RC24033 (W0608-072) August 23, 2006
Computer Science

IBM Research Report

Before-Commit Client State Management Services for AJAX Applications

Paul Castro, Frederique Giraud, Ravi Konuru, John Ponzo, Jerome White*
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*Currently at California Institute of Technology, Pasadena, CA

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Before-Commit Client State Management Services for AJAX Applications

Paul Castro, Frederique Giraud, Ravi Konuru, John Ponzo, Jerome White

***Abstract*—Heavily script-based browser applications change the manner in which users interact with web browsers. Instead of downloading a succession of HTML pages, users download a single application and use that application for a long period of time. The application is not a set of HTML pages, but rather a single page that can possible modify its own presentation based on data exchanged with a server. In such an environment, it is necessary to provide some means for the client to manage its own state. We describe the initial results of our work in providing client-side state management services for these script-based applications. We focus on browser-based services that can help the user *before* any data is committed on the server. Our services include state checkpointing, property binding, operation logging, operational replay, ATOM/RSS data updates, and application-controlled persistence.**

***Index Terms*—Web Applications, Scripting, Programming Environments, Data Handling, Data Management**

I. Introduction

THE term AJAX (Asynchronous Javascript and XML) appeared in 2005 [12], and succinctly describes a collection of existing technologies like Javascript [11] and XMLHttpRequest [28] that have matured and are transforming the role of the browser from a passive *view renderer* to a more active participant in web applications. In a traditional web application, the browser displays the user interface and mainly passes user input to a back-end server that is executing the core application logic. This insures that web applications can run on base configurations of most browsers but can result in an inferior user experience when compared to an average desktop application. The problem arises because 1) base configurations of browsers do not have access to a rich set of user interface elements and 2) the browser is locked to application code and state residing on a remote server. Most user input causes a mandatory page load, making the application feel less responsive and more disruptive for the user.

Original manuscript received June 22, 2006.

P.C. Author is at IBM Research (email: castrop@us.ibm.com)
F.G. Author is at IBM Research (email: giraud@us.ibm.com)
R.K. Author is at IBM Research (email: rkonuru@us.ibm.com)
J.P. Author is at IBM Research (email: jponzo@us.ibm.com)
J.W. Author completed work while visiting IBM Research. Author is currently at the Computer Science Department, California Institute of Technology, Pasadena, CA (e-mail: jerome@cs.caltech.edu).

AJAX applications leverage the browsers' ability to execute Javascript and communicate with the server using asynchronous messaging. Application code and application state can be moved from the server to the browser, resulting in applications that have richer interfaces and improved responsiveness. More importantly, from the perspective of this paper, the browser has much more autonomy to manage application state separate from the server-side components. For example, browser-based office applications may only require a server to store data documents. The browser handles all other document management operations. Highly interactive web applications such as Google Maps [13] and Writely [26] are representative of AJAX applications.

Because of the large improvement in user experience, it is likely that AJAX applications will be an important, if not dominant, type of web application in the future. This has implications in browser design, application programming models, and performance enhancing strategies for the web infrastructure.

In this paper we focus on the increased autonomy an application has to manage its own state locally. We are particularly interested in applications that may operate in a browser for extended periods of time with only periodic communications to the server. We refer to the *glide time* of an AJAX application, which is the length of time the browser can operate on application state autonomously before it contacts the server to commit data changes or receive updates. We are interested in browser-based data-centric services that can 1) help the application with client-side state management during glide time and 2) assist the application with reconciling application state with the server when the glide time has ended.

Data persistence and synchronization issues arise in many contexts, such as distributed file systems [16], P2P data dissemination [23], and mobile computing [4]. In this paper, we focus on the limitations of current browsers and propose enhanced browser capabilities to better support the new usage models enabled by AJAX design patterns. Our client-side approach is only a starting point for investigating this rich and complex research area in the context of AJAX applications.

We look at several browser-related issues:

- *Persistence abstractions and mechanisms?* Client-side state is read-write application data. This application state may be a replica of data retrieved

from the server, or may be generated locally. Some applications may not need to preserve this state beyond the current application session while others may require some type of persistence between application sessions and/or browser sessions. Persistence is a thorny issue as base browser configurations often limit access to the local file system for obvious security reasons.

- *Data replica divergence and synchronization?* AJAX applications enable more complex user interactions to happen completely on the browser. Clients may operate more autonomously with increased periods of time between server requests. During these times, application state may change significantly from the last time the client and the server interacted. This is a critical issue if the client has replicated application state locally and needs to reconcile this state with the server.
- *Browser abstractions for the programmer and user?* Browser abstractions focus on navigating a passive set of web pages retrieved from different servers. UI components such as copy/paste features, and the back and forward button have little meaning to an AJAX application. Client-side state is typically hidden from the user and surfaced only through the application view. As the application modifies its state, are there useful abstractions for how an application might copy and paste application state to different applications? Or how an application might enable undo/redo functions?

In this paper, we present an initial design and prototype implementation of Ripple-X. Ripple-X provides a set of components in the form of Javascript libraries and browser extensions that provide client-side state management services. Our current focus is on services that are useful to the client *before* it needs to commit application state to a server though some of our components, like the Ripple-X connectivity manager, do utilize server-side resources. Our intention is to eventually expand our design such that our services can be useful across a spectrum of application requirements.

The remainder of this paper is as follows. In Section II, we provide an overview of the issues for client-side state management and how our services address these issues using the Model-View-Controller architecture. In Section III, we describe our approach in the context of an "investment club" AJAX application. Section IV provides details about the Ripple-X libraries. In Section V we discuss some of the outstanding issues. Section VI and VII contain related work and conclusions, respectively.

II. Separation of Application from Data in AJAX Applications

In this section, we provide an overview of our basic approach based on separating data from presentation components in AJAX applications.

A. Browser Document Object Mode

Major browsers such as Internet Explorer and Mozilla Firefox provide a Document Object Model (DOM) [8] interface to the currently rendered web page. The DOM is a container for all page objects such as presentation elements and script. The structure of the DOM freely mixes data and the presentation elements that use them. For example, a text field element contains a "value" attribute which should be displayed in the text field.

In a traditional web application, the server side naturally maintains a separation between application and data. Application code usually exists in some type of application framework separate from data, which is normally managed by transactional components like databases or content management systems.

AJAX applications move some of the management responsibility for data to the client. Current browser abstractions provide little or no facility to manage this state without custom Javascript code. What was naturally separated on the server-side becomes intertwined as a single web page on the client-side. The browser DOM must function as a "catch-all" container for application script, presentation elements, and non-visual application data. Fundamentally, the AJAX application force fits its requirements as an *application model* onto a *document model* that is not optimized for it. For example, since it is optimized for visual elements, the browser DOM is not an efficient storage mechanism for non-visual data. Base DOM elements contain a slew style attributes that are only useful for presentation elements, increasing the footprint unnecessarily when storing non-visual data. An AJAX application may benefit from not instantiating certain application data as DOM elements at all. Current browser implementations generally do not allow this.

In Ripple-X, we propose separating application state from the traditional browser DOM. Our proposal is described in the next section.

B. Model View Controller for Browsers

The Model-View-Controller (MVC) design pattern separates an application into model, view, and controller components as a means to build maintainable and robust code [2][3][21]. In MVC, the model can be thought of as application data, the view is the user interface for interacting with the model, and the controller processes user events from the view to update both the model and view as needed. This pattern is often implicitly followed by traditional web applications with the client embodying the view of an application and the server implementing both the model and controller. [6]

In an AJAX application, the MVC pattern does not separate so cleanly between client and server. Portions of the model and controller will reside on the client to increase application responsiveness (with the additional benefit of potentially

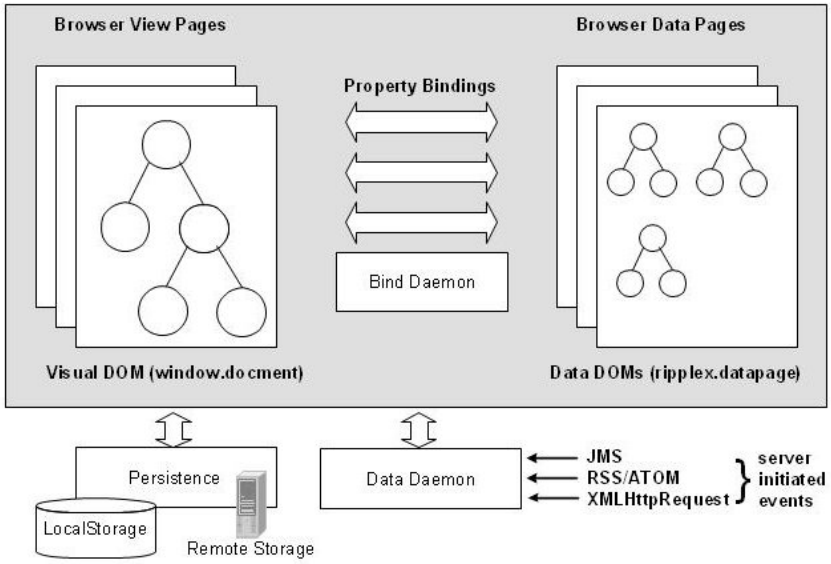

Figure 1 - Separation of AJAX application into view and data pages

reduced server load).

Browsers currently do not have the right programming abstractions to support the MVC approach. In Ripple-X, we extend the browser to support model abstractions. We provide an overview of our approach in the next section.

C. View Page and Data Page

Figure 1 shows how Ripple-X structures an AJAX application. Instead of a single document, we separate the AJAX application into a view page and a data page. The view page contains the browser DOM and represents all the presentation elements and scripting code. The data page is a container for one or more data DOMs that represent non-visual client-side state in XML form. For example, a data DOM could represent the values from a form, or a list of customers that will be displayed as a select list in the user interface. As shown in the figure, data DOMs are bound to the presentation element via a binding mechanism. This binding is active and when the underlying data DOM changes, the presentation elements bound to it receive an event so they can update themselves.

This separation has several advantages to the program designer and user: First, it provides cleaner abstractions for both the developer and user for client-state in an AJAX application. Client-state is simply the collection of data DOMs that are associated with the view page. This state is independent of the current view page and is portable to different AJAX applications, requiring only a definition of the bindings as shown in Figure 1.

Secondly, it allows for optimized management of data separately from application. AJAX applications that deal with large data sets in either XML or JSON format benefit from optimized data representations on the browser side. The browser DOM is not optimized as a container for non-visual data. For example, there is no support for compact data representations or stream-based (lazy) loading of data from the server. In our abstraction, the data page can implement browser-based data services such as efficient storage, retrieval, and synchronization with backend servers. These services would be available to any AJAX application loaded into the browser. This design is well aligned with the roadmap for future browsers which will include unified storage management for browser data (bookmarks, cache, cookies), e.g. through a small footprint relational database [25].

In our abstraction, an AJAX application has access to a single datapage. Datadoms stored in that datapage are only accessible to that application. Thus, the browser manages a datapage for each application it downloads

An AJAX application implies a data-centric approach to interacting with the server. Since the browser generates its presentation locally, communication with the server is mainly for propagating changes of the local application state back to the server. The server can commit this state to a transactional component or even propagate this state to other clients, e.g. through ATOM and RSS syndicated news feeds.

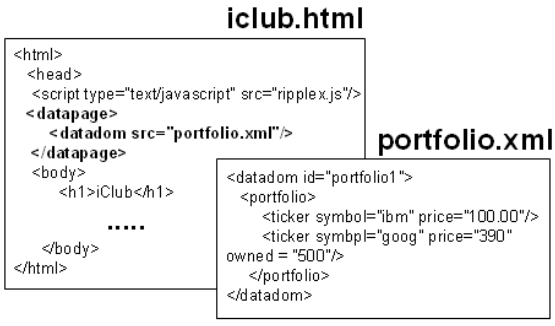

Figure 2 - iClub application with Ripple-X datapage and datadom elements

D. Before-Commit Data Services

AJAX applications can act more like desktop applications that engage the user for longer periods of time. In this mode, applications may still be server-managed but contact the server less frequently. Earlier, we referred to the glide time of an AJAX application as the period of time the application can reasonably run without contacting the server. Clearly, this is an application-specific measure; glide-time can be quite lengthy, such as for browser-based office applications, or quite short, such as browser based instant messaging applications.

We use the term glide time to refer to the time period during application runtime where there is a current gap in browser support for managing application state; we hope to provide browser-based data-centric services that can fill this gap. The goal of these services is to ease the programmer's burden in managing its own client state, and to help prepare the application for when it needs to contact the server to reconcile its application state.

We have identified the following services in our initial development of Ripple-X, which we believe provide value to typical AJAX applications We will describe these in more detail in Section III and IV.

- Checkpointing
- Operator Logging
- Feed-based updates
- Connectivity Masking

It should be noted that these four services can have value even in the case where the browser frequently contacts the server.

III. The Anatomy of a Ripple-X AJAX Application

In this section we provide an overview of an AJAX application implemented using Ripple-X. We use the example of a prototype investment application that helps investors manage a shared portfolio. Our current prototype is a testbed for the Ripple-X data services and not a fully usable application.

A. iClub Shared Portfolio Manager

The iClub Portfolio Manager allows users to collaborate on a shared investment portfolio. Users can submit information about their investment styles through a profile page, as well as make proposals to buy and sell equities to the rest of the members in their club. In an actual application, users would vote on each proposed action and any proposal that receives a majority positive vote will be processed. For simplicity, the current iClub prototype assumes that all proposals have received a majority vote and commits the proposed changes to the server without reconciling conflicts.

Figure 2 shows the HTML markup of the portfolio page where users can view the current portfolio and make buy and sell proposals. In the figure, the HTML markup includes a new element with the name `datapage`. The `datapage` wraps a declaration for an element called `datadom` that references an external file. The content of the external file is an XML

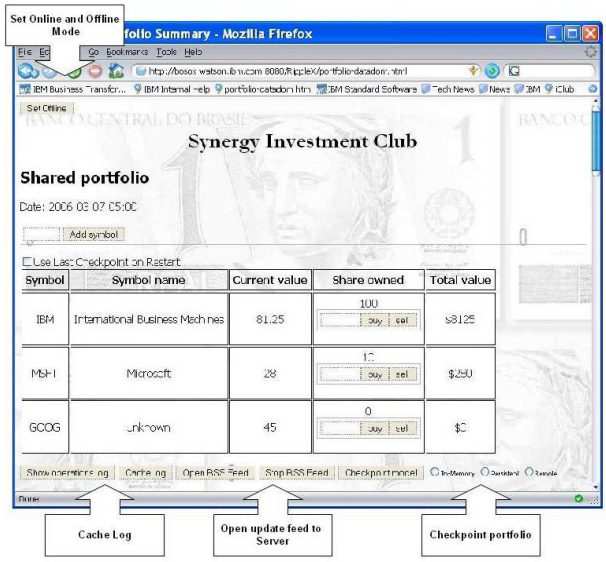

Figure 3 - iClub Portfolio Screen with Data Services

document that represents the state of the portfolio at the time the browser accesses the application from the server

In the iClub application, the browser downloads a replica of the portfolio data from the server and primarily modifies that copy. Other browsers may also access the iClub application and receive their own copy of the portfolio instance. As clients make modifications to the portfolio, the browser updates the local copy of the portfolio. The browser propagates these changes back to the server dependent on whether the application is in online or offline mode.

Figure 3 is a screenshot that shows the view of the portfolio as seen by one client. The buttons highlighted in the figure use the Ripple-X services as applied to managing the portfolio state:

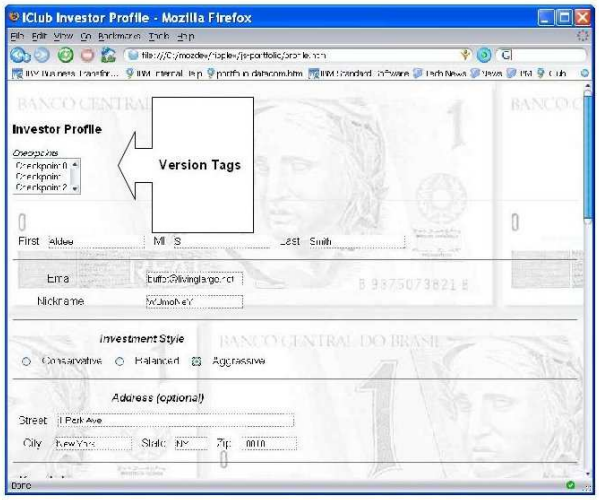

Figure 4 - Profile page with automatic versioning of application state

- *Create multiple tagged portfolio proposals*: the user can modify the currently displayed portfolio and then save it for later retrieval. The user can create an arbitrary number of versions of its local copy of the portfolio.
- *Log all updates and cache locally*: the user can elect to keep track of all updates made to the local portfolio using an available logger. The logger will keep track of a sequence of high-level user operations (like "buy 100 shares of IBM") or just low level DOM update calls (e.g. `setNodeValue` and `setAttribute` methods) depending on the application configuration
- *Work in offline mode*: the user can elect not to send any updates the server but still update the local copy of the portfolio. When the user goes online again, the portfolio changes are sent to the server as either a log of changes or the entire local portfolio state.
- *Subscribe to update feed from server*: the user can elect to receive the latest updates about the portfolio from the server, even if the user is idle. The iClub application will read an RSS or ATOM feed that contains portfolio updates from other clients and update the page as new data arrives

Ripple-X provides a library of client-side state management services that the iClub application uses to implement the above features. These services are generally useful to other applications as well. For example, Figure 4 is a screenshot of another part of the iClub application that allows the user to fill out an investment profile. Similar to the portfolio page, the profile page contains datapage and datadom elements that represent the profile data being rendered by the profile page. The user can edit the current profile as in any standard form, which updates the local copy of the profile. In this scenario however, as the user makes changes to the local profile copy, the application automatically makes a version history. The select list widget at the top of the page lists all the versions of the profile and allows the user to go back and forth between older and newer versions. When the user selects a version from the select list, the profile page automatically reverts to displaying that version of the profile.

Both pages are just manifestations of the Ripple-X data services and highlight how an application might use them in practice. In the next section, we provide details of these services.

IV. Ripple-X Packages

In this section, we describe packaging of the Ripple-X libraries and the features of each library.

Figure 5 shows the abstractions as implemented in Ripple-X. The left side of the figure illustrates the Ripple-X libraries that run on the browser. Ripple-X is completely written in Javascript and requires no extensions to the browser. The one exception is if the application requires support for persistence (see Section IV-I).

Figure 5 lists 5 core components that build on each other to provide enhanced client-side state management services. These services provided by these components are: 1) an operator framework, 2) property bindings between data and presentation elements, 3) change logging and checkpointing, 4) processing of server initiated updates, and 5) management for intermittent connectivity. We describe each of these in this section.

A. Packaging

Standard access to the different Ripple-X components is through Javascript libraries. In the current implementation, the components are organized into different libraries though some components depend on others being present (e.g. the logger

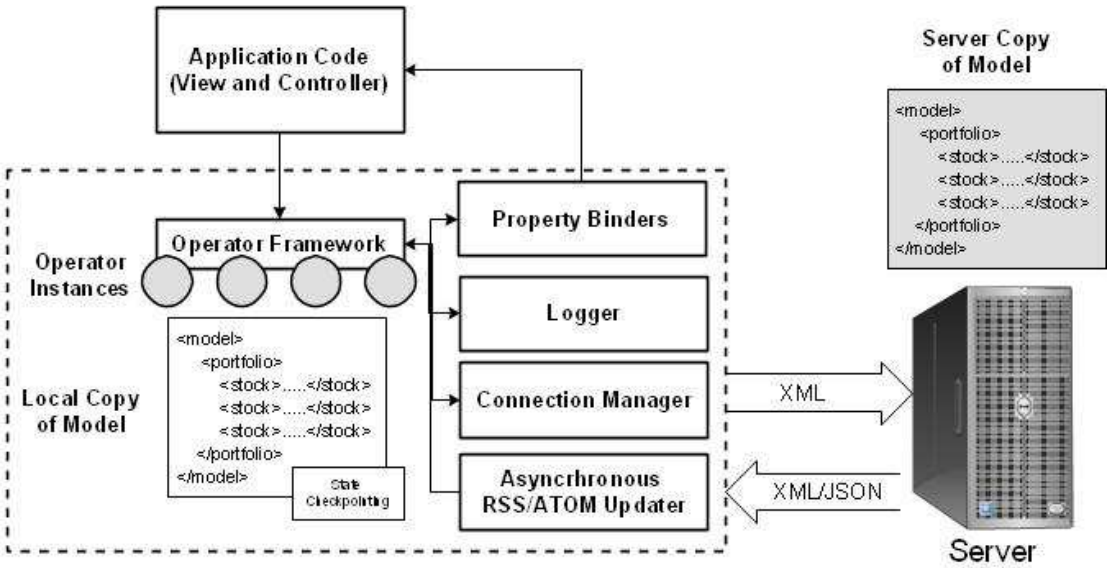

Figure 5 - Ripple-X implementation on browser

looks for the operator framework). The basic libraries are:

- ripplex-core.js – contains definitions for the datapage and data doms
- ripplex-logger.js – contains the logger
- ripplex-rssupdate.js – contains the RSS/ATOM updater
- ripplex-connection.js – contains the connection manager
- rippex-propertybinder.js – contains the code for binding a model to presentation elements.

The operator framework is included as a set of templates that the application programmer can use to define basic operations in Javascript. The operator templates follow traditional operator models by allowing pre- and post-conditions to be checked. We describe the operator framework in Section IV-E.

We use standard Javascript techniques to provide "namespacing" for each of the Ripple-X libraries. Ripple-X "objects" are prefaced using "ripplex." This mimics Java style namespacing of packages.

Ripple-X uses the Dojo Javascript libraries, which can be downloaded freely [9].

B. Core Ripple-X Libraries: ripplex-core.js

The core Ripple-X library implements the view and data abstractions described previously. We preserve access to the view DOM using the standard Javascript `document` element. In addition to the `document` element, there is now a `datapage` element, which is the container for 0 or more data items in the form of `datadom` elements. Applications access the datadoms through the datapage using basic get and put semantics.

For example, the Javascript for accessing the shared portfolio in the iClub application would be:

```
var datapage = ripplex.getDataPage();
var portfolio = datapage.get("portfolio1");
```

where "portfolio1" is a unique identifier for the portfolio datadom. In Ripple-X, each datadom contains a unique (in the context of the datapage) identifier. In the code above, the variable "portfolio" is a datadom, which provides access to the checkpointing and property binding services.

To create a datadom, an application can call:

```
var datadom =
     datapage.createDataDOM(uuid,xml);
datapage.put(uuid,datadom);
```

where "uuid" is the unique identifier for the datadom, and "xml" is a string containing an XML version of the datadom being created (as seen in Figure 1, the external file reference to a datadom). If no string is specified, the datapage creates a datadom with no instance data. The application can later assign instance data to the datadom using the `setRoot()` method call. The final line in the code adds the datadom to the datapage.

Datadoms represent model instances as XML documents that applications can access using standard XML APIs. The current implementation of Ripple-X supports the standard DOM API for navigating and manipulating the underlying XML document though we are investigating the use of XPath or XQuery processors for future work.

The datadom is a meta-structure that wraps XML instance data. To access the actual portfolio data in the iClub application, the code must access the instance of the portfolio. It has the option of accessing the datadom-level tag or the content-level tag for the instance data. For example, in the two calls:

```
var datadomTag = portfolio.getRoot();
var contentTag = portfolilo.getContent();
```

the first call returns the root of a DOM pointing to the "datadom" element in the instance data. Applications use this mode to retrieve metadata about the datadom, such as its UUID, and service-level attributes. The second call returns the root of a DOM pointing to the "portfolio" tag in the instance data. This gives the most direct access to the portfolio data for display. Applications may update the portfolio by retrieving the instance data from the `getContent` call directly, but the preferred method for performing updates is through the operator framework (see Section IV-E).

Checkpointing allows applications to maintain a tagged version of the datadom. Ripple-X provides a base level means to tag and checkpoint the current state by calling a `checkpoint(tagname, mode)` method on the datadom. There are three modes for checkpointing:

1. *In-Memory mode* – In this mode, the datadom creates the checkpoint and stores it in-memory. This checkpoint exists for only the current application session.
2. *Persistent mode* – in this mode, the datadom creates a checkpoint and serializes it into a UTF-16 formatted string. The application can store this string in an appropriate persistence mechanism. In the current implementation, we have implemented a small browser extension for Mozilla Firefox that uses an XPCOM interface for writing the checkpoint to the local file system. In the future, applications should be able to select the local persistence service, e.g. through the Dojo toolkit storage interface currently under development [20]
3. *Remote mode* – this mode is similar to the persistent mode, except the serialized version of the datadom is sent to a storage service located on the network. For security reasons, this service must originate from the same domain as the AJAX application. Using the remote mode, users download their checkpoints to any web browser on

any client; also, users can share checkpoints with each other via the server. In the current implementation, users specify the URI of the persistence service as an attribute of the datadom. In the current implementation we have a prototype server that uses basic get and put calls for storing checkpoints. Future implementations will need a way to ensure that the client can conform to the persistence service interface, e.g. using a Service Oriented Architecture (SOA) approach.

In all modes, the Ripple-X package keeps the checkpoint as a serialized version of the datadom. This serialized version is the same as the contents of the external file in Figure 1.

The datadom can easily restore a checkpoint by calling the `restoreCheckpoint(tag,mode))` method on the datadom. The mode is an optional parameter in this call – the default behavior is to assume a tag is unique across checkpointing modes and if the caller does not specify the mode, it will check in-memory, persistent, and remote checkpoints in order to find the tag. It will return the first checkpoint it finds with a matching name, otherwise the call will have no effect on the current datadom.

For example, to checkpoint the current state of the datadom, the application calls:

```
portfolio.checkpoint("MyCheckpoint",ripplex.
MEMORY_MODE)
```

which will store the checkpoint using the in-memory mode. Restoring the checkpoint requires the call:

```
portfolio.restoreCheckpoint("MyCheckpoint")
```

In persistent and remote mode, applications can use checkpointing to restore a datadom to a previous state across page reloads and browser sessions. The current implementation allows the application to declare an "initialize" checkpoint. If such a checkpoint exists, then the datadom will set its instance data to that checkpoint every time the application loads. For example,

```
portfolio.setInitialize(true,
"MyCheckpoint", mode);
```

The datadom will store the checkpoint "MyCheckpoint" using the mode specified by the "mode" parameter. Clearly, this mode should either be persistent or remote. The next time the application loads, the datadom representing the portfolio will search for its starting state as the checkpoint named "MyCheckpoint". If "MyCheckpoint" cannot be found, the datadom will set its state to the value defined by the server, if any.

C. Property Binding: ripplex.propertybinding.js

The MVC approach requires the programmer to specify the linkage between model components and the view. If done manually, this process can be fairly tedious and error prone. In our current implementation we have adopted a function-based approach to specify model and view bindings. In contrast, XForms provides a starting point for how one might take a declarative approach to specifying property bindings. A declarative approach has advantages over a function-based approach in some cases (see Section IV-H) and we are currently moving our implementation to a more fine-grained property binding approach using a declarative API.

In the current implementation, presentation elements bind themselves to datadoms using a simple subscription API. The API is:

```
datadom.subscribe (subid, object, method
name1, method name2);

datadom.unsubscribe (subid, object, method
name1, method name2);
```

Where:

- `subid` is a unique identifier for the subscription.
- `Object` is the object instance name that contains `method name1.`
- `method name1` is the method name to invoke to update the model.
- `method name2` is the method name to invoke to update the presentation.

For example, if we want to trigger an update to the presentation when the user creates a buy proposal, the code would look like:

```
portfolio.subscribe(
"MyTrigger",
buyOperator,
"execute",
"updatePortfolioView").
```

where "buyOperator" is an operator object bound to the portfolio and "updatePortfolioView" is a function call that will trigger an update of the portfolio view.

The current API provides a simple mechanism that relies on presentation elements subscribing to presentation events when the content of the datadom changes. A finer-grained approach may be more advantageous and as future work we are looking at different binding mechanisms where presentation elements bind themselves to elements of the instance data in the datadom. The binding uses XPath to express what elements the presentation element is bound to. For a property binding mechanism, only a subset of XPath is likely needed and one interesting problem is to find what subset of XPath is "safe" to use in the binding context.

Once the presentation binds to the datadom, updates to the datadom trigger updates to the presentation. In the iClub portfolio page from Figure 1, the presentation of the portfolio reflects the state of the portfolio datadom at all times. However, since the local copy of the portfolio is not committed to the server, iClub modifies the presentation depending on the change propagation mode of the application.

In iClub, the application can be either on-line or off-line. When working off-line the local model is updated optimistically according to the operation at hand. The presentation is refreshed, but with a visual clue that this update is *proposed* but not *committed* to the server. When working on-line, the client sends updates to the server immediately in the form of a serialized operator.

To summarize, the portfolio datadom can change in several ways:

1- An application defined operator is invoked, usually by pressing a button in the view. As mentioned above, when such operator is invoked, the presentation is updated along with the datadom, but the operation is not committed.
2- A server-initiated update using the RSS/ATOM feed information (see Section IV-G)
3- Restoring a previous checkpoint of the application state. This off-line operation restores a previous state of the data model and related pending operations.

D. Connection Management: ripplex-connection.js.

Ripple-X supports intermittent connectivity to the server by providing a connectivity manager. AJAX applications break the traditional request response model between browser and server by relying on asynchronous communications enabled by the XMLHttpRequest. The Ripple-X connectivity manager assumes that this connection is intermittent and relies on a store and forward mechanism for server communication. The connectivity manager provides three queues for organizing outgoing state change requests to the server. As the client and server exchange information about application state, the connectivity manager helps the client keep track of what data change operations are currently in process between the client and server

The connection manager is automatically invoked when the application updates a datadom through the operator framework. When the application executes an operator, the connection manager runs a queuing protocol. It attempts to send messages to the URL specified by the datadom's "src-uri" attribute. If no "src-uri" attribute is specified, then it attempts to contact the originating URL of the AJAX application.

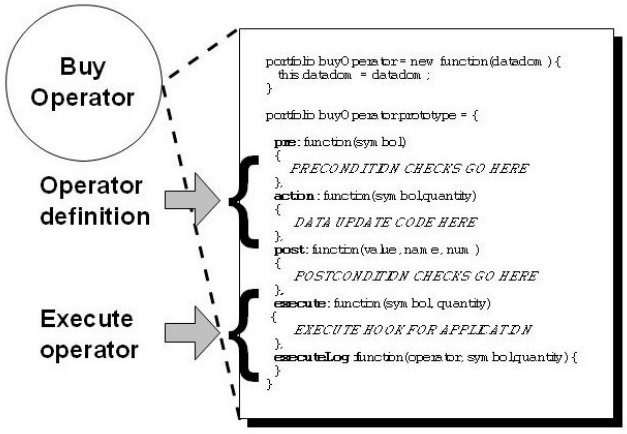

Figure 6 - Definition of Buy Operator using the operator framework

Update messages are in the form of serialized versions of operators in the Ripple-X operator framework. For example, if a client executes a buy operator, this is transformed into a *change string*, which serializes the details of the buy. An application forwards this change string to the connectivity manager. By default, the connectivity manager places the change string in a *store & forward* queue. When a string is the store & forward queue, the connectivity manager creates an update message and sends it to the server. Then the connectivity manager moves the operator call into a *pending* queue of sent messages that are awaiting a server response. When the client receives a response, the connectivity manager moves the change string from the pending queue into the *committed operations* queue, which represents the committed changes to the server.

Applications use the queue organization to provide the user with visual cues as to the state of their data modifications. For example, if the application wants to modify a shared portfolio by selling 100 shared of IBM stock, then the view of the portfolio can tag the IBM stock entry as "pending" and then committed as the serialized operator calls move between the queues.

In addition to organizing the messages, the logger can serialize the contents of the store & forward, pending, and committed queues. Applications that end abruptly can resume execution using the state of their last known interactions with the server. This is potentially useful for application recovery and rollbacks. As future work, we plan on investigating more robust recovery mechanism.

E. Operator Framework

The operator framework allows applications to specify operators over the model instance. These operators are application-specific and semantically represent high-level operations over data. For example, a user can modify a shared portfolio by buying and selling equities.

The operator template captures application-specific operations over a datadom. This allows the enforcement of high level update policies that may not be expressible or supported in low level schema languages. This is particularly useful to handle the case where one value might depend on another (e.g. if you delete a user from iClub, then you should also delete any proposals initiated by that user).

Ripple-X uses the operator framework to filter what data update operations should be logged, and act as a foundation for application-level synchronization mechanisms.

Figure 6 shows the operator template and an implementation of a buy operator. Applications create operators using this template and Javascript. The template says that an operator must implement an `action()` method, and optionally `pre()` and `post()` condition methods. The latter methods

are checks an operation may want to perform before and after the actual update operation described in `action()`.

The `execute()` method is passed any parameters needed by the aforementioned methods. The `executeLog()` method is not user specified but acts as a hook into the logger code.

Once the operator is defined, the application creates operators bound to a datadom. For example:

```
var buyOperator = new
portfolio.BuyOperator(portfolio);
buyOperator.execute("IBM",100");
```

where the variable "buyOperator" is bound to the datadom variable "portfolio."

Operators form the foundation for a richer form of data synchronization. In conjunction with the logger, it provides a mechanism for sending high-level descriptions of changes made to the data to the server. The server can "replay" these changes in a more robust manner since pre and post condition calls can encapsulate conflict handling routines. We plan to investigate the use of application-specific operators vs. a lower-level approach for AJAX applications as part of our research.

In the current prototype application, we use the operator framework to create a log of high-level updates and as an implementation of a command pattern when informing the server that the client has initiated an update. We envision that operator definition is not mandatory; the application can also access a default set of operators which represent low-level mutations of the DOM such as *setNodeValue()* and *setAttribute()*.

F. Logging: ripplex-logger.js

AJAX applications may experience extended periods of autonomous operation before communicating with the server. If the application is sharing state with the server, it needs a convenient way to communicate data changes it has made since the last communication. Ripple-X provides a logging component that works in conjunction with an operator framework for keeping track of these changes. Applications can send a serialized version of this log to the server in lieu of sending the actual data itself. At the same time, servers can send a log of operations to the client to be replayed as a means to synchronize data.

To get access to the logger, an application calls

```
  var buyOperator = new
portfolio.BuyOperator();
  var logger =
ripplex.getDatapage.getLogger();.
  Logger.attatch(buyOperator);
  buyOperator.execute("IBM",100);
  var log = logger.serialize();
```

The code creates a buy operator. and retrieves the logger service from the datapage. It then binds the buy operator to the logger using the `attach` method. When the buy operator executes, the logger creates a change entry in the log. A serialized version of the log contents is available via the `serialize` method.

The log contains a serialized representation of the executed operators. The current implementation stores:

- The name of the operator, which is assumed to be unique for the current application
- The UUID of the model instance over which the operator was executed
- A name-value pair list of any parameters that the application passed to the operator.

For example, if we buy 100 shares of IBM stock using portfolio with id="portfolio1", the log would contain the string

```
name=buy&modelID=portfolio1&symbol=IBM&n
umber=100
```

We can optionally include a timestamp in the log for each entry.

In the current version of Ripple-X, it is left to the application to process the log (on the server or client) for data synchronization, though we hope to extend our capabilities in this area.

G. Server-initiated Updates: ripplex-rssupdate.js

In the iClub application, multiple clients share portfolio data with each other. A user should receive updates made by other clients even if the user is idle. Currently, browsers have very limited facilities for receiving events from servers. For example, maintaining an open HTTP connection between the browser and server can be inefficient and unreliable because of timeouts that can result in undesired application behavior.

To support server initiated updates, Ripple-X provides an RSS/ATOM updater component that subscribes to a standard RSS or ATOM feed to receive changes made to the data on the server. Browsers traditionally use RSS and ATOM to subscribe and aggregate news feeds. Both types of feeds have support for polling and some support for asynchronous messaging. These specifications define a standard type of XML document that contains information regarding news feeds but can be tailored to carry arbitrary data. As RSS and ATOM are supported by base configurations of most browsers, we can leverage this to open "data consistency"

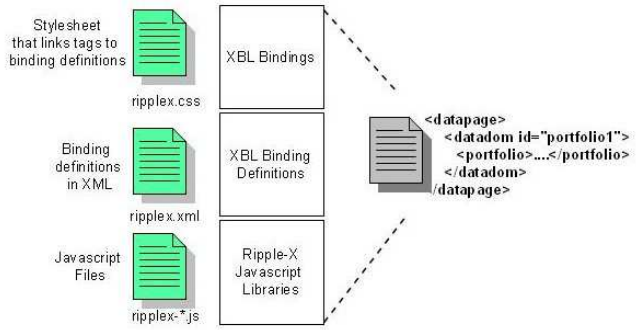

Figure 7 - Declarative interface implementing using Mozilla XBL

channels with servers to keep local data copies up to date.

The feed can contain changes in the form of a log, diffgrams [7], or just the latest copy of the data. In the iClub application, the server puts its copy of the portfolio state in the feed.

For example, the portfolio datadom can receive updates from the server through the following code:

```
portfolio.setUpdateFeed(url);
portfolio.subsribeToFeed(5);
```

which will set the URL of the RSS feed to the variable *url* and poll this feed every 5 seconds for updates. Alternatively, an application can set the "rss-url" attribute on the datadom element to specify the RSS/ATOM feed to subscribe to for changes. For security reasons, this feed must have the same origin as the AJAX application.

In the current implementation, the RSS/ATOM updater allows applications to poll an RSS feed representing server initiated changes to the local model. Currently, the application extracts the entire state of the model from the server from the application feed and overwrites the current local model. A more robust application would implement some reconciliation procedure to synchronize the two copies of the data on both the client and server.

H. Declarative and Javascript APIs

For convenience, we have implemented a declarative API for some of the Ripple-X libraries. Figure 2 shows how a datapage and datadom are declared using the ripple-x namespace

In Ripple-X, we take advantage of the Extensible Binding Language (XBL) [27] facility in Mozilla to define custom tags that provide an application with access to the core Javascript libraries. This requires that the application include a new style sheet in its definition that describes the definitions of the custom tags. The style sheet refers to a Ripple-X XML binding specification that must be accessible to the application. This binding specification defines the functions that can be called on the custom tags and their implementation.

Application code can access the underlying Ripple-X libraries using the `document.getElementById(id)` method where "id" is the id of the custom datapage or datadom tag. This call is standard on all base browser configurations that support Javascript. Once the application has a handle to that element, it can just call the Ripple-X libraries as if it were directly calling the Ripple-X Javascript code.

Figure 7 shows the mapping between the declarative API and the underlying Javascript code. In the figure, custom tag names are defined by the style sheet. Behaviors for elements bearing the custom tagname are defined in the XBL binding files. In Ripple-X, this binding file links to the Ripple-X libraries. As a simple example, for the declaration in Figure 7, the following code can retrieve the datapage:

```
var datapage =
document.getElementById("datapage");
var portfolio =
datapage.getInstance("portfolio1");
```

We utilize the declarative interface to extend Ripple-X features into existing HTML elements. Using style sheets and XBL, we have added checkpointing to standard HTML elements. For this to work, the application only needs to import the ripplex-core.js package and the included Ripple-X CSS styles and XBL specifications. With these installed, the application can create an HTML element and assign its class to be "checkable." This effectively extends the element with the checkpointing methods described previously. Thus, application programmers can checkpoint forms and visual elements in the same manner as Ripple-X checkpoints application state.

I. Persistence

Existing persistence mechanisms such as browser cookies or Internet Explorer *UserData* limits the size the application state. To have more flexible persistence options, we implemented two experimental local persistence mechanisms to support the Ripple-X services. Both currently require the use of the Mozilla browser as they are implemented as Mozilla extensions.

The first mechanism, OpenCache, uses the standard browser cache as a store. We added a callback mechanism to the cache, which allows the application to register for callbacks when changes occur to a cache object. This version requires changes to the core browser code.

The second mechanism, OpenCache-JS, is accessible as a browser extension and stores application data on the file system. The current version does not support callbacks but we plan to add this in the future. OpenCache-JS uses the *nsLocalFile* XPCOM component in Mozilla to write data files to a user configurable "cacheroot" directory. The default location of the cacheroot is the current user profile directory.

The iClub application uses OpenCache-JS to store application state. This state includes serialized datadom, serialized log contents, and initial state checkpoints.

V. Discussion

In this section we discuss some of the issues in designing a client-side state management system and its implications for browser design and web infrastructure.

A. Security & Privacy

Ripple-X does not change the security model for AJAX applications. Browsers policies nominally rely on same origin semantics for access control of backend data sources. An AJAX application can only communicate back to the same domain from which it originated. Cross-domain scripting is not allowed for AJAX applications originating from a server but allowable if the AJAX application is loaded from a local file on the client.

For Ripple-X, potential security and privacy issues arise from the persistence of application state on the client or remotely on the server. Two important issues are 1) access

control of local application state and 2) prevention of snooping application state from non-browser-based applications.

For the first issue, same origin semantics may be sufficient for limiting access to application data [15]. This may be too limiting if we want to enable client-based mash-ups [18], where we can create a new application that uses data from other applications. Allowing cross-domain scripting is dangerous from a privacy perspective since it may allow stealth user tracking. It may be possible to employ certificates to control access to data to a collection of authorized applications.

For the second issue, persistence mechanisms may not have protection mechanisms for sensitive application data that is stored locally on the client. For instance, browser cookies are fully readable by text editors. Clearly, some type of encryption could be employed to prevent snooping, though how this might interact with mash-ups is not clear.

For all security and privacy issues, it is critical that the user have control over security policies through clear and consistent mechanisms.

B. Standardization of models/data instances

In Ripple-X, we have adopted certain syntax to enable the MVC pattern on the browser. Similar efforts are underway by other communities, e.g. data islands in IE, XForms models, and MVC enablement by the Dojo Toolkit. Clearly, AJAX application developers would benefit from a standard syntax for defining the MVC pattern in the browser in much the same way as the DOM API is standard across browsers. For example, XForms models could exist independently of the XForms specification and provide a starting point on which to build a rich set of client- and server-based data-centric services for AJAX applications.

C. Standardization of data synchronization control formats

Ripple-X relies on RSS/ATOM news feeds to support server-initiated updates. RSS and ATOM are fairly generic but not optimized for the representation of consistency information. We utilize these feeds because code to support this is ubiquitous for browsers; a consistency mechanism could potentially be pushed into core browser code using existing components. As is the case for model and data definitions, AJAX developers may benefit from a standardized form of data feed that provides information about data changes. One example format could be the diffgram.

D. Browser Abstractions and AJAX Applications

The browser UI is primarily geared for surfing the web. Newer browser implementations, like Flock, add extra UI features that allow closer ties to socially oriented web services like shared photo databases and blogging. If may be useful to augment or modify the UI of the browser to support client-side state management utilities. For example, we have implemented a Datapage Inspector, which is a Mozilla extension for Ripple-X that displays the datapage and datadoms for a webpage in a tree view. This is similar to the DOM Inspector that can be installed in Mozilla but is tailored to the datapage and datadom abstractions. Another example for a UI component could be a versioning navigator like the automatic profile versioning from Figure 4. A versioning navigator could implement the browser's back and forward button semantics for an AJAX application based on how the application state has evolved.

In addition to tooling, UI design may need to provide user-centered feedback regarding the status of the data on the client. Applications may need to provide visual cues for data that is only locally updated and not committed to the server. Deeper abstractions, such as the concept of *external synchrony* [20] presented in file I/O research, may be applicable to UI issues regarding data persistence for AJAX applications. In *external synchrony*, the file system processes updates asynchronously but changes are buffered and not output to the user until committed to the file system.

E. Support for Disaggregation

The impact of AJAX applications may be felt across all tiers of the web infrastructure. Clearly, security and performance are impacted by changing user behavior and application workloads. As browsers become aggregation points for data and applications, it is possible that the trend will lead to further disaggregation of the web infrastructure on both the client and server tiers.

For example, the two dominant browsers in today's market IE and Mozilla Firefox, have moved more towards providing programming platforms. IE is closely integrated into the Windows operating system while Mozilla provides a comprehensive SDK to build non-browser applications with Mozilla libraries; for example, Songbird is a media player [22] which combines the open source media player VLC with Mozilla code and runs outside a browser. Using these platforms, developers can create applications where being disconnected is the exception rather than the rule. Internet-ready applications are already appearing without need for a browser. For example, Konfabulator [17] allows authors to create useful desktop widgets using Javascript for accessing information like stock prices, the weather, photographs, etc. from Internet sources.

This trend may lead to disaggregation on the server-side as well. Server-side mash-ups may give way to composable applications on the client. For example, the site Housing Maps [14] combines Google Maps [13] with housing information from housing lists to provide a geographically organized listing of available housing in major metropolitan areas. In the future, this mash-up can occur directly on the client, bypassing previous server-side aggregation points on the network. Clients may have a personalized version of housing maps by culling information directly from the source data providers.

VI. Related Work

The MVC pattern is first described by Burbeck [2], and later in many papers regarding application design. The application of this pattern to web application design is

prevalent in web literature, recent examples include [21][1]. Ripple-X applies the MVC pattern to browser-based applications. Toolkits like Dojo [9] and specifications like XForms [10] also adopt the MVC pattern. Ripple-X extends Dojo by providing client-side state management facilities. Ripple-X specifies MVC differently from XForms models, but is generally compatible and we are investigating the integration of our data services into an extended version of the XForms model.

The term AJAX first appeared in [12], and defines a programming style using a collection of existing technologies widely available on browsers, e.g. Javascript [11] and XMLHttpRequest [28]. Using AJAX techniques, developers can create browser-based applications that are much closer to their desktop counterparts in both look and responsiveness. AJAX is still the subject of many industry articles and books [6].

Data durability and synchronization is large and richly discussed research topic, e.g. [5][16][23], and our project takes initial steps to address some of the larger issues in an AJAX context.. In this paper we focus on browser-based services that can assist the client before committing data to the server. The current version of Ripple-X assumes that an external entity will perform data synchronization; we have implemented an application level policy-based synchronization service in Java [4] and hope to migrate this to the browser context.

Saving application state locally requires browser-based extensions with access to system resources. While we have developed OpenCache and OpenCache-JS, there are other efforts, e.g. AMASS integration into a generic Dojo storage framework [19].

VII. Conclusions

Ripple-X is a set of Javascript libraries and namespace extensions that provides data-centric services to AJAX applications. Ripple-X is designed using the MVC approach and overlays this separation of concerns on AJAX applications. Using Ripple-X, AJAX applications have access to logging, checkpointing, RSS/ATOM updates, and a connectivity manager.

We are currently implementing a new version of Ripple-X that will incorporate support for fine-grained property bindings. We are investigating mechanisms to support lazy loading of large models from the server onto the client. As part of this work we are looking at several client profiles, including mobile devices.

Acknowledgment

The authors thank Lionel Villard, Apratim Purakayastha, Charlie Weicha, and Rich Thompson at IBM Research for their help designing and developing this project.

References

[1] F. Bellas, D. Fernandez, A. Muino, " A Flexible Framework for Engineering "My" Portals," Proceedings of the 13th International World Wide Web Conference WWW 2004, May 2004, New York, NY USA

[2] S. Burbeck, "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)," University of Illinois in Urbana-Champaign (UIUC) Smalltalk Archive. http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[3] R. Cardone, D Soroker, A Tiwari, "Using XForms to Simplify Web Programming," Proceedings of the 14th International World Wide Web Conference WWW 2005, May 2005, Chiba Japan

[4] P. Castro, F. Giraud, R. Konuru, A. Purakayastha, D. Yeh:, " A Programming Framework for Mobilizing Enterprise Applications,". Proceedings of the IEEE Workshop for Mobile Computing Systems and Applications,WMCSA 2004, Nov 2004

[5] J. Cho, H. Garcia-Molina, "Synchronizing a Database to Improve Freshness". SIGMOD 2000.

[6] D. Crane, E. Pascarello, D. James, "Ajax in Action," Manning Publications, Greenwich CT USA 2006

[7] Diffgrams, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpcondiffgrams.asp

[8] Document Object Model, http://www.w3.org/DOM/

[9] Dojo Toolkit, http://dojotoolkit.org

[10] M. Dubinko, L. Klotz, R. Merrick, T. Raman, "XForms 1.0," World Wide Web Consortium (Recommendation) October 2003. http://www.w3.org/MarkUp/Forms.

[11] ECMA International Standard EMCA-262, "ECMAScript Language Specification," 3rd edition, http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[12] J. Garrett, "AJAX: A New Approach to Web Applications," http://adaptivepath.com/publications/essays/archives/000385.php

[13] Google Maps, http://maps.google.com

[14] Housing Maps://www.housingmaps.com

[15] C. Jackson, D. Boneh, A. Bortz, J.C. Mitchell, "Protecting Browser State from Web Privacy Attacks." Proceedings of the 15th International World Wide Web Conference, WWW 2006, May 2006.

[16]] J. Kistler, M. Satyanarayanan, "Disconnected Operation in the Coda File System," ACM Transactions on Computer Systems, 10(1):3-25, February 1992.

[17] Konfabulator, http://widgets.yahoo.com

[18] D. Merrill, "Mashups: the New Breed of Web App," IBM DeveloperWorks, http://www-128.ibm.com/developerworks/library/x-mashups.html?ca=dgr-lnxw16MashupChallenges, Aug 2006

[19] B. Neuberg, "Now in a Browser Near You: Offline Access and Permanent, Client-Side Storage, Thanks to Dojo.Storage," available at http://codinginparadise.org/weblog/2006/04/now-in-browser-near-you-offline-access.html

[20] E. Nightingale, K. Veeraraghavan, P. Chen, J. Flinn, "Rethink the Sync," Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, November 2006

[21] T. Parr, "Enforcing Strict Model-View Separation in Template Engines," Proceedings of the 13th International World Wide Web Conference WWW 2004, May 2004, New York, NY USA

[22] Songbird, http://www.songbirdnest.com

[23]] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, C. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage Sys-tem," SOSP 1995.

[24] VLC Media Player, http://www.videolan.org/vlc/

[25] V. Vukicevic, "Mozilla2 Unified Storage, http://wiki.mozilla.org/Mozilla2:Unified_Storage:

[26] Writely, http://www.writely.com

[27] XBL, Extensible Binding Language, http://www.mozilla.org/projects/xbl/xbl.html

[28] The XMLHttpRequest Object, W3C Working Draft 19 June 2006, http://www.w3.org/TR/XMLHttpRequest