely adopted. (A second consequence of the single-reference model is that the application programmer must be trusted to mark an object as updated.)

9.2 COSMOS State-Machine Models of Synchronizable Data Stores

Several parties expressed interest in the underlying MNCRS data-synchronization mechanisms without the Java API. Unfortunately, the behavior of these mechanisms was specified only indirectly, in terms of the behavior of the Java methods in the framework. This interest was expressed in questions like the following:

- "Can I use MNCRS data synchronization to synchronize with data on a non-Java device?"
- "Can MNCRS data synchronization form the basis of an enterprise-wide data-synchronization strategy accommodating non-Java data servers?"
- "Can I implement an MNCRS sync store in C?"
- "I want to synchronize MNCRS sync stores with other data stores, but I'm not interested in providing the application interface to sync stores. What does it mean to 'conform' to MNCRS without conforming to the full API?"
- "I'm trying to design a protocol for synchronizing MNCRS sync stores, but it is hard to deduce from the data-synchronization API what I can assume about the relationships among sync versions before a synchronization, or what relationships must be true after a synchronization. Can you provide a rigorous specification?"

In response to these concerns, we undertook to separate the underlying logic of data synchronization from the details of creating, maintaining, and querying data stores. We defined a formal state-machine model of a synchronizing data store, called the Co-Operative State Machine for Object Synchronization, or COSMOS.

The state of a COSMOS state machine includes the contents of a data store that is to be synchronized with other data stores. Such data stores contain values associated with keys. The contents of a data store may be modified by a *local mutation* (typically performed by an application program running on the platform that hosts the data store). Synchronization brings two state machines into states such that there is a one-to-one correspondence between the keys in one data store and the keys in the other, and also a one-to-one correspondence between the values associated with these keys. (Thus, COSMOS synchronization brings two state machines into isomorphic, but not necessarily identical, states.)

When the value associated with a given key is modified independently in two different state machines and synchronization propagates the change from one state machine to the other, the receiving COSMOS must *reconcile* the conflict, determining a value that should be associated with that key. There is a function *reconcile*(v_1,v_2), defined individually for every COSMOS, which takes a local value v_1 and a conflicting remote value v_2, and yields the value that should be used to reconcile the conflict. The *reconcile* function may evaluate either to one of its arguments or to some third value.

A COSMOS state-machine definition defines the set of updates generated by a COSMOS in a given state (in some cases, filtered to include only updates later than a specified version); the

latest update version already reflected in the state of a given machine; and the state transition that occurs when a remote update is applied to a machine in a given state. It does not define the form of the messages in which these versions and updates are conveyed or the transport conveying them. Neither does it specify the implementation of the *reconcile* function. The COSMOS model does not specify an application-programming interface for performing local mutations or determining the value currently associated with some key. Indeed, a COSMOS state machine acting only as a synchronization server need not have any such interface.

Questions about "non-Java MNCRS synchronization" can be understood as questions about non-Java implementations of the COSMOS state machine. An MNCRS sync store can be understood as a Java object implementing COSMOS rules, storing bound values in Java `Reconcilable` objects, and computing the COSMOS *reconcile* function by invoking an object's `reconcile` method. However, it is possible to implement a COSMOS state machine that has no connection to Java. Any two COSMOS implementations can synchronize with each other, provided that they use agreed-upon message formats and protocols.

We soon recognized the value of extending the COSMOS approach to data-synchronization models other than the MNCRS model. We developed simplified state-machine models for data stores synchronizing in a star topology and for pairs of data stores synchronizing only with each other. In fact, there are many topologies of interest. Among them, in order of increasing generality (and thus increasing complexity and overhead), are:

- the dedicated-pair topology (two stores synchronizing only with each other)
- star topologies
- hierarchical topologies with synchronization only between immediate parents and children
- hierarchical topologies with synchronization between arbitrary ancestors and descendants
- peer-to-peer topologies (the model originally derived from the MNCRS model)

We also recognized that each of these models reflected certain policy decisions. One such decision is whether transport should be assumed reliable (leading to a simple model) or unreliable (leading to a model that explicitly addresses error detection and recovery, and allows for the integration of error detection and recovery algorithms with synchronization algorithms). Other policy decisions involve tradeoffs between efficiency of normal operation and efficiency of error recovery. Work is currently underway to define families of COSMOS models and synchronization dialogs reflecting a wide variety of policies and topologies. This effort should lead to a better understanding of the performance implications of various policy decisions. The COSMOS effort will also provide a catalog of synchronization schemes that can be used to foster interoperability among independently developed products. Finally, COSMOS state machines provide a formal basis for proving properties of synchronization protocols.

9.3 The Mobile Data Synchronization Service

The Mobile Data Synchronization Service, or MDSS, is discussed in detail in [But00], so it will be described only briefly here. The MDSS architecture allows a variety of clients, including a Java object store, to synchronize with a variety of central enterprise databases. The architecture includes a mid-tier server that accepts query and update requests from mobile or desktop clients

and passes them on to a database-specific adapter. Clients and adapters communicate through the Mobile Data Synchronization Protocol, or MDSP. MDSP defines the form of an XML document for data exchange. Insertions, modifications, and deletions of items in a data store can be described in MDSP documents. MDSP documents are transmitted between the client and the mid-tier server by MQ Series Everyplace, a lightweight message queuing facility that provides a subset of MQ Series functionality. In particular, MQ Series Everyplace provides guaranteed delivery to and from mobile platforms. MDSP documents are encoded into WBXML [Mar99], a succinct encoding of XML submitted to the World Wide Web Consortium, and the WBXML byte stream is sent by enqueuing it on an MQ Series Everyplace queue as a single message object. When this message object is dequeued at the receiving end, the WBXML byte stream is decoded into an MDSP XML document.

We used our reference implementation of the MNCRS data-synchronization framework as the starting point for the MDSS Java object-store client. However, the MNCRS framework provides more general capabilities than are needed by MDSS. In MDSS, all synchronization is with a central server, and MDSS blocks updates to a client data store while synchronization is in progress. As a consequence of this restriction, and of the structure of an MDSS synchronization session, concurrent-update conflicts will never arise at an MDSS client. Therefore, the MNCRS conflict-detection and conflict-resolution capabilities are not needed in MDSS. We modified our implementation of the MNCRS framework to exploit the restricted way in which MDSS clients will use the framework, improving the client's efficiency. In addition, we plugged in the new implementation of the `Synchronizer` interface described in Section 6.4, which uses MQ Series Everyplace to send and receive updates encapsulated in WBXML-encoded MDSP documents.

This page intentionally left blank.

Acknowledgments

This paper describes the results of two collaborative efforts, one by the data-synchronization working group of the MNCRS consortium to specify the framework and one by IBM researchers at the Thomas J. Watson Research Center to implement it. All of the participants in both these efforts could be considered coauthors of this paper; however, the opinions expressed about the strengths and shortcomings the framework are my own.

My colleagues in the MNCRS working group were Lonnie Hansen of Arkona; Henry Kings of Ericsson; Yoshifumi Miyata of Fujitsu; Yoshinori Kishimoto of Hitachi Ltd.; Maria Butrico, Henry Chang, and Apratim Purakayastha of IBM Research; Shinsuke Mitsuma and Hiroki Murata of IBM Japan; Jeremy Jones of Lotus Development; Tetsuo Maeda of Matsushita; Seiji Fujii, Masahiro Kuroda, Hideaki Okada, Ryoji Ono, and Mariko Yoshida of Mitsubishi; John Howard and Luosheng Peng of Mitsubishi Electric Information Technology Center America; Ken Chan of Nortel; Rafiul Ahad and Jiader Day of Oracle; Takao Ikoma of Sharp; Teck Yang Lee, Brian Raymor, and Roger Riggs of Sun Microsystems, Inc.; and Hidekazu Izumi, Satoshi Hoshina, and Tetsuro Muranaga of Toshiba. Henry Chang and Masahiro Kuroda preceded me as chairmen of the working group; Roger Riggs preceded me as editor of the framework API, and organized our disparate early thoughts and discussions into coherent descriptions of the architecture and API . Maria Butrico, Henry Chang, Jeremy Jones, Masahiro Kuroda, Hideaki Okada, Luosheng Peng, Teck Yang Lee, and Roger Riggs played especially active roles in the working group and profoundly influenced the framework.

My colleagues on the IBM implementation team were Maria Butrico, Henry Chang, John Givler, Ajay Mohindra, and Apratim Purakayastha.

This page intentionally left blank.

References

[App98] Apparao, Vidur, Byrne, Steve, Champion, Mike, Isaacs, Scott, Jacobs, Ian, Le Hors, Arnaud, Nicol, Gavin, Robie, Jonathan, Sutor, Robert, Wilson, Chris, and Wood, Lauren, eds. *Document Object Model (DOM) Level 1 Specification, Version 1.0*. W3C Recommendation REC-DOM-Level-1-19981001, October 1, 1998 <URL: http://www.w3.org/TR/REC-DOM-Level-1/>

[But00] Butrico, Maria, Cohen, Norman, Givler, John, Mohindra, Ajay, Purakayastha, Apratim, Shea, Dennis, Cheng, Josephine, Clare, Don, Fisher, Gerry, Scott, Rob, Sun, Yudong, Wone, May, and Zondervan, Quinton. Enterprise data access from mobile computers: an end-to-end story. *Proceedings of the 10th IEEE Workshop on Research Issues in Data Engineering*, February 27-28, 2000, San Diego, California (to appear)

[Dav85] Davidson, Susan B., Garcia-Molina, Hector, and Skeen, Dale. Consistency in partitioned networks. *ACM Computing Surveys* **17**, No. 3 (September 1985), pp. 341-370

[Dem94] Demers, Alan, Petersen, Karin, Spreitzer, Mike, Terry, Douglas, Theimer, Marvin, and Welch, Brent. The Bayou architecture: support for data sharing among mobile users. In Cabrera, Luis-Felipe, and Satyanarayanan, Mahadev, eds., *Workshop on Mobile Computing Systems and Applications*, December 8-9, 1994, Santa Cruz, California. IEEE Computer Society Press, Los Alamitos, California, 1995, pp. 2-7

[Emb98] Embedded systems software. IBM Tokyo Research Laboratory <URL: http://www.trl.ibm.co.jp/projects/embtec/embsft_e.htm>, June 30, 1998

[Fis82] Fischer, Michael J., and Michael, Alan. Sacrificing serializability to attain high availability of data in an unreliable network. *Proceedings of the ACM Symposium or Principles of Database Systems*, March 29-31, 1982, Los Angeles, California, pp. 70-75

[Gam95] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, 1995

[Gel85] Gelernter, David. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* **7**, No. 1 (January 1985), pp. 80-112

[Gol99] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D. *HTTP Extensions for Distributed Authoring—WEBDAV*. Request for Comments 2518, Internet Engineering Task Force. <URL: http://www.ietf.org/rfc/rfc2518.txt>

[Guy90] Guy, Richard G., Heidemann, John S., Mak, Wai, Page, Thomas W., Jr., Popek, Gerald J., and Rothmeier, Dieter. Implementation of the Ficus replicated file system. *Proceedings of the Summer USENIX Conference*, June 1990, Anaheim, California, pp. 63-71

[Ham97] Hamilton, Graham, ed. *JavaBeans*, version 1.01. Sun Microsystems, July 24, 1997. <URL: http://java.sun.com/beans/docs/beans.101.pdf>

[Hap98] Hapner, Mark, Burridge, Rich, and Sharma, Rahul. *Java Message Service*, version 1.0.1. Sun Microsystems, October 5, 1998. <URL: http://java.sun.com/products/jms/jms-101-spec.pdf>

[Jav99] *JavaSpaces Specification*, version 1.0. Sun Microsystems, January 25, 1999. <URL: http://www.sun.com/jini/specs/js.pdf>

[Kaw92] Kawell, Leonard, Jr., Beckhardt, Steven, Halvorsen, Timothy, Ozzie, Raymond, and Greif, Irene. Replicated document management in a group communication system. In David Marca and Geoffrey Bock, eds., *Groupware: Software for Computer-Supported Cooperative Work*. IEEE Computer Society Press, Los Alamitos, California, pp. 226-235

[Knu73] Knuth, Donald E. *The Art of Computer Programming. Volume 1, Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, Massachusetts, 1973

[Kum93] Kumar, Puneet, and Satyanarayanan, M. Supporting application-specific resolution in an optimistically replicated file system. *Fourth Workshop on Workstation Operating Systems*, October 14-15, 1993, Napa, California. IEEE Computer Society Press, Los Alamitos, California, 1993. pp. 66-70

[Kum95] Kumar, Puneet, and Satyanarayanan, M. Flexible and safe resolution of file conflicts *Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*, January 16-20 1995, New Orleans, Louisiana, n.p.

[Lad92] Ladin, Rivka, Liskov, Barbara, Shrira, Liuba, and Ghemawat, Sanjay. Providing high availability using lazy replication. *ACM Transactions on Computer Systems* **10**, No. 4 (November 1992), pp. 360-391

[Lu94] Lu, Qi, and Satyanarayanan, M. Isolation-only transactions for mobile computing. *Operating Systems Review* **28**, No. 2 (April 1994), pp. 81-87

[Mar99] Martin, Bruce, and Jano, Bashar. WAP binary XML content format. W3C Note, June 24, 1999. <URL: http://www.w3.org/TR/wbxml/>

[Mon98] Montenegro, Gabriel. MNCRS: industry specifications for the mobile NC. *IEEE Internet Computing* **2**, No. 1 (January-February 1998), pp. 73-77

[Ope98] Open Group. *Network Computing Client*. Open Group Technical Standard C801, October 1998

[Par83] Parker, D.Stott, Popek, Gerald J., Rudisin, Gerard, Stoughton, Allen, Walker, Bruce J., Walton, Evelyn, Chow, Johanna M., Edwards, David, Kiser, Stephen, and Kline, Charles. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering* **SE-9**, No. 3 (May 1983), pp. 240-247

[Pet97] Petersen, Karin, Spreitzer, Mike J., Terry, Douglas B., Theimer, Marvin M., and Demers, Alan J. Flexible update propagation for weakly consistent replication. *SIGOPS '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 5-8, 1997, Saint-Malo, France, pp. 288-301

[Rat96] Ratner, David, Popek, Gerald J., and Reiher, Peter. Peer replication with selective control. UCLA Technical Report CSD-960031, July 1996

[Rat97] Ratner, David, Reiher, Peter, and Popek, Gerald J. Dynamic version vector maintenance. UCLA Technical Report CSD-970022, June 1997

[Sar87] Sarin, Sunil K., and Lynch, Nancy A. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering* **SE-13**, No. 1 (January 1987), pp. 39-47

[Ter94] Terry, Douglas B., Demers, Alan J., Petersen, Karin, Spreitzer, Mike J., Theimer, Marvin M., and Welch, Brent B. Session guarantees for weakly consistent replicated data. *Proceedings International Conference on Parallel and Distributed Information Systems (PDIS*), September 1994, Austin, Texas. IEEE Computer Society Press, Los Alamitos, California. pp. 140-149

[Ter95] Terry, Douglas B., Theimer, Marvin M., Petersen, Karin, Demers, Alan J., Spreitzer, Mike J., and Hauser, Carl H. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SIGOPS '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 3-6, 1995, Copper Mountain Resort, Colorado, pp. 172-182

[Whi98] Whitehead, E. James, Jr., and Wiggins, Meredith. WEBDAV: IETF standard for collaborative authoring on the web. *IEEE Internet Computing* **2**, No. 5 (September-October 1998), pp. 34-40

[Wyc98] Wyckoff, P., McLaughry, S.W., Lehman, T.J., and Ford, D.A. T Spaces. *IBM Systems Journal* **37**, No. 3 (1998), pp. 454-474