Automated Object Persistence for JavaScript

Brett Cannon
University of British Columbia
201-2366 Main Mall
Vancouver BC V6T 1Z4
drifty@cs.ubc.ca

Eric Wohlstadter
University of British Columbia
201-2366 Main Mall
Vancouver BC V6T 1Z4
wohlstad@cs.ubc.ca

ABSTRACT

Traditionally web applications have required an internet connection in order to work with data. Browsers have lacked any mechanisms to allow web applications to operate offline with a set of data to provide constant access to applications. Recently, through browser plug-ins such as Google Gears, browsers have gained the ability to persist data for offline use. However, until now it's been difficult for a web developer using these plug-ins to manage persisting data both locally for offline use and in the internet cloud due to: synchronization requirements, managing throughput and latency to the cloud, and making it work within the confines of a standards-compliant web browser. Historically in non-browser environments, programming language environments have offered automated object persistence to shield the developer from these complexities. In our research we have created a framework which introduces automated persistence of data objects for JavaScript utilizing the internet. Unlike traditional object persistence solutions, ours relies only on existing or forthcoming internet standards and does not rely upon specific runtime mechanisms such as OS or interpreter/compiler support. A new design was required in order to be suitable to the internet's unique characteristics of varying connection quality and a browser's specific restrictions. We validate our approach using benchmarks which show that our framework can handle thousands of data objects automatically, reducing the amount of work needed by developers to support offline Web applications.

Categories and Subject Descriptors

D.1.5 [**Software**]: Programming Techniques—*Object-oriented Techniques*; D.3.2 [**Software**]: Programming Languages—*JavaScript*; E.2 [**Data**]: Data Storage Representations—*Object Representation*

General Terms

Design

Keywords

JavaScript, JSON, HTML5, Web Storage, object persistence

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

1. INTRODUCTION

Offline Web applications are a new class of application on the Web which can support page browsing and execution of scripts when disconnected from the Internet. Some popular examples include Gmail, WordPress, MySpace, and AutoDesk Online. The technical considerations needed to provide such offline use are currently being standardized as part of the new HTML5 standard [16].

Although most browsers have historically had the capability to cache some components of a Web page [9], such as HTML and media resources, this did not include the intermediate results of computations performed by script logic. As script logic has become a more integral part of application functionality, this deficiency has made the deployment of complex, feature-rich, Web applications difficult.

Offline Web applications allow for data created and managed by script logic to be stored locally in the browser. This allows Web application use to be suspended and resumed offline, and also for offline data to be synchronized on the Web at some time in the future. However, unlike the automatic caching of page components by Web browsers, the persistence of intermediate script computations unfortunately requires careful consideration and manual intervention by script programmers.

Historically in non-browser environments, distributed programming language environments [4, 20] have offered automated, object persistence to shield the developer from these complexities. In this research we investigate the feasibility of automating persistence of data objects for JavaScript. Developers can work with objects as they normally do, by getting/setting properties and calling methods, while having persistence of objects managed for them.

Through this research we have identified and tackled several requirements and challenges of object persistence which are unique in the specific case of support for Web browser embedded JavaScript objects. These challenges and requirements can be described roughly in three categories.

1. Interoperability
   The common Web browser based JavaScript execution environment is standardized through a set of standards such as ECMAScript 5 and HTML. While ECMAScript defines the core JavaScript language semantics, other standards such as HTML4 and the emerging HTML5, define sets of APIs that are available to Web application authors. As part of the requirements for transparent JavaScript object persistence, we decided to investigate what techniques were possible while still conforming to standards. This is challenging since

it means we would not have the capability to make changes to the JavaScript interpreter layer itself, but must make automated object persistence work entirely at the application level. This decision also induces our focus on the next two specific technical challenges.

2. Extensibility facilities in JavaScript
   Compared to many other languages, JavaScript has only very primitive support for extensibility. JavaScript is also a dynamic programming language where introducing a compilation step would be a shift in development for a typical JavaScript developer. This has led us to use a different approach from previous work which typically relied on compiler modifications [6, 8, 17]. We leverage the use of JavaScript *property accessors*, an extensibility mechanism being introduced in ECMAScript 5, but also describe how we have needed to work around some lack of support for extensibility in JavaScript.

3. JavaScript's concurrency model
   Also different from most modern programming languages is JavaScript's model and support for concurrency. Unlike the thread model used by most languages, JavaScript is purely event-based. This has a direct consequence for our purposes: it is not possible for a separate thread of control to be used for managing persistence. Operations which manage persistence must either: suspend application progress or delay until the application is idle. These two options both offer different tradeoffs which we have needed to investigate in our research. We show how application level maintenance tasks implemented by timed event-handlers can efficiently maintain a persistent heap.

We validate our approach using benchmarks which show that our framework can handle thousands of data objects automatically, to reduce the amount of work needed by developers to support offline Web applications.

The remainder of the paper is structured as follows: in Section 2 we describe a motivating example application, in Section 3 we present an overview of our work, in Section 4 we present technical details of how to detect mutations of JavaScript objects, in Section 5 persisting objects locally is discussed, in Section 6 the server communication is covered, in Section 7 we present an evaluation, in Section 8 we present related work and we conclude in Section 9.

2. MOTIVATING PROBLEM

Consider a scenario where a developer would like to deploy an offline Web application to support a Web-based e-mail client. Without offline support, a user would not be able to access their inbox while disconnected from the Internet. Furthermore, a user would not be able to compose new e-mail messages while disconnected. This is because such modern Web applications make use of JavaScript for the management of data items such as individual e-mails. This practice is common on today's Web as one of the central techniques of the Ajax application paradigm.

Persistence of data for offline use by JavaScript is now being widely deployed in all major browsers. Some support is provided for both simple persistent hash maps [14] and also for SQL-style database APIs [13]. However, none of the existing mechanisms directly support either object structured data or automatic persistence. This means that developers still must manage several difficult problems by themselves.

First, developers must determine how and when to save data from heap-allocated JavaScript objects to local browser storage. This must be done carefully to balance several tradeoffs. If changes are saved too frequently, the overhead of each save action could degrade performance. Also, since JavaScript is single threaded, writing a single large batch of changes will suspend the application from servicing any event-handlers: such as those which respond to user actions or timers which control activities such as page animations. For example, on the one hand, if every keystroke while composing a new e-mail triggered an expensive saving action it could have visible delays in the user experience. However, on the other hand, leaving changes unsaved for an extended period increases the chances of data loss; no one wants that email draft to be lost because the browser crashed.

To deal with this problem, we have investigated the use of *persistence by reachability* [17] for JavaScript. In this approach, the object heap is divided into transient and persistent sub-graphs. Our runtime framework consists of several time-scheduled event handlers which process a work list of persistence-related tasks periodically. In this way, a developer could use JavaScript data objects to build a persistent data structure representing a user's email folder. When a new e-mail object is added to the data structure, our framework would ensure the new e-mail is automatically saved to local storage in such a way as to not degrade application response when dealing with thousands of objects.

Second, a developer must determine how and when to synchronize offline data stored in a browser with a server copy of the data. This is difficult because copies of data may become inconsistent if they are made available to more than one browser session. For example, an e-mail draft might be copied to two different browsers that are used by the same user. If the user edits one of the copies offline, it may become inconsistent.

To deal with this problem, we have investigated and developed an application framework for automated persistence at an individual object granularity. This allows the browser to communicate updates of data for individual objects which map directly to the object model created by the application developer. Conflicts between object versions are detected automatically which triggers a conflict resolution callback. The developer simply needs to implement the callback to apply any semantically relevant resolution strategy. For example, an e-mail draft composed while offline would automatically be synchronized to a server when an internet connection is re-established. Also if that same e-mail draft was in conflict with other edits made while the browser was offline the user could be notified of the conflict in order to resolve it.

While previous work on object-oriented databases and distributed object languages have similar motivation, we have had to deal with different technical issues that are unique to the browser embedded JavaScript programming environment. We had to provide specific mechanisms to work around JavaScript's lack of threading, primitive support for language extensibility, and make sure that support does not require the use of any features that are not part of browser standards.

3. OVERVIEW

To solve problems similar to the one presented in Section 2 we have implemented an automated object persistence framework for JavaScript. The architecture of our framework is shown in Figure 1. Our approach involves detecting mutations of persistent objects, serializing the objects to local storage, and then synchronizing the serialized form of the persisted objects to a server on the internet.

We use two approaches to detecting mutations of persistent objects in the application (Section 4). First, we have a maintenance task that detects *dynamic property additions* which periodically scans all of the persisted objects under use during the current execution of the browser for changes to the objects (Section 4.3). The set of persisted objects under use is called the *live object set*. We also use *accessors* that we have attached to persisted objects which detect mutations at the time of assignment (Section 5). Two separate approaches for detecting mutations is required to work around shortcomings in JavaScript (Section 4.2).

The detection of a mutation leads to two actions occurring locally in the browser (Section 5). The first is that the mutated object is serialized into a format called *JSON* (JavaScript Object Notation) which is subsequently put into *local storage*. This makes sure that if the browser crashed the state of the persisted object is stored for later retrieval. The second action after detecting mutation is to add any new objects to the live object set.

Periodically all mutations of objects are pushed to a server on the internet through a remote synchronization task (Section 6). The browser sends the new serialized form of any persistent objects that have mutated since the last time the browser and server communicated. The server responds with any updates to objects that the browser is lacking. Optionally, the developer can assign objects to specific groups to improve locality.

Normal reading and writing of persisted objects is allowed while offline so that the web application can continue to work as normal. Upon reconnection with the server any conflicts created while offline are detected by the server to the browser so the web application has an opportunity to resolve the conflict.

4. DETECTING MUTATIONS

4.1 Background

In any application, data naturally falls into two classes: *transient* and *persistent*. Transient data is applicable only for the lifetime of a session, whereas persistent data must survive across individual application sessions. In object-oriented languages, object persistence is a technique for automatically binding object instances to secondary storage, making them persistent. As mentioned previously, this allows developers to work with objects as they normally do, by getting/setting properties and calling methods, while having persistence of certain objects managed for them.

Even when using an object persistence framework, developers still must apply their knowledge of application semantics to classify objects as transient or persistent. *Persistence by reachability* has often been advocated as providing the simplest solution for classification. In this approach, the object heap is divided into transient and persistent sub-graphs. A persistent "root" object is made available to developers and any object that can be reached transitively from this root is considered to be persistent [17]. In our approach, we automatically make a special root object available, `persist.root`, that is attached to the global `window` object in JavaScript. To support this notion of persistence, we will need to have some way of monitoring the object heap to know when objects are made reachable or unreachable.

4.2 Detecting Reachability in JavaScript

If we were able to modify a JavaScript interpreter directly, this problem would be straightforward. Even still, there is some support in JavaScript to provide a subset of what is required. In JavaScript, special functions called *accessors* (see Figure 1) can be used which are called transparently when a specific object property is read or written. One can *bind* individual accessors to a property for assignment/writing and a separate accessor for accessing/reading. In our framework, when a write accessor is triggered it records that the object was mutated and stores the written value so it can be returned by our read accessor. By masking the function call behind standard property manipulation syntax the object's interface remains consistent while its behaviour changes to that of a function call.

Accessors allow us to detect some mutations implicitly from the fact that an assignment has taken place. Still there are two more related problems to consider. First, saving the new state of the application when a mutation is detected could take some time. Since JavaScript is not multi-threaded, this process will block any other application functions, so we consider this problem carefully in Section 5. Second, JavaScript is a dynamic programming language, this means we must consider not only changes to existing object properties but also the dynamic addition of properties.

Unlike languages with static object schemas (e.g. Java or C++), properties in JavaScript can be added to an object at any time. Although some dynamic languages provide support for intercepting the event of a new property being added to an object (e.g. Python), JavaScript does not support this feature.

To deal with this problem, we make use of a scheduled *dynamic property addition task* (see Figure 1) which iterates over specific groups of persistent objects to search for new properties, known as *stabilization* [17]. This task is a JavaScript event handler running at a developer-specified interval. JavaScript supports tasks to be run at timed intervals through the use of the standard function `window.setTimeout`. This process must be done efficiently so that it does not block the main application from executing its own event handlers.

Deletion of properties is handled by the assignment of `undefined` to a property that has our write accessor set. A transient property that is created and then deleted before the detection of the new properties is of no consequence to us.

4.3 Detecting Dynamic Property Additions

When a persistent object is loaded, it is added to the "live" object set (see Figure 1). This set is periodically iterated over and each object is processed by the function `detectNewProperties` shown in Figure 2. This function iterates over each property in an object to see if it already has accessors bound to it, initially assuming that there are no new properties (reflected through the `foundNew` variable).

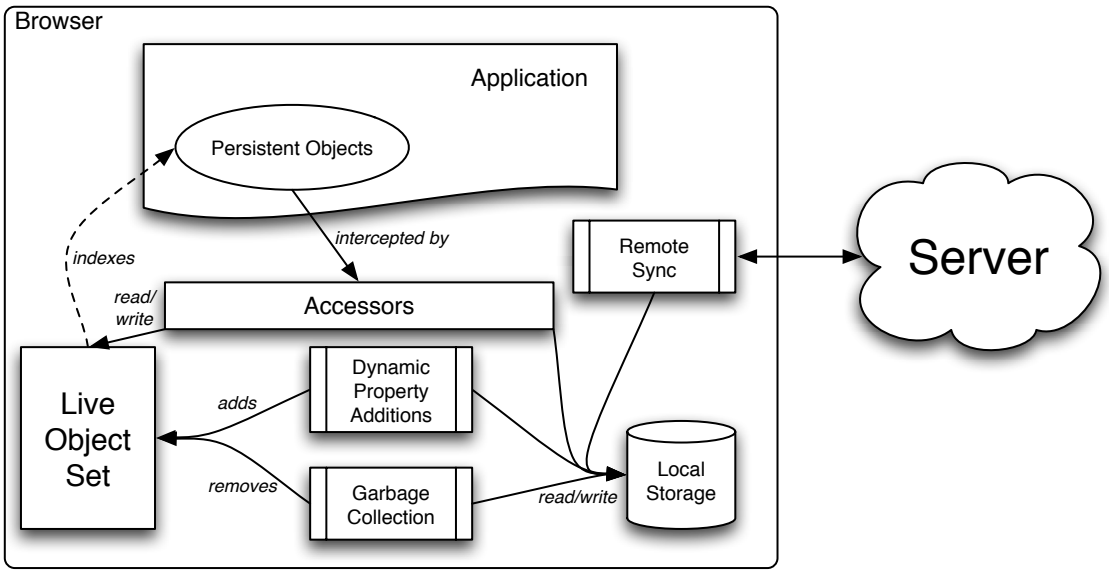

Figure 1: Architecture diagram. *Accessors* act as a read barrier to alert us to when a persistent object has mutated. Mutations (both accessors and the *dynamic property addition* maintenance task) cause objects to be added to the *live object set* and have their state stored in *local storage*. *Garbage collection* removes dead objects from local storage. Finally, mutations are sent to a server during *remote synchronization*.

```
function detectNewProperties(object) {
  var foundNew = false;
  for(property in object) {
    if (isNewProperty(object, property)) {
      foundNew = true;
      if (object[property] instanceof Object) {
        var newObject = object[property];
        registerObject(newObject);
        persistObject(newObject);
        log(newObject, "created");
        detectNewProperties(newObject);
      }
      addAccessor(object, property);
    }
  }
  if(foundNew) {
    persistObject(object);
    log(object, "updated");
  }
}
```

Figure 2: Pseudocode for detecting new properties and persisting any new parts of the object graph. Described in Section 4.3.

For each property on the object being processed, we check to see if the property is new. If the property is new according to our framework, we mark the object being processed as mutated by assigning `true` to `foundNew`. If the new property points to an object itself we consider it a new object that we must persist as we reached it through another persistent object. The new object must be registered with our framework, persisted for safe keeping, logged as new for eventual sending to a server on the internet, and then itself checked for other new objects.

During the window of time when a set of objects is retrieved and the maintenance task of searching for new properties executes, all persisted objects must be considered unstable (i.e. data could be lost in a crash) as we can be unaware of new properties that have occurred on persisted objects [17]. Once we have performed our iteration, though, the objects are guaranteed stable as we will have checked all objects which have been loaded and thus checked all objects that could have been mutated. JavaScript's concurrency model guarantees no objects are mutated while we iterate over the set.

We restrict this object graph traversal to only those objects that have been loaded during the current execution of the browser as those are the only objects that could have been mutated. We also allow the stabilization task to be paused after a certain amount of time has elapsed to prevent the locking up of the user interface. By setting a small interval between iterations over the live object set to stabilize it, we can minimize the window where data mutation could be lost by a browser crash. This needs to be balanced against the cost of the maintenance task, so we explore this issue in the evaluation (Section 7).

5. PERSISTING DATA LOCALLY

Stabilization from searching for new properties along with mutations detected by accessors gives us the knowledge of what needs to be persisted locally. To do so we need both a persistent data store in the web browser as well as a serialization format for JavaScript objects.

For storing objects, we use the persistent key/value map called `localStorage` (referred to as "local storage" in Figure 1) that is being standardized as the W3C Web Storage standard [14]. Saving an object to storage may also require that any objects referenced by the object are saved. This is currently not handled by any existing JavaScript standard. So, when an object is saved to storage, a traversal of the object is performed to flatten the transitive graph of object references into a set of objects. The traversal transforms each object property that holds an internal JavaScript reference to an external globally unique identifier [4, 5, 20]. These GUIDs are then used to insert a serialized version of each object into the map.

For serialization we rely on the standard JavaScript Object Notation (JSON) to represent each individual object after the graph has been flattened. This promotes interoperability with any components on the synchronization server, discussed in Section 6. Note that while there is already standard browser support for serializing JavaScript objects using a JSON API, this API does not support cycles in an object graph; only trees are supported. Additionally, a serialized JSON object tree is represented by a single string, which does not allow for the indexing, retrieval, and update at the granularity of individual objects. Using GUID references we can isolate objects in the object graph.

A side-effect of introducing the concept of GUIDs as references is it allows for the lazy loading of persistent objects from local storage. When a property of an object is accessed that points to a persisted object, the live object set is checked for the object, and if found is returned. However, if the object is not live, the GUID is used to load the object from local storage and has its properties bound to accessors before caching the object in the live object set and returning the object.

The consequence of this strategy is that if a persisted object is never used during an execution of the browser it is not loaded from storage, saving the cost of loading it from disk and deserializing it. This allows our accessors to act as a read barrier to make sure that an object gets loaded from whatever storage location is the fastest (memory, local, or remote). It also speeds up start-up costs by avoiding having to wait for all persistent objects to be loaded from local storage before work can begin.

5.1 Garbage Collection

Ideally when a persisted object is deleted or made to be no longer reachable, we would like receive immediate notification of property deletions. In some languages, it is possible to use a deletion accessor or tie into the garbage collector and denote the fact that an object has been deleted [17]. Unfortunately JavaScript provides no such mechanisms to be notified when an object is deleted.

In order to know when a persisted object is no longer reachable from `persist.root` we implemented our own *persistent garbage* collector task at the application level (see Figure 1). This is reasonable, because this garbage collector is used to remove dead objects from secondary storage, not from main memory, so it does not need to execute frequently and incur its overhead cost. While also clearing out dead objects from local storage, our garbage collector simultaneously removes dead objects from the live object set to prevent including them during stabilization.

```
function gc() {
  intiallyLive = clone(live);
  tempLive = {};
  worklist = [persist.root];
  for(object in worklist) {
    if(id(object) in tempLive) {
      continue;
    }
    tempLive.add(id(object));
    for(property in object) {
      subObject = object[property];
      if (subObject instanceof Object))
        lookAt.append(subObject);
    }
  }
  live = initially_live;
  for(key in guidsInStorage()) {
    if(!tempLive[key]) {
      localStorage.removeItem(key);
      delete(live[key]);
      log(key, "deleted");
    }
  }
}
```

Figure 3: Pseudocode for our garbage collection step. Described in Section 5.1.

We use a mark-and-sweep algorithm to implement our garbage collector as shown in Figure 3. We initially clone the live object set to know what it was set to before we begin walking the object graph. We construct a work list starting at `persist.root` so we can perform a breadth-first search of the object graph. Any objects that are reached during the traversal are considered live and added to the `tempLive` set. Once we have finished our traversal we set the live object set back to what it was before we began the traversal. While having to temporarily load all persistent objects defeats lazy loading, the fact that garbage collection is not expected to be run until late into the execution of the web application allows the benefit of lazy loading to manifest as increased startup responsiveness.

Now knowing what objects are reachable from `persist.root`, we go through each known persistent object in local storage to see if it was marked in our traversal as stored in `tempLive`. If an object that exists in local storage but was not marked is discovered, the object is removed from local storage, removed from the live object set, and then logged as deleted.

6. PERSISTING DATA REMOTELY

When dealing with a distributed system that allows for offline use, two approaches can be used: optimistic or pessimistic updating of the data while offline [19]. In an optimistic updating scheme you allow for offline mutation of data, with the hope that when it comes time to synchronize

the data you will most likely be able to without conflicts. A pessimistic scheme is the reverse of optimistic updating: assuming that reconnection will lead to conflicts with changes made while offline, you do not allow a user to make any mutations while offline (reading is of course allowed).

To support the user in doing work offline, we need to use an optimistic approach. For those instances where the server automatically detects a conflict during an update (discussed in Section 6.2.1), we follow the tradition of letting the application handle how to semantically resolve conflicts [10]. This allows developers to create web applications where data is shared; either structure the objects such that only a single user edits any one object at a time or develop their own approach to resolving conflicts while synchronizing persistent objects with the server, when they know it is going to be a recurring issue.

Supporting persistent updates at the level of individual objects is difficult in a browser setting because it must be done at the application level for interoperability, however as described below, this is necessary to support efficient pushing and pulling of fine-grained updates for synchronizing with a server.

6.1 Browser-Side Logging

For each mutated object its GUID, what triggered the mutation (creation, update, or deletion of an object), the JSON stored in local storage for the object, and the object's group labels (discussed in Section 6.2.2) are sent. Each log entry is stored individually in local storage so that the log can be reconstructed if the browser is closed before log is sent to the server. We chose to update the server in a batch – which requires a log – instead of after each individual mutation, to prevent having many mutations in a short period of time trigging a large number of network connections being created.

6.2 Synchronization With A Server

While the client-side log lets us know what needs to be pushed to the server, it does not tell us what needs to be pulled from the server. In the case of updating the local copy of the persisted object graph there are two situations. One is where the local copy is "cold"; either the browser is loading a web application for the first time or has been disconnected for quite some time. The other situation is when the local copy of the object graph is "hot"; the difference between what the browser has and what the server has is minimal.

6.2.1 "Hot" Synchronization

To start we will address "hot" synchronization. If the churn rate on data mutation is low then it is not hard for the browser to stay in sync with the server. When the browser is synchronized to a specific point in time with the server we say the browser is a "snapshot" of the server.

When it comes time to sync with the server (see Figure 1), the browser sends the log of mutated objects (Section 6.1) along with the timestamp for the last snapshot. The browser leaves the connection open, waiting for a response from the server.

On the server's end, each mutated object is to be stored if no conflicts exist. For each object, the server stores at least its GUID, JSON representation, timestamp of the last mutation, and what groups it belongs to (groups are discussed in Section 6.2.2).

As the server processes the log sent by the browser, each object is checked to see if there is a conflict with the proposed update from the browser. For newly created objects, a conflict occurs if the object already existed. For updated objects, a conflict occurs if the timestamp on the object is newer than the snapshot timestamp the browser sent with the log. A deletion of an object is in conflict if the object has mutated since the last sync [1].

Preferring the version of an object in the server over the browser provides master copy semantics [2, 3]. If there is no conflict the server's version of the object is updated based on the values sent over in the log from the browser.

As a return value sent to browser, the server sends all objects (and their respective details) whose timestamp is newer than the browser's snapshot timestamp. An array of all objects' GUIDs who were found in conflict are also sent along.

Once the response is received by the browser for the log it sent over, the browser processes the data. First, each object that the server sent over as mutated is processed. All updates are stored to `localStorage` immediately. For each updated object another callback is optionally called so the application is aware of any changes that have occurred. If the object is in the live object set, it is updated in-place automatically.

Next, all found conflicts are processed. For all conflicts, a callback is called which passes in the state of the object as found on the server and the conflicted state the browser originally sent to the server. This gives the web application a chance to resolve conflicts in an application-specific way [10].

Synchronizing with the server occurs at regular intervals through asynchronous XMLHttpRequest calls. By synchronizing regularly and on short intervals the local copy of the object graph can be kept in sync to the server with minimal skew between the two. But as our framework supports offline use of objects, it is entirely possible that the snapshot stored by the browser becomes far out of sync with the master copy the server contains based on a developer-specific amount of time passing. This "cold" browser scenario is especially acute in a browser that has never accessed the web application before.

6.2.2 "Cold" Synchronization

If the browser is "cold" and still downloading the initial set of persistent objects, the developer has the option of using a callback mechanism similar to the standard XMLHttpRequest API to ensure that all objects that an event handler requires are loaded into memory before executing. This prevents the handler from blocking when an object it depends on has not yet been fetched. In the callback request, developers specify a list of *object groups*, and the callback is called when all the objects in the specified groups have been retrieved. An object group is simply a numeric tag to serve as indexes for efficient retrieval and to scope sets of objects. An object is added to a group by calling a function with the object and the group to add it to: `assignGroup(object, groupId)`.

The developer is shielded from the details of where the objects in the specified groups are currently located. This prevents the browser from becoming unresponsive while the initial set of persistent objects is downloaded if meaningful work can be done with only a subset of objects.

	Objects / Properties per object							
Experiment	10/10	10/20	100/10	100/20	1000/10	1000/20	10,000/10	10,000/20
Control	10,273.6	5458.6	1303.4	611.03	136.23	61.24	11.78	4.86
Framework	2262.4	1795.8	265.14	202.32	25.32	18.8	2.12	1.52

Table 1: Comparison of serializing an object tree w/ JSON to our framework. Measurements are operations per second. Shown for a varying number of objects, and number of properties per object.

The framework is also continually fetching entire object groups, one at a time, in order to end up in a "hot" synchronized state. If the browser tries to read an object that has not been fetched yet without using the callback mechanism, it is retrieved on-demand in a blocking fashion from the server, which also returns the entire object group the requested object belongs in. The fetching of object groups automatically in the background is done in such a way that the main browser thread has a chance to schedule other events between object group requests. This allows for improved interactivity and perceived application startup time by not blocking the browser until all persistent objects are received from the server.

7. EVALUATION

7.1 Benchmarks

Here we demonstrate the feasibility of our prototype framework on several micro-benchmarks and a use-case based on eBay Web service data. Our focus is to determine the scalability of application-level persistence under a variety of object configurations. We focus only on measurements of the JavaScript client-side portion of the framework since this is the contribution of the research. We have implemented a server-side object repository in Python for use with our prototype, but its performance characteristics are not a salient part of this work as no novel contribution is dependent on server performance.

All reported benchmark numbers were taken using the Dromaeo JavaScript performance testing suite [23]. Each benchmark is executed as many times as possible until the total elapsed time surpasses one second, leading to a unit of "operations/second". This allows for extremely fast benchmarks to be accurately measured as the accuracy of the clock in browsers can be as coarse as 15 ms; executing for at least one second makes a 15 ms skew lead to at worse-case 1.5% mis-report of results. Each benchmark is measured five times and the average is reported.

The machine used to take the benchmark measurements is an Apple MacBook laptop running a 2.2 GHz Intel Core 2 Duo processor with 4 GB of RAM. The operating system is OS X. We report numbers from tests on the Apple Safari 4 browser.

We also tested our prototype on Mozilla Firefox successfully, however the numbers were far worse. For example, just in the case of JSON serialization Firefox 3.6b1 is nearly twice as slow as Safari. As for simple object property access Firefox is over three times slower and when accessors are involved is over six times slower. With both pieces of technology essential to our framework all of our benchmark numbers were subsequently nearly two times slower or more on Firefox.

At the time of submission of this paper, the Chrome and Opera browsers do not support the standard Web storage API (although Chrome subsequently added support by final publication). Internet Explorer 8 supports W3C Web storage but it does not yet support ECMAScript 5 accessors for JavaScript objects. Thus, our evaluation reflects what is possible working within proposed emerging standards, but not already widely deployed standards. Clearly, offline Web applications are a new area and browser vendors are still working towards finalizing their implementations.

7.1.1 Micro Benchmarks

For our first benchmark we measured the cost of persisting objects (Table 1). The control test takes the object described in each column, serializes it to JSON, and then stores it to local storage. Our framework test takes the reference object and persists all objects reachable from it. This illustrates the up front cost of an entire new object sub-graph becoming reachable instantaneously, by constructing the graph as transient objects and then attaching the root of the sub-graph to the persistent root.

Analyzing the results of the first benchmark shows our approach taking over 4.5x longer to just over 3x longer. While the use of the framework clearly adds overhead, this would most likely not be noticeable by a user until the sub-graph was over 1000 objects and thus take longer than 150ms [7]. Developers using our framework would need to take care when attaching entirely new sub-graphs of much larger sizes, but that scenario seems unlikely, as described in Section 7.1.2.

	Objects / Properties per object			
Experiment	1000/10	1000/20	10,000/10	10,000/20
Stabilizing	160.62	71.34	14.4	6.67

Table 2: Stabilization measurements. Measurements are operations per second. Shown for a varying number of objects, and number of properties per object.

The second benchmark measured the overhead of stabilization (Table 2). This table shows the overhead of simply detecting that no new properties were added to the object graph. This is the minimum cost paid for every stabilization of the object graph assuming no objects had new properties added. As the data shows we could check 10,000 objects with roughly 15 properties each in 100 ms.

The final micro benchmark measured the overhead from the garbage collection step (Table 3). All benchmarks include the creation of the test objects and the stabilization of the object graph. Each row in the figure represents whether all persisted objects were cached in memory (hot) or not (cold) and whether all or no persisted objects were actually collected. This means in the "Live/All" row that af-

Experiments	Objects / Properties per object			
Object set / How much collected	1000 / 10	1000 / 20	10,000 / 10	10,000 / 20
Live / All	5.49	3.3	0.5	0.3
Cold / All	5.55	3.22	0.48	0.3
Live / Nothing	5.65	3.18	0.5	0.28
Cold / Nothing	3.66	1.99	0.32	0.18

Table 3: Garbage collection measurements. Unit of measurement is operations per second. Shown for a varying number of objects, and number of properties per object.

ter stabilization occurred the test object was deleted from `persist.root`. Compared to the "New" row in Table 2, we also measured the cost of clearing out local storage and the live object set of all objects.

For "Cold/All" we cleared out the live object set, reloaded `persist.root` and deleted the property the test object was assigned to. The added cost compared to Table 2 is similar to the "Live/All" measurements, except the live object set had nothing removed.

The "Live/Nothing" measurement, after stabilization, simply triggered the garbage collector without any deletions. This means the benchmark measures the cost of traversing the entire object graph to discover that no objects need to be removed from local storage or the live object set.

Finally, the "Cold/Nothing" measurement involves clearing the live object set before traversing the object graph to discover that no objects are to be deleted. The overhead here is that while nothing is deleted, all objects need to be revived from local storage in order to traverse the graph to discover nothing needs removal.

Looking at the numbers for all garbage collector scenarios in Table 3 compared to the "New" row in Table 2 you will notice that only the "Cold/Nothing" scenario incurs any great overhead. This makes sense as the other scenarios either do not have to load objects or already have them loaded to traverse. But in the "Cold/Nothing" scenario the entire object graph needs to be brought out of local storage for examination.

Luckily garbage collection should be a rare occurrence as it should only be run after the web browser has been executing for some time and thus has at least some portion of the object graph live. A task such as collecting garbage *on disk* can be triggered by the user synchronously, through a developer provided GUI button, such as "Empty Trash". For this reason, the overhead shown in these measurements looks reasonable.

7.1.2 eBay Benchmark

Our micro-benchmarks were scaled to specifically demonstrate the limits of our approach. For a quantitative evaluation of our framework using real-world data we used eBay's web service. We queried the service using the search term "iPod" and requested 100 auctions in JSON format. We stored the JSON as a literal in our test code. The control experiment converted the JSON to an object representation, deserialized that object, saved the JSON representation to `localStorage`, and then converted the object to XML (to exercise accessing every property on every object). For benchmarking our framework we deserialized the JSON, attached the object to `persist.root`, stabilized the object graph, and then converted `persist.root` to XML.

The benchmark measures the overhead of our approach compared to a very simple approach to storing an object tree. By fully stabilizing the object graph for our framework we not only measure the cost of persisting the data but also the cost of adding accessors for all properties. And by converting the object to XML we make sure to visit every object and property, thoroughly exercising our accessors.

The results of the benchmark are that our framework can operate 15.8 times/second versus 58.51 times/second for the control. While there is an obvious overhead incurred by our framework, the cost contains several features that are lacking in our simple control benchmark. One, our framework supports cyclic object graphs while native JSON only supports an object tree. Two, if a property were to be mutated we would know about it immediately, minimizing potential data loss from a crash. Three, we also log all data for synchronization to a server for remote backup.

Although there is overhead, an operation that can be performed 15.8 times/second is not going to be noticed by the user [7]. This shows that our framework would be efficient for scenarios such as querying a web service and persisting the results of the query.

7.2 Experience

In addition to quantitative measurements, we wanted to gain some experience using this approach to better understand its use in an application. For this purpose, we both ported an existing Web application to use the persistence provided by the framework, as well as comparing it to an existing JavaScript toolkit that provides some support for persistence.

7.2.1 Moi Web Application

To initially make sure that our framework not only worked, but understand how it could be used effectively, we ported a Web application of our own design, called *Moi*, to our framework. The application was designed to allow the authors to report to each other what the other had done during the prior week using a simple web page form to enter data and an Atom feed to consume the data. The application supported offline use so that updates could occur at any time.

The original Moi application consisted of objects made up of the date at the start of a work week and the summary of what had occurred that week. When a user changed/added an entry we saved the entry locally, added it to a log of changed entries, and then submitted it to a web service to store it, if there was a network connection.

When moved over to our framework this simplified Moi greatly, dropping the LOC from 281 to 58 – a nearly 80% reduction. The objects were attached to `persist.root` with property names consisting of the Monday date in the object.

Our framework then did the rest. That eliminated all of the code we had written to manage the storage of our data. It also allowed us to focus on the usage and structure of our data instead of its storage since our framework took that over for us. In the case that a large number of entries were necessary to be stored on a client, we can use object groups to divide entries by their year. This would allow the framework to automatically chunk 52 entries in each batch. In the case of where data was updated while offline, we registered a callback for when new data was received. This allowed us to update the current view if needed. We also provided a simple view to handle conflicts by popping up a window with the data we originally tried to push, so the user could view the updated data and manually decide how to resolve the conflicts.

7.2.2 *Dojo Toolkit*

Probably the closest existing JavaScript library that provides data persistence and offline support is the Dojo Toolkit. Dojo provides the `dojox.data.JsonRestStore` for reading and writing data to a server using a HTTP REST API [9]. Dojo also provides `dojox.rpc.OfflineRest` with `JsonRestStore` to provide offline access to the data they manipulate through `JsonRestStore`.

`JsonRestStore` allows a developer to work with objects directly, like our approach, and not through a low-level data manipulation API. The library does require, though, developers register objects for storage. To save any mutations a developer must mark an object as "dirty" when they have mutated and then explicitly call a global save function:

```
// Marking an object as ‘‘dirty’’ and saving it
jsonStore.changing(item);
item.attr = newValue;
jsonStore.save();
```

The explicit registering of objects for saving and the need to mark mutated objects as dirty along with an explicit save call prevent `JsonRestStore` from being as automatic as the approach in this paper. `JsonRestStore` also does not process mutations to sub-objects; every object that changes must be marked as dirty to have changes be picked up.

A possible bigger difference between our framework and `JsonRestStore` is that communication with the server is one-way from the browser to the server. There is no support to pull new objects from the server nor check to make sure data unseen by the browser is not overwritten on the browser. `JsonRestStore` takes the view that the browser is the master copy of data, as opposed to our framework which makes sure all browsers used by a user are consistent.

Dojo provides a simple e-mail application, Dojo Mail, to demonstrate their GUI toolkit. We used this as a basis for gaining further experience porting applications to use our framework. Dojo Mail contains a tree representing the various folders an email user may have. That tree is rendered using `dijit.Tree` and data exposed through an object implementing the `dijit.tree.model` interface. The interface itself exposes data in a tree structure and allows for notifying the tree widget when the data it is rendering has changed.

By implementing the model interface that Dojo's requires for use with `dijit.tree.model`, we created a persistent version of the model. This shows that it is possible to take our persistence framework and use it with pre-existing JavaScript libraries transparently.

8. RELATED WORK

8.1 Networked File Systems and Databases

A modification of the AFS file system by Huston and Honeyman allowed for disconnected use using an optimistic update policy much like we use [18]. Callbacks were also used to notify the user if conflicts arose from disconnected mutations. The Coda filesystem also used optimistic updating [19]. Coda viewed local data as second-class replicas of the master data, in line with our master-copy view. The Bayou system influenced us by making it clear that metadata should never contain server-specific details as that prevents scaling and thus all metadata should be globally applicable and usable[24]. The Ficus file system emphasized "no lost update" semantics to retain data consistency [12]. The authors also pointed out that users act as their own write token, minimizing actual conflicts in data from multiple updates while offline. While strictly not a networked file system, Balasubramaniam and Pierce wrote about synchronizing files where they pointed out optimistic updating is a must for disconnected operation [1]. While all of these researchers contributed to how we handle offline mutations, their work focused on file systems and not object persistence. Gray et al noted that conflicts from offline mutations should be handled by the application as each application has their own unique requirements [10].

8.2 Persistent Programming Languages

Brown et al showed that persistence can be treated as orthogonal to the language, although it does require some rather low-level access by the language [6]. Eliot and Moss implemented the Mneme object store which used pointers to represent edges in the object graph like in this research [8]. Hosking and Chen created an orthogonal persistence system that tied into the garbage collection step of a programming language, finding it acceptable to persist only when GC is run and thus with a slight delay [17]. Our work differs from these papers by adding object persistence to a pre-existing programming language without interpreter/compiler modification.

8.3 Distributed and Persistent Object Systems

Research on object-oriented, distributed programming languages solves many issues with how to transport objects over a network to remote systems. The Emerald and Modula-2 programming languages were some of the the first languages to provide location transparency but the language also did not hide details that may be needed for performance reasons [4, 20]. We borrowed ideas on how to identify objects from these works even though they focus on moving objects for execution while we have focused on persistence.

The Thor object-oriented database is a persistent object system that supported disconnected operation in an optimistic fashion [11, 21, 22]. It was designed for desktop applications operating in a WAN environment where server redundancy was key. In comparison to our work, our design space was the internet where we had to operate within the confines of the browser and not with the full power of C++ and the underlying OS. This required eager caching of all objects in the browser to avoid latency issues, along having to work within JavaScript's restrictions where we could not tightly integrate into the language.

8.4 Web Browsers

The Gears plug-in introduced the idea of local storage in web browsers, which helped lead to the W3C Web Storage API [14]. The Web Workers specification [15], while allowing for concurrent processing in the web browser, can only be passed copies of objects consisting of only data while also not being able to access `localStorage`. These limitations make Web Workers nearly useless to our framework.

9. CONCLUSION

Relying exclusively on mechanisms which are described in emerging or proposed standards, we have created an automated object persistence framework for Web browsers. We are able to work around shortcomings of the JavaScript programming language in order to detect mutations of persistent objects as quickly as possible; either on a set interval for new properties or instantly for mutations of known properties. All detected mutations are logged to allow for synchronizing the mutations to a server on the internet.

For communication between a browser and a server, we developed a communication scheme from scratch. It had the requirement that it should work with only the browser initiating a connection. It also was designed to work with a potentially latency-heavy internet connection.

The framework not only works with a possibly sub-par internet connection, but with no internet connection at all. By utilizing offline storage our framework can persist objects while disconnected from the internet. When a connection is re-established all changes are sent to a server where they are then persisted and potential mutation conflicts are detected.

Our evaluation of our work shows that thousands of objects can be persisted with acceptable performance overhead on select browsers. Support for our framework is available in three of the four major browsers, although one of the supported browser only existed after initial evaluation and another had extremely poor performance characteristics.

10. REFERENCES

[1] S. Balasubramaniam and B. Pierce. What is a file synchronizer? In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 98–108. ACM New York, NY, USA, 1998.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.

[3] P. A. Bernstein and N. Goodman. A sophisticate's introduction to distributed concurrency control (invited paper). In *VLDB '82: Proceedings of the 8th International Conference on Very Large Data Bases*, pages 62–76, San Francisco, CA, USA, 1982. Morgan Kaufmann Publishers Inc.

[4] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the emerald system. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86, New York, NY, USA, 1986. ACM.

[5] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 97–108, New York, NY, USA, 1999. ACM.

[6] A. Brown, G. Mainetto, F. Matthes, R. Mueller, and D. McNally. An open system architecture for a persistent object store. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume 2, 1992.

[7] J. R. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 317–318, New York, NY, USA, 2001. ACM.

[8] J. Eliot and B. Moss. Design of the mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, 1990.

[9] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[11] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected operation in the thor object-oriented database system. In *WMCSA '94: Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, pages 51–56, Washington, DC, USA, 1994. IEEE Computer Society.

[12] J. Heidemann, R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with ficus. In *in Proceedings of the Second Workshop on the Management of Replicated Data*, pages 2–5, 1992.

[13] I. Hickson. W3C Web Database. `http://www.w3.org/TR/webdatabase/`, September 2009.

[14] I. Hickson. W3C Web Storage. `http://dev.w3.org/html5/webstorage/`, July 2009.

[15] I. Hickson. W3C Web Workers. `http://www.w3.org/TR/workers/`, July 2009.

[16] I. Hickson and D. Hyatt. Html 5. `http://www.w3.org/TR/html5/`.

[17] A. L. Hosking and J. Chen. Mostly-copying reachability-based orthogonal persistence. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 382–398, New York, NY, USA, 1999. ACM.

[18] L. B. Huston and P. Honeyman. Disconnected operation for afs. In *MLCS: Mobile & Location-Independent Computing Symposium on Mobile & Location-Independent Computing Symposium*, pages 1–1, Berkeley, CA, USA, 1993. USENIX Association.

[19] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[20] H. M. Levy and E. D. Tempero. Modules, objects and distributed programming: issues in rpc and remote object invocation. *Softw. Pract. Exper.*, 21(1):77–90, 1991.

[21] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *In European Conference for Object-Oriented Programming (ECOOP*, pages 230–257. Springer-Verlag, 1999.

[22] B. Liskov, M. Day, and L. Shrira. Distributed object management in thor. In *Distributed Object Management*, pages 79–91. Morgan Kaufmann, 1993.

[23] Mozilla Foundation. Dromaeo. `http://dromaeo.com/`.

[24] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5):172–182, 1995.