# Object-Relational Event Middleware for Web Applications

Peng Li and Eric Wohlstadter

University of British Columbia
{lipeng, wohlstad}@cs.ubc.ca

## Abstract

Web based applications are increasingly being used for highly reactive systems where clients expect to be notified of broadcast style information with relatively low latency. Software development of these applications has partially been addressed by technologies associated with the Ajax and Comet architecture for Web programming. While such applications are beneficial to end users, they create additional burdens for software developers. In particular, this push-style development is not integrated with the object-oriented model of data used by application-tier developers. In this paper, we investigate an event-driven style of programming to allow event-based subscription and notification of changes to application object state. This requires a new framework to maintain consistency for developers between two key elements. First, consistency must be maintained between application-tier objects and data-tier state. Second, consistency must be maintained between subscriptions across multiple hosts in a server cluster, so that notifications of changes to object state are disseminated to all appropriate browser clients. We use a running example from the Java-based LightPortal open-source social network Web application to describe the approach. We also evaluate performance implications on the RUBiS Web auction application benchmark.

## 1 Introduction

Web based applications are increasingly being used for highly reactive systems where clients expect to be notified of broadcast style information [27] with relatively low latency according to user's specific interests. Compared to traditional request-driven Web applications, these applications require that clients are notified efficiently, when data of interest is updated on the server, asynchronously to that user's actions.

These server push-based applications make use of a publish-subscribe style architecture. Here the application is primarily responsible for managing a large, shared, data repository for users. At the application level, users express interest in some subset of the data provided by the application service. These interest generating actions from the user are then interpreted by application code to manage delivery of updated data to interested users.

For example, in a social networking Web site, users might share information by posting short messages on each other's profile pages (i.e. on their "wall"). In this case, users who are viewing the wall for another specific user should receive notification of the updates to that wall interactively (i.e. in soft real-time). So an action such as viewing a profile page implicitly creates a subscription for notification of those wall updates. Users expect such notification to occur asynchronously, through Ajax mechanisms.

As in **Figure 1**, User X might visit such a page in step 1. Later (step 2), User Y causes an action which updates some application state on which that page depends. So, in step 3, User X will be notified.

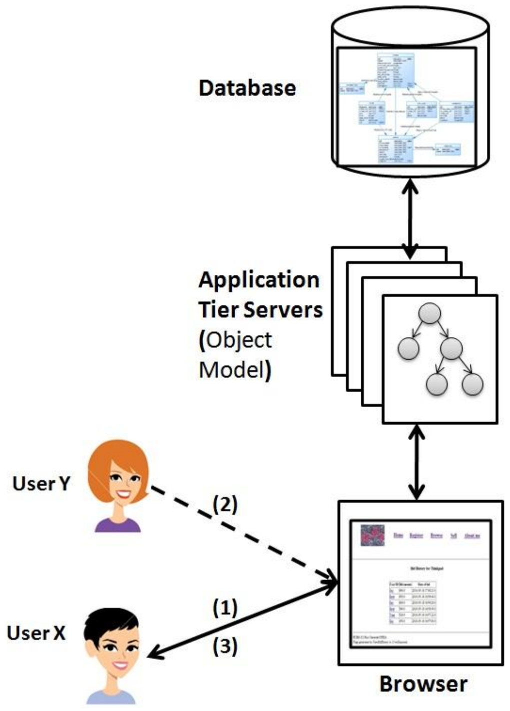


Figure 1: Overview of Object Relational Events (ORE). ORE manages consistency between the database and application-tier object model, and manages notification to subscriptions placed by multiple servers in the application-tier. Example: User X visits a profile page for a particular user. Later (step 2), User Y causes an action which updates some application state on which that page depends. So, in step 3, User X will be notified through server-side push.

Software development of these applications has partially been addressed by technologies associated with the Ajax architecture for Web programming. Recently, this approach is being used more aggressively to add the high-level abstraction of server to browser push notification using techniques under the name Comet [9]. A closely related technology, known as Web Sockets, is also being considered for W3C standardization. Applications using these new approaches can be found in Gmail, online chat rooms, social networking and other applications which need notifications to occur interactively.

While such applications are beneficial to end users, they create additional burdens for software developers in the application-tier. In particular, this push-style development is not integrated with the object-oriented programming practices used in typical application-tier development. Standard practice is generally to focus development around an object-oriented model of data (as in the application-tier of **Figure 1**).

This object model can then be mapped to relational data storage through the use of specific design patterns and automation techniques known as Object-Relational Mapping (ORM). While the ORM approach is beneficial, it does not directly support an event-driven style of development, making it cumbersome to develop push-style applications using such middleware. In this paper we address this problem by investigating the integration of event-driven application programming with the object-relational model in server-push Web applications. We hope to enable an event-driven style of programming where software developers can easily register for notification of updates to objects. When the object state changes, this information can be pushed to any interested browser clients.

Unfortunately, ORM frameworks do not provide consistency guarantees once an object view has been extracted from the database. For example, when all the wall postings for a specific user have been retrieved from the database, the ORM will not guarantee the consistency between the wall post objects in the application-tier and those represented in a database. So, in this case, when a post is inserted by another user, the postings in memory could become inconsistent with those in the database. This was never seen as a problem before, since traditional Web applications only use such object replicas transiently, in the processing of a client request. This inconsistency problem manifests itself in two ways, which we consider in this research.

First, since object class properties are mapped to a specific view (e.g. projection) of the database tables, the underlying data which corresponds to object state can be modified in multiple ways. For example, it is possible for two distinct object classes to have properties that map to the same column in the database. This problem is closely related to the problem of view maintenance in the database literature [14]. However, this problem has not previously been considered for object-relational frameworks or server-push technology.

Second, push-based Web applications typically include notifications of data representing interactions between users (e.g. sharing data on a profile wall). However, medium- and large-scale Web sites are handled by multiple physical hosts in an application-tier cluster (indicated by the layering of the application-tier in **Figure 1**). Relying on the data-tier to provide consistency between end-user views of data would create additional latency that is a bottleneck to interactivity. This paper describes how we have mitigated this problem specifically for dissemination of updates to object state monitored by an application-tier programming framework.

We have investigated the integration of event-driven application programming with the object-relational model and describe the details of our research through reference to our implementation called Object-Relational Events (ORE). The contributions of this paper can be summarized as follows:

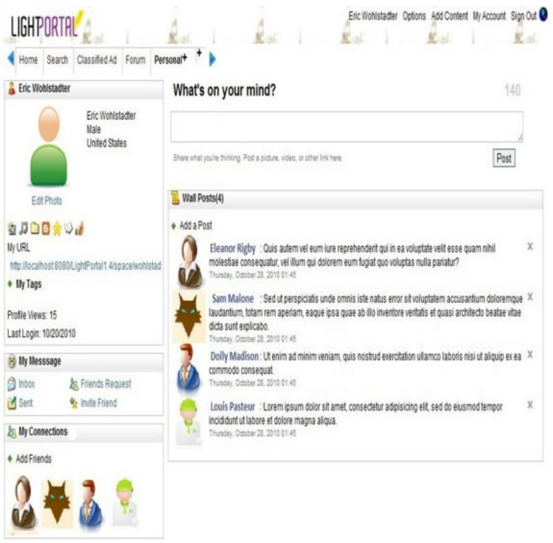
1. Design, implementation and evaluation of ORE, a Web application platform for event-driven programming at the Web application-tier, to resolve both: (i) conceptual consistency between relational and object-oriented data models and (ii) to manage notification dissemination for server-push Web application clusters.
2. An evaluation demonstrating that ORE has low overhead compared to a manually implemented server-push applications.
3. Evaluation of optimization for server-push notification using bloom filter based subscription compression.

The rest of the paper proceeds as follows: in Section 2 we further describe a motivating example from the LightPortal project. We present some technical background in Section 3. In Section 4 we present more technical details of the implementation, then an evaluation in Section 5. Related work is presented in Section 6. Finally, the paper concludes in Section 7.

## 2 Motivating Example

To motivate the problems described in the introduction, we use a running example of the LightPortal social network application [20]. LightPortal is an open source Java Web application. LightPortal provides support for several kinds of user interactions, e.g. registering a new account, creating social links, joining groups, leaving posts on a user's wall, etc.

**Figure 2** shows an example of the LightPortal profile page. For a more interactive application, developers may want to provide for new entries of the profile postings to scroll on to the end of the log without additional user interaction.



Eleanor Rigby : Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?
Thursday, October 28, 2010 01:45

Sam Malone : Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo.
Thursday, October 28, 2010 01:45

Dolly Madison : Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Thursday, October 28, 2010 01:45

Louis Pasteur : Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Thursday, October 28, 2010 01:45

Figure 2: The LightPortal profile page. When new posts are made, additional entries should be appended asynchronously. Using ORE, developers can register for notification of updates to all postings associated with a particular profile object.

For the implementation, the application developer will need to choose between an approach based on client-side polling or else use server-side push mechanisms to handle delivery to clients. Polling approaches integrate well with existing server-side programming models. They follow the traditional request-response workflow and offer a fairly simple strategy for creating modern Ajax-style Web applications. However, under heavy load, polling approaches can cause considerable strain on the server compared to push-based approaches. Thus developers have turn to server-push to achieve low-latency notification with scalability.

A server-side developer needs to address two concerns in this example: first, to detect when a new post is added/removed/changed on a profile, and second, to notify all users who are viewing the wall.

Detecting such changes is normally hard for an application-tier programmer using object-oriented programming because the application-tier may project any number of different object views of the underlying database for different parts of the application. As a simple example, suppose the database includes two tables as in **Figure 3**.

```
Profile(
  pId PRIMARY KEY,
  userName)

WallPost(
  wpId PRIMARY KEY,
  wallProfileId FOREIGN KEY REFERENCES Profile(pId),
  postedProfileId FOREIGN KEY REFERENCES Profile(pId),
  messageText)
```

Figure 3: Example SQL table definitions for LightPortal `Profile` and `WallPost`. In the conceptual model of the data there are two one-to-many relationships between `Profile` entities and `WallPost` entities (types are not shown and elided SQL syntax is used).

So, `wallProfileId` is a foreign key referencing the profile of the user whose wall will contain the message, and `posterProfileId` is a foreign key referencing the profile of the user who posted the message. Thus in the conceptual (entity-relationship) model [24] of the data there are two one-to-many relationships between Profile entities and Wall post entities.

The possibility of inconsistency between object views and database state can now be seen concretely in light of **Figure 4**. This object model of data corresponds closely to the conceptual (ER) model, with the aforementioned one-to-many relationships materialized as Set collections. Notice that conceptually, when a new `WallPost` object is added to the set of posts that a user has made to their friend's wall (i.e. added to the set `postedToFriendWall`), it should seemingly cause a new `WallPost` to appear in some other profile's wall posts (i.e. added to some object's `onProfileWall` set).

However, such conceptual consistency is not enforced by any existing ORM frameworks. This is because in a traditional client-pull, request-driven, Web application such inconsistency is not generally a problem. In such request-driven applications object views of the database are reconciled at each client request (such construction might take place from an application-tier cache, so this does not necessarily imply a need to access the data-tier). Thus, such inconsistency would only be present in the course of a single request and the developer of that single request's logic could manage the problem manually.

```
class Profile {
    int id;
    String userName;
    Set⟨WallPost⟩ onProfileWall;
    Set⟨WallPost⟩ postedToFriendWall;
    ...
}
```

Figure 4: Example Java class definition for LightPortal `Profile`. This object model of data corresponds closely to the conceptual (ER) model, with the aforementioned one-to-many relationships materialized as `Set` collections.

However, in push-based applications, subscriptions for interesting events are long-lived and possibly persistent. So the lifetime of a subscription would span many application requests, making it difficult for a programmer to manually reason about inconsistency.

Ideally mutations occurring to objects which correspond to updates of interest for users would be detected immediately, so that notification to any interested users could be processed. For example, in ORE, a server-side programmer could create a subscription to be notified when new posts appear on a profile's wall as in **Figure 5**.

```
//Create notification callback function

profileWallCallback = function(pushConnection, event) {
   //Use event data to create JSON notification
   //and submit to pushConnection
}

//Use ORE API to register for any changes to the profile
//object's onProfileWall collection

ORE.addCollectionListener(profile,
                          "onProfileWall",
                          profileWallCallback, ..);
```

Figure 5: Using ORE a developer can register a long-lived subscription for notification when posts are added to a profile wall. ORE will ensure conceptual consistency when posts are added through object views, possibly different from the one used to bind the subscription. (Syntax shown in JavaScript for conciseness, although implementation is server-side Java).

Still, ensuring consistency between object views and the underlying logical database model is not the only problem to provide a programmer friendly, scalable, event-notification framework for Web applications. This is because most Web applications are deployed across several hosts in a server cluster. ORE will also need to manage the fact that replicas of objects will be distributed across the various hosts which manage the session for interacting users.

For such reasons, we believe frameworks for supporting object-oriented, event-driven programming, should be integrated with ORM frameworks used commonly in application-tier logic. In the remainder of the paper we discuss more details of our prototype platform for investigating this approach.

## 3 Background

Before describing our technical research, we describe some useful background on important server-side Web application programming technologies currently used in industry. Later, in Section 6, we return to describe other closely related research.

### 3.1 Comet and Web Sockets

Since HTTP is inherently a request-response based protocol, pushing asynchronous updates of data from the server to browser based clients can be difficult. Although a proposal for server-push was put forward by Netscape in the late 1990's, this mechanism was never supported across all major browsers [8].

Due to this limitation, Web developers have used a number of techniques to try and work around this problem. Traditionally, a polling based approach was used, in which a Web page would be programmed to automatically refresh after a certain timeout period. This solution led to a poor user experience as the browser was forced to render an entire page resulting in a noticeable interrupt of the user interaction.

Newer approaches based on the popular Ajax mechanisms provide for a less intrusive user experience. The Ajax mechanisms include an RPC-like abstraction which enables the browser to poll for updates from the server in the background. As content is retrieved from the server, scripts can be used to partially render incremental changes of the user interface. However, this approach still suffers from two serious problems.

First, since polling is used, the polling interval determines a best possible latency between the time of an update to data on the server and the time of delivery to the client. Second, reducing the polling rate has the effect of increasing the workload on the server, so managing tradeoffs between this latency and overall server throughput can be difficult.

The demand for more responsive Web applications has led to interest in an alternative approach, commonly referred to as Comet, for example using the mechanism of long-polling. In this approach, the browser creates an HTTP connection to the server, which is held idle by the server until some update of data is available for delivery to the client.

Since a browser script programmer is able to set the timeout on the client, the browser can hold the connection open indefinitely, until data is pushed to the client along the ordinary HTTP response flow. While there has been interest in the approach for some time, it was not practical before, since on traditional threaded Web servers, each held connection resulted in consumption of costly thread resources on the server. More recently, since the introduction of asynchronous event-based Web server architectures [26], there has been an increasingly widespread use of this approach.

Research in [4] has compared the server performance cost for Comet and the polling approach based on the same work load and shows that under high workloads Comet provides far better scalability.

### 3.2 Object-Relational Middleware

Web applications usually rely on a relational data source for handling the ACID requirements of information storage and retrieval. The conceptual model of relational data for most online transaction processing applications, such as a typical Web application, corresponds closely to an object-oriented class structure. However, at the implementation level, the relational model and object model are diverse.

To solve this impedance mismatch, developers turn to object-relational mapping middleware to try and maintain the best of both worlds. In an ORM, entity relationships and properties from a database schema are explicitly mapped to properties of object classes in the application-tier. Relationships such as one-to-many and many-to-many can be mapped to collection classes of an object-oriented programming language, such as Set, List, or Map.

## 4 Technical Details

To describe the technical details, we start by briefly discussing the high-level programming model exposed to application developers and then describe the further technical details of the ORE framework.

### 4.1 ORE Framework Architecture

**Figure 6** shows the high-level architecture of the ORE framework runtime components, placed in the context of the Web application-tier. Clients communicate with an ORE-based application over two types of connections.

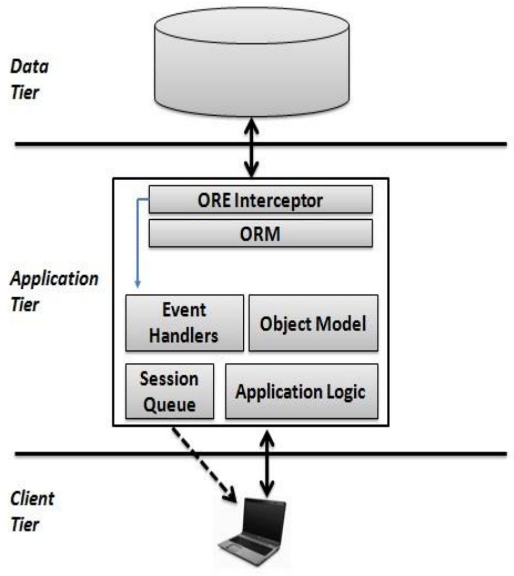


Figure 6: Internal architecture of ORE framework, in the context of a three-tier Web architecture. Client communicates with ORE-based application through two types of connections (i) a traditional HTTP request (bold arrow) and (ii) an HTTP long-polling request.

First, as in traditional applications, clients issue HTTP requests which are handled by *Application Logic*, implemented by a request handler (e.g. a Java Servlet, CGI script, String Template [23], etc..) targeted by a URI for fetching dynamic Web page content. During request handling, a developer typically manipulates data as an *Object Model* to complete HTML page generation or generation of Ajax updates (i.e. XML or JSON content). Object data is retrieved as views from the data-tier through an ORM. Unlike traditional Web applications, application programmers can use the ORE API to registered long-lived subscriptions for notification to changes of ORM managed state.

Generated content simply contains a snapshot representation of some resource state [12]. However, developers might want to maintain consistency between the client and server state for some temporally sensitive objects. Using ORE, developers can use an API to attach *Event Handlers* (i.e. listeners) to properties of objects in the object model. The attachment of handlers to objects is correlated with the session of the client making the request. When that property is mutated, the event handler will be queued for execution, as described later in Section 4.4.

At the time of execution, the handler is provided with some information specific to that event as well as a handle to an output stream which will be pushed to the client browser. For example, in the case of a new `WallPost` in LightPortal, the event would contain a reference to the wall posting information as an object. The handler implementation can simply be programmed to marshal an appropriate representation of the event to the output stream; the resulting output will be appropriately routed to the corresponding session for delivery to the client.

### 4.2 ORE Framework API

The top of **Figure 7** shows three representative functions provided for ORE programmers. The first two, `addPropertyListener` and `add-CollectionListener` accept similar argument types, but provides different semantics.

For these functions the argument `entity` is a reference to an ORM object to monitor. The `property` describes a specific property whose mutation should cause notification to the callback provided by `listener`. The final argument `persistent` describes whether the subscription should remain durable across multiple sessions from the same user.

The first case, `addPropertyListener`, covers detection of mutations to the value of a `property`, whereas the second, `addCollectionListener`, covers addition, removal, or update of elements from a collection referenced by the `property`.

The third function, `addQueryListener`, provides support for listening to updates of an ad-hoc collection of objects, selected by an object query. In our implementation we rely on the Java based Hibernate Query Language (HQL) but other object query languages could be supported. Details of the ORE implementation to support these functions are described below (in Section 4.3).

```
//Sample of functions from ORE API
class ORE {
  addPropertyListener(Object entity,
                      String property,
                      PropertyListener listener,
                      boolean persistent);

  addCollectionListener(Object entity,
                      String property,
                      CollectionListener listener,
                      boolean persistent);

  addQueryListener(String oql,
                      QueryListener listener,
                      boolean persistent);
...
}

//Example API usage
ore.addPropertyListener(profile,
                        "status",
                        statusListener,
                        true);

ore.addCollectionListener(profile,
                        "posts",
                        postingListener,
                        false);

ore.addQueryListener(''SELECT posts FROM Profile
                     WHERE ..'',
                     false);
```

Figure 7: Sample of functions from ORE API (top) and example of usage (bottom).

The bottom of **Figure 7** shows three examples of use. The first covering a subscription to be notified when the `status` of a particular `profile` object changes; the second covering a subscription to be notified when any elements are added, removed, or updated from the set of `posts` related to the `profile`. The last example also subscribes to a set of posts, but using an ad-hoc query which might filter on additional values from the database.

## 4.3 Object-Relational Consistency

The details of detecting updates for the purpose of notification depend on the particular API functions that are used. As in **Figure 6**, an interceptor (ORE Interceptor) is used to monitor the SQL `UPDATE` and `INSERT` statements issued by the ORM to the database. The interceptor wraps the original API used by the ORM so it can be plugged in transparently. These details are described for the three most important functions from the API.

### 4.3.1 `addPropertyListener`

In an ORM-based application, every primitive (i.e. non-collection) typed property of a class corresponds to a single column of some singleton view of the database state (i.e. a query that returns one row). For views which do not contain aggregation, each primitive property can be mapped back to a single column in one of the base relations of the database.

Here we monitor updates made the ORM to detect when an update is made to a row of this relation with the same primary key as that given by the `entity` passed to `addPropertyListener`. If the update changes the value of `property`, then ORE passes the new value as an event to the `listener` callback.

As described in the literature, not all views generated by aggregation can be mapped to base relations [14] without reissuing the original database query. For those limited cases, developers would still need to rely on application specific consistency logic.

### 4.3.2 `addCollectionListener`

In the case of a collection type, such properties correspond to a view of the database which contains one column. The column maps directly to a column in a base relation that is a foreign key referencing the primary key of the relation identified above (in `addPropertyListener`). Here we monitor updates made to the ORM to detect when an insert, delete, or update is made to the table containing this column.

In each case, if the value of the column matches the same primary key as that given by the `entity` passed to `addCollectionListener`, an event is generated. Depending on the type of operation, the event passed to the `listener` callback contains an object either representing the data added, removed, or changed in the collection.

#### 4.3.3 `addQueryListener`

For ad-hoc queries, we compute the intersection between the query passed as a subscription and the `UPDATE` or `INSERT` generated by the ORM. This intersection is computed as described, for example, in previous work on Web application cache invalidation [3]. As described, bulk updates with conditional criteria may require an additional query to the database, in order to resolve intersection.

These types of bulk updates appear to be are rare in end-user Web applications, since users typically interact with set of objects that have already been presented in the browser (e.g. in a HTML table), and users actions do not typically result in an ad-hoc set of database rows being changed. Even still, this can be supported with a small amount of additional overhead.

### 4.4 Server-Push

Some Web applications can tolerate a reasonably high latency between updates on the server and notification to interested clients. For some Web applications a push based solution is likely to be more efficient, to avoid the cost of repeated application-level connection setup. Several HTTP push solutions have been proposed recently, including a proposal for changes to include server-side push in the W3C HTML 5 standard. In our current implementation we make use of the Comet long-polling approach.

When an event is triggered by the ORE Interceptor it checks to see if the client for which the event is targeted is currently holding on to a connection. If they are, we can simply execute the handler immediately, and push the resulting output stream as a response to the held HTTP request.

However, although long polling helps alleviate some performance concerns in Web applications, it is still not based on a connection-oriented protocol. So for delivering content to clients, we may need to store some state information in the event that they are in the process of creating a new long-connection.

This state consists of the set of event handlers which have been triggered between the time of the client's last long polling request, held in the Session Queue. Later, when a new connection is made by the client, these queued event handlers are executed on behalf of the client and a response based on the output can be returned immediately.

### 4.5 ORE Cluster Details

Using ORE on an application-tier hosted by a single server machine host can be achieved as previously described. However, medium- and large-scale deployments of Web applications will require the application-tier to be hosted across a cluster of server hosts or VM instances in a cloud computing platform. In these settings, user requests are generally forwarded by a load balancer to some host in the cluster.

To ensure that clients have continuous access to session state stored in host main memory, sticky sessions can be used so that client requests from the same user continue to be forwarded to the same host. For an ORE application, we will additionally need to require a mechanism to ensure that updates made by a user on one host are disseminated as notifications to users that may be connected to other hosts in the cluster. This situation is illustrated in **Figure 8**, where two clients are shown with sessions hosted by different ORE cluster hosts.

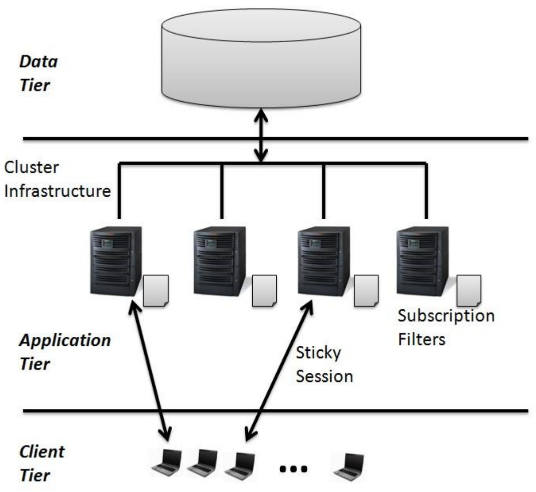


Figure 8: Architecture of ORE clustering.

In traditional client-pull Web applications, strong data consistency across application hosts is generally achieved by access to the data-tier. In order to prevent the data-tier from becoming a bottleneck, weaker data consistency is provided by distributed caching [16, 10, 11] in the application-tier. Neither of these approaches are ideal for notification dissemination as they are used to update passive data storage which requires a client-driven request to pull information. While a large amount of research exists for efficient notification in publish-subscribe systems [27, 6, 17], such research results do not apply directly to push-based Web applications. Also closely related is research on scalable distributed storage for Web-based systems [7, 15]. However, like research on Web caching, that research is applied to applications that are pull-driven by client requests.

In a Web application, user actions which generate the need for new subscriptions could take place at any time through a user's browsing session. This could even depend on relatively transient session state, such as which page in a Web site a user is actively viewing. Thus the need to generate new subscriptions for clients could come at any time and at a large scale. For this reason, ORE uses a peer to peer topology for notification dissemination to avoid potential bottlenecks associated with centralizing subscription information or notification routing. ORE peers discover the identity of existing peers in a cluster using a central discovery server since changes to cluster membership happen relatively infrequently. This is true even in an elastic cloud computing scenario since membership changes would happen on the order of minutes.

The ORE clustering implementation is currently built on top of a Java Messaging Service (JMS) layer, although any efficient point-to-point asynchronous messaging middleware could be used. Updates to objects on a single host are detected as described, requiring notification to any hosts which are currently managing users interested in such updates.

To avoid broadcasting notifications to all peers, notifications are filtered by subscription information broadcast periodically by every host to all peers. Since modern Web application hosts can manage many thousands of user simultaneously, this subscription information is compressed as lossy set membership information, using the well known Bloom Filter [2, 5, 25] data-structure. Each filter encodes the subscriptions currently active for some host in the cluster. This allows us to filter notifications before forwarding to distinct cluster hosts. Since bloom filters are probabilistic, there is a chance of false positives, where some notification is sent unnecessarily. We configured an existing implementation (Apache open-source), to a false positive rate of 2%. In our experiments this provided a minimum reduction of 48% in bandwidth cost and reduces CPU costs associated with processing subscription filters.

We use a Counting Bloom Filter [25], since subscriptions will need to be added and also removed depending on users browsing behavior. As long as one user at a particular host is interested in an update of a particular type, that host will need to receive relevant notifications. For this reason, each host maintains a counter for each unique subscription currently active at that host. Two subscriptions are considered identical when either: the `entity` and `property` arguments given to the API have the same value (in the case of `addPropertyListener` and `addCollectionListener`) or when the query statement given to the API is the same (in the case of `addQueryListener`).

When the counter is incremented from 0 to 1, a key is generated to represent that subscription type, and is added to the bloom filter maintained by the host. Later, when the counter is decremented from 1 to 0, this key is removed from the filter. In our experiments, we use a subscription filter broadcast period of 200ms, since this provides a low latency for interactive applications while only introducing negligible overhead. This is considering that Web application hosts are typically already processing on the order of hundreds or thousands of requests per second.

## 5 Experimentation and Examples

We applied our implementation to two open-source Java-based Web applications. We looked for cases where ORE would be useful to extend the capabilities of these traditional Web applications with server-push. We also used the applications to run performance and scalability experiments with realistic applications. All measurements are averages over 10 trials. In both cases the base experimental setup was as follows.

*Hardware environment*: All machines used in the experiments are a 64-bit Intel Xeon dual core 2.8GHz processor with 7.5GB main memory. The workload generation is partitioned between multiple clients to ensure that the client simulation does not become a bottleneck in the experiments. These machines are connected through a 1Gbps ethernet local area network.

*Software environment*: The application server is Jetty 7 for Java and the database is MySQL 5.1. We used Jetty for its support of asynchronous I/O on long-polling connections. This design will soon become a Java standard through the Servlet 3.0 API. We use the Hibernate 3.5.2 ORM.

## 5.1 RUBiS

First, we applied our ORE framework to part of the Rice University Bidding System (RUBiS) [1] Web auction application. This application simulates the activities for an online auction site. RUBiS provides support for several kinds of user interactions, e.g. registering a new account, submitting an item for auction, placing a bid for an item, leaving comments on another user, etc.. Currently RUBiS is programmed as a traditional Web application where users are required to pull new information from the server by explicitly refreshing a page representing some content which they have read.

## 5.2 RUBiS Application Example

To evaluate and test the implementation of ORE, we applied it to a common use-case for server-push Web applications, monitoring of bid history. We found that this interaction in RUBiS contains a source of potential inconsistency in the object model, which could be overlooked by developers, in the case of relationships between Auction Items and Bids (shown in **Figure 9**).

Consider a developer who wants to implement a server-push version of RUBiS auction monitoring. Looking at the `ViewBidHistory` code, they could see that each item object contains a reference to its collection of bids (i.e. `item.getBids()`). So they may mistakenly assume that changing the RUBiS code to monitor additions to this collection would help detect whenever a new bid was added to an item.

However, after looking at the `StoreBid` servlet (bottom of **Figure 9**), we see that other parts of the code insert bids directly to the database, creating a relationship to an item only by virtue of a foreign key.

In order to implement a new server-push version of RUBiS auction monitoring, we applied an ORE event handler to manage updates of changes to the underlying one-to-many relationship between Auction Items and Bids. We changed the HTML of RUBiS to accept these JSON updates and apply them as incremental changes to the client-side Document Object Model (DOM).

```
class ViewBidHistory extends HttpServlet {
  doGet(request, response) {
      itemId = request.getURLParameter("itemId");
      item = orm.selectByPrimaryKey(itemId);
      response.print(item.getBids());
  } }

class StoreBid extends HttpServlet {
  doGet(request, response) {
      itemId = request.getURLParameter("itemId");
      newBid = new Bid(itemId, /*form input from user*/);
      orm.insert(newBid);
  }
}
```

Figure 9: RUBiS Java Servlet classes to (top: to insert a new bid for an auction item) and (bottom: to generate content displaying a log of bids on an auction item). Notice that the entity relationship between Items and Bids is being manipulated through different object views (top: through a collection property `Item.bids`) and (bottom: directly through a mapping to the `Bid` table).

## 5.3 RUBiS Experiment

In this experiment we modify the bidding client workload of RUBiS, which uses a mix of the RUBiS benchmark workload where 80% of users are read-heavy and 20% of users are write-heavy. We took CPU load measures over a 5 minute interval with a 1 minute warm up interval to build up to a 4000 user workload. This particular evaluation is performed using a single server to isolate the CPU usage overhead for three implementations.

The Polling case uses a traditional polling approach for delivering bid history updates to clients across Ajax. The x-axis plots measurements for a range of concurrent simulated users. In the Polling case, each read-heavy user polls at a frequency of 2 seconds for updates.

The measurements for Manual and ORE both use a server-push implementation over long-polling. In the Manual case, we hand-coded the event notification by locating appropriate points in the Java code where relevant object state was mutated. The ORE approach uses an event handler attached to an item's `bids` property.

From **Figure 10**, we see that the CPU usage for both the ORE and manual implementation are similar, although ORE induces a fix overhead of between 2% and 10%. This overhead is due to the additional costs of query interception and analysis.

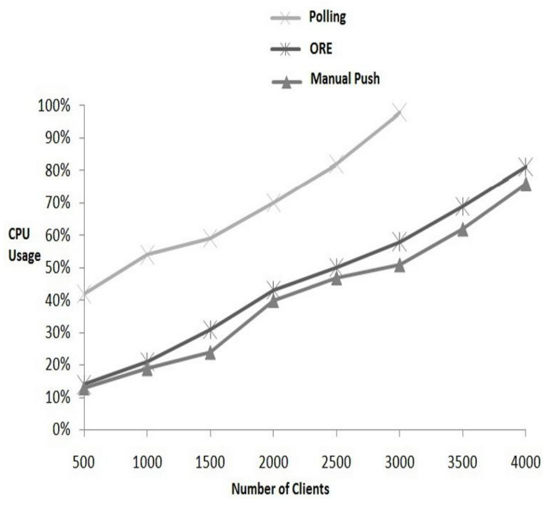


Figure 10: CPU Usage in RUBiS Use-Case for three different implementations.

## 5.4 LightPortal Experiment

Similarly to the RUBiS example, we have applied ORE to the use-case of the LightPortal profile pages, as described in Section 2. Here we consider performance implications of ORE as applied to this scenario and deployed on a cluster of six server hosts. Again we use a mix where 20% of simulated users write to a profile periodically and 80% of simulated users only read from profiles.

Across the experiments, there is a fixed client workload of 3000 simultaneous clients (per cluster host). Each client is simulated by a thread which changes its target profile page (reading or writing) every 5 seconds. Thus, each writing thread will write once every 5 seconds, and every reading thread will receive any updates to a specific profile page which occur during that 5 second interval.

Clearly most users in a real application would likely view a page for more than 5 seconds, however, using a shorter interval allows us to test the scalability of our subscription filter dissemination protocol, since changes to page viewing correspond to changes in subscription state.

As shown in **Figure 11**, along the x-axis we vary the client browsing activity according to a probability that two users will choose to read/write a single specific page during a 5 second interval. So, for the case of 0%, all users are always active on a separate page, and for the case of 100%, there is only a single profile page which is visited by users. This allows us to investigate differing implementation strategies for sharing subscription and notification information between cluster hosts. For example, in the case of a 0% overlap, there is no need for separate machines to ever share notifications; in the case of 100% all updates to the profile page must be broadcast to all other hosts.

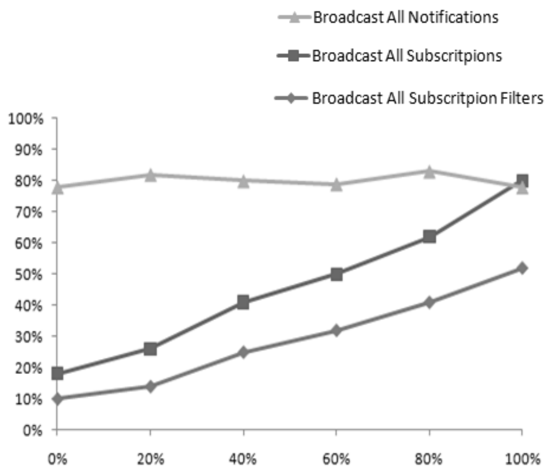


Figure 11: CPU Usage in LightPortal Use-Case for differing client overlap in page activity.

The y-axis shows the CPU usage on one single machine. Since the experiment is symmetric with respect to all six hosts, the measurements for any host are roughly identical.

Three implementations are evaluated. First, a naive approach in which all notifications are broadcast for any update to a page (Broadcast All Notifications). Second, an approach which broadcasts subscriptions uncompressed at 200ms intervals (Broadcast All Subscriptions). Third, an approach which broadcasts similarly, but which compresses subscription information with the previously described counting bloom filter (Broadcast All Subscription Filters).

Since the naive approach broadcasts all notifications regardless of necessity, its CPU usage is fixed across all client simulations. We see from the figure that the ORE cluster implementation using a subscription bloom filter reduces CPU usage by up to 25%. The CPU reduction is due partially to the fact that the serialized bloom filter can be directly copied from the middleware message buffer into a form suitable for testing subscription membership. In the case of the uncompressed filter, each subscription must be unmarshalled and used to build a new data-structure suitable for notification filtering.

# 6 Related Work

## 6.1 Caching Dynamic Web Content

Caching is often used to reduce Internet latency and server load for Web applications. While HTTP includes specific support for caching of static data [22], caching of dynamic content is much more difficult. Dynamic pages are generated based on the results of queries to back-end data sources. In this case, a solution is needed to determine when cached content becomes invalidated due to updates of data sources.

Iyengar et al. [16] provide a data update propagation algorithm to maintain the consistency of query results which may be cached as dynamic content and the underlying data store. When a change to the underlying data happens, cache managers query the dependence information which has been stored in an object dependence graph to determine which objects should be invalidated. Such dependence graphs can be automatically generated from queries [10].

AutoWebCache [3] is a middleware solution which can transparently cache dynamically generated content in J2EE applications. The first main contribution of their research is to provide a solution for caching dynamic Web page at the front-end and maintaining consistency with the back-end database using query intersection run-time analysis. We use a similar approach for detecting updates to object views. ORE uses some similar techniques for detection of updates to relevant database state. Different from caching work we considered translation of updates as push-based notification and dissemination in distributed application-tier clusters.

## 6.2 Automated Ajax

Due to the coordination of asynchronous messaging between client and server, client-side Ajax programming can be difficult. Other projects have previously identified the essence of Ajax as being the detection of updates to specific application state and the propagation of these updates to the browser's DOM tree.

Ajax Report Pages [13] is a programming model for interacting with an SQL data store directly from client-side script code. In this approach application code is restricted to a specific superset of SQL, which allows the framework to automatically detect changes to the database affecting a previously rendered report. In this respect, that research is similar to ORE. However, that research does not consider the requirements for object-oriented application programming, server-side push technology, or application-tier clustering.

FlapJax [21] is a client-side data-flow programming languages for Ajax. In FlapJax, data-flow paths created by variable assignments are tracked by the language runtime to support automatic propagation of variable mutations. Different from ORE, this project focuses on traditional Ajax client-side programming, whereas our concern is on server-side push technology.

## 6.3 Event-Driven Architecture

Event driven programming is a well established practice in database management through the use of triggers and more extensively in the Event-Condition-Action paradigm [19]. While some work has been done to add event-driven programming to object-oriented databases systems [18], the ORE research considers support specifically for an object-relational model where consistency between the application-tier and database must be managed.

## 6.4 Pub-Sub Systems

One of the most important problems in wide-area publish subscribe systems is the efficient forwarding of messages between routers [6, 27]. Routers should filter any published messages which are not useful to any other downstream subscriber. However, the size of subscription information can be too large and volatile to efficiently manage. For this reason, lossy subscription information can be leveraged in the form of bloom filters. In ORE we have also leveraged this form of compression for sharing subscriptions between application-tier cluster hosts.

## 7 Conclusion

Currently, the data consistency between resource updates and an object-oriented model in Web applications is not guaranteed by conventional object-relational middleware such as the popular ORM frameworks. However, for emerging push-style Web applications, this consistency guarantee is important to detect changes which may take place on multiple differing object views of the same underlying data.

In our performance evaluation we saw that the ORE framework approach could provide better performance than a traditional polling based implementation and less than a 10% reduction of performance compared to a manually implemented push-based based implementation. In our cluster experiments, we found the use of bloom filtering important for large-scale subscription information with a high churn rate.

## References

[1] C. Amza, A. Ch, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, and W. Zwaenepoel. W.: Specification and implementation of dynamic web site benchmarks. In: IEEE Workshop on Workload Characterization, pages 3–13, 2002.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13:422–426, 1970.

[3] S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, and W. Zwaenepoel. Caching dynamic web content: Designing and analysing an aspect-oriented solution. In Proc. of the Internation Middleware Conference, 2006.

[4] E. Bozdag, A. Mesbah, and A. van Deursen. Performance testing of data delivery techniques for ajax applications. Journal of Web Engineering, 2009.

[5] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In Internet Mathematics, pages 636–646, 2002.

[6] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In Proc. of the SIGCOMM Conference on Applications, technologies, architectures, and protocols for computer communication, pages 163–174, 2003.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In USENIX Symposium on Operating System Design and Implementation, pages 205–218, 2006.

[8] N. C. Corporation. An exploration of dynamic-documents. `http://www.hoolan.net/spec/WWW/pushpull/`, 1998.

[9] D. Crane and P. McCarthy. Comet and Reverse Ajax: The Next Generation Ajax 2.0. APress.

[10] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A middleware system which intelligently caches query results. In Middleware '00: IFIP/ACM International Conference on Distributed systems platforms, pages 24–44, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[11] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In IEEE/ACM Transactions on Networking, pages 254–265, 2000.

[12] R. T. Fielding, D. Software, and R. N. Taylor. Principled design of the modern web architecture. ACM Transactions on Internet Technology, 2:115–150, 2002.

[13] Y. Fu, Y. Papakonstantinou, K. Kowalczykowski, K. K. Zhao, and K. W. Ong. Ajax-based report pages as incrementally rendered views. In International Conference on Management of Data (SIGMOD), 2010.

[14] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In SIGMOD Conference, 1993.

[15] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazons highly available key-value store. In In Proc. SOSP, pages 205–220, 2007.

[16] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, pages 49–60, 1997.

[17] H. A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V.Muthusamy, R. S. Kazemzadeh: The PADRES Publish/Subscribe System. Principles, Applications, of Distributed Event-Based Systems, pages 164-205, IGI Global, 2010.

[18] G. Jagadish, N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In In Proc. of the ACM SIGMOD International Conference on Management of Data, pages 81–90, 1992.

[19] K. G. Kulkarni, N. M. Mattos, and R. Cochrane. Active database features in sql3. In Active Rules in Database Systems, pages 197–219. 1999.

[20] LightPortal. http://www.lightportal.org/. Accessed on 27/06/2011.

[21] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In OOPSLA, 2009.

[22] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In Proc. of the SIGCOMM Conference on Applications, technologies, architectures, and protocols for computer communication, 1997.

[23] T. Parr. Enforcing strict model-view separation in template engines. In Proc. of the International World Wide Web Conference, 2004.

[24] P. P. shan Chen. The entity-relationship model: Toward a unified view of data. ACM Transactions on Database Systems, 1:9–36, 1976.

[25] H. Song, J. Turner, S. Dharmapurikar, and J. Lockwood. Fast hash table lookup using extended bloom filter: An aid to network processing. In Proc. Conf. on Applications Technologies Architectures and Protocols for Computer Communications, pages 18–192, 2005.

[26] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In Proc. of the Symposium on Operating Systems Principles, 2001.

[27] T. Yan and H. Garcia-molina. The sift information dissemination system. ACM Transactions on Database Systems, 24:529–565, 2000.