An Application-Agnostic Replication System for Ubiquitous Computing

James O'Brien, Marc Shapiro
Microsoft Research
7 J J Thompson Ave.
Cambridge, UK
i-jameso@microsoft.com

Abstract: *In this paper we present Joyce, a platform and programming framework that enables applications to form the kind of variably-connected, data-sharing group that typifies ubiquitous computing. The platform captures the semantics of application operations via a system of actions and constraints, which it distributes to peers in a group using epidemic propagation. Our system presents several advantages: (1) we capture a rich semantic model of application usage across all participants in a group, (2) we persist this rich semantic model independently of the constituent applications, (3) propagation of the semantic model is tolerant of varying connectivity (4) we use the semantic model to reconcile concurrent modifications from separate participants, (5) we encapsulate all of this functionality in a piece of application-agnostic middleware.*

Keywords: mobility, collaboration, reconciliation, frameworks

1 Introduction

There are two implicit themes in ubiquitous computing ('ubicomp') that current applications are ill-equipped to meet. The first is the pervasive sharing of data between devices and between people in a mobile context. As a user moves between computational environments, he might be disconnected for long periods, but he would still like to be able to access and modify his data during the disconnection.

The second theme is collaborative work. Collaboration infrastructures in ubicomp must not only support multiple collaborators on a data-set but must also facilitate a smooth transition between synchronous and asynchronous collaborative modes, which result either from user choice or from varying connectivity. Application interaction in both collaborative modes should look and feel as similar as possible to the user. Moreover, it will be common that both modes will be in use within the same collaborative group.

This is an inversion of the model used to create applications today (which we will term the 'personal computing' model). This model assumes that data is modified by one user using one device and that the data is essentially owned by

and contained within that user/device combination. This assumption is reflected in the programming frameworks used to construct these applications.

Certain classes of application (most notably PIM applications) have attempted to solve the varying connectivity problem. However, the techniques employed have been very specific to the data domain and very intrusive to the application. The application must dedicate a significant amount of specialized logic, often to the detriment of its core functionality. Further, these applications are usually designed around lock-step synchronization of devices and require a fair amount of user intervention; they do not support the fluid, multi-synchronous collaboration requirements of an ubicomp environment. Finally, each application implements its own synchronization logic, leaving them segregated and incompatible with one another.

Even if all of these systems were already suitable for ubicomp we should not think of 'ubicomp applications' but rather applications that happen to be running in an ubicomp environment. The mechanics of participating in that environment should be separate from the application; encapsulated in a piece of ubicomp middleware.

In this paper, we propose just such a system. Our platform, named Joyce, is an application-agnostic replication system that enables devices to form adhoc, data-sharing groups. Joyce has application-independent logic for group formation, variable connectivity, data replication and reconciliation, and multi-synchronous collaboration. It delivers this functionality to the distributed applications as unobtrusively as possible, by intercepting the standard Model-View-Controller interaction cycle.

This paper proceeds as follows. In Section 2 we give a brief description of the key issues that must be addressed, and from these derive a set of requirements for the framework. Section 3 gives a system overview and describes the concepts that underlie the framework. Section 4 describes the application model defined by our framework. Section 5 describes the components supplied by the framework to enable applications to participate in data-sharing groups. Section 6 describes how shared data is kept synchronous. Section 7 concludes.

2 Key Issues

The kind of ubicomp environment that the community has envisioned since Weiser introduced the topic [Weiser 91] has been characterized by a proliferation of mobile devices that form spontaneous, data-sharing networks as they move through the "computational landscape" [Abowd 00].

Supporting such groups involves several key issues that differentiate mobile ubicomp applications from isolated, personal computing applications. It is the role of the framework to deal with these issues while masking the problems involved from the application logic.

An ubicomp application is never alone in modifying a piece of data, but it is always a part of a group of participants, even while it is disconnected. Although the set of members of a group may be fairly static, connectivity changes dynamically; members may be disconnected for unbounded periods. In spite of this, applications should be able to access their data whether they are connected to a group or not and that data should be kept consistent with the modifications from the rest of the group. Varying connectivity however, rules out any kind of strong consistency guarantees. Instead, we provide an *optimistic* replication system, where data is replicated and each device is allowed to modify its local copy [Saito & Shapiro 2004]. At some later point, the concurrent modifications from the different devices are *reconciled* to create a common state.

The issues above suggest the key problems that should be in the domain of the framework rather than the application.

- **Update propagation:** The framework needs to propagate modifications from one participant to all the others. The propagation scheme used should ensure that, despite varying connectivity, every group member will receive the modifications of every other member. Furthermore, since Joyce is designed to be application agnostic we must represent modifications in an application-independent way.

- **Reconciliation:** Replicas may be independently modified in a conflicting manner. Reconciliation is the process of bringing replicas to a consistent state by detecting and resolving conflicting concurrent modifications.

  The process of detecting and resolving conflicts depends on application semantics and user intent. Existing reconcilers [Balasubramaniam & Pierce 1998] are confined to a single data type. Joyce remains application agnostic, by representing application semantics and user intents explicitly.

- **Consistency:** Individual replicated states are allowed to diverge in the short term. However, all states must eventually be made consistent according to the results of reconciliation. Joyce has a mechanism for bringing a state to consistency, concurrent with the user independently modifying that state. The application must remain as responsive as possible throughout this process.

- **Application Usability:** Joyce should not degrade the performance and responsiveness of the application. This is especially important in several key areas: transitioning from connected to disconnected states, transitioning between asynchronous and synchronous collaborative modes, and bringing a state to consistency.

3 System Overview

Joyce is a middleware framework designed to provide solutions to the requirements above and decouple the details of those solutions from applications.

Joyce connects *participants* that are all working on the same shared data set and distributes the modifications made by one participant to all the others. It allows participants to disconnect and reconnect without loss of information or responsiveness; an application can continue to run while disconnected and modifications will be propagated to it on reconnection.

There is no fully automatic solution to reconciliation, since resolving conflicting concurrent actions depends on the application semantics and user intents. To avoid "wiring in" knowledge of application internals, Joyce is based on a reified model of application semantics.

In this section, we present a high-level view of how the system works and the core concepts used.

3.1 Joyce Groups

A Joyce *group* represents the set of participants working together on some shared data store. Each participant works with a local, replicated instance of the data. At any point in time, a member is connected to zero or more other members. A participant remains a member of its group even when it has no connectivity.

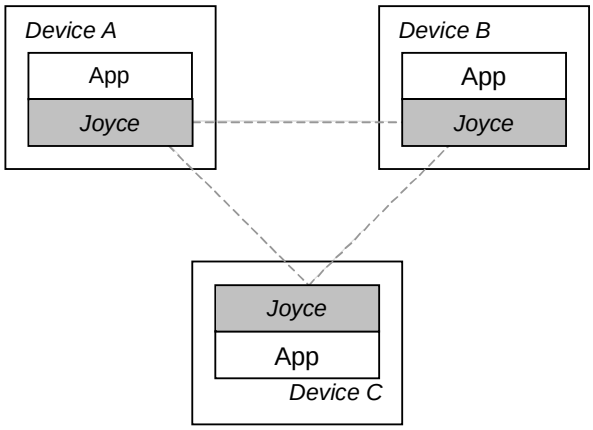

Figure 1 Three applications on different devices use Joyce to form a collaborative group. In this instance the group is fully connected. Note that the members of the group are the *applications* working on the data, not the devices. The applications are still part of the group even if their devices are disconnected.

3.2 Logging

To participate in the group, applications log their data modifications using *actions* and *static constraints*, which are designed to explicitly articulate the intended semantics. An action is an object that reifies the invocation of some

application operation, such that it can be transmitted and applied at any member. A static constraint is an object that reifies a semantic invariant, which the system is entrusted to maintain.

As a member makes modifications to the shared state, they are recorded as actions into a log. User intents and application semantics are recorded as *log constraints*, a sub-class of static constraints.[1] The concurrency semantics of shared objects constitute *object constraints*, another sub-class of static constraints.

An action has an `execute` method that is used to replay a particular invocation on a state. It may also have a `compensate` method that is used to un-apply the invocation from a particular state.

Static constraints are semantic relations between actions that must hold, irrespective of the state that actions are run on. Joyce uses the set of constraints defined by Preguiça et al. [2003a], explained next. Joyce also supports *dynamic constraints*, which are assertions about the current state that must hold at run time.

3.2.1 Object constraints

Object constraints indicate how *classes* of action relate to each other. Object constraints are exported as a set of methods comparing two actions. Joyce currently defines the following set of object constraints:

- **Commutes:** Do the supplied actions commute? Is the result of executing the two actions independent of execution order? This information is used by the Joyce *scheduler* to optimise both forward execution and rollback.
- **Helps:** Does running the first action before the second *increase* the chances of the second action's dynamic constraints succeeding? If so, the scheduler will try to run them in that order.
- **Hinders:** Does running the first action before the second *decrease* the chances of the second succeeding? If so, the scheduler attempts to run them in opposite order.
- **Enables:** Can the second action be run *only* if the first action has succeeded? If so, the scheduler will schedule them accordingly.
- **Prevents:** Does running the first action *prevent* the second action from succeeding? If so, the scheduler attempts to run them in opposite order.

These object constraints represent the semantic relations between pairs of concurrent actions, i.e., the concurrency semantics of shared objects.

[1] This is in contrast with previous systems, where the log records only the chronological order of operations [Petersen et al. 1997].

3.2.2 Log Constraints

Log constraints express invariants that must hold between action instances that share a log (as opposed to object constraints that express invariants between *classes* of action). We have factored log constraints into two categories: *grouping* constraints and *ordering* constraints. Currently there are two types of group:

- **Parcel**: Confers atomicity to the grouped actions (i.e. either all the actions must be executed or none of them can be.) The scheduler will either schedule all, or none of them.
- **Alternative**: Indicates that only one of the grouped actions can be executed. The reconciler chooses between them.

Ordering constraints indicate that the constrained actions should execute in the specified order. Following Preguiça et al. [2003b]:

- **Strong ordering** indicates that, if the predecessor is not executed, then neither can the successor be.
- **Weak ordering** indicates that if the successor has already executed, then the predecessor may *not* (but the other way around is OK).

Log constraints are typically used to express higher level application semantics (i.e., a subtask within an application that consists of more than one command) or user intents within a particular task.

3.3 Epidemic Propagation

Joyce uses an *epidemic propagation* [Demers 87] scheme to distribute modifications around the variably connected Joyce group. Epidemic propagation distributes logs by making a series of pair-wise exchanges between connected peers in the group. Currently, exchanges may happen as the result of a connectivity status change (for example when a member joins a group), they may be timed to occur at certain intervals or they may be timed to happen during an interaction pause (i.e. if no new actions have been appended to a log after a certain period.)

Epidemic propagation guarantees (with high probability) that each member receives each other member's log, possibly through intermediate members, given sufficient connectivity [Demers 87]. Epidemic propagation is also well adapted to multi-synchronous collaboration. When connectivity is good, we propagate modifications frequently, thus minimizing divergence. When connectivity is nil, we allow the states to diverge until connection is restored. When connectivity is poor (e.g., over a cell phone connection) we can still take advantage of available bandwidth to propagate "important" updates with highest priority. Since the same epidemic mechanism is used in all cases, we can support a mixture of connectivity modes. [Edwards 97]

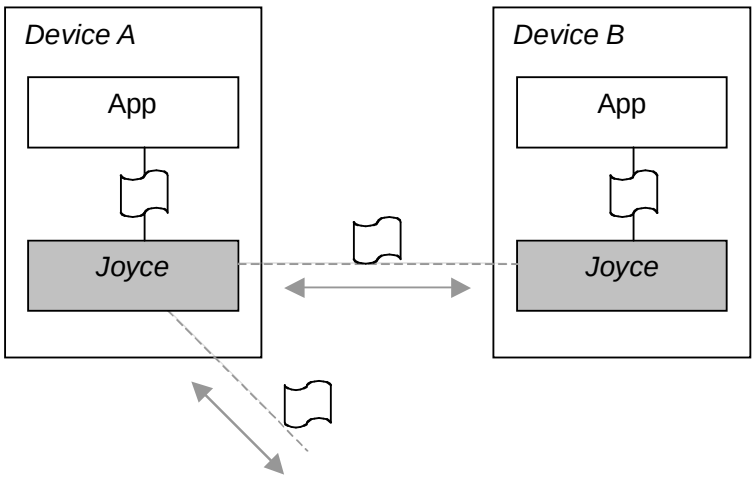

Figure 2 Participants in a group log their operations. The logged operations are then propagated to the other participants by Joyce.

3.4 Reconciliation

Modifications on individual participants are made without coordination with the other members of a group. This helps us maintain the performance of a participant and allows it to continue working when disconnected. However, the data on each participant is supposed to reflect a shared state common to all group members. Every isolated modification causes a participant to *diverge* from that common state.

Reconciliation is the act of merging concurrent logs to produce a common, non-conflicting schedule. Most existing reconcilers merge according to a pre-determined order (for example timestamp order in Bayou). In contrast, Joyce builds on our previous reconciliation system IceCube [Preguiça et al. 2003a], which uses the semantic information provided by constraints to produce a best schedule that preserves the application semantics.

IceCube treats reconciliation as an optimization problem. It merges all the logs supplied to it into one large semantic graph with actions as nodes and ordering constraints as edges. A schedule is a traversal of this graph such that all the static constraints are satisfied. Any actions not traversed are dropped. If a dynamic invariant fails, the scheduler backtracks and tries a different traversal. The heuristic scheduling finds a traversal that minimizes the value of the dropped actions.

One member of the group, the *primary*, is responsible for reconciling the logs received from all the members and *committing* a reconciled schedule. The actions committed in this schedule generate the common state.

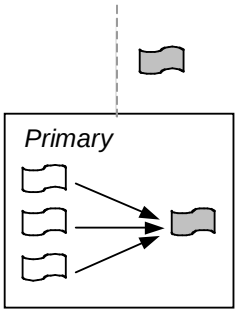

Figure 3 The *primary* member reconciles concurrent modifications from the other participants into a *commit log*, which it then propagates.

Knowledge of the common state is distributed to the non-primary participants by propagating the commit result in the same way as the other logs. The participants can synchronize themselves to the common state by running the committed actions but they may choose not to do so immediately for a number of reasons:

- Synchronizing to the common state may involve halting interaction as the committed actions are applied. It may be more important not to interrupt the user.
- A modification the user has made may conflict with a committed modification. The user should be given an opportunity to fix the conflict.
- The application may be disconnected from the network. Instead of disallowing modification when disconnected we should let the user continue to modify his local data and synchronize to the common state when he reconnects.

For these reasons, applications monitor both the local, divergent state and the common state produced by the primary and should preferably highlight the difference between them to give the user an idea of how far his state has diverged.

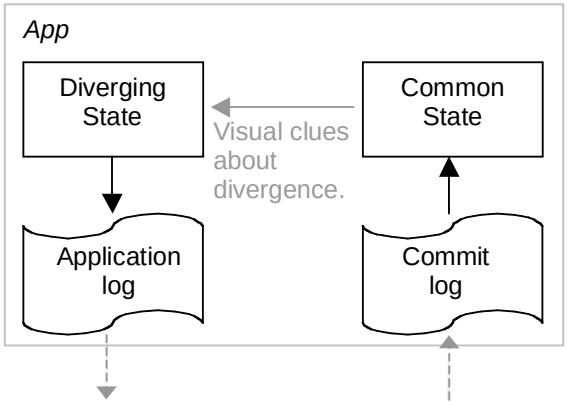

Figure 4 Applications may diverge from the common state. They should preferably give some visual clue about the extent of the divergence.

4 Application Model

Most current application design is based around the Model-View-Controller partitioning pattern [Krasner 88]. This design pattern was introduced with Smalltalk and is encapsulated in frameworks such as the MFC, ATL, WinFX and Cocoa.

The pattern introduced a standard *interaction cycle* where input from the user is evaluated into a set of model modification messages by the *controller*. These messages are sent to the *model* component, which applies them and sends a set of model change messages to the *view*. The view reflects the effects of the model changes on some output device.

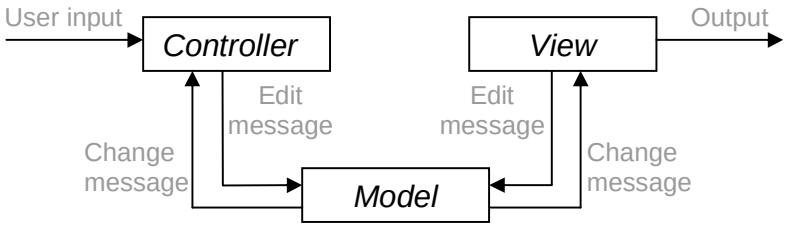

Figure 5 The traditional MVC interaction cycle

This model simplifies the construction of applications but is insufficient for mobile, collaborative environments since it assumes that applications are isolated. More specifically, it assumes that modifications to the model always come from the local controller (and inversely those modifications from the controller are always destined for the local model.) The pattern also has the more subtle assumption that the local controller is the authoritative source of all modifications and modifications are linear – it has no notion of concurrent modifications, conflicting modifications or reconciliation.

The interaction cycle in Joyce must not make any of these assumptions. We expand MVC by introducing another component, the *coordinator*, which is responsible for interfacing with the Joyce system.

4.1 The Interaction Cycle in Joyce

In Joyce, the interaction cycle operates in two modes. The most common mode is active when the user is modifying this local state without regard to the other participants. That is, he is causing his local state to *diverge* as outlined in *section 3*.

This mode is much like the traditional MVC interaction cycle; user interaction is transformed by the controller into a set of actions and constraints that are logged and executed on the model.

This cycle (illustrated in *figure 6*) is called the *tentative* cycle and the actions generated are *tentative actions*. The actions are tentative since they have only been applied to the local model. For the actions to become permanent they must be committed by a primary.

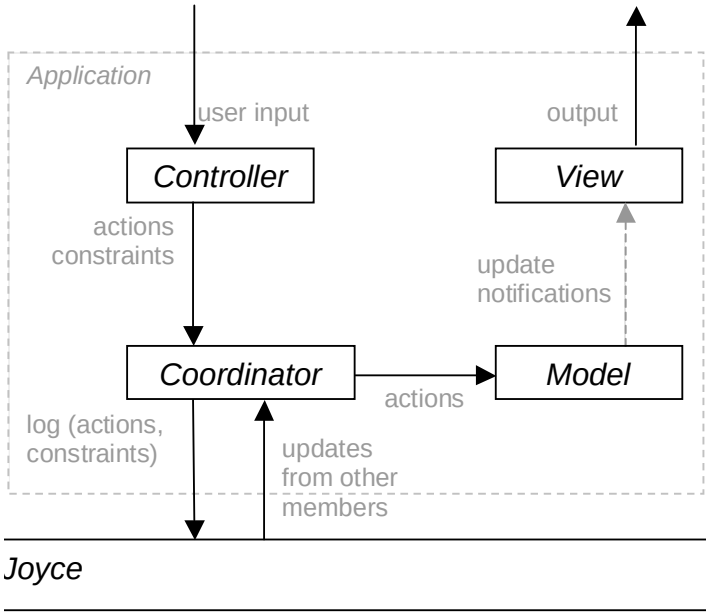

Figure 6 The tentative interaction cycle in Joyce. The controller and sends them to the coordinator for execution and logging.

There follows a brief overview of the pattern components.

4.1.1 Model

The model component encapsulates the application's state; it is this state that is replicated across the participants in a group.

Joyce does not need to know anything about the structure of the model. The interface for modifying it is entirely encapsulated in the set of actions supplied by the application. The model is modified by applying an action instance to it.

4.1.2 View

Views represent the graphical representation of the model. As with standard MVC views update themselves in response to notifications. Unlike standard MVC, our notifications may also refer to status change information from the Joyce system.

4.1.3 Controller

Controllers translate user input into model modifications that, in Joyce, are encapsulated as a constrained set of actions. Therefore, the controller's job is more specific in our framework – it is responsible for generating a set of actions and constraints that represent a modification to a state in response to a user interaction.

4.1.4 Coordinator

The coordinator is the bridge [Gamma 95] between Joyce and the application. It sits between the controller and the model in the interaction cycle and is responsible for sending actions to the log and keeping the model consistent with commitment results.

5 The Framework

The framework supplies both the coordinator component and the components required for interacting with a Joyce group.

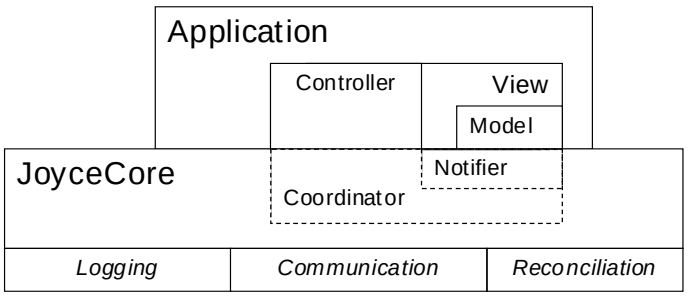

Figure 7 The architecture of a typical application using the Joyce system. The model, view(s) and controller remain specific to the application whilst the coordinator is supplied by Joyce.

There were two major considerations when designing Joyce. Firstly we must provide a stable API that is easy to understand, easy to program to and that removes as much of the reconciliation burden as possible. Secondly, we must ensure that Joyce can facilitate new research into reconciliation, accommodate improvements from that research and deliver those improvements to application programmers.

Joyce provides a set of components that implement the techniques described in section 3 and a library, *JoyceCore*, that acts as a wrapper façade [Schmidt 00] between the application and these components. This architecture allowes improved components to be plugged into JoyceCore without any changes to the applications. The components cover the following functional areas:

- **Logging**: Defines how a group member should log modifications.
- **Communication**: Describes how modifications are propagated to other participants in the group. In particular, this area defines the format of the *multi-log* used in epidemic propagation.
- **Reconciliation**: Describes how the entity providing the reconciliation service should behave. Defines what the input to a reconciler looks like and what the output from a reconciler should look like.

We give a brief overview of how each functional component accomplishes its job followed by a description of the application model used by the framework.

5.1 Logging

This component implements the logging logic used to record application operations. The format of the log produced has to meet two requirements: firstly,

it must implement the action/constraint graph model described in Section 3. Secondly, it should be easy to exchange logs and fragments of logs between nodes in a Joyce group, and such exchanges should be as resilient as possible to connectivity failures.

In the current implementation a log is implemented as a collection of *log elements*, each of which has a monotonically increasing sequence number. Each log element has a property `originatorID` that identifies the application that generated the element (for example "`com.microsoft.word`") and a property, `nodeID`, that identifies the group member on which the element originated.

We currently define two types of log element:

1. **Action elements** represent an action (as described in section 3). It contains an identifier, the `actionType`, of the command object that was invoked and the set of parameters it was invoked with.
2. **Meta-action elements** provide information about actions that precede them in the same log.

The most common use for meta-actions is to record log-constraints. For example, consider an application that resolves a drag-and-drop user interaction into two operations, *cut* and *paste*, which must be executed atomically. *Figure 8* shows the sub-log recording the two actions.

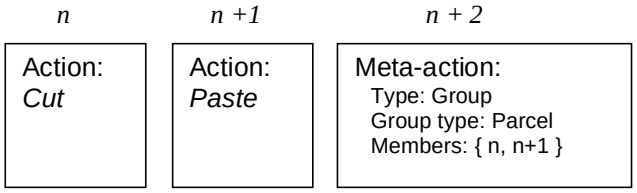

Figure 8 A drag-and-drop is recorded as a parcel containing *cut* and *paste* actions. The meta-action specifies the type of constraint and the sequence numbers of the constrained actions.

Sometime later, the user runs a spell-check that results in the pasted text being altered. This is resolved into a replace-text operation that must be strong-ordered after the paste operation (*figure 9*).

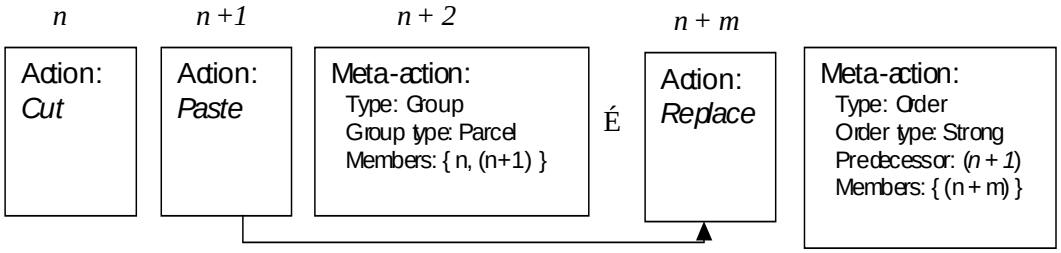

Figure 9 A replace action caused by a spell-check must be strong ordered after the insertion of the text it has changed. The meta-action specifies the sequence number of the predecessor and successor actions.

Log constraint meta-actions can be thought of as instructions to a member receiving the log about how to build the semantic graph that the log describes.

Using the meta-action scheme allows us to capture all the information about a log in a sequential collection of elements.

Each log also keeps track of a position in the log called the *last-commit mark*. This indicates the point in the log at which this replica was last in a non-divergent state (viz., when it applied the committed log, explained shortly). Every log element that was appended after this mark is *tentative* .

5.1.1 Multi-logs

For each member of a group the Joyce system on that member maintains an object called the *multi-log* that represents that member's view of the group. The multi-log will contain at least two logs; one for the member on which the table resides, termed the *node log*, and one that indicates which actions have been committed and aborted, termed the *commit log*. Furthermore, a member retains a log in its multi-log for every member that it ever has heard from. The multi-log forms part of the bridge between the Joyce system and the application. When a coordinator logs actions and constraints it logs them to its local multi-log.

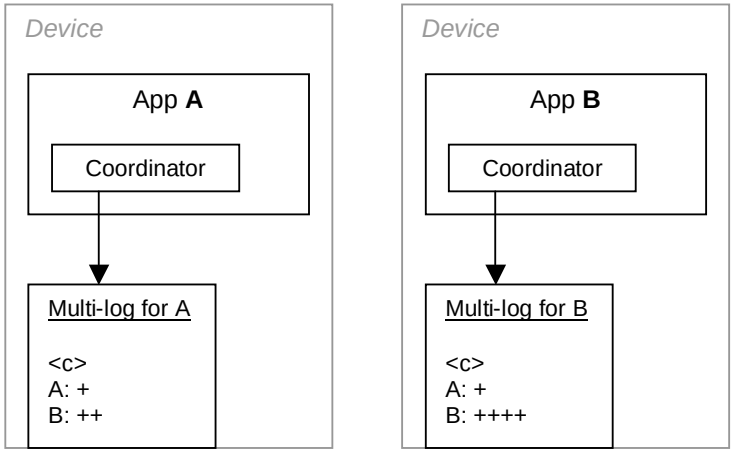

Figure 10 Applications on two devices, though currently disconnected, have entries in their multi-logs for each other. The commit logs (labeled <c>) are empty.

5.2 Communication

The combination of the multi-log construct and the sequential numbering of elements in a log allows us implement a very simple scheme for the exchange of updates. The multi-log is kept up-to-date via a series of pair-wise *vector-clock exchanges* between members that have a direct connection.

5.2.1 Vector-clock Exchanges

A vector-clock records how fresh a multi-log is by recording the last sequence number for each log in the multi-log. Periodically a member will send its vector-clock to a peer and receive the peer's vector-clock in return. Both members

examine the vector-clock they receive to see if there are any fresher logs on the other member. If a member discovers that its peer member has a fresher record for a log then it asks the peer for all the new log elements in that log. This process is called a *vector-clock exchange*.

Consider a case where we have a Joyce group of only three members, we start at a point where all members are disconnected:

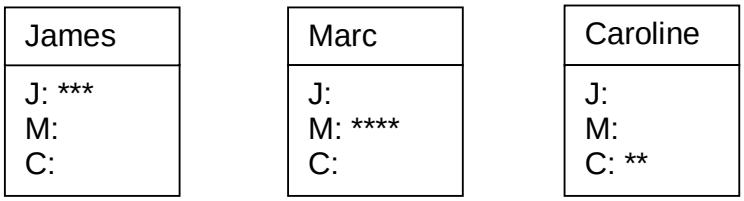

Figure 11 James, Marc and Caroline have been making modifications while disconnected. ***** represents a log element. The commit log is omitted for clarity.

Each member of the Joyce group has been making modifications whilst disconnected. James has generated 3 log elements, Marc has generated 4 and Caroline 2. Because of the disconnection, no node is aware of peer modifications. Now a link is established between Caroline's node and Marc's node and they exchange vector-clocks. After examining the vector-clock she received, Caroline asks Marc for the updates that she does not have, and vice-versa.

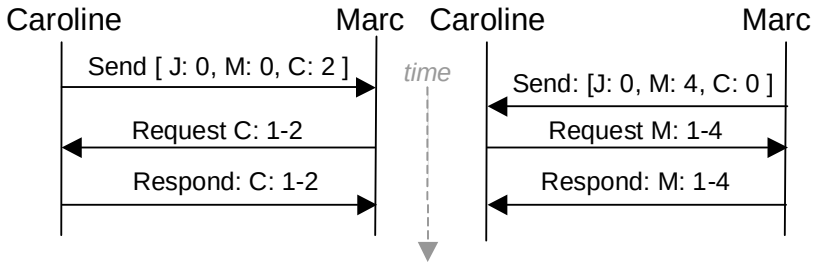

Figure 12 Caroline sends her vector clock to Marc and receives Marc's in return. She sees that Marc has a fresher record for his own log and requests sequence numbers 1 – 4 from that log. In parallel, Marc examines the vector clock that initiated the exchange, sees that is contains a fresher log for *C* and requests the updates for that log.

At the end of the exchange Marc and Caroline's multi-logs mirror each other:

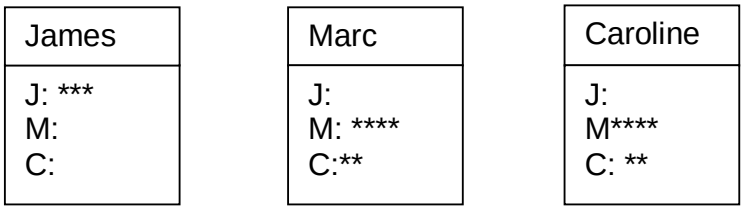

Figure 13 After the exchange Marc and Caroline's multi-logs are the same.

Sometime later, James contacts Marc and a similar exchange happens, this time however James discovers that Marc has fresher records for *two* logs and so makes two requests for updates. These update requests may also happen concurrently:

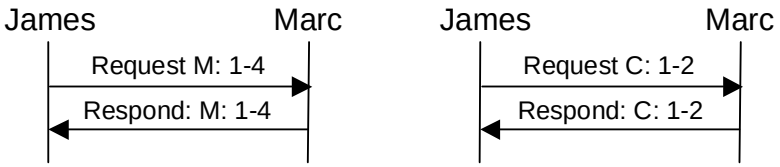

Figure 14 James discovers Marc has fresher records for two logs and so makes two concurrent update requests.

The result after this exchange will be:

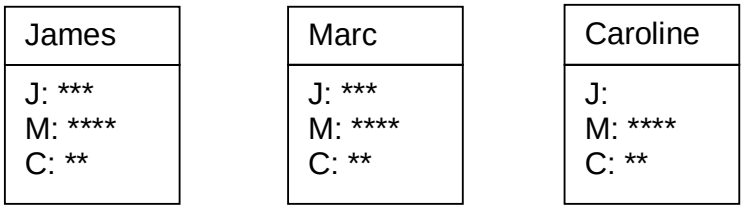

Figure 15 Caroline's changes have propagated to James.

Caroline's changes have *propagated* to James despite the fact that no direct connection has been made between them; it is this technique that allows us to accommodate varying connectivity.

5.3 Reconciliation

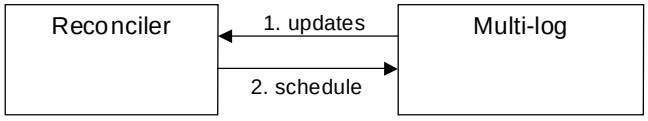

Figure 16 The multi-log of a primary node will have a reference to a reconciliation engine. The engine takes in and produces an update to the **commit log**

Reconciliation creates a schedule of actions from the multi-log that satisfies the constraints. *Commitment* is the act of irrevocably selecting a reconciliation schedule (out of the many possible ones) for execution on every site, in order to make them consistent. The schedules that have been committed are recorded in the multi-log as the *commit log*. The commit log consists of *commit* and *abort* meta-actions that identify the actions that are committed (irrevocably scheduled for execution) or aborted (irrevocably excluded from execution). The commit log may also contain ordering constraints, providing an irrevocable ordering of non-commuting committed actions.

Schedules are generated by a *reconciliation engine* provided by the framework. This reconciliation engine is a strategy object [Gamma 95] implementing a reconciliation algorithm & driven by the multi-log. The engine

takes the tentative updates from the multi-log and splits them into those that are executed and those that are rejected. If the user is satisfied with the schedule it is appended to the commit log and propagated.

6 Commit Log Updates

The tentative cycle is interrupted when the application decides to apply a commit log update (i.e. when the application synchronizes its state to the common state).

The commit log represents the authoritative version of the replicated state via the actions that must be run to generate that state. It is updated by either a local reconciliation (if the member is primary) or a vector-clock exchange. Either way, a commit log update indicates that the application should synchronize its local state with the commit log if it wishes to stop diverging.

When applying an update we have a range of meta-actions bounded by the commit log's *last commit mark* and its *current position* mark that need to be applied to the model via the coordinator. We must also bear in mind that any one of these actions may conflict dynamically with a tentative action in the node log.

Since applying a commit log update involves looking-up the actions referenced by it, Joyce must ensure that those actions are already in the multi-log. On the primary member, this is not an issue since the commit log update was generated from actions already in the primary multi-log. On members that are not primary, Joyce preserves this property by appending updates to peer member logs before it applies updates to the commit log (epidemic propagation ensures that authoritative updates do not arrive before the peer updates they reference.)

Joyce applies commit log updates with the following algorithm:

1. Roll back all node log actions that were appended after the node log's *last commit mark*.
2. For each action A referenced by a commit/abort meta-action
3. If A is committed execute it via the coordinator.
4. Set the *last commit mark* in A's log to be immediately after A.
5. For each action N after the node's last commit mark
6. Attempt to execute N against the new model using the coordinator
7. If N has a dynamic failure
8. Mark this action as invalid against the new model.

Consider the multi-log *James* which has just appended meta-actions referring to "d e f G H I" to its commit log but has yet to apply them (*figure 11*)

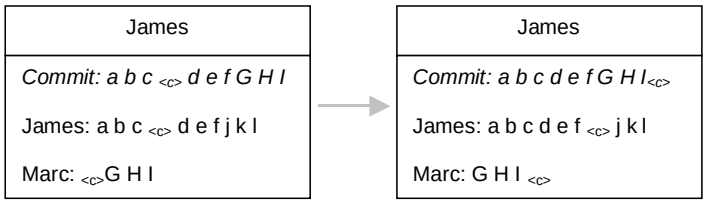

Figure 17 The multi-log for James before and after applying a commit update

The first step in applying the auth update is to return to the last commit mark in the node log. We then enumerate over the actions referenced by the commit log, executing the committed ones. Finally we must re-run the node log actions "j k l", these are tentative actions that have yet to be reconciled and may conflict dynamically with "G H I".

During the application of a commit update we must suspend the tentative cycle. The actions referenced by the update are applied to the model in the same way as the actions from the local controller (via the `execute` method) but we must briefly halt the supply of actions from the controller while this takes place, this causes an interruption to the application circuit as illustrated in figure 17.

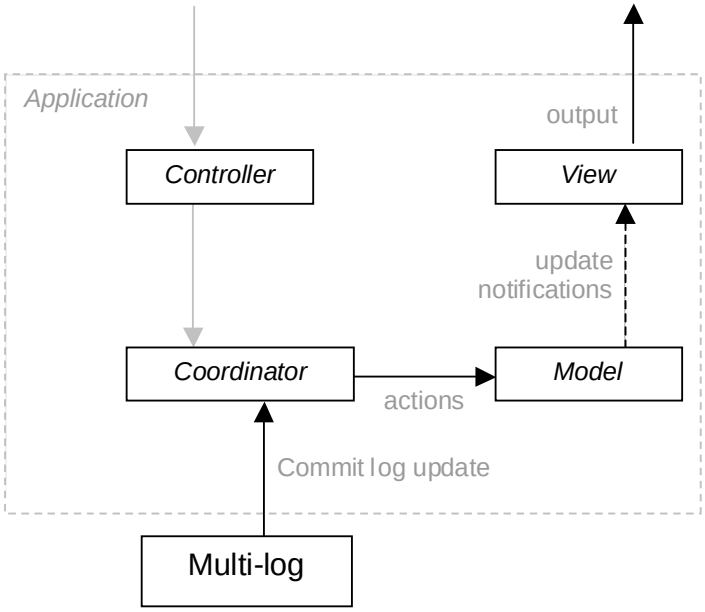

Figure 18 When a commit update arrives the tentative cycle is suspended. The application state and view is updated to reflect the commit log.

6.1 Invalid Tentative Actions

Suspending the supply of actions from the controller whilst applying a commit update does not necessarily imply pausing *interaction* with the application. The

user interface may remain responsive and tentative actions may continue to be constructed whilst a commit update is applied in the background. However, it may be the case that tentative actions become invalid in the time it takes the user to generate them.

Say we have a calendar application in which the user creates an appointment by selecting some cell corresponding to the appointment time and entering the details of the appointment. He selects the cell corresponding to 10:00 and starts creating an appointment. In the background however, an update to the commit log arrives that contains a committed 10:00 appointment. When the user completes his 10:00 appointment it will be submitted as part of a log fragment to the coordinator and (if the application is written correctly) will fail a dynamic precondition constraint.

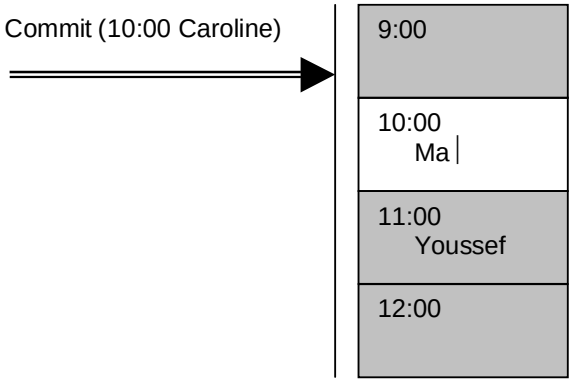

Figure 19 A commit arrives that will invalidate the action being constructed

When this happens the coordinator will detect the precondition failure and mark the failed action as *invalid* by appending an *invalidation meta-action* that references the failed action to the submitted fragment. The controller must also mark as invalid any other actions in the fragment that are *dependent* on the failed action (via an ordering or parceling constraint.) Further, if coordinator has already applied actions that are parceled with the failed action it must *un-apply* them using compensation actions. If any of the un-applied actions do not have compensation actions the coordinator must try to restore the model in some other way, typically this will involve rolling back to the last committed milestone and rolling forward over all the tentative actions less the failed ones.

6.2 Preventing Invalid Tentative Actions

Obviously, invalid tentative actions are undesirable. Applications should try to prevent invalid tentative actions being generated by listening for *committed action* and *aborted action* coordinator notifications which are fired during application of a commit update. If an application is alerted to a condition that will lead to an invalid action it should notify the user in some way and, preferably, have the user change his input so that the action no longer conflicts.

In the example above, the application is aware that the user has started constructing a 10:00 appointment. If the application has registered for the *committed action* notification it will be alerted when the *10:00 (Caroline)* action arrives and should highlight the 10:00 cell in some suitably alarming way. To resolve the potential conflict the application should prompt the user to drag the appointment he is constructing to a free cell (say 12:00.)

Preventing tentative actions can become more complex when the action under construction will become invalid because of a dependency, for example if the action under construction is a strong successor of the action that has become invalid. The application designers may use the logging functionality provided by Joyce to detect the invalidation or they may depend on their own logic. Either way they must balance the complexity of attempting to detect the invalidation beforehand against letting the under-construction action go and fixing the invalidation after the coordinator has detected it.

6.3 Do we need Invalidation Meta-Actions?

Invalidation meta-actions are not strictly necessary, if they were not present the invalid tentative actions would fail at reconcile time (with the same dynamic failure that happened at tentative execution) and would be aborted. Invalidation meta-actions are designed to be a guide to log implementations and log editing tools. For example if we *know* that a tentative action will abort and the user attempts to put a later tentative action in a parcel with it we can inform the user that this later action will never be committed.

At first glance the process of preventing invalid actions may look similar to the process of fixing aborted actions (see above.) The key difference is that only the primary can fix aborted actions and cause a re-reconciliation whereas preventing invalid actions happens on every type of member. If the Joyce group is configured such that all the members are primary then obviously the application is free to choose whether to alter the aborted action or the tentative action on a case-by-case basis.

7 Discussion and Future Work

The increased concentration on ubiquitous or pervasive computing has already resulted in growing numbers of applications that collaborate over some shared data. A centralized, application-specific approach to consistency in such situations is often infeasible due to the high degree of mobility in the participating devices.

In contrast, we have described our application-agnostic reconciliation system. We provide a programming framework and set of services that mobile

applications can use to distribute their modifications within a variably connected group of collaborators.

Many of the design choices used in Joyce have been seen before. For example, application-agnostic reconciliation came from our previous IceCube [Kermarrec 01] project and Bayou [Edwards 97] employed an epidemic propagation system to distribute writes to a replicated database. The contribution of Joyce is to employ these methods in a programming framework that enables applications to be written or modified to participate in data-sharing groups with the minimum of effort.

We define an application model that introduces the concept of non-local, non-authoritative modifications into the well-established MVC partitioning pattern and we supply a flexible set of components behind this model to handle the various aspects of participating an adhoc, mobile, collaborative group.

The greatest challenge when writing applications for Joyce is describing functionality in our actions and constraint model. As we gain experience writing applications future work will involve extending the framework or layering new frameworks on top of Joyce that automate the extraction of constraints from a given data domain.

8 References

[Abowd 00] Abowd GD, Mynatt ED: Charting Past, Present and Future Research in Ubiquitous Computing. ACM *Transactions on Human-Computer Interaction* 7:29-58, 2000

[Balasubramaniam & Pierce 1998] S. Balasubramaniam and C. Pierce: What is a File Synchronizer? In *Int. Conf. On Mobile Comp. And Netw. (MobiComp 98).* ACM/IEEE, Oct. 1998

[Demers 87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker. HH. Sturgis, D. Swinehart, and D. Terry: Epidemic Algorithms for Replicated Database Management. In *Proceedings Sixth Symposium on Principles of Distributed Computing*, Vancouver, B. C., Canada, August 1987, pages. 1-12

[Edwards 97] W.K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer. Designing and Implementing Asynchronous Collaborative Applications with Bayou. P*roceedings User Interface Systems and Technology*, Banff, Canada, Oct 1997, pages. 119-128

[Gamma 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Reading Massacheusetts. Addison-Wesley, 1995

[Kermarrec 01] A.-M Kermarrec, A. Rowstron, M. Shapiro and P. Druschel: The IceCube Approach to the Reconciliatino of Replicas, in *20th Symposium on Principles of Distributed Computing (PODC)*, pp. 210-218, Aug 2001

[Krasner 88] G.E. Krasner, S. T. Pope: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk 80 System. *Journal of Object-Oriented Programming* 1:26—49, 1998

[Preguiça et al. 2003a] Nuno Preguiça and Marc Shapiro and Caroline Matheson: Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Tenth Int. Conf. on Coop. Info. Sys. (CoopIS)*, Nov 2003

[Preguiça et al. 2003b] Nuno Preguiça and Marc Shapiro and J. Legatheaux Martins: Automating semantics-based reconciliation for mobile Transactions. *CFSE'3, conference francaise sur les systemes d'exploitation*, Oct 2003

[Saito & Shapiro 2004] Y. Saito and M. Shapiro: Optimistic Replication, *Tech. Rep. MSR-TR-2003—60*, Microsot Research, Cambridge, UK, Oct 2003

[Schmidt 00] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann: Pattern-Oriented Software Architecture volume 2, Patterns for Concurrent and Networked Objects, Chichester Sussex, Wiley, 2000

[Weiser 91] Weiser M: The Computer of the 21st Century. *Scientific American*, 265, 3, 66-75, 1991