Integrating Long Polling with an MVC Web Framework

Eric Stratmann, John Ousterhout, and Sameer Madan
Department of Computer Science
Stanford University
{*estrat,ouster,sameer27*}*@cs.stanford.edu*

Abstract

Long polling is a technique that simulates server push using Ajax requests, allowing Web pages to be updated quickly in response to server events. Unfortunately, existing long poll approaches are difficult to use, do not work well with server-side frameworks based on the Model-View-Controller (MVC) pattern, and are not scalable. Vault is an architecture for long polling that integrates cleanly with MVC frameworks and scales for clustered environments of hundreds of application servers. Vault lets developers focus on writing application-specific code without worrying about the details of how long polling is implemented. We have implemented Vault in two different Web frameworks.

1 Introduction

In its earliest days the Web supported only static content, but over the years it has evolved to support a variety of dynamic effects that allow Web pages to interact with users and update themselves incrementally on the fly. In the past most of these updates have been user-driven: they occurred in response to user interactions such as mouse clicks and keystrokes. However, in recent years more and more applications require *server push*, where Web pages update without any user involvement in response to events occurring remotely on a Web server or even other browsers. Server push is particularly important for collaboration applications where users want to be notified immediately when other users perform actions such as typing messages or modifying a shared document. Server push is also useful for a variety of monitoring applications.

The most common way to implement server push today is with a mechanism called *long polling*. Unfortunately, long polling is not easy to use. Existing implementations tend to be special-purpose and application-specific. They do not integrate well with Model-View-Controller (MVC) Web frameworks that are commonly used to build Web applications, often requiring separate servers just for long polling. Finally, it is challenging to create a long polling system that scales to handle large Web applications.

In this paper we describe an architecture for long polling called Vault, along with its implementation. Vault has three attractive properties:

- It integrates with MVC frameworks in a natural and simple fashion using a new construct called a *feed*, which is an extension of the model concept from MVC.
- It generalizes to support a variety of long polling applications (using different kinds of feeds), and to support multiple uses of long polling within a single Web page.
- It includes a scalable notification system that allows Vault to be used even for very large applications and that spares Web developers from having to worry about issues of distributed notification. There are two interesting architecture decisions in the design of the notification system. The first is that Vault separates notification from data storage, which makes it easy to create a scalable notifier and to incorporate a variety of data storage mechanisms into long polling. The second is that extraneous notifications are allowed in Vault, which helps simplify the implementation of the system and crash recovery.

Overall, Vault improves on existing work by making it easier to create Web applications that use long polling.

We have implemented Vault in two Web frameworks. We first built Vault in Fiz [10], a research framework, and then ported it to Django [3] to demonstrate the general applicability of Vault in MVC frameworks. This paper uses the Django implementation for examples, but the concepts and techniques could be used in most other MVC frameworks.

The remainder of this paper is organized as follows. We present background information on long polling and the problems it introduces in Section 2. Section 3 describes the major components of Vault and how they work together to implement long polling. Section 4 presents a few examples of feeds. Section 5 addresses the issue of long polling in a clustered environment with potentially thousands of application servers. Section 6 analyzes the performance of the Vault notification mechanism. We discuss integrating Vault with Web frameworks in Section 7. Section 8 describes the limitations of Vault, and Section 9 compares Vault with other approaches to long polling. Finally, Section 10 concludes.

2 Background

2.1 Long polling

The goal of long polling is to allow a Web server to initiate updates to existing Web pages at any time. The updates will reflect events occurring on the server, such as the arrival of a chat message, a change to a shared document, or a change in an instrument reading.

Unfortunately, the Web contains no mechanism for servers to initiate communication with browsers. All communication in the Web must be initiated by browsers: for example, a browser issues an HTTP request for a new Web page when the user clicks on a link, and Javascript running within a Web page can issue a request for additional information to update that page using mechanisms such as Ajax [8]. Web servers are generally *stateless*: once a server finishes processing a browser request it discards (almost) all information about that request. Although a server may retain a small amount of information about each active client (using *session* objects) the server typically doesn't retain the addresses of all its clients; even if it did there is no way for it to initiate a connection with the browser. A Web server can only respond to requests initiated by the browser.

Thus Web servers cannot update Web page content without browser assistance. A simple approach used by many early applications is polling: once a Web page is loaded, it issues Ajax requests to the server at regular intervals to check for updates. When an interesting event happens on the server it cannot immediately notify the browser; it must save information about the event until the next poll, at which point the Web page gets updated. Although polling can provide an appearance to the user much like true server push, it requires a trade-off between fidelity and efficiency. A short interval for polling provides high fidelity (events are reflected quickly on the browser) but wastes server resources and bandwidth responding to poll requests. A long interval for polling reduces overhead but may result in long delays before an event is reflected in the browser.

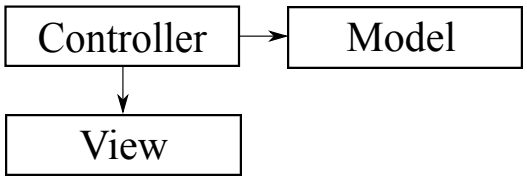

Figure 1: The Model-View-Controller pattern for Web applications. Each HTTP request is handled by a controller, which fetches data for the request from models and then invokes one or more views to render the Web page.

Long polling, also known as Comet [14] or reverse Ajax, is a variation on the polling approach that improves both fidelity and efficiency. The browser polls the server as described above, but if no updates are available the server does not complete the request. Instead it holds the Ajax request open for a period of time (typically tens of seconds). If an event occurs during this time then the server completes the request immediately and returns information about the event. If no event occurs for a long period of time the server eventually ends the request (or the browser times out the request). When this happens the browser immediately initiates a new long poll request. With long polling there is almost always an Ajax request active on the server, so events can be reflected rapidly to the browser. Since requests are held on the server for long periods of time, the overhead for initiating and completing requests is minimized.

However, long polling still has inefficiencies and complexities. It results in one open Ajax connection on the server for every active client, which can result in large numbers of open connections. Browsers typically limit the number of outstanding HTTP connections from a given page, which can complicate Web pages that wish to use long polling for several different elements. But the most challenging problem for long polling is that it does not integrate well with modern Web frameworks; these problems are introduced in the next section, and solving them is the topic of the rest of the paper.

2.2 Model-View-Controller Frameworks

Most of today's popular server-side Web frameworks are based on the Model-View-Controller (MVC) pattern [12] [13]. MVC divides Web applications into three kinds of components (see Figure 1): *models*, which manage the application's data; *views*, which render data into HTML and other forms required by the browser; and *controllers*, which provide glue between the other components as well as miscellaneous functions such as ensuring that users are logged in. When a request arrives at

the Web server, the framework dispatches it to a method in a controller based on the URL. For example, a request for the URL `/students/display/47` might be dispatched to the `display` method in the `Student` controller. The controller fetches appropriate data from one or more models (e.g., data for the student whose id is 47), then invokes a view to render an HTML Web page that incorporates the data.

Unfortunately, MVC frameworks were not designed with long polling in mind. As a result, it is difficult to use long polling in Web applications today. Most frameworks assume that requests finish quickly so they bind a request tightly to a thread: the thread is occupied until the request completes. Some frameworks have only a single thread, which means there can be only one active request at a time; this can result in deadlock, since an active long poll request can prevent the server from handling another request that would generate the event to complete the long poll. If a framework is multi-threaded, it can use one thread for each active long poll request while processing other requests with additional threads. However, the presence of large numbers of threads can lead to performance problems. Fortunately, some frameworks (such as Tomcat 7.0) have recently added mechanisms for detaching a request from its thread, so that long poll requests can be put to sleep efficiently without occupying threads.

Another problem with MVC frameworks is that they were not designed for the flow of control that occurs in long polling. In traditional Web applications requests are relatively independent: some requests read information and pass it to the browser while other requests use information from the browser to update data, but there is no direct interaction between requests. With long polling, requests interact: one request may cause an event that is of interest to one or more active long polls; the framework must provide a mechanism for managing these events and moving information between requests. Notifications become even more complicated in clustered Web servers where an action on one Web server may impact long poll requests on other servers in the cluster.

Because of these problems, applications that need long polling typically use special-purpose solutions today. In many cases long polling is handled by different Web servers than the main application, with a special (non-MVC) framework used for the long poll servers (see Section 9.3). The internal mechanisms used for long polling are often application-specific, so that each long polling application must be built from scratch. For example, some implementations of long polling tie the notification mechanism to a particular data model such as message channels, which requires the notification mechanism to be reimplemented if a different data model is used.

Our goal for Vault was to create an architecture for long polling that integrates naturally with existing MVC frameworks, generalizes to support a variety of long polling applications and encourage code reuse, and provides reusable solutions for many of the problems shared across long polling applications such as managing the long poll protocol and creating a scalable notification system.

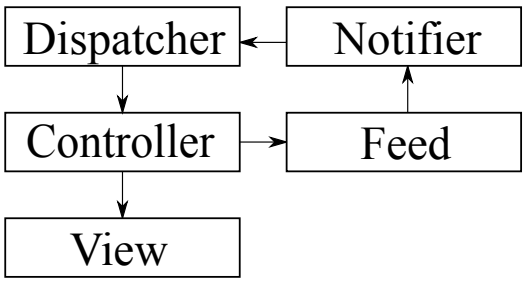

Figure 2: The architecture of Vault. Vault is similar to an MVC framework except models have been replaced with feeds and two new components have been added (the notifier and the dispatcher).

3 The Vault Architecture

Figure 2 shows the overall architecture of Vault. Vault extends an MVC framework with three additional elements: feeds, a notifier, and a dispatcher. A *feed* is similar to a model in a traditional MVC framework except that it provides extra capabilities for long polling. As with traditional models, a feed is responsible for managing a particular kind of data, such as a table in a database, a queue of messages, or an instrument reading. A traditional model answers the question *what is the current state of the data?* and also provides methods to modify the model's data, validate data using business logic, etc. A feed provides these same facilities, but in addition it contains methods to answer questions of the form *how has the data changed since the last time I asked?*. The new methods make it easy to create long polling applications. One of our goals for Vault is to make it as simple as possible to convert a model to a feed; we will illustrate this in Section 3.3.

One of the key issues for a long polling system is notification: when there is a change in the state of data (e.g. a value in a database is modified, or an instrument reading changes) there may be pending long poll requests that are interested in the change. The *notifier* is responsible for keeping track of long poll requests and waking them up when interesting events occur. Feeds tell the notifier which events are relevant for each long poll request, and they also tell the notifier when events have occurred; the notifier is responsible for matching interests with events.

logs.py

```
def show(request):
    logs = Logs.get_new_instances()
    return render_to_response("show.html", {"logs": logs})

def get_latest_logs(request):
    logs = Logs.get_new_instances()
    if logs.count() == 0:
        return
    return append_to_element("logs-div", "log-entries.html", {"logs": logs})
```

show.html

```
...
<% add_long_poll "logs/get_latest_logs" %>
<div id="logs-div">
    <% include "log-entries.html" %>
</div>
...
```

log-entries.html

```
<% for log in logs %>
    <p><%= log.type %>: <%= log.message %></p>
<% endfor %>
```

Figure 3: A simple application that uses Django and Vault to display a list of log entries. `logs.py` is a controller; the `show` method displays the overall page and `get_latest_logs` updates the page with new log entries during long-poll requests. `show.html` is a template that renders the main page, and `log-entries.html` is the template for rendering a list of log entries.

Vault notifications do not contain any data, they simply say that a change has occurred; it is up to feeds to verify exactly what data changed. The Vault notifier is scalable, in that it can support Web applications spanning many application servers: if a feed on one server announces an event to its local notifier, the notifier will propagate information about the event to all other servers that are interested in it. The notifier API is discussed in Section 3.2 and a scalable implementation is described in Section 5.

The third component of Vault is the long poll dispatcher. The dispatcher provides glue between Javascript running on Web pages, controllers, and the notifier. Among other things, it receives long poll requests and invokes controller methods at the right time(s) to complete those requests. Its behavior will be explained in Section 3.4.

Of these three components, feeds are the primary component visible to Web application developers. The dispatcher is almost completely invisible to developers, and the notifier is visible only to developers who create new feeds.

3.1 A Simple Example

This section illustrates how the Vault mechanism is used by a Web application developer (using Django as the framework), assuming that an appropriate feed already exists. Writing an application with Vault is similar to writing an application using MVC. Developers write controller methods that fetch data and use it to render a view. Although they must indicate to Vault that they would like to use long polling to update a page, most of the details of the long poll mechanism are handled invisibly by Vault.

The example page (see Figure 3) displays a list of log entries and uses long polling to update the page whenever a new log entry is added to a database. New log entries appear to be added to the page instantaneously. A user who opens the page will initially see a list of existing log entries, and new ones will show up when they are created. Log entries are assumed to be generated elsewhere.

The page (URL `/logs/show`) is generated by the `show` method. `show` has the same structure as normal MVC controller methods: it retrieves data from a model and then invokes a view to render a Web page using that data.

The page differs from traditional MVC pages in two ways. First, `show` invokes a new feed method called `get_new_instances` to retrieve its data; `get_new_instances` returns all of the current rows in the table and also makes arrangements for future notifications. Second, to arrange for automatic page updates, the view calls `add_long_poll`. This causes Vault to enable long polling for the page, and it indicates that

```
all() => [Record]
filter(condition) => [Record]
save() => void
get_new_instances(request) => [Record]
```

Figure 4: A few of the methods provided by `DatabaseFeed`. `get_new_instances` is the only method that would not be found in a regular model. `[Record]` indicates that the method returns an array of database records. The `request` object represents the state of the HTTP request currently being processed.

the method `logs/get_new_logs` should be invoked to update the page.

Vault arranges for the page to issue long-poll requests, and it invokes `get_latest_logs` during the requests. The purpose of this method is to update the page if new log entries have been created. `get_latest_logs` is similar to a method in an MVC framework for handling an Ajax request: it fetches data from a model, invokes a view to render that data, and returns it to the browser (`append_to_element` is a helper method that generates Javascript to append HTML rendered from a template to an element). However, it must ensure that it generates no output if there are no new log entries. This signals to Vault that the request is not complete yet. Vault then waits and invokes this method again when it thinks there may be new log entries.

The `DatabaseFeed` (see Figure 4) is used to determine if any new log entries have been created. Its `get_new_instances` method returns any new log entries that have been created since the last time the method was invoked for the current page, or an empty array if there are none. The first time it is invoked, it returns all existing log entries, as in the `show` method. All feeds have methods analogous to `get_new_instances`: these methods either return the latest data, if it has changed recently, or an indication that the data has not changed. Aside from this method, `DatabaseFeed` also has ordinary model methods such as `all` and `filter`, which return subsets of the data, and `save`, which writes new data to the database.

Vault automatically handles many of the details of long polling so that application developers do not have to deal with them. For example, the dispatcher automatically includes Javascript in the page to issue long-poll requests, the feed and notifier work together to determine when to invoke methods such as `get_latest_logs`, and the feed keeps track of which log entries have already been seen by the current page. These mechanisms will be described in upcoming sections.

```
create_interest(request, interest) => boolean
remove_interest(request, interest) => void
notify(interest) => void
```

Figure 5: The methods provided by the Vault notifier.

3.2 The Notifier

One of the most important elements of a long polling system is its notification mechanism, which allows long poll requests to find out when interesting events have occurred (such as a new row being added to a table). Most libraries for long polling combine notification with data storage, for example by building the long polling mechanism around a message-passing system. In contrast, Vault separates these two functions, with data storage managed by feeds and notification implemented by a separate notifier. This separation has several advantages. First, it allows a single notifier implementation to be shared by many feeds, which simplifies the development of new feeds. Second, it allows existing data storage mechanisms, such as relational databases, to be used in a long polling framework. Third, it allows the construction of a scalable notifier (described in Section 5) without having to build a scalable storage system as well.

The Vault notifier provides a general-purpose mechanism for keeping track of which long poll requests are interested in which events and notifying the dispatcher when those events occur (see Figure 5). Each event is described by a string called an *interest*. The `create_interest` method associates an interest with a particular HttpRequest (which represents the state of an HTTP request being processed by the server, usually a long poll request); the `remove_interest` method breaks the association, if one exists. The `notify` method will call the dispatcher to wake up all requests that have expressed interest in a particular event. As will be seen in Section 3.3, feeds are responsible for calling `create_interest` and `notify`. Interest names are chosen by feeds; they do not need to be unique, but the system will operate more efficiently if they are.

For Web applications that exist entirely on a single server machine the notifier implementation is quite simple, consisting of little more than a hash table to store all of the active interests. However, its implementation becomes more interesting for large Web sites where the sender and receiver of a notification may be on different server machines. Section 5 describes how a scalable distributed notifier can be implemented with the same API described by Figure 5.

Extraneous notifications are acceptable in Vault; they affect only the performance of the system, not its correctness. Extraneous notifications can happen in sev-

```
1  class DatabaseFeed:
2      def get_new_instances(self, request):
3          interest = "DB-" + self.model.name
4          possibly_ready = Notifier.create_interest(interest, request)
5          if not possibly_ready:
6              return []
7
8          old_largest_key = PageProperties.get(request, interest)
9          if old_largest_key == None:
10             old_largest_key = 0
11         current_largest_key = self.get_largest_primary_key()
12
13         PageProperties.set(request, interest, current_largest_key)
14
15         if current_largest_key > old_largest_key:
16             latest = self.filter(primary_key__greater_than=old_largest_key)
17             return latest
18         else:
19             return []
20
21     def on_db_save(table_name, instance, is_new_instance):
22         if is_new_instance:
23             Notifier.notify("DB-" + table_name)
```

Figure 6: A partial implementation of `DatabaseFeed`, showing the code necessary to implement the `get_new_instances` method. `get_new_instances` records the largest primary key seen for each distinct Web page and uses that information in future calls to determine whether new rows have been added to the table since the page was last updated. For brevity, code for the other feed methods is not shown and a few minor liberties have been taken with the Django API, such as the arguments to `on_db_save`.

eral ways. First, it is possible for different feeds to choose the same string for their interests. As a result, an event in either feed will end up notifying both interests. In addition, extraneous notifications can happen when recovering from crashes in the cluster notifier (see Section 5). If an extraneous notification happens in the example of Section 3.1 it will cause the controller method `get_latest_logs` to be invoked again, even though no new rows have been created. The `get_new_instances` method will return an empty array in, so `get_latest_logs` will generate no output and the dispatcher will delay for another notification.

Vault interests are similar to condition variables as used in monitor-style synchronization [9]: a notification means "the event you were waiting for *probably* occurred, but you should check to be certain". Vault interests can be thought of as a scalable and distributed implementation of condition variables, where each interest string corresponds to a distinct condition variable.

3.3 Implementing Feeds

A feed is a superset of a traditional MVC model, with two additional features. First, it must maintain enough state so that it can tell exactly which data has changed (if any) since the last time it was invoked. Second, it must work with the Notifier to make sure that long poll requests are reawakened when relevant events occur (such as the addition of a row to a database) . The paragraphs below discuss these issues in detail, using the `get_new_instances` method of `DatabaseFeed` for illustration (see Figure 6).

In order to detect data changes, feeds must maintain state about the data they have already seen. For example, `get_new_instances` does this by keeping track of the largest primary key that has been seen so far (`old_largest_key`). In each call, it compares the largest key seen so far with the largest primary key in the database (lines 8-11). If the latter is larger, the feed returns all the new rows; otherwise it returns an empty array. The type of state will very from feed to feed, but a few possibilities are a primary key of a table, the contents of a row, or a timestamp.

Because a user many have several pages open, state such as the largest key seen must be page specific. For example, a user might have the same URL opened in two different browser tabs; they must each update when new rows are created. Vault accomplishes this through the use of *page properties*, which are key-value pairs stored on the server but associated with a particular Web page. Page properties can be thought of as session data at the page level instead of the user or browser level. If a page property is set during one request associated with a particular page (such as when `get_new_instances` is invoked by the `show` method of Figure 3 during the initial page rendering), it will be visible and modifiable

in any future requests associated with the same page (such as when `get_new_instances` is invoked by `get_latest_logs` during a subsequent long poll request). The `DatabaseFeed` uses page properties to store the largest primary key seen by the current page. For details on how page properties are implemented, see [11]. The Fiz implementation of Vault uses an existing page property mechanism provided by Fiz; Django does not include page properties, so we implemented page properties as part of Vault.

The second major responsibility for feeds is to communicate with the notifier. This involves two actions: notification and expressing interest. Notification occurs whenever any operation that modifies data is performed, such as creating a new row or modifying an existing row. For example, `on_db_save`, which is invoked by Django after any row in the table has been modified, calls `Notifier.notify` (line 23) to wake up requests interested in additions to the table. Expressing an interest occurs whenever any feed method is invoked. `get_new_instances` invokes `Notifier.create_interest` to arrange for notification if/when new rows are added to the table. The interest must be created at the beginning of the method, *before* checking to see whether there are any new instances: if the interest were created afterward, there would be a race condition where a new row could be added by a different thread after the check but before the interest was created, in which case the notification would be missed.

The `DatabaseFeed` uses interests of the form "DB-*ttt*", where *ttt* is the name of the table. This ensures that only requests interested in that table are notified when the new rows are added to the table.

Note that `Notifier.remove_interest` is not invoked in Figure 6. The feed intentionally allows the interest to persist across long poll requests, so that notifications occurring between requests will be noticed. Old interests are eventually garbage collected by the notifier.

Feeds can take advantage of an additional feature of the notification mechanism in order to eliminate unnecessary work. In many cases the notifier has enough information to tell the feed that there is no chance that the feed's data has changed (e.g., if the interest has existed since the last call to the feed and there has been no intervening notification). In this case `create_interest` returns false and `get_new_instances` can return immediately without even checking its data. This optimization often prevents a costly operation, such as reading from disk or querying a remote server. Furthermore, most of the time when the feed is invoked there will have been no change (for example, the first check for each long poll request is likely to fail).

All feeds have methods with the same general structure as `get_new_instances`. In each case the method must first create one or more interests, then check to see if relevant information has changed. The exact checks may vary from feed to feed, but they will all record information using page properties in order to detect when the information on the page becomes out of date. For example, a feed that implements messages with serial numbers might store the serial number of the last message seen by this page and compare it with the serial number of the most recent message.

3.4 The Dispatcher

The Vault dispatcher hides the details of the long poll protocol and supervises the execution of long poll requests. The dispatcher is invisible to Web application developers except for its `add_long_poll` method, which is invoked by views to include long polling in a Web page (see Figure 3). When `add_long_poll` is invoked the dispatcher adds Javascript for long polling to the current Web page. This Javascript will start up as soon as the page has loaded and initiate an Ajax request for long polling back to the server. When the long poll request completes the Javascript will process its results and then immediately initiate another long poll request. Only one long poll request is outstanding at a time, no matter how many long poll methods have been declared for the page.

Vault arranges for long poll requests to be handled by the dispatcher when they arrive on the Web server. In the Django implementation this is done by running Vault as middleware that checks each request for a special long poll URL, and if present sends the request to the dispatcher. The dispatcher finds all of the long poll methods for the current Web page (`add_long_poll` records this information using the page property mechanism described in Section 3.3) and invokes all of them in turn. If output was generated by any of the long poll methods then the dispatcher returns the output to the browser and completes the request. If none of the methods generated any output then the dispatcher puts the request to sleep until it receives a notification for this request from the notifier. Once a notification is received the dispatcher invokes all of the long poll methods again; once again, it either returns (if output has been generated) or puts the request to sleep.

In normal use it is unlikely that the first invocation of the long poll methods during a request will generate any output, since only a short time has elapsed since the last invocation of those methods. However, it is important to invoke all of the methods, which in turn invoke feeds, in order to ensure that interests have been created for all of the relevant events.

```
create_channel(channel_name) => void
create_user(user_name) => void
get_new_messages(request, user_name) => [Message]
subscribe(user_name, channel_name) => void
post_message(channel_name, message) => void
```

Figure 7: The methods provided by our example Messaging feed .

4 Feed examples

In order to evaluate the architecture of Vault we have implemented two feeds; these feeds illustrate the approaches that we think will be most common in actual Web applications.

The first feed is the `DatabaseFeed` discussed earlier in 3. We implemented two feed methods, `get_new_instances` and `get_modified_instance`, and designed a third, `get_attribute_changes`.

`get_new_instances` has been discussed earlier in Section 3.1. Briefly, `get_new_instances` returns new rows from a database. It uses interests of the form `DB-`*ttt*, where *ttt* is the name of the table. `get_new_instances` detects new rows by saving the highest primary key seen in the past for any row in the table and comparing this with the current highest key in future calls (it assumes that primary keys are chosen in ascending order).

`get_modified_instance` allows a Web page to observe changes to a particular row; it returns the latest value of the row, or `None` if the row has not changed since the last call for this row on the current Web page. It uses interests of the form `DB-`*ttt*`-`*rrr*, where *ttt* is the name of the table and *rrr* is the primary key of the row under observation; all methods in the model that modify existing rows must notify the interest for that row. The implementation of `get_modified_instance` is similar to that of `get_new_instances` in Figure 6: it records the last value of the row seen by the current Web page and compares this against the current contents of the row in future calls.

Although these two methods are simple to implement, one can imagine more complicated methods that are not as easy to implement in Vault. One example is a method to monitor changes to a column (or *attribute* in model terminology), called `get_attribute_changes`. If any values in the column have changed since the last invocation for this table in the current Web page the primary keys for the affected rows are returned. It is difficult for the feed to tell whether a column has changed recently unless it records a copy of the entire column, which would be impractical for large tables. One solution is to create an auxiliary data structure to keep track of changes. A new table can be added to the database with each row representing one change to the given column: the auxiliary row contains the primary key of the row in the main table that changed. This table will need to be updated any time the model modifies a field in the column under observation: once the auxiliary table exists, an approach similar to `get_new_instances` can be used to implement `get_attribute_changes`. The method stores the largest primary key (of the auxiliary table) used by previous method calls and queries the table for all rows with a larger primary key.

The `DatabaseFeed` described above introduces overhead to notify various interests when relevant changes occur in the database; in some cases a single modification to the database might require notification of several different interests (e.g., modifying a row would notify both that the row was modified and any column changes). However, as will be seen in Section 6, notifying an interest is considerably faster than typical database operations. Thus we think a variety of interesting database feeds can be implemented with minimal overhead.

Our second feed implements a publish-subscribe messaging protocol (see Figure 7) somewhat like Bayeux [1] (see Section 9.1). It contains a collection of channels on which text messages can be posted, and each user can subscribe to channels that are of interest to that user. The message feed provides a method `receive_messages` that is analogous to `get_new_instances` in `DatabaseFeed`: it returns all the messages that have arrived for a given user since the last call to this method for the current Web page. The feed uses a separate interest for each user with names of the form `message-`*uuu*, where *uuu* is an identifier for a particular user. `receive_messages` creates an interest for the current user, and `post_message` notifies all of the users that have subscribed to the channel on which the message is posted. In order to tell whether there are new messages for a user, the feed serializes all of the messages for each user and records the serial number of the most recent message that has been delivered to each Web page.

We have created a simple application using the message feed that provides Web pages for posting messages, subscribing to channels, and observing incoming traffic for a particular user.

We believe that many feed implementations are

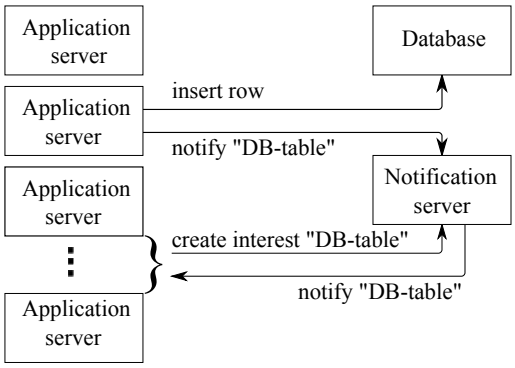

Figure 8: Cluster architecture. Notifications must go through notification servers where they are relayed to servers with matching interests.

likely to be similar to the ones above where they either (a) observe one or more individual values (or rows) like `get_new_instances` and `get_modified_instance`, (b) observe a sequence of messages or changes like `receive_messages` and `get_new_instances`, or (c) make more complex observations like `get_attribute_changes`, in which case they will create an auxiliary data structure in the form of a sequence.

5 Cluster Support and Scalability

Large-scale Web applications must use clusters of application servers to handle the required load; the cluster size can vary from a few servers to thousands. Typically, any application server is capable of handling any client request (though the networking infrastructure is often configured to deliver requests for a particular browser to the same server whenever possible); the data for the application is kept in storage that is shared among all of the servers, such as a database server.

Introducing long polling to a cluster environment complicates the notification mechanism because events originating on one server may need to propagate to other application servers. Many events will be of interest to only a single server (or may have no interests at all), while other events may be of interest to nearly every server.

One possible solution would be to broadcast all notifications to all application servers. This approach behaves correctly but does not scale well since every application server must process every notification: the notification workload of each server will increase as the total number of servers increases and the system will eventually reach a point where the servers are spending all of their time handling irrelevant notifications.

For Vault we implemented a more scalable solution using a separate *notification server* (see Figure 8). When an interest is created on a particular application server, that information gets forwarded to the notification server so that it knows which application servers care about which interests. When an interest is notified on a particular application server, the notification is forwarded to the notification server, which in turn notifies any other application servers that care about the interest. With this approach only the application servers that care about a particular interest are involved when that interest is notified. The notification server is it similar to the local notifier, but it works with interested servers, not interested requests. In particular, both have the same basic API (create interest, remove interest, and notify).

For large Web applications it may not be possible for a single notification server to handle all of the notification traffic. In this case multiple notification servers can be used, with each notification server handling a subset of all the interests. Local notifiers can use a consistent hash function [15] on each interest to determine which notification server to send it to.

One of the advantages of the API we have chosen for notification is that it distributes naturally as described above. Furthermore, the decision to allow extraneous notifications simplifies crash recovery and several other management issues for notification servers.

Crash recovery is simple in Vault due to these properties. If a notification server crashes, a new notification server can be started as its replacement. Each of the application servers can deal with the crash by first recreating all of its local interests on the new server and then notifying all of those interests locally (just in case a notification occurred while the original server was down). Most of the notifications will be extraneous but the feeds will detect that nothing has actually changed. This behavior is correct but may be slow depending on the number of interests.

There is an additional crash recovery issue in Vault, which occurs if a feed crashes after updating its data but before notifying the notification server. In Figure 6, this could happen if the server crashes on line 22. If an application server crashes, there is no way to tell whether it was about to send notifications. To avoid losing notifications, every existing interest must be notified whenever any application server crashes. Another alternative is to use a two-phase notification, but the associated overheard makes the first alternative a more attractive option.

6 Benchmarks

We ran a series of experiments to measure the performance and scalability of the Vault notification mechanism. Our intent was to determine how many servers are needed to handle a given load. We ran our experiments

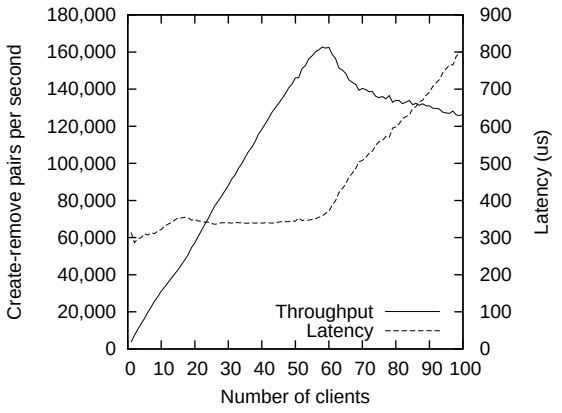

Figure 9: Performance measurements of the notification server for creating and removing interests. Each load generator repeatedly messages the notification server to create an interest and then remove the interest. Throughput measures the number of create-remove pairs handled by the notification server per second and latency measures the time from the start of creating the interest to the end of removing the interest.

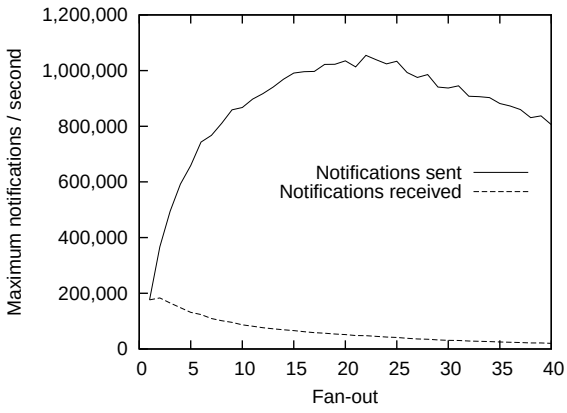

Figure 10: Performance of the notification server for processing notifications. Fan-out indicates the number of application servers that must ultimately receive each notification received by the notification server. The chart shows the maximum rate at which the server can receive notifications (measured by varying the number of issuing clients) and the maximum rate at which it sends them to individual application servers.

on 40 identical servers. Each one has a Xeon X3470 (4x2.93 GHz cores), 24GB of 800MHz DDR3 SDRAM, and an Intel e1000 NIC. For some benchmarks, the number of application servers is larger than the number of machines, in which case some machines run multiple processes (the impact on the measurements is negligible because the notification server bottlenecks before any of the application servers do).

Our first experiment was to determine the number of create-interest and remove-interest operations each notification server can handle per second. Creates are performed by an application server when a long poll creates an interest locally and the application server has not yet relayed the interest to the notification server. Removing an interest occurs when the application server determines that there are no longer any interested long polls. Creating and removing an interest may happen as frequently as several times per page request so they should be fast.

The experiment measured the total number create-remove pairs handled by the notification server per second and the latency from the start of the create to the end of the remove, as shown in Figure 9. Each load generator ran a loop sending a create message and waiting for the response and then sending a remove without waiting for the response. Our results show a linear increase in throughput as the number of load generators is increased. The throughput maximum is around 160,000 create-remove pairs per second, but drops about 10% after the maximum is reached. Latency remains roughly constant until the throughput limit is reached, at which point the latency begins to grow linearly.

Our second experiment measured the cost for the notification server to process events and relay them to interested application servers. We measured the maximum throughput for the notification server with different fan-outs per notification, as seen in Figure 10. The figure shows the total number of notifications received by the notification server per second and the number of notifications it passes on to application servers. The notifications received starts at just under 200,000 per second with one interested server, then drops roughly linearly as the fan-out increases. The total number of notifications sent rises as fan-out increases, peaking at around 1,000,000 notifications per second with a fan-out of 15-20. As the number of servers continues to increase, the total number of notifications sent drops to around 800,000 per second.

Figure 10 does not contain a data point for interests with a fan-out of zero. For notifications where no server is interested, the notification server can process about 2,000,000 per second. Notification servers may get many notifications for which there are no interested servers so it is important for them to be quick.

7 Framework integration

Although the principles of Vault are applicable to any MVC framework, integrating Vault with a Web framework requires some basic level of support from the framework. We have identified two such requirements: the ability to decouple requests from their threads, and page properties.

Many existing frameworks do not currently allow re-

quests to be put to sleep. For example, Django does not support this, so we modified the Django Web server to implement request detaching. Our initial implementation in Fiz was easier because Fiz is based on Apache Tomcat 7.0, which provides a mechanism to *detach* a request from the thread that is executing it, so that the thread can return without ending the request. The Vault dispatcher saves information about the detached request and then returns so its thread can process other requests (one of which might generate a notification needed by the detached request). When a notification occurs the dispatcher retrieves the detached request and continues processing it as described above.

If a framework does not allow requests to be detached but does support multi-threading, then the dispatcher can use synchronization mechanisms to put the long poll thread to sleep and wake it up later when a notification occurs. However, this means that each waiting long poll request occupies a thread, which could result in hundreds or thousands of sleeping threads for a busy Web server. Unfortunately, many systems suffer performance degradation when large numbers of threads are sleeping, which could limit the viability of the approach.

If a framework only allows a single active request and does not support request detaching, then the framework cannot support long polling; applications will have to use a traditional polling approach. However, Vault could still be used in such a framework and most of Vault's benefits would still apply.

Secondly, a framework must provide page properties or some equivalent mechanism to associate data with a particular Web page and make that data available in future requests emanating from the page. If a framework does not support page properties, they can be built into Vault on top of existing session facilities (a separate object for each page can be created within the session, identified with a unique identifier that is recorded in the page and returned in long poll requests).

8 Limitations

We are aware of two limitations in the Vault architecture. The first limitation is that some styles of notification do not map immediately onto interest strings, such as `get_attribute_changes` in Section 4. We believe these situations can be handled with the creation of auxiliary data structures as described in Section 4, but the creation of these structures may affect the complexity and efficiency of the application. There may be some applications where a better solution could be achieved with a richer data model built into the notification mechanism.

The second limitation of Vault is that the crash recovery mechanism can produce large numbers of extraneous notifications, as described in Section 5. We have experimented with alternatives that reduce the extraneous notifications, but they result in extra work during normal operation, which seems worse than the current overheads during crash recovery.

9 Related Work

9.1 CometD/Bayeux

CometD [2] is the most well known long polling system. It is an implementation of Bayeux [1], a protocol intended primarily for bidirectional interactions between Web clients. Bayeux communication is done through channels and is based on a publish/subscribe model. Clients can post messages to a channel, and other clients subscribed to the channel receive the messages.

CometD works well for publish/subscribe communication but does not generalize to other uses (for example, there is no way to implement database notifications using CometD). Since CometD is a stand-alone framework, it does not work with existing MVC frameworks. By tying the protocol to the message-based model used, CometD limits the ability of developers to write applications using other data models. In addition, the current implementation of CometD does not seem to have a mature cluster implementation.

9.2 WebSockets

The WebSocket [7] API is a proposed addition to Web browsers that allows for bi-directional communication between the browser and the server. Although WebSockets are often seen as a "solution" to long polling, they do not fix any of the major issues associated with long polling. Notification, scalability, and integration with MVC will still be issues with WebSockets. WebSockets only fix some minor inconveniences such as the need to hold Ajax requests at the server and some of the associated complexity of long polling. If widely adopted (which is likely when HTML5 becomes supported by all browsers) WebSockets could become the new transport mechanism for Vault, in which case Vault would not have to worry about Ajax requests timing out or the need to wait for an Ajax request to return from the browser to send another update.

9.3 Event-based Web Frameworks

Due to the difficulty of using long polling in traditional Web frameworks, event-based frameworks have become popular for handling long polling. This is typically done by running two Web frameworks side-by-side, one to handle normal requests and an event-based one to handle long polls, with the two communicating though some

backend channel. This approach is easier than trying to include long polling in existing MVC frameworks but is not as clean as keeping all application logic in one framework.

Event-based Web frameworks differ from other frameworks because they only run one thread at a time, eliminating the thread issues traditional MVC frameworks have when implementing long polling. Since there is only one thread, it is important that it does not block. If an expensive operation is performed, such as disk or network access, a callback is specified by the caller and the server stops processing the current request and starts processing a different request. When the operation finishes, the server runs the callback, passing in values generated by the operation. This style of programming makes it easy to resume a request at a later time as is required with Vault. Node.js [5] is an javascript event-based IO library used to write server-side javascript applications. Tornado [6] is a similar Web server for Python, based on FriendFeed [4].

10 Conclusion

Long polling allows for new interactive Web applications that respond quickly to external events. Existing implementations, however, have not been general or scalable, and they require too much glue code for application developers. We have presented a modification of the MVC pattern that allows developers to write their applications in a familiar style without needing to know about the details of how long polling is implemented. Furthermore, Vault makes it easy to develop a variety of feeds that can be reused in many different applications. Internally, the Vault implementation handles long polling in a scalable way using a distributed notifier to minimize the amount of work that is required per request.

We hope that the Vault architecture will be implemented in a variety of mainstream frameworks in order to encourage the development of interesting applications based on long polling.

11 Acknowledgments

This research was supported by the National Science Foundation under Grant No. 0963859. Thanks to Tomer London, Bobby Johnson, and anonymous program committee members for reviewing various drafts of this paper.

References

[1] Bayeux protocol. `http://svn.cometd.com/trunk/bayeux/bayeux.html`.

[2] Cometd homepage. `http://cometd.org/`.

[3] Django. `http://www.djangoproject.com/`.

[4] Friendfeed homepage. `http://friendfeed.com/`.

[5] Node.js homepage. `http://nodejs.org/`.

[6] Tornado web server homepage. `http://nodejs.org/`.

[7] Web socket api. `http://dev.w3.org/html5/websockets/`.

[8] GARRETT, J. J. Ajax: a new approach to web applications, February 2005. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`.

[9] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in mesa. *Commun. ACM 23* (February 1980), 105–117.

[10] OUSTERHOUT, J. Fiz: A Component Framework for Web Applications. Stanford CSD Technical Report. `http://www.stanford.edu/~ouster/cgibin/papers/fiz.pdf`, 2009.

[11] OUSTERHOUT, J., AND STRATMANN, E. Managing state for ajax-driven web components. *USENIX Conference on Web Application Development* (June 2010), 73–85.

[12] REENSKAUG, T. Models-views-controllers. Xerox PARC technical note `http://heim.ifi.uio.no/~trygver/mvc-1/1979-05-MVC.pdf`, May 1979.

[13] REENSKAUG, T. Thing-model-view-editor. Xerox PARC technical note `http://heim.ifi.uio.no/~trygver/mvc-2/1979-12-MVC.pdf`, May 1979.

[14] RUSSEL, A. Comet: Low latency data for the browser, March 2006. `http://alex.dojotoolkit.org/2006/03/comet-lowlatency-data-for-the-browser/`.

[15] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11* (February 2003), 17–32.