RC24544 (W0805-002) May 1, 2008
Computer Science

IBM Research Report

CANSYNC: Middleware for Data Checkpointing and Synchronization for Disconnected Browsers

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

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

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

CANSYNC: Middleware for Data Checkpointing and Synchronization for Disconnected Browsers

Paul Castro[1], Frederique Giraud[1], Ravi Konuru[1]

[1] IBM T. J. Watson Research Center
Hawthorne, NY, USA
{castrop, giraud, rkonuru}@us.ibm.com

Abstract. As web applications offload more application processing into the browser, there is growing interest in disconnecting browsers from servers. In a disconnected browser, applications are responsible for caching and managing application state without access to server resources. In this paper, we present CANSYNC, a lightweight data layer that executes on both the browser and server to manage data checkpointing and synchronization tasks for a disconnected web application. CANSYNC provides change logging and undo/redo features for browser-based application state management, as well as a customizable synchronization engine for reconciling browser state with the server. The CANSYNC API is accessible in Javascript on the client and does not require the installation of browser extensions. CANSYNC server-side components can be accessed using HTTP calls. We describe the design and implementation of CANSYNC in the context of a prototype bookmark sharing application.

1 Introduction

Safely and seamlessly disconnecting clients from server-based resources has been an area of research in many areas including file systems [17], databases [7], and web services [25]. Developers are increasingly interested in enhancing the availability of web applications by providing a disconnected mode of operation for code executing in the browser. At first glance, disconnecting a web application from the network can seem contradictory; one touted benefit of a web application over its desktop counterpart is the centralization of administration and resources on a server, where clients only need a browser and a network connection to run the application. However, as users become more mobile, having a network connection is difficult to guarantee. Even with intermittent connectivity, a disconnected mode for an application can be beneficial; aggressively placing application logic and data in the browser can increase application responsiveness, make the application more fault-tolerant, and mask an application from QoS variations in network service.

Currently, disconnecting a browser is achievable through Google Gears [12], an open-source, cross-platform browser extension that provides four basic building blocks that developers can leverage to implement a disconnected mode for their applications: a local database for persistence, a local proxy to redirect application

calls to local resources, a resource manager to replicate server resources to the browser, and a worker pool that enables a close approximation of multi-threading for a Javascript application for the purpose of keeping the browser UI responsive. Google Gears is designed to incrementally improve the browsers ability to disconnect from the server and so by design it does not provide everything a developer needs; notably absent are facilities to help the application manage and reconcile its replicated application state with that on the server.

Managing application state within the basic browser programming model can be difficult to implement. While there are proposals to create separate data storage APIs for the browser [9], the current browser's heritage as a document renderer has resulted in an ad hoc programming model where each individual visual element may manage its own data. Untangling data from the application UI is difficult as it is fundamental to the design of the browser's HTML-based Document Object Model (DOM) [8]. Although this HTML DOM is not optimized to manage non-visual application state, developers use it as a catch-all container on the client for presentation, application logic, and data.

In this paper, we present a framework that improves a browser's ability to manage application state and helps facilitate disconnection for browser-based applications. It is likely that Google Gears, or some future variant that provides similar capabilities, will provide the basic building blocks to enable disconnection; we extend this work by providing a thin data services layer that is split between the client and server and address two areas we believe are fundamental to application state management in the browser:

- a mechanism to checkpoint local application state that enables features like undo/redo, suspend/resume, version management, simplified audit trails, and improved fault tolerance for a web application.
- a customizable synchronization engine that implements different synchronization modes (e.g. fast vs. slow synchronization) and executes application-specific policies for synchronizing a local copy of application state with that on the server.

In this paper we present the Checkpointing ANd SYNChronization (CANSYNC) data layer, which provides checkpointing and synchronization as services accessible to application programmers as Javascript libraries that are easily integrated into application code. As part of the design of CANSYNC, we look at a user-level widget model that provides visual metaphors for accessing the less esoteric aspects of CANSYNC. The use of these widgets is optional; both the checkpointing and data synchronization services in CANSYNC have public Javascript APIs for client-based components, and an HTTP-based protocol for accessing server-based components.

The remainder of this paper is as follows: in Section 2, we motivate data-centric disconnection and describe an example application that uses CANSYNC; in Section 3, we provide a design overview of CANSYNC in relation to the Model-View-Controller design pattern; in Section 4 we present details regarding the widget programming model in CANSYNC and the implementation of the CANSYNC checkpointing and data synchronization services; in Section 5 we discuss related work; we conclude with a discussion of some of the challenges of state management framework like CANSYNC and describe future work in Section 6.

2 Example Application that Uses CANSYNC

In this section, we discus a data centric view of disconnection and present an example of an application that primarily uses locally cached data.

2.1 Data Centric View of Disconnection

Disconnecting an application relies on aggressive caching. A browser application is typically one or more HTML pages that have dependent files such as style sheets, external source files, and media files. In general, disconnection involves caching these pages and their dependent files. It may also involve replicating portions of the server-side execution environment needed by these pages onto the client. Replicating the latter can be complex, so one simplifying assumption is that all application logic needed during disconnection is already in the HTML pages or dependent files, and any feature in the application that requires server-side code can be safely disabled.

Automatically scanning and downloading an HTML page and its dependent files is relatively straightforward [17]. It is more difficult to determine the application state needed on the client if it is managed on the server through some opaque means, e.g. in a database. Replicating an entire data store onto the client is not usually feasible. In general, the web application page will contain a snapshot of the data needed for its initial view; if subsequent calls to a server-side database are needed for additional views, then the application may fail to execute. As an example, Google Maps [13] displays a scrollable map, which locally caches a subset of map images needed for the currently displayed view. If Google Maps were disconnect-able, we would require knowledge about additional map images to cache since the user may wish to consult maps beyond the initial view during disconnection. Much research has been done in predictive caching, for example in file systems [16] and browser caches [14]. For a browser-based application, it would be useful for application developers to have the ability to explicitly declare the data needed to be cached. This can be done in parallel with page-level replication.

In this paper, we take a data-centric view of disconnection, where we assume that the initial page-level caching is already done, e.g. by a developer using the managed resource module in Google Gears to specify what files on a server need to be replicated to the client for disconnection. Instead, we focus on the post-initialization phase of disconnection, where the browser manages a local replica of non-visual data. In this model, the client works primarily with locally cached data, and separate data centric services manage the data, e.g. keep the locally cached data synchronized with the server. We constrain ourselves to solutions that can work in base configurations of browsers, though we utilize installable browser extensions, such as the embedded SQlite in Google Gears, if they are available.

Our goal is to provide programmers with browser-specific facilities to specify, manage, and synchronize application state. We take a lower-level view of data synchronization and create a synchronization framework based on basic concepts derived from other data centric approaches, such as SyncML [20].

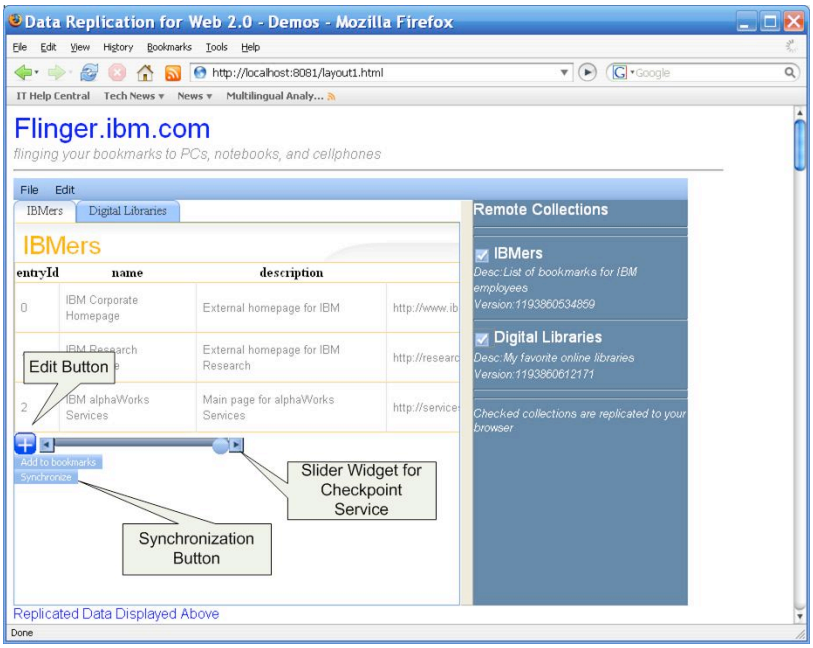

Fig. 1. Screenshot of the prototype Flinger bookmarking application

2.2 *Flinger* Bookmarking Application

Flinger is a prototype browser-based, collaborative web application developed as a test-bed for the data-centric design in CANSYNC. In Flinger, users create and exchange collections of semantically related bookmarks via a browser-based interface and a Flinger server. For example, a user could be researching a possible stock purchase. During the research process, she bookmarks sites that have relevant information about the stock. Flinger allows the user to group these bookmarks into a collection, tag them, and then publish the collection to a Flinger server to share with other users. Users can browse the published collections on the Flinger server and download the collections they would like loaded into their browser.

Rather than channeling all update operations on collections to the Flinger server, Flinger replicates data to the browser. Users work with local copies of bookmark collections and only contact the server to propagate changes. This potentially reduces the overall load on the server, but allows the local copies of collections to diverge. This is analogous to the Asynchronous Javascript and XML (AJAX) [11] design pattern, which relies on asynchronous messaging and locally executed Javascript code to create a browser application with a richer, more responsive interface. Flinger inherently favors a disconnected mode of operation where users can update cached copies of collections without contacting the Flinger server.

Figure 1 is a screenshot of the main window of the Flinger application. One the right hand side of the window, Flinger displays a list of the available collections on the Flinger server. The list includes the collection's name, a short description of the content, and a version number. Next to the collection's name is a checkbox. When the checkbox is checked, Flinger replicates the collection from the server to the browser. Flinger renders the collection into a table located in the tabbed windowpane to the left of the collection list. In Figure 1, the client has checked off two lists, which are replicated locally.

2.3 CANSYNC in *Flinger*

Figure 1 highlights two visual widgets below the table, a slider bar and a synchronization button, which provide visual metaphors for the checkpointing and data synchronization services in CANSYNC. The table is fully editable; the user can select a cell in the table and double-click it to enter the table edit mode. In edit mode, the user can change the value of a table cell and then un-select the cell (by clicking away from the cell or pressing the return key on the keyboard) to commit the change. The table is tied to an in-memory datastore that contains a string representation of the table contents using the Javascript Object Notation format (JSON) [10]. Users can bypass the table and edit this string directly in Javascript, which will similarly commit the change and also trigger the table to update its display to the value represented by the updated JSON string.

The user can access CANSYNC's checkpointing feature via the slider bar. When a user makes a change to the content of the table, CANSYNC automatically creates a new entry in a change history for the table. By moving the slider to the left, the user restores the table's data to a previous checkpoint in the history, which triggers a redraw of the table to that version. Moving the slider to the right, the table redraws to a newer version.

The user can access CANSYNC's synchronization feature by pressing the synchronization button. When the user synchronizes, CANSYNC propagates all local changes to the Flinger server. In this example, these local changes are the log of changes recorded by CANSYNC's checkpointing service. The server reacts by updating its copy of the collection and recording the changes in a log. It also generates a message back to the client that contains any changes made to the server's copy since the last time the client synchronized. Client-side code in CANSYNC processes these updates in the browser and updates the local collection accordingly.

CANSYNC synchronization executes a Flinger-specified policy to resolve update conflicts. This policy is based on "last write wins" semantics. Under this policy, the Flinger server will always commit the updates it receives from the client. The server will only forward updates to the client made since the last synchronization with that client if they do not conflict with the most recently submitted updates from the client. This is sufficient for an application like Flinger, which can function with weakly consistent replicas and has no hard requirement for eventual consistency of all clients. However, CANSYNC allows the application to create any custom synchronization policy using the change provider synchronization API (see section 4.2.2).

We note that there is no special disconnected mode in Flinger. Because Flinger always works with local copies, it only requires a network collection to initially cache bookmark collections from the server. To select which collections to cache, the user selects the collection from the collection list. Flinger replicates the selected collections automatically. When the user reconnects to the network, she can simply press the synchronization button to synchronize her local copy with that on the server.

Clearly, other applications will have differing requirements than Flinger regarding adaptation to disconnection, reconciliation of replicas with conflicting updates, and the policy for selecting items to replicate to the browser. Part of the design philosophy of CANSYNC is to allow applications to heavily customize the checkpoint and synchronization services to meet their requirements. Application developers can exploit application-specific features of the content to optimize caching and reconciliation mechanisms.

3 CANSYNC Design as MVC Data Services Layer

The Model-View-Controller (MVC) design pattern separates an application into model, view, and controller components as a means to create more robust and maintainable code [3]. 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. In these applications, the browser is the view, and the server implements both the model and controller.

Web applications are increasingly moving more Model and Controller parts to the browser, which is pushing the original design of the browser as a document renderer to its limits. Emerging client programming toolkits are beginning to define a data layer to manage these the new responsibilities of browser-based code. For example, the Dojo Toolkit [9] is a popular Javascript library that, in addition to providing a palette of enhanced user interface (UI) widgets, defines an MVC programming model that specifies standard storage and data retrieval APIs for code executing in a browser. By separating an application in the browser into separate model, view and controller components, data can be managed independently of the browser Document Object Model (DOM) or application UI code. Applications that handle large data sets would benefit from optimized data representations in the browser, such as the planned integration of SQLite into upcoming versions Mozilla Firefox [26]. Independent data management makes browser UI processing and style sheet logic re-usable across the changes in data.

3.1 Disconnection and Browser MVC Models

Disconnection toolkits introduce MVC into the browser because they rely on caching data into a local store. For example, in the Google Gears approach, the application UI communicates with a data switch, which can direct data calls to local or remote data resources. This isolates application UI code from separately stored data. CANSYNC

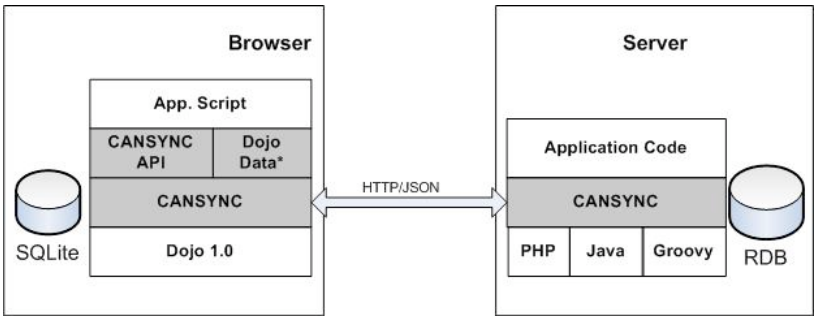

Fig. 2. The CANSYNC Data layer on client and server

extends the features of the local data store by allowing application programmers to create multiple versions of stored data organized as a history of checkpoints. This can be exposed as an API in the data switch, which allows an application to specify which version of data it would like to access.

Dojo has defined its own disconnection API, the Dojo Offline Toolkit (DOT) [18], which uses Google Gears but adds some functionality above the basic Google Gears building blocks. DOT provides facilities for recording user-level action logs containing application operations that were performed while the application was offline. In DOT, action logs can be translated to HTTP calls that the user would have sent if she had been online.

CANSYNC extends DOT by providing a more comprehensive framework to implement data synchronization for an application. CANSYNC spans both client and server and defines a synchronization communication protocol between the two that can be used by a wide range of applications. In CANSYNC, applications can define custom synchronization policies using the CANSYNC API that isolates the policy from the CANSYNC synchronization engine. CANSYNC's data-centric synchronization provides facilities for performing store-level synchronization, as opposed to the user-level action logs in DOT.

3.2 CANSYNC Data Layer

CANSYNC is designed as a relatively lightweight data layer that sits between application logic in the browser, and the lower-level data access layer on both the client (for disconnection) and server. CANSYNC purposely attempts to minimize the introduction of new data abstractions. As such, CANSYNC adopts a programming model that allows developers to write data handling functions in standard Javascript and just instantiate a limited set of CANSYNC defined Javascript objects to access CANSYNC services. CANSYNC also extends the data API specified by the Dojo Toolkit to provide a richer API for data checkpointing and synchronization.

Figure 2 shows the positioning of CANSYNC as part of a web application. On the left side of the figure, CANSYNC consists of client components written in Javascript

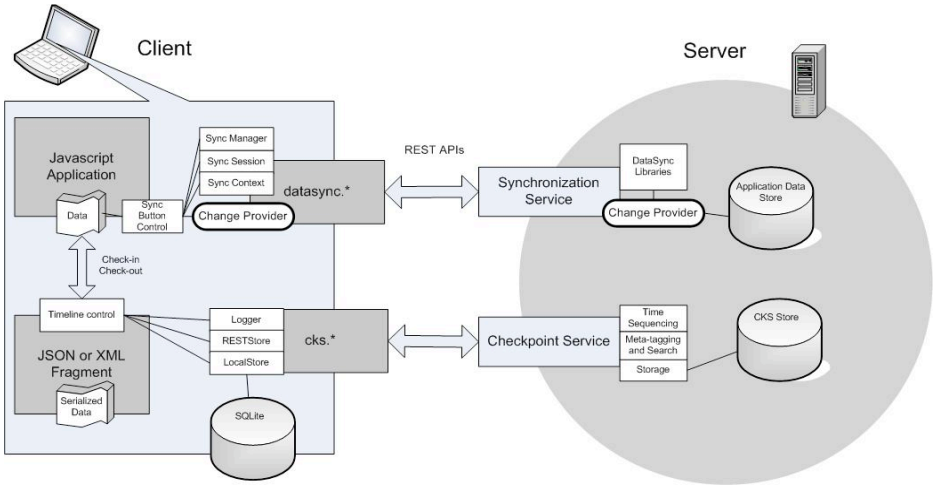

Fig. 3. Architectural Overview of CANSYNC

that optionally can use the SQLite database included in Google Gears. CANSYNC is developed as an extension to Dojo 1.0 though it provides both a Dojo API and a custom non-Dojo API for application scripts.

CANSYNC provides programmatic means for an application developer to separate the management of non-visual data from presentation elements. For example, the Flinger application replicates a bookmark collection represented as a JSON string. The application transfers the management of this string to CANSYNC. Internally, CANSYNC persists this JSON data into a local (or remote) store.

We refer to the deployment model of CANSYNC as an instance of model-based replication. In model-based replication, the client can replicate both data and data services from the server. The data services implement the core functions of CANSYNC. Clients can enable CANSYNC by accessing the features via a model API without the need to install browser extensions such as Mozilla Firefox plug-ins.

CANSYNC communicates with its peer components on the server-side using HTTP with JSON as a data exchange format. In the current version, CANSYNC server-side components are implemented in PHP, Java and Groovy. As part of our on-going development, we have a PHP implementation of CANSYNC that provides support for clients on mobile devices.

4 CANSYNC Implementation and Programming Model

We have implemented the checkpointing and data synchronization services in CANSYNC. Figure 3 is an overview of the architecture of CANSYNC. In the figure, CANSYNC provides client and server components to enable the checkpointing and synchronization services. The client is executing a Javascript application, which uses

```
// initialize CANSYNC widget
var jsonString = "{'IBMers':[{'name':'IBM Homepage','url':'www.ibm.com','desc':'IBM home'}]"
var timecontrol = new cks.TimelineControls(sliderRoot,jsonString,"local");
timecontrol.setValueNow();

// get data from widget and use
var data = timecontrol.checkout();

// normal access pattern
data.IBMers[0].desc = "IBM homepage for external visitors";

// put data back in CANSYNC timeline widget to trigger diff detection
timecontrol.checkin(data);

// checkpoint
timecontrol.checkpoint("mycheckpoint");

// register with synchronization service
var syncManager = new datasync.SyncManager();
syncManager.register("myUserId");

// create a synchronization session
var syncSession = new dataSyncSession();
syncSession.setSyncContext(… parameters for sync context…);
syncManager.add("mysession",session);


// synchronize data with server
syncSession.sync();
```

Fig. 4. Client code example using SMW2.0

some non-visual data. The application can transfer management of this data to CANSYNC via the timeline control or the sync button control widgets. These widgets act as front ends to the different service components, described in more detail in this section. On the client, these widgets are implemented in Javascript and the Dojo Toolkit [9].

The server hosts the necessary synchronization libraries as well as a remote checkpoint store. The latter is useful for clients that may not require disconnection and need an off-board place to store checkpoints. The client and server communicate using an HTTP/JSON-based API. CANSYNC includes a reference implementation of the synchronization service, which uses JSON formatted data on both client and server side. This reference implementation allows a client to synchronize two JSON objects. Flinger uses this reference implementation.

CANSYNC has a widget-based API and a lower-level Javascript API based on extensions to Dojo. Figure 4 is example of client code that uses the CANSYNC widget API. The current client footprint of the checkpointing service in CANSYNC is ~58kb of Javascript code. The synchronization service is ~80kb of Javascript code. Each service can use the Dojo packaging scheme to select pieces of each service to reduce the overall footprint.

4.1 CANSYNC Checkpointing

The checkpointing service in CANSYNC is both a client-only and client-server service that provides an application with the ability to log changes to application state, create checkpoints of that state, and retrieve the state using key-based queries or a checkpoint history timeline abstraction. Using the checkpointing service, an

application can enable a simple undo and redo feature that can be persistent across application and browser sessions. The checkpointed state can be stored locally or use a server-based checkpoint store

The checkpoint service consists of three components: 1) timeline control widget, 2) change logger, and 3) a persistence API. Each component can be used independently of the other so applications can customize them as needed. We describe the widget API following sections. Note that the checkpoint service also has a low-level API for directly accessing CANSYNC checkpointing in Javascript based on extensions to the Dojo Data specification. For brevity we do not cover the low-level API in this paper.

4.1.1 Timeline Control Widget API

The timeline control is an HTML widget that represents a change history for a particular piece of application data. The timeline control organizes the change history along timeline, and provides the application with the necessary controls to navigate this timeline.

In CANSYNC, the timeline control provides a simple, 1 dimensional history. As part of our research, we plan to investigate more complex history schemes, such as tree schemes that allow histories with multiple branches.

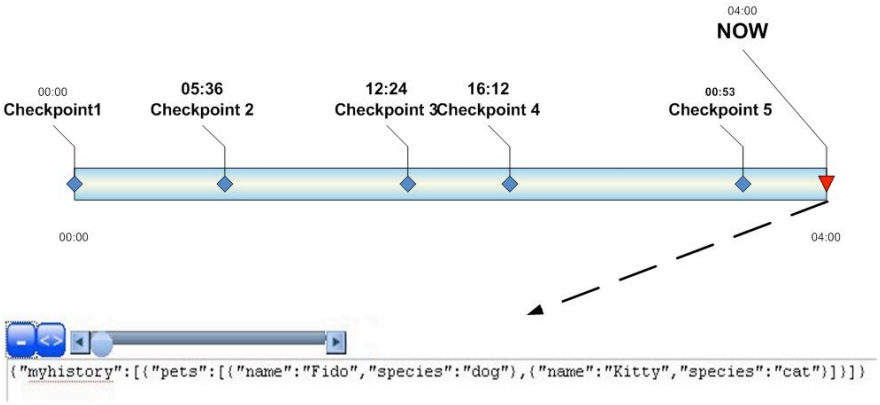

Fig. 5. Close-up of timeline control with editing window

Figure 5 shows an example timeline that describes the update history of an object and how this maps to the timeline control abstractions. In the figure, the timeline has a "now" object, which is the latest version of the data, and a history of checkpoints. As the user moves the slider bar, she moves a cursor between earlier and later checkpoints. The "now" object is always the value of the cursor. Using the timeline control, the application state can be thought of as the "now" object, with past and potential updates available for restoration. When the application restores a checkpoint, this checkpoint is copied and the copy is appended as the latest entry in the timeline. The "now" object then points to that entry.

In Figure 4, the application instantiates a timeline control with a reference to the parameters `jsonStr` and `sliderRoot`, which are the source data and the DOM element in the document, which will render the slider bar, respectively. The timeline

control initially assigns the "now" object as this string and has an empty history timeline.

The timeline control plus JSON-formatted data is the model for the checkpoint service. Note that the combination of the timeline control and the synchronization libraries, described in Section 4.2, provide the necessary consistency logic to enable disconnection and reconciliation. In CANSYNC, the timeline control should be used to coordinate access to the data. This will ensure that different elements that rely on the data are always referring to the same version of the data.

For coordination, the timeline control provides `checkout()` and `checkin()` functions to access the data. When checkout is called, the timeline control returns the "now" object in the timeline. Internally, it hydrates the original JSON string into a Javascript object hierarchy and returns that to the application. In the figure, the application retrieves the data from `checkout()` and then updates the data using standard Javascript code.

When the application wishes to make commit the updates, it uses the checkin function. The checkin() function will do two things: 1) if the timeline control is configured to detect changes, it will scan the updated object and compare it to the original object using an internal change logger object. The logger will append any detected updates to a log. 2) the new object is serialized and replaces the original object.

The `checkout()` and `checkin()` functions can be customized to generate events to inform elements that the data has been updated. The timeline control has an `addListener()` function to subscribe to these events.

The timeline control provides `checkpoint()` and `restoreCheckpoint()` functions to create and retrieve application state. When the application calls `checkpoint()`, the timeline control creates a new entry in the timeline using its "now" object. The application can provide additional parameters to the checkpoint to enable different modes of navigating the timeline. For example, the application can pass in a label "key1" which will be used to uniquely identify that checkpoint. It also passes in key words such as "bookmark." Key words are tags on the checkpoint and can be used to search for checkpoints using key word searches.

The `restoreCheckpoint()` function retrieves a checkpoint in the timeline and replaces the current "now" object with the retrieved checkpoint. The application can access a stored checkpoint in several ways. First, the application can use the unique key of the checkpoint if it has one. It can also identify a checkpoint by naming its position in the timeline using an index number where 0 is the earliest checkpoint. Finally, it can use the cursor controls to assign the "now" object to that checkpoint. In the cursor controls, a call to `getNext()` or `getPrevious()` will move the cursor one step forward or backward in the timeline, respectively. This will restore the checkpoint at that position. If there is no `getNext()` or `getPrevious()`, as is the case when the cursor is at the extreme ends of the timeline, the "now" object does not change and the cursor does not move.

The timeline control also offers window-based access to checkpoints in the timeline. In this mode, the application specifies the starting and ending positions of the window to the `getWindow()` function. This will return an array representing the checkpoints that fall within the specified window. This is useful for prefetching data

to increase the responsiveness of the timeline control since the checkpoint history can be stored on a network-based store (see Section 4.1.3).

The time control has optional visual elements that can be displayed and used to access the checkpoint history. Figure 5 shows the slider bar and how the internal functions are mapped to the control. In the figure, the "+" button can be used to show the underlying JSON string that represents the "now" object. This timeline control displays this string in an editable textarea. The user can change the value of this string in this textarea.

4.1.2 Checkpoint Change Logger

The timeline control contains a checkpoint logger so the application does not have to directly access the change logger API. The checkpoint change logger is a client-side component that is responsible for detecting and logging changes to locally cached JSON data. Logging of updates is fundamental to enabling disconnection, and also useful for implementing versioning and undo/redo features. The change logger implements an automated CRUD-level, operation-transfer logging capability to detect changes to JSON objects. In CANSYNC, the checkpoint history is internally recorded as a log of CRUD-level update operations over a base version of the application state. CANSYNC also has a customizable checkpoint logger based on the Dojo API that can record application defined operations. For brevity, we cover only the automated checkpoint logger in this paper.

```
{ log : [ { path : "customer.0.name", operation : "update", newvalue : "Samuel", oldvalue : "Sam"} ,
        { path : "customer.1.name", operation : "update", newvalue : "Sally M", oldvalue : "Sally"} ,
        { path : "customer.2.name", operation : "add", newvalue : "{customer:{name:"Joe",status:"new"}}"}
] }
```

Fig. 6. A JSON log entry.

Figure 6 shows the log items created by the automated checkpoint logger for JSON objects. In CANSYNC, the change logger provides a `diff()` function that attempts to detect the difference between the original and updated state. The `diff()` function will take two Javascript object hierarchies and perform a depth-first comparison of the objects. This is optimized for cases when the updates represent incremental changes to application data without schema changes, for example, when the application wishes to detect the difference between the entries in two versions of a form. In general, the scan is reliable if the topology between the original and updated objects are the same, or if additions of subtrees to the topology are always appended as the last child of the parent object; the current implementation relies on path information to uniquely identify objects and object properties. By only appending as the last child, i.e. the tail of an array, this ensures that path identifiers do not dynamically change. Although not currently part of the implementation, we will extend the `diff()` function to detect deletes. In object models like JSON, explicit delete of properties is not supported. One solution is to mark properties with a special deletion mark (known as tombstoning), which can then be detected and used to perform the delete using some other process. In addition to delete, we will also support a "REPLACE" operation,

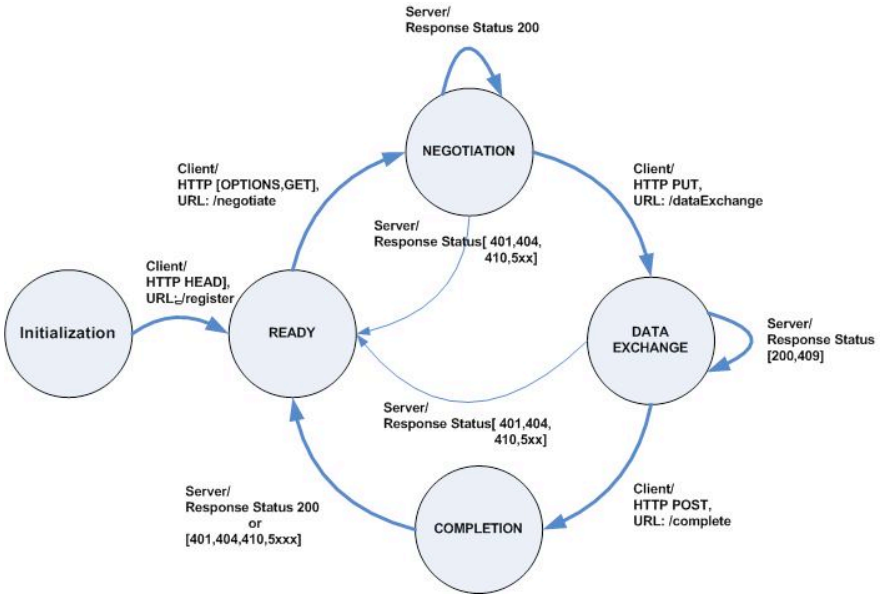

Fig.7. Finite State Machine for the Synchronization service

which represents a drastic change in a JSON object hierarchy such as a schema change, or an update to every property in the JSON hierarchy.

4.1.3 Checkpoint Persistence API

The timeline control can persist the checkpoint history using the persistence API. The storage API provides timeline abstractions for retrieving the checkpoint history, and is responsible for interpreting the updates in the log in order to restore the application state from a log of update operations.

The persistence API provides hash table semantics for retrieving checkpoints by keys, and also implements the cursor functions of the timeline control such as `getNext()` and `getPrevious()`. It also implements the `getWindow()` function.

When the timeline control is instantiated, the application can specify whether the checkpoint history is persisted locally, or on a remote checkpoint store. For the latter, CANSYNC provides a Javascript RESTStore object that communicates with server-based objects to persist the checkpoint history. As the name implies, access to the store is based on a REST API so the use of the RESTStore API is optional. Applications can choose to directly communicate with the store using HTTP.

Clearly, to support disconnection, the client may wish to store the checkpoint history locally. CANSYNC provides a LocalStore Javascript object. The interface to LocalStore is the same as the RESTStore but all persistence if done locally. The current implementation of LocalStore uses the SQLite database included with Google Gears with the expectation that future versions of Mozilla Firefox will embed SQLite as part of the base configuration of the browser. We will move to this version in future versions of the LocalStore.

4.2 CANSYNC Synchronization

In this section, we describe the data synchronization service in CANSYNC. It consists of a lightweight, general purpose sync engine providing client-side and server-side components that can be customized to implement application-specific synchronization policies. Like the checkpoint service, it provides applications with client-side APIs that enable the synchronization features.

4.2.1 Synchronization Engine

The sync engine extends the synchronization capabilities of the Dojo Offline toolkit by providing a richer set of synchronization options. The synchronization semantics of the engine can be represented as a Finite State Machine (FSM) that internally progresses from state to state via HTTP calls. In the following sections, we first describe the basic abstractions used by the synchronization engine, followed by a description of the client-side components and finally the corresponding server-side components.

The synchronization service borrows many high-level concepts from the Open Mobile Alliance Data Synchronization standard (OMA DS) [20], although it is not an implementation of that standard. The synchronization service takes advantage of some of the features of the HTTP protocol to simplify the engine implementation: it does not provide security features, but relies on an external implementation of the `basicAuth` protocol of HTTP for authentication and on HTTPS for encrypted connections.

4.2.2 Synchronization Finite State Machine

The client and server components of the sync engine support the synchronization of client-side replicated data with server-side source data according to agreed upon parameters. Multiple clients may share the server-side source data. The client-side replicas are subsets of the source data on the server.

A synchronization cycle between such pair of source data and a replica is implemented as an FSM. The FSM is driven by the HTTP messages exchanged between the client and the server. This process guarantees the timely completion of the synchronization cycle, once initiated.

Figure 7 shows the four main phases of a synchronization cycle: READY, NEGOTIATE, DATA EXCHANGE, and COMPLETE.

The cycle enters the READY phase after the client registers with the server. During registration, the client sends some initial parameters such as a unique user id and the version number of its client software. It may optionally also pass a max id size, max object size and an expected max timeout for the client session. The server performs initialization of the new client. A client registers itself only once with the server. If authentication is enabled, the client may be required to login. Once registration is complete, the client can initiate multiple synchronization cycles with the server for a specific piece of data.

The synchronization cycle enters the NEGOTIATION phase when the client sends an HTTP OPTIONS call to the URL suffix “/negotiate.” In the negotiation state, the

client identifies the entities to synchronize and the desired synchronization characteristics: namely the sync 'type', i.e. mono or bi-directional update transfers; the sync 'mode', either 'fast' or 'slow', and the sync 'effect', which may be set 'actual' or 'inform'. In a slow sync, the client and server may exchange their entire state – this can be used for the first synchronization or if the client and server cannot determine their last synchronization point. In a fast synchronization, the client and server exchange just their updates from the last synchronization.

If the sync effect is set to actual, then both client and server commit updates based on the application-specific reconciliation policies. The 'inform' sync effect allows the client to conscientiously diverge its data from that on the server. In such case, neither client nor server updates its data. Instead, the server informs the client of the potential consequences of synchronization. The client may use the information to display the data differences in the browser, for example.

The next phase in the synchronization cycle is the DATA EXCHANGE phase, which is started when the client issues an HTTP PUT to the URL suffix "/dataExchange." In this phase, updates from the client and /or server are propagated according to the synchronization parameters agreed upon. In a bi-directional sync, the client updates are first propagated to the server where the server invokes the application-specific update reconciliation policies for the updates. The server compiles individual status messages for each update and then sends them to the client. The server also sends its own updates to the client in the same response.

The finals phase in the cycle is the COMPLETION phase, which is reached when the client issues an HTTP POST to the URL suffix "/completion." This state is reached when all updates and corresponding status messages have been exchanged. The synchronization state machine returns to the ready state. Subsequent synchronization cycles of the client's replica and the server source data may be initiated with different sync parameters.

In the figure, there are additional transitions based on server responses to the client requests. Under normal operation, the server should respond with an HTTP status code that indicates server acknowledgment. These can be simple HTTP status 200 or 204 codes, or may be additional codes that indicate acknowledgment but also carry additional information. For example, the server may respond with status code 409 during the data exchange phase, which represents that a conflict has occurred. If the server responds with a failure status code, the sync cycle is aborted and the synchronization cycle returns to the ready state to restart the session. For example, failure transitions occur if the client and server somehow get out of step during a synchronization session. If the server is in the negotiate phase and the client tries to move to the complete phase, the sync cycle restarts with the server and client returning to the ready phase.

To keep track of updates, both the client and server rely on markers indicating when the client and server last synchronized successfully. The synchronization service lets the application specify the nature of these markers. In the reference implementation in CANSYNC (and used in Flinger), it is a millisecond resolution timestamp acting as a version number. During the negotiation phase of a synchronization cycle, the server compares its last marker to that sent by the client to determine the sync mode and identify how much data to exchange in order to be synchronized again.

4.2.2 Client-side Components

The CANSYNC architectural overview in Figure 3 shows the client-side components of the synchronization service. In the figure, the client components are the Sync Control Button, Sync Manager, Sync Session, Sync Context, and Change Provider.

The Sync Control Button is a widget that application developers can embed in their applications. The widget captures the most important parameters for managing data synchronization. The client can instantiate a sync control button with a Sync Context, which contains the parameters that specify that synchronization type, mode, effect, and other needed information. When the user clicks the synchronization button, it will call the client-side synchronization calls in the correct order, and return events informing the client about the current status of the synchronization cycle. Use of this widget is optional, and application developers can always choose to access the synchronization API directly via the Sync Manager and Sync Session components. Figure 4 shows example code using the component API.

The Sync Manager handles the client registration with server, and also manages the client's Sync Session objects. A sync session represents a resource pair and associated sync parameters. The client may have multiple Sync Sessions as long as the session's resource pairs do not overlap.

A Sync Session implements the core functionality of an individual synchronization cycle between the two given data resources. In the figure, the application instantiates a Sync Session object and passes as arguments the identifiers for the client and server resources, the type, mode and effect of the synchronization requested. In addition, the application must also pass its client-side Change Provider object instance and the name of its server-side Change Provider class for the Sync Session.

The synchronization engine requires the developer to implement a `ChangeProvider` interface on both the client and server. This interface allows developers to provide custom update reconciliation policies between client and server resources. CANSYNC provides a reference implementation of the `ChangeProvider` interface that assumes the data is JSON on both the client and the server-side. The reference implementation relies on the checkpoint logger to keep an update log on the browser.

Figure 8 shows the Java specification of the `ChangeProvider` interface. The `ChangeProvider` on the client is in Javascript, though it has an equivalent interface. For each update in a set of updates, the synchronization invokes `applyChange()` on the `ChangeProvider`. Here, the application developer implements the reconciliation policy for the application, and also connects the synchronization engine to the application's data store. The `ChangeProvider` also specifies the `informChange()`

```
public interface ChangeProvider {
      public JSONArray getChanges(SyncContext si);
      public JSONObject applyChange(SyncContext si, JSONbject update);
      public JSONObject informChange(SyncContext si, JSONbject update);
      public String generateNextAnchor(SyncContext si);
      public boolean hasChanged(SyncContext si);
      public boolean handleStatus(SyncContext si,Object Status);
}
```

Fig. 8. Change Provider Interface

method, where the application developer can simulate the application of the update but not actually commit any change to the underlying store.

Although not in the current implementation, we plan to have visual widgets that provide feedback to the user regarding the consistency state of the locally cached data based on the status returned by `informChange()`. For example, a table could display different background colors for individual cells where the color indicates if that cell has an update available on the server.

4.2.3 Server-side Components

The server side components implement the server responsibilities for the FSM. Like for the client-side, the application developer implements a `ChangeProvider` interface to describe application-specific reconciliation policies. This object is invoked by the engine on behalf of the application.

5 Related Work

The challenge of disconnecting an application is an instance of optimistic replication. Saito and Shapiro provide a recent and comprehensive survey of this area [23]. Coda [17] and Ficus [22] are examples of pioneering work done for optimistically replicating file systems. Bayou [24] is a much cited work on peer-to-peer replication. Bayou used epidemic protocols to propagate changes. Research in databases has looked at synchronizing relational stores, for example [7].

CANSYNC provides web-specific APIs for enabling checkpointing and synchronization applications in a browser. CANSNC is most related to browser disconnection frameworks like Google Gears [12], which provide basic building blocks for disconnecting browser based applications. CANSYNC differs from Google Gears by focusing on data services for replicated data in the browser that are not available in current frameworks. For example, Google Gears does not provide synchronization capabilities for developers. CANSYNC allows developers to utilize a generic synchronization engine they can customize for their applications. Google Gears also supports simple versioning of replicated components using its managed resource module; however, this is specifically for managing the download of application components and does not enable undo/redo for users or allow checkpointing of browser state.

Research in disconnecting web applications has focused on novel mechanisms to predict what components should be projected from the server to the client. Terry and Ramasubramanian [25] have looked at disconnecting web applications that communicate using web services protocols like SOAP. Chandra, et.al looks at issues disconnecting web services using mobile code [6]. CANSYNC is synergistic with this work by providing a systems-level perspective on how developers can incorporate these mechanisms into web applications using a lightweight framework.

CANSYNC extends our earlier work [5] that looked at MVC. The MVC design pattern is first described by Burbeck [3], and later in many papers regarding application design. The application of this pattern to web application design include [1][4]. Fluid Computing [2] also advocated an MVC approach to providing

replication services to mobile devices. Fluid Computing uses DOM-based data model as their unit of replication, and utilizes an epidemic protocol for change propagation.

Joyce is a novel system for enabling a rich history of undo/redo for desktop applications [19]. We are inspired by this work and hope to provide simple mechanisms that enable a general undo/redo model for browser-based applications. This is potentially useful in allowing different abstractions for utilizing the navigation features in a browser for a web application; for example, the back and forward buttons that are standard on all browsers can be used to navigate the checkpoint timeline in CANSYNC.

6 Discussion and Future Work

Data replication for web applications is used widely on the server-side to enhance availability and scalability. CANSYNC focuses on replica management for user-facing components of an application. CANSYNC looks at reducing the burden of using a data layer on the developer, minimizing the footprint for installation (CANSYNC is downloaded with an HTML page), and providing a customizable architecture to meet application needs. In previous work [5], we presented Ripple-X, a declarative approach to separate view from data, which was a precursor to the checkpointing service in CANSYNC. In this approach we created a data specific namespace using the XML Binding Language (XBL) feature in Mozilla Firefox. This allowed users to use a special datapage tag in their web page that acted as a container for non-visual data represented by XML fragments. The XML fragments could be created declaratively or programmatically and placed in the datapage using a Javascript API. As part of this work, we investigated model-based replication by automatically wrapping the non-visual XML fragments in a data dom object that provided the checkpointing service for that data. From the developer perspective, this allowed them to access additional checkpointing and persistence features on DOM elements.

CANSYNC provides a core engine for synchronization and a reference implementation based on synchronizing JSON-based data rather than XML, but does not directly address optimal policies for reconciliation. In general, optimizing reconciliation relies on exploiting knowledge about the data being synchronized. CANSYNC allows the developer to customize the synchronization engine by encoding this knowledge in the `ChangeProvider` interface. However, there is a long history of work in optimistic replication for areas like distributed file systems and distributed databases that could be applied to CANSYNC.

One possible direction for CANSYNC would be to couple a checkpoint history with the synchronization engine to allow checkpoint-based rollbacks to resolve update conflicts. In this approach, if the server detects a conflict, the server can send the client a conflict document that specifies the nature of the conflict. The client could then search its history of checkpoints to find the latest checkpoint that would not produce the conflict. The user would be presented with an option to resolve the conflict by systematically choosing a checkpoint that requires little or no modification to synchronize successfully with the server. This could also be done without the users

participation by finding a checkpoint that does not produce a conflict, and then merging that checkpoint with the latest checkpoint to produce a minimally changed version of the data that can be synchronized with the server. Obviously, automatically resolving conflicts can be disorienting to the user. Judicious application of automated policies plus proper feedback to the user through the UI may be the best approach.

CANSYNC (like Google Gears) advocates a framework approach, which allows applications to be retrofitted with features that enable state management features. For CANSYNC, this is required because application developers must specify application-specific reconciliation strategies for synchronization, and CANSYNC provides UI elements that encapsulate the CANSYNC service API. However, it is compelling to imagine a facility that would allow even existing applications to disconnect from the network with minimal changes. This mechanism would have to exist outside of the browser, and have the ability to mask any application from the status of the network connection.

In general, providing "disconnection-in-a-box" via a proxy is a complex problem given the open-ended nature of web application protocols. Constraining the problem to a subset of possible client-server interactions may simplify the problem. For example, there is interest in industry in making data on servers available as syndicated feeds such as RSS and ATOM. These syndicated feeds provide more structure to the messages between client and server and provide hints on cacheable items. Like distributed file systems, where the unit of replication is a file, the unit of replication in a syndicated feed could be a feed entry.

Security as an on-going issue for disconnected web-based applications. CANSYNC assumes simple authentication and authorization can be handled at the HTTP-layer. The Dojo Offline Toolkit allows users to encrypt data stored in the local database. However, none of these approaches address larger issues surrounding user mashups and replication of a users' data on a multi-tiered web application. We hope to look at these issues as part of future research.

Acknowledgements

The authors would like to thank John Ponzo and Apratim Purakayastha of IBM Research, Chris Mitchell of IBM Software Group, and Jerome White in the Computer Science Department, California Institute of Technology, for sage commentary, deft direction setting, and early prototypes of our research.

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. D. Bourges-Waldegg, Y. Duponchel, M. Graf, M. Moser: The Fluid Computing Middleware: Bringing Application Fluidity to the Mobile Internet. SAINT 2005: 54-63

3. 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
4. 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
5. P. Castro, F. Giraud, R. Konuru, J. Ponzo, J. White, “Before-Commit Client State Management Services for AJAX Applications”, IEEE HotWeb 2006
6. B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Nayati, A. Razzaq, A. Sewani, “Resource Management for Scalable Disconnected Access to Web Services”, Proceedings of the 10th International WWW Conference, WWW10, 2001.
7. J. Cho, H. Garcia-Molina, “Synchronizing a Database to Improve Freshness”. SIGMOD 2000.
8. Document Object Model, http://www.w3.org/DOM/
9. Dojo Toolkit, http://dojotoolkit.org
10. ECMA International Standard EMCA-262, “ECMAScript Language Specification,” 3rd edition, http://www.ecma-international.org/publications/standards/Ecma-262.htm.
11. J. Garrett, “AJAX: A New Approach to Web Applications,” HYPERLINK "http://adaptivepath.com/publications/essays/archives/000385.php"http://adaptivepath.com/publications/essays/archives/000385.php
12. Google Gears, http://gears.google.com
13. Google Maps, http://maps.google.com
14. Z. Jiang, L. Kleinrock, “An adaptive network prefetch scheme”, IEEE Journal on Selected Areas in Communications , April 1998, 6(3): 358-368
15. A. Kerrmarrec, A. Rowstron, M. Shapiro, P. Druschel, “The IceCube approach to the reconcilliation of divergent replicas”, Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC), 2001.
16. G. Kuenning, P. Reiher, and G. Popek, "Experience with an Automated Hoarding System," Personal Technologies, September 1997
17. J. Kistler, M. Satyanarayanan, “Disconnected Operation in the Coda File System,” ACM Transactions on Computer Systems, 10(1):3-25, February 1992.
18. B. Neuberg, Dojo Offline, http://dojotoolkit.org/offline"http://dojotoolkit.org/offline
19. J. O'Brian, M. Shapiro, “Undo for anyone, anywhere, anytime”, Proceedings of the 11th workshop on ACM SIGOPS European Workshop 2004
20. Open Mobile Alliance Data Synchronization (SyncML), http://www.openmobilealliance.org/release_program/ds_v112.html
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. P. Reiher, J. Heidemann, D. Ratner, G. Skinner, G.. Popek, "Resolving File Conflicts in the Ficus File System," USENIX Conference Proceedings, pages 183-195. USENIX, June 1994
23. T. Saito, M. Shapiro, “Optimistic replication”, ACM Computing Surveys, 37(1), 2005.
24. 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.
25. D. Terry, V. Ramasubramanian, “Caching XML Web Services for Mobility”, ACM Queue, May 2003, pp.70-78
26. V. Vukicevic, “Mozilla2 Unified Storage, http://wiki.mozilla.org/Mozilla2:Unified_Storage:
27. XBL, Extensible Binding Language, http://www.mozilla.org/projects/xbl/xbl.html