

# An Application Framework for Nomadic, Collaborative Applications

James O'Brien<sup>1</sup>, Marc Shapiro<sup>2</sup>

www.jaimz.org

james@jaimz.org

INRIA Rocquencourt, France and LIP6, Paris, France

marc.shapiro@acm.org

**Abstract.** To maintain availability and responsiveness, mobile applications sharing data often work on their own copy and transmit local changes to other participants. Existing systems for recording, transmitting and reconciling concurrent changes are usually ad-hoc and specific to particular applications. In contrast, we present Joyce; a general application programming framework for creating highly dynamic mobile, collaborative applications. The framework abstracts application semantics using an action-constraint formal model and provides communication and consistency services based on this model. The framework exposes an interface that allows application programmers to concentrate on core functionality without worrying about these issues. Applications made with the framework can run seamlessly across changing combination of devices, users and synchrony. We discuss the principles behind the framework, its implementation and evaluate its utility by creating a complex, shared application.

## Introduction

Today's computing environment is increasingly nomadic; applications run on laptops and devices that are not geographically fixed, and it is increasingly collaborative; applications are often used concurrently by more than one person or device. Such an environment is characterized by a high degree of change in the number of participants, change in connectivity between those participants, and change in the synchrony of collaboration. Programmers need tools to create good collaborative, nomadic applications: applications that adapt to mobility, adopt a collaborative posture and retain the richness and control of desktop applications.

The major difficulty with such applications is maintaining the consistency of shared data. Commonly used application architectures, for example Model-View-Controller [Krasner 88], implicitly assume that data is modified by one user using one device. Many applications fail to benefit from collaboration and mobility due to the prohibitive cost of re-architecting to take account of concurrency control issues.

Certain classes of application, for example *personal information managers*, are designed specifically to be shared between mobile devices. The techniques used however, are specific to the domain of the application and intrusive to the application logic. Moreover, most of these applications use some form of lock-step synchronisation which requires the user's intervention. Finally, the concurrency control wheel

tends to be reinvented with each application, extending development time and resulting in segregated, incompatible systems. This is not an approach that scales well to general application construction and the increasing popularity of pervasive, mobile computing is likely to underscore its shortcomings.

Functionality time-consuming to implement and common between different applications is usually encapsulated in an *application framework*. An application framework is designed to handle the logic common to all applications sharing a particular aspect: for example Apple's Cocoa framework [Cocoa] handles interaction with the windowing system for graphical desktop applications. Frameworks differ from libraries in that applications using them exhibit an *inversion of control* [Schmidt 00]; it is the framework logic, rather than the application logic that controls the execution of the application process.

In this paper we describe an application framework called *Joyce* that introduces a new programming pattern for highly dynamic, collaborative applications and provides an implementation of that pattern. Joyce enables applications to run across changing combinations of devices, changing combinations of users, and changing combinations of synchrony. We describe what we believe are the current and future requirements of collaborative, nomadic applications and why current techniques do not meet these requirements, we then go on to explain the principles behind our system and describe a realistic application, "Babble", created to evaluate the system.

## Requirements

Applications created with our framework must meet the following expectations:

- *We expect to be mobile and only occasionally connected*: the applications will be used concurrently by a mixture of users on a mixture of devices. Devices may transition between on-line and off-line at any time so we cannot assume constant connectivity or a complete knowledge of the collaborative group membership. We also cannot assume any particular physical device configuration (e.g. local storage).
- *We expect nomadic, collaborative applications to be as rich as current single-user, single-device applications*: the applications must be at least as responsive and featureful as current desktop applications and will preferably exhibit improvements in usability.
- *We expect to be fully aware of group activity but we do not expect to be bound to a distracting WYSIWIS environment*: these environments (**W**hat **Y**ou **S**ee **I**s **W**hat **I** See) attempt to keep the application display of each participant precisely in sync. Where such a scheme is necessary (conferencing applications for example) we expect the framework to allow us to build it. However, in applications where real-time collaboration is not the objective, WYSIWYS produces a display that constantly distracts the user from his local task. This leads to a feeling of loss of control which in turn leads to application usability lower than the single-user equivalent; as we have already stated, this is unacceptable. We expect to be continuously *aware* of group activity but also in *control* of how and when the activity is applied.
- *We expect to be aware of the group history of the application state and we expect a manipulatable history that works well in collaborative environments*: projects such

as FlatLand [Edwards et al. 00] and GINA [Berlage et al. 93] have demonstrated the benefits of manipulatable history but current implementations of undo/redo in a collaborative environment are complex and application specific. [Sun 02].

To meet these expectations and remain generic the framework needs to be adaptable across two major criteria. Firstly, the framework must be able to cope with different degrees of *coupling* between the participants [Berlage et al. 93]. Coupling is the degree of co-ordination between participants. For example, when syncing mobile devices all the devices involved are connected and they all receive each other's updates at the same time. In contrast, collaborative systems can fall anywhere between same place/same time systems where collaborators work "shoulder-to-shoulder", to different place/different time systems where collaborators may be dispersed across time zones. We should be able to use the framework to build applications anywhere within this spectrum.

Secondly, any concurrency control system is closely linked to the semantics of the object being shared [Munson et al. 96]. In traditional database systems this semantic is one of read/write operations to some storage. This was found to be too restrictive and techniques were developed to expose a richer set of semantics based on the programmatic interface of the shared data structures [Munson et al. 96][Schwarz et al. 84]. This allows more concurrent activity by more narrowly defining what constitutes a conflict. From a user's perspective however, a modification has more semantics than can be expressed solely in data structure interfaces; our framework must be able to express higher-level application semantics and user intentions.

From these general requirements we developed a more concrete list of problems to be moved from the domain of the application to the domain of our framework:

1. **Modeling activity:** Joyce should provide an application-agnostic way of representing concurrency semantics that is rich enough to articulate object, application and user-level semantics.
2. **Communicating activity:** The framework should ensure that, even with partial connectivity, modifications from one participant will propagate to all the others.
3. **Consistency:** Joyce must provide an application-agnostic mechanism for bringing diverging, replicated states to consistency, concurrent with the user modifying that state.

In satisfying these problems it is vital that Joyce not degrade the performance and responsiveness of the application.

## Previous Work

An early approach to concurrency control was simply to acquire a lock on a piece of data before modifying it, the data being stored at some central location. If the lock could not be acquired then the application either blocked until the lock was available or failed. Many early research systems were based around a locking mechanism called floor-control [Sarin et al. 85] in which one participant modified the shared object while the others observed, waiting their turn. This approach has the advantage of simplicity and is still used in web-based collaborative systems such as Wiki [ Wiki ] and

JotSpot [JotSpot]. However, locking has proven problematic for mobile applications since it requires a constant connection to the central data store, and even if a connection is present an application may spend a great deal of time blocked until a lock becomes available.

The DistView [Prakash et al. 94] framework used replicated lock tables to prevent blocking becoming too great a hindrance and the GroupKit [Roseman et al. 96] system allowed operations on shared data whilst a lock was pending; if the lock request was refused the operations were undone. The concept of *tickle locks* [Greif et al. 86] was developed to minimise the amount of time waiting on a lock - essentially the requester would 'tickle' the participant holding the lock and, if there was no response, the lock would be transferred.

Even with these improvements, locking proved restrictive and lead to awkward interaction as applications either blocked or backed-out failed changes. Instead, mobile applications often adopt an *optimistic replication* scheme [Saito et al. 05] in which each participant takes a local replica of the shared state and modifies that replica without regard to concurrent changes from other applications. At some later point all the replicas are synchronised to produce a common state. The technique is termed optimistic since the applications 'optimistically' assume that their local changes will not conflict with concurrent changes at other replicas. This is the approach used in our framework since local states require no locking and the applications can remain responsive.

The dOPT algorithm of Ellis and Gibbs [Ellis et al. 89] introduced *operational transform* (OT) in which remote operations are 're-written' so that their effect locally is the same as their effect where they were issued, regardless of any local operations that have happened in the mean time. OT has proven particularly popular in real-time collaborative text editing systems such as ShrEdit [McGuffin et al 92], Grove [Ellis et al 88] and SubEtherEdit [ SubEtherEdit ].

The use of OT leads to very responsive applications but the technique is more a mechanism to maintain consistency despite out-of-order messaging than a synchronisation mechanism. Moreover, although the technique itself is generic, OT implementations are usually application specific and very complex. The semantics of an operation is obfuscated by the transform and often lost entirely if an incoming operation has to be transformed against many prior operations. If a history mechanism (such as undo/redo) is required this leads to further application-specific complexity [Sun 02]. There are also known scenarios where current OT techniques may lead to an inconsistent state [Li 04]. Finally, OT is intended primarily for real-time, synchronous editing systems rather than multi-synchronous, occasionally-connected systems.

Bayou [Edwards 97] introduced several mechanisms that support multi-synchronous distributed applications. Bayou is a log-based optimistic replication system that models operations using a read/write semantic augmented with application-defined conflict detection and resolution mechanisms. Operations are communicated using an epidemic propagation scheme that guarantees updates from one participant will reach all the others given sufficient connectivity [Demers 87]. Bayou has good solutions for maintaining communication in the face of occasional connectivity but forces applications to adhere to the limited read/write semantic.

Although concurrency control has been studied extensively and many techniques have been developed we find none of the principles and algorithms suitable to be in-

tegrated into the general application development cycle. Either the techniques are too application specific (as with OT), do not work in a multi-synchronous environment (as with floor-control) or do not wholly express application and user semantics (as with Bayou).

## The Multi-log

Joyce is a programming framework built around an operation-based replication and collaboration system designed specifically for applications operating in the kind of dynamic environment described in section 1. Joyce connects participants working on replicated copies of shared data 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.

The core data-structure used by Joyce is a distributively maintained, shared, semantic data-store: the *multi-log*. The multi-log is designed to provide a fine-grained model of activity within a collaborative group, based on a reified model of application semantics. It is a graph structure in which vertices represent data modifications made by applications and edges represent the semantics of those modifications in terms of invariant relations that must hold between them. The framework is responsible for synchronising both the multi-log and the replicated states.

## Basic Definitions

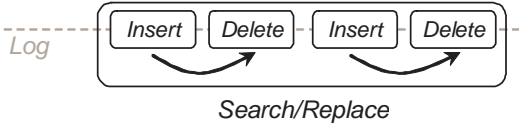
We define a *data object* as the distinguishable unit of data that is being shared, this may be anything from a calendar to a document to a database. Each data object has an associated *group* which is the set of all nodes working on replicated copies of that data object; a node being some application process that is modifying the data. It is possible that the members of the group may change from one moment to the next as may the connectivity between members. We cannot require that any member have a complete knowledge of all the others but we do provide a mechanism that any one node can use to discover a peer group – the subset of the group that can be contacted. The framework ascertains the peer group either by broadcasting an announcement and listening for replies or by joining an application-level multicast tree [Castro et al. 02] corresponding to the shared object.

## Modeling Application Activity

Joyce defines an action/constraint formalism that allows applications to define a fine grained model of their concurrency semantics, at both the object and user level.

Following the command pattern [Gamma et. al. 95], Joyce applications are architected primarily as a set of commands that modify a particular kind of data object. Command invocations are recorded in an application log as a series of *actions*. Joyce also records a set of *constraints* that describe the semantics of the modification that

the command invocation was part of. These constraints are guaranteed to be preserved by the framework.

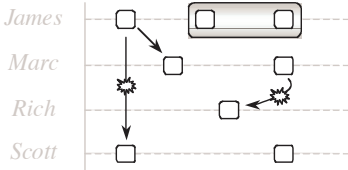


**Fig. 1.** Joyce logs modifications with their semantics. A text editor may model search and replace as insert and delete actions that are ordered and atomic.

A requirement outlined in 2.1 is that the framework be able to represent both object and application-level semantics; this is achieved by defining *object* and *log constraints*. Object constraints represent semantic invariants between *classes* of commands, and by extension the data object that those commands are designed to modify, whereas log constraints express invariants between actions that share a log. Log constraints are used to express user intent and application semantics and stand in contrast with previous systems where only the chronological order of operations is recorded [Petersen et al. 1997]. The set of object and log constraints have been derived from those constraints that have proven expressive in our previous work on reconciliation. Readers interested in the motivation behind these constraints are advised to consult [Kermarrec 01] and [Preguiça et al. 03].

**Modeling Group Activity**

The multi-log is a semantic graph formed by processing individual application logs. Vertices in the multi-log are actions and edges represent the constraints between them. Edges are placed between actions from differing source logs to indicate that a modification from one peer is dependant on or mutually exclusive with a modification from another. In this way we create a picture of the activity within a group that is independent of the chronology of the actions. Instead of trying to use timestamps to derive dependency information we use the invariants expressed in the multi-log semantic graph.



**Fig. 2.** This multi-log describes a semantic graph containing an ordering constraint, two conflicts and a parcel.

It is a vital task of the framework to keep the multi-log on each node as representative of group activity as possible. To achieve this, the multi-log is distributively main-

tained using an epidemic propagation scheme [Demers 87]. Epidemic propagation is well known to exhibit good behaviour in the face of varying connectivity since a node's updates may still propagate through intermediaries even if that node is no longer connected [Demers 87].

### **State Consistency**

Problem 3 in section 2.1 requires Joyce to have a method of bringing divergent states to consistency. To achieve this, we provide a reconciliation engine, based on our previous IceCube engine, that can calculate a consistent subset of actions from the multi-log. A consistent subset of actions is one in which no actions conflict and all the constraints in the subset are satisfied. IceCube treats this as an optimisation problem: each action has an associated weight indicating how important the action is; the IceCube algorithm heuristically determines the subset of actions from the multi-log such that the total value of the actions not in the set is minimised.

The consistent subset produced by the reconciliation engine forms a *schedule* a sequenced ordering of actions that may be selected for *commitment*. Commitment is the act of irrevocably selecting a reconciliation schedule for execution at every member in order to make their replicated states consistent. The schedules that have been committed are recorded in a special multi-log entry called the *commit log* which consists of commit and abort meta-actions that reference actions in the multi-log.

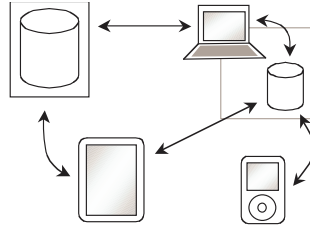
A node that generates commit-log updates is called a *primary* and there is usually only one per Joyce group. By default, Joyce assigns the creator of a data object to be the primary for that object, but other mechanisms, for example consensus mechanisms [Lamport 98] may be used. Epidemic propagation ensures commit log updates arrive at all nodes in the right order and eventual consistency is reached.

### **Multi-log Persistence**

The traditional file system storage model is cumbersome when applied to nomadic, collaborative applications. Nomadic devices may not have local storage and continuous connection to a file server is not feasible. Joyce provides an automatic persistence service wherein one or more *storage nodes* join a collaborative group and persist the multi-log to backing store.

Joyce applications take snapshots of their data at specific times. A snapshot is most often taken when a state has reached some milestone in the editing process or when the framework detects that the state has been brought to consistency. Taking a snapshot of the consistent state allows the framework to truncate the multi-log by removing the committed and aborted actions - all future actions can be issued against the consistent snapshot.

If a group member disconnects or crashes the act of re-joining the group, and re-contacting the storage node, restores the application state with a minimum of data loss.

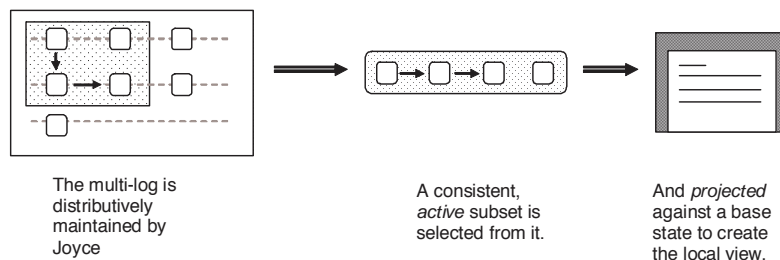


**Fig. 3.** A Joyce group containing two storage nodes: a 'server' node and a laptop running its own node.

### Application Model

Joyce provides a skeleton architecture designed to foster applications that meet the expectations outlined in section 2. The key principle of the architecture is that the user interacts with a *local view* of the global activity which is as responsive as a corresponding single-user application would be. The user should feel in full control of this local view and not overwhelmed by group activity.

The multi-log is the history of the global activity within a collaborative group. The local view is a *projection* of a subset of this global history. The framework maintains a consistent subset of actions from the multi-log, the *active subset*, that is run against some base application state to generate the local view.



**Fig. 4.** The local application is a projection of the global history.

The active subset contains two kinds of action: actions that have been committed by the primary and a consistent subset of *tentative actions* - actions generated locally or remotely that have not yet been committed or aborted. It is by manipulating which actions are included in this tentative set that the user and application controls what appears in the local view.

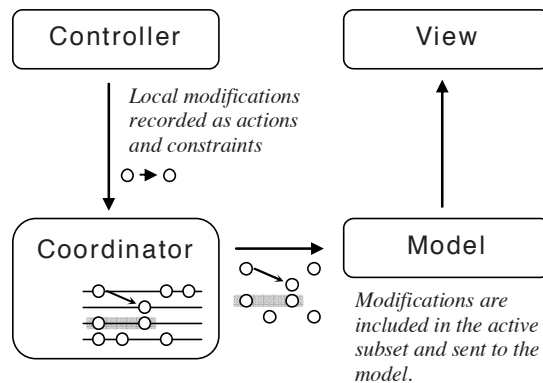
The framework is designed to keep the local view responsive by adding locally generated actions to the active subset immediately, implementing undo/redo as local operations and filtering incoming updates to determine which should appear in the active subset.

## The Tentative Interaction Cycle

To reflect local modifications quickly, the architecture populates the active subset using an interaction cycle derived from the Model-View-Controller pattern [Krasner 88]. An interaction cycle is the programmatic path between a user triggering a local modification and the result of that modification being reflected in the application output. MVC introduced a cycle in which input from the user is evaluated by a controller into a set of modification messages for the model; the model applies the modifications and sends a set of update messages to the view which reflects the modification back to the user.

This pattern simplifies the construction of GUI applications but assumes that modifications always come from a local (i.e. in-process) controller; and inversely that modifications from the controller are always for the local model. The pattern also has the more subtle assumption that the local controller is the authoritative source of the modifications - it has no notion of a global state that might be defined elsewhere.

We expand MVC by introducing a *coordinator* component, whose job is to maintain the active subset and apply it to the model. During our interaction cycle (figure 5) user input is evaluated into a set of actions and constraints; these are sent to the coordinator, which logs them in the multi-log and immediately includes them in the active subset - causing them to be applied to the model and reflected in the view. We call this the *tentative interaction cycle* since the actions applied to the state are local, tentative actions.



**Fig. 5.** The tentative interaction cycle in Joyce. The controller generates modifications and sends them to the coordinator for execution and logging.

When an update to the multi-log arrives, the coordinator interrupts this cycle to recalculate the active set. It uses the reconciler to create a consistent schedule from the updated pool of tentative actions in the multi-log, which becomes the new active subset. Note that this reconciliation has no effect on the globally consistent state as defined by the commit log - it is local to the receiving node.

If the multi-log update includes a commit log update, the aborted actions and their dependants are removed from the tentative action pool and the active subset is pre-populated with the committed actions before the local reconciliation occurs.

The actions in the active subset are recorded relative to a base state, usually a snapshot of a previous stable state. To apply a new active subset, Joyce restores the base state, then runs the new active subset against it. The schedule produced by the reconciler is guaranteed to respect ordering constraints and so can be executed sequentially.

### **Filtering, Undo and Redo**

A user can define which applications are included in his active subset by defining *filters* over the set of tentative actions in the multi-log. A filter is simply a predicate that pre-excludes matching tentative actions from a reconciled schedule. This prevents the coordinator including the filtered action and its dependants in the active subset.

The simplest example of filtering is masking out specific collaborators. Here, the filter matches every action from a particular source. Actions from the source will not be accepted into the active subset and thus will not contribute to the local state. It is important to note that filtering does not remove actions from the multi-log, just from the tentative action set. All information about group activity is retained, an important expectation (section 2). Later, the filter may be removed, allowing the previously masked work to be reintegrated into the view.

Undo is implemented as a filter that masks out a specific action. To undo a modification the user selects the action and creates the filter; when the active subset is recalculated it will be equal to the previous active subset less the undone action and its dependants (those actions that are parceled with or strong ordered after it). If subsequent remote actions arrive that are dependant on the undone action the process ensures those actions will not appear in the active subset.

Since constraint information is used to calculate the dependants, undo in Joyce is selective. The undo operation is confined only to those operations directly effected [O'Brien 04] and the corresponding redo can be done at any time if no intermediate arrivals conflict with the undone action. This contrasts with the stack-like, linear model used in most applications.

### **An Example Application: Babble**

To refine Joyce for real-world development we created a free-flow collaborative editor called *Babble*. A text editor is complex enough to exercise the whole framework but familiar enough that the contributions of Joyce are well highlighted; particularly fluid collaboration, selective undo/redo and passive storage.

When 'opening' a file in Babble, Joyce discovers and joins the collaborative group for the document, restores the most recent snapshot it can find and brings the local multi-log up to date. Babble is then notified of the reconstructed state and the local interaction cycle can begin.



Fig. 6. Babble will synchronise to the current group state on start-up.

Edits from collaborators can be “tagged”: on mouse-over we can display context information about the edit taken directly from the multi-log.

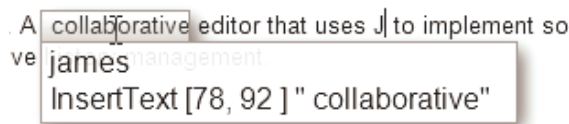


Fig. 7. A tagged edit

If concurrent edits conflict, Joyce will choose an edit for Babble to apply and instruct Babble to highlight the effected content with a red shading. In keeping with the local view principle, Joyce will chose local edits over remote ones unless specifically instructed otherwise.

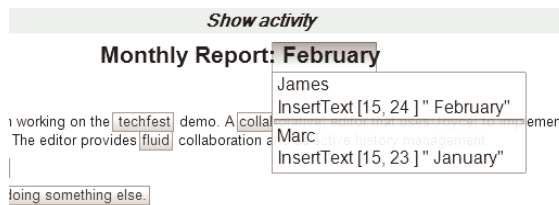
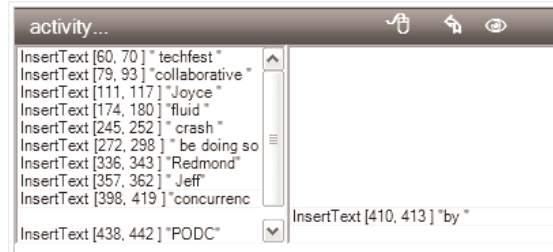


Fig. 8. Viewing a conflict

Active subset manipulation (that is, selective undo, redo and filtering) is triggered through a *history editor*. Actions are arranged in the editor according to character position and dependencies. In the figure below, the action in the second column has a dependency on the action in the first. Selecting the first column action also highlights the second-column action and the modifications that both actions made are highlighted in the content display. In this way the user can visualize the extent of a prospective undo.



**Fig. 9.** The history editor

Since this representation can be confusing in a large document the history editor may be set to display only the *local history* – the history at the current carat position. The history editor may also display higher-level operations such as a search-and-replace, which is implemented as a parcel of inserts and deletes. Selection of the constraint representation also selects its constituent actions. When an undo is triggered a filter is placed on the selected actions, the active set is recalculated, and the results displayed. The effect on the content is that the highlighted content modifications have been undone but the modifications of non-dependant actions remain in place.

### Representing Text Editing in Joyce

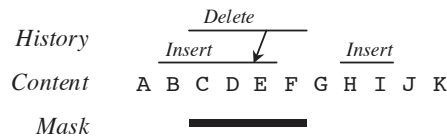
Applications built with Joyce are architected as a collection of *actions* that implement the application commands and *constraints* that represent the concurrency semantics of those commands. To implement Babble we needed a set of actions and constraints that encapsulated text editing.

Text editors are usually built around a linear character buffer addressed using character coordinates from 0 (before the first character) to N (after the last character). Two operations modify this buffer: *insert(p, c)*, that inserts character *c* at position *p*, and *delete(p, n)* that removes a range of *n* characters starting at position *p*. Most shared text editors are built around the same structure but use *operational transforms* to rewrite remote inserts and deletes such that their local effect is the same as their effect at their source. Essentially, the edit points of inserts and the edit points and spans of deletes are shifted in order to compensate for operations that have been applied to the local state. This gives good performance in distributed, real-time editing but is complex to implement, especially if multi-synchrony and undo-redo are required, intrinsic qualities of Joyce applications.

Babble borrows the idea of translating edit points from OT but uses a more systematic approach that meets the requirements of the Joyce framework. Our representation of a text buffer is more complex than a simple character array but captures the dependencies between edits and allows us to show, hide, re-combine and re-order editing operations as directed by Joyce.

The representation is in three parts:

1. **The content:** a linear text buffer similar to that used by non-concurrent and OT editors. However, with the exception of snap-shots and undo/redo, characters are only ever inserted into the buffer, not removed.
2. **The mask:** a collection of character position intervals that indicates deleted text. Masked text is not displayed and therefore cannot be edited.
3. **The history:** a hierarchical collection of character position intervals that record the operations that have been applied to the content.

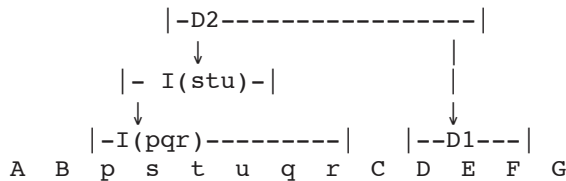


**Fig. 10.** Babble represents a text buffer in three layers. This buffer is displayed as ABGHIJK

The actions defined by Babble are:

1. **Insert (p, s):** insert the string s into the content at position p.
2. **Delete (p, a):** insert a mask of length a into the mask structure at position p.

To define constraints, we say that one Insert must follow another if the edit point of the second intersects the span of the first. A Delete must follow another Delete *or* Insert if the spans of the two actions intersect. This is recorded in the history and communicated to Joyce using *ordering* log constraints. In the buffer depicted below there have been two inserts and two deletes and the appropriate constraints have been set.



**Fig. 11.** Ordering constraints are set in the history and multi-log according to the intersections of operation spans.

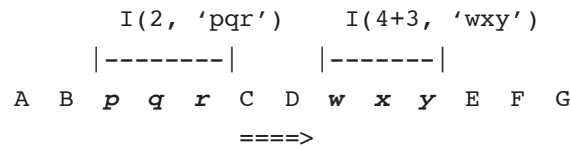
Note that ordering constraints are transitive in Joyce so there is no need to set a constraint from D2 to I(pqr).

### Replaying Out-of-order Changes

Babble is required to be able to replay local and remote operations in any order, since they may be recombined in any consistent order by the Joyce reconciler when the active subset is calculated.

When replaying an action, Babble uses the history to detect whether the action being replayed needs to be transformed. The replay mechanism uses the history struc-

ture to keep track of the mutations to the content: i.e. where content has been inserted. When an action is replayed out of order, its edit point is shifted according to the mutations that have happened to the content since that action was first issued. For example, if an action at site *A* inserts text of length 3 at position 2 then receives an action *b* from a remote site that inserts text at position 4, then the edit point of *b* is shifted to take account of the local action. The mask data structure allows us to apply the same mechanism to deletes since no content is actually removed.



**Fig. 12.** Remote action  $I(4, 'wxy')$  is shifted due to a prior mutation in the content.

When comparing local and remote edits, Babble will raise a conflict if the edit points of two insertions are identical or the span of a delete intersects with another delete or the edit point of another insert. The conflict is expressed to Joyce with a *mutual exclusivity* constraint and highlighted in the user interface as above.

## Summary and Future Work

Joyce is a programming framework that provides three main contributions: a clearly defined idea of what collaborative, nomadic applications should be, a systematic model for creating such applications and an implementation of the principles and mechanisms described in the model.

Babble demonstrates that the creation of a complex, shared application is possible with the framework. One developer was able to take the application from design to functionality in little over two months since the framework abstracted away both maintenance of occasionally-connected groups and concurrency control mechanics. The result is a full-featured, shared text editor with demonstrable advantages over similar applications: improvements in the undo/redo and storage user experience compared to contemporary single-user editors, and greater control over the local state than contemporary collaborative editors.

The creation of Babble was greatly simplified by Joyce but was still not as simple as we would have liked. Re-casting an application into Joyce's action/constraint model is difficult and requires an approach unfamiliar to most application developers. How to extensively unit test such applications remains unclear. Future work should investigate whether constraints can be automatically derived from a data type.

With regard to the programming model, strict adherence to the MVC cycle is preferable but can lead to unacceptable performance. Pure MVC implies an asynchronous model in which programs depend only on events to be notified of model changes. In reality, most MVC applications shortcut from the controller to the view to provide more immediate feedback.

In Babble there is a similar, probably typical, compromise in that local actions are constructed synchronously in the history structure and appended to the multi-log on completion. If a multi-log update arrives, special code exists to detect whether the action being constructed is *going* to conflict. If MVC is any guide, this will be a typical compromise in Joyce applications; we should anticipate it and provide a lower-level API to the reconciler so that applications can detect possible conflicts themselves.

The toolkit and application described in this paper was implemented at Microsoft Research Cambridge using .NET. Our immediate focus is producing and releasing a streamlined Java version of the toolkit along with a more advanced, styled-text version of Babble and a presentation tool.

We expect further developments of the kind of application described in this paper to raise interesting and difficult questions in the areas of user-interface, application construction and security. Using Joyce, we can cope with dynamic reconfigurations of devices, users and synchrony but we can't reconfigure an *application instance* to adapt to the device it is running on or the scenario it is being used in. An interesting approach may be to completely de-couple actions from applications. Joyce applications lessen the requirement on the user to switch mental 'modes' since his focus is always on the artefact being created. Decreasing modality increases usability. Future implementations may go further and disintegrate actions from applications completely to further lessen modality across the whole system. Actions may be associated with particular data types and always triggered in the same way. If we create a set of actions and constraints for editing XML we may be able to declaratively *generate* applications by using an XML file to weave together actions that have registered against XML schema types in a central system pool.

## References

- [Berlage et. al. 93] T. Berlage and A. Genau. "A Framework for Shared Applications with a Replicated Architecture". *Proc. ACM Symposium on User Interface Software and Technology*. 1993
- [Cocoa ] <http://developer.apple.com/cocoa>
- [Cooper 03] A. Cooper, R. Reimann, R.M. Reimann, H. Dubberly. "About Face 2.0: The Essentials of Interaction Design". *John Wiley & Sons, Inc.* 2003
- [Castro et al. 02] M. Castro, P. Druschel, A. M. Kermarrec, A. Rowstron. "SCRIBE: A large-scale and decentralized application-level multicast infrastructure". *IEEE Journal on Selected Areas in communications (JSAC)*, 2002
- [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". *Proc. Sixth Symposium on Principles of Distributed Computing*, Vancouver, B. C., Canada, 1987
- [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". *Proc. User Interface Systems and Technology*, Banff, Canada, 1997
- [Edwards et. al. 00] W.K. Edwards, T. Igarashi, A. LaMarca, E.D. Mynatt. "A Temporal Model for Multi-Level Undo and Redo". *Proc. User Interface Systems and Technology*, San Diego, CA, 2000
- [Ellis et al 88] Ellis, C., Gibbs, S.J. and Rein, G., "Design and Use of a Group Editor", *MCC Technical Report Number STP-263-88*, Sept. 1988

- [Ellis et al 89] C.A. Ellis and S.J. Gibbs. "Concurrency Control in Groupware Systems". *Proc. SIGCHI Conference on Human Factors in Computing Systems*, Portland, OR, 93
- [Gamma et. al. 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns, Elements of Reusable Object-Oriented Software". *Addison-Wesley*, 1995
- [Greif et al. 86] I. Greif, R. Seliger, and W. Wehl (1986). "Atomic Data Abstractions in a Distributed Collaborative Editing System". *Proc. of the Thirteenth Annual Symposium on Principles of Programming Languages* St. Petersburg, Florida. 1986
- [ JotSpot ] <http://www.jotspot.com>
- [Kerमारrec 01] A.M. Kerमारrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proc. of Twentieth ACM Symposium on Principles of Distributed Computing PODC*, Newport, RI USA, August 2001
- [Krasner 88] G.E. Krasner and S.T. Pope. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system." *Journal of Object Oriented Programming*, 1988
- [Li et al. 04] D. Li and R. Li. "Ensuring Content and Intention Consistency in Real-Time Group Editors", *24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, 2004
- [McGuffin et al 92] L. McGuffin, and G. Olson, "ShrEdit: A Shared Electronic Workspace", *CSMIL Technical Report, Cognitive Science and Machine Intelligence Laboratory*, University of Michigan, 1992
- [Munson et al. 96] J. Munson and P. Dewan. "A Concurrency Control Framework for Collaborative Systems". *Proc. ACM Conference on Computer Supported Cooperative Work*. 1996
- [Lamport 98] L. Lamport. "The Part-time Parliament". *ACM Transactions on Computer Systems*. May 1998
- [O'Brien 04] J. O'Brien and M. Shapiro, "Undo for Anyone, Anywhere, Anytime". *Proc. SIGOPS European Workshop*, 2004
- [Peterson et. Al 97] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, A.J. Demers. "Flexible Update Propagation for Weakly Consistent Replication". *Proc. Sixteenth ACM Symposium on Operating System Principles (SOSP)*, Saint-Malo, Franco, 1997
- [Prakash et al. 94] Prakash, A. and Shim, H. S. 1994. "DistView: support for building efficient collaborative applications using replicated objects". *Proc. ACM Conference on Computer Supported Cooperative Work (CSCW '94)*, Chapel Hill, North Carolina, 1994).
- [Preguiça et al. 03] N. Preguiça, M. Shapiro, C. Matheson: Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Tenth Int. Conf. on Coop. Info. Sys. (CoopIS)*, 2003
- [Roseman et al. 96] M. Roseman, S. Greenberg. "Building real-time groupware with GroupKit, a groupware toolkit". *ACM Trans. Comput.-Hum. Interact.* Mar. 1996
- [Saito et al. 05] Y. Saito and M. Shapiro. "Optimistic replication". *ACM Comput. Surv.* 37, 1, 2005
- [Sarin et. al. 85] S. Sarin and I. Greif. "Computer based real-time conferencing systems". *Computer* 18, 10 October 1985
- [Schmidt et. al. 00] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. "Pattern-Oriented Software Architecture Volume 2, Patterns for Concurrent and Networked Objects", *Wiley*, 2000
- [Schwarz et. al. 84] P.M. Schwarz and A.Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems* 2, 3, 1984
- [SubEtherEdit] <http://www.codingmonkeys.de/subethaedit/>
- [Sun 02] C. Sun, "Undo as concurrent inverse in group editors", *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2002
- [ Wiki ] <http://c2.com/cgi/wiki?WikiWikiWeb>