HTTP Mailbox - Asynchronous RESTful Communication

Sawood Alam
Dept. of Computer Science
Old Dominion University
Norfolk, Virginia, USA
salam@cs.odu.edu

Charles L. Cartledge
Dept. of Computer Science
Old Dominion University
Norfolk, Virginia, USA
ccartled@cs.odu.edu

Michael L. Nelson
Dept. of Computer Science
Old Dominion University
Norfolk, Virginia, USA
mln@cs.odu.edu

ABSTRACT
We describe HTTP Mailbox, a mechanism to enable RESTful HTTP communication in an asynchronous mode with a full range of HTTP methods otherwise unavailable to standard clients and servers. HTTP Mailbox allows for broadcast and multicast semantics via HTTP. We evaluate a reference implementation using ApacheBench (a server stress testing tool) demonstrating high throughput (on 1,000 concurrent requests) and a systemic error rate of 0.01%. Finally, we demonstrate our HTTP Mailbox implementation in a human assisted web preservation application called “Preserve Me!”.

Categories and Subject Descriptors
H.3.5 [Information Storage and Retrieval]: Online Information Services

General Terms
Design, Implementation, Evaluation

Keywords
HTTP Mailbox, Messaging, REST, Asynchronous, Linda

1. INTRODUCTION
Alice wanted to keep track of her tasks and maintain a to-do list. She found Bob’s shared, hosted task manager service. She created her initial tasks list (Table 1) and started working on the highest priority task. Once it was finished, she wanted to mark that task as done. Hence, Alice made an HTTP PATCH request [12] to the specific task URI on Bob’s server to modify the task partially. Unfortunately the server was down so this communication failed. Alice tried again after some time when the server was up but got a 501 Not Implemented HTTP Response [15] from the server. Alice talks to Bob regarding this issue and Bob replied that there is a non-RESTful way of doing this on the task server. They wished there was an extra layer of indirection to provide a RESTful interface to Bob’s server, even if indirectly.

Table 1: Alice’s tasks
| ID | Description  | Priority | Status |
|----|--------------|----------|--------|
| 1  | Write a paper. | HIGH     | Pending |
| 2  | Go on vacation. | LOW      | Pending |

HTTP Mailbox provides a layer of indirection. It allows sending any HTTP message (request or response), encapsulated in the message body to a URI relative to the HTTP Mailbox service using an HTTP POST request. Resulting messages can be retrieved by making an HTTP GET request to the HTTP Mailbox. Multiple HTTP messages to the same recipient can be pipelined in a single HTTP POST request. The HTTP Mailbox also provides multicast and broadcast messaging capabilities, an enhancement not possible using HTTP.

In past years, general web services used only the GET and POST methods of the HTTP protocol while several other HTTP methods like PUT, PATCH, and DELETE were rarely used. Until recently, the Web was mainly accessed by humans using web browsers and clicking on hyperlinks or submitting HTML forms. Clicking on a link is always a GET request while HTML forms only allow GET and POST methods [39, 40]. Recently, several web frameworks/libraries (like Ruby on Rails [38], CakePHP [6], Django [21], and .NET [32]) have started supporting RESTful web services through APIs. To support extra HTTP methods in browsers, these frameworks have used hidden HTML form fields as a workaround to convey the desired HTTP method to the server application. In such cases, the web server is unaware of the intended HTTP method because it receives the request as POST. A middleware between the web server and the application may override the HTTP method based on special hidden form field values. On one hand, this limitation is only with HTML and not Ajax requests. On the other hand, Ajax requests suffer from same-origin policy because support for Cross Origin Resource Sharing (CORS) is in the working draft of XMLHttpRequest [3]. While modern web browsers have recently started supporting cross-origin Ajax requests [22], this feature is not available in old browsers.

Unavailability of the servers is another factor that affects the communication. Because of the stateless and synchronous nature of HTTP, a client must wait for the server to be available to perform the task and respond to the request. By introducing HTTP Mailbox as another layer of indirection, we can address these issues.

2. BACKGROUND

2.1 REST
REpresentational State Transfer (REST) [14, 16] is a software architecture for large scale distributed system which has emerged as the preeminent design pattern. It utilizes
existing HTTP methods to generalize the interfaces of the web service by mapping resource actions like Create, Read, Update, and Delete (CRUD) [4] to corresponding HTTP methods POST, GET, PUT, and DELETE respectively. Remote Procedure Call (RPC) on the other hand encourages application designers to define their own application specific methods. A typical implementation of RPC on the Web is Simple Object Access Protocol (SOAP) [20] that allows querying available procedures and associates arguments on a remote server then a client can invoke those procedures remotely using XML as the medium of exchange.

If Bob’s tasks server on example.com was REST compliant, then after completing the first task Alice could have made an HTTP PATCH request to the task URI to mark it done as Code 1 (Lines: 1-6) and gotten the modified task resource in response as Code 1 (Lines: 8-12). It is not mandatory to return an entity body in the response to a PATCH request but in our example, we will assume that server will send the updated resource in the response. Media types text/task and text/task-patch are not defined, these are used here for illustration purpose only.

| Method | LAMP | HTML | Ajax | DMOZ |
|--------|------|------|------|------|
| GET    | Default Support | Link, Form | Yes | 100% |
| POST   | Default Support | Form | Yes | 40.3% |
| PUT    | Extra Config. | None | Yes | 1.7% |
| DELETE | Extra Config. | None | Yes | 1.8% |
| PATCH  | Extra Config. | None | Yes | 1.3% |

### 2.2 Linda

Linda [7] is a model based on generative communications [19] to facilitate distributed computing by sharing objects (e.g., data, computation requests and computation results) called tuples in a shared virtual memory called tuplespace. Processes query the tuplespace based on some criteria and perform a destructive or non-destructive read. Once the result of the process is ready, it is written back to the tuplespace where it can be picked up by another process.

Linda provides means for asynchronous (time-uncoupled) communication in which sender and recipient(s) do not need to meet in time. It also facilitates space-uncoupling as the sender and the recipient(s) do not need to know the identities of each other. This allows criteria based group messaging.

Linda has four basic operations or functions defined as:

- “in” – a destructive read,
- “rd” – a non-destructive read,
- “out” – producing a tuple, and
- “eval” – creating a process to evaluate a tuple and producing a result tuple if applicable.

Now, assume that a client application on Alice’s machine is communicating with Bob’s task manager process via a shared tuplespace using the Linda model. To mark the first task completed, Alice’s client may perform an out function to generate a tuple in the tuplespace for processing by Bob’s service when available.

```plaintext
out("task", 1, "Done")
```

This means create a tuple for task with id 1 to mark it done. This tuple will remain in the tuplespace until Bob’s service (or any other process) performs a destructive read using in function.

```plaintext
in("task", 1, ?id, ?status)
```

This read query using the in function will match the Alice’s tuple of “task”, assign “1” to “id” and “Done” to “status”, and remove it from the tuplespace.

Bob’s service then can create a live/active tuple using the “eval” function to create a new process for marking the task with id 1 as done and update the tasks table to reflect the changes permanently. Bob’s service may also wish to keep log of the changes.

```plaintext
eval("log", 1, changedStatus("Done"))
```

1. Linux, Apache, MySQL, and PHP, Perl or Python.
In this case, output of the live tuple will result in a passive tuple after the “eval” function is done, that can be stored in the tuplespace.

\[
\text{rd("log", ?id, ?status)}
\]

This log tuple can be read using “rd” function several times without removing it from the tuplespace by Alice’s client, Bob’s server or any other entity that has access to the tuplespace. If Bob’s service does not want to remove it from the tuplespace then a similar “rd” function can be used.

We took the simplicity of this model and considered implementing it on the Web scale to store and forward HTTP messages (requests and responses). Linda is a pre-web model mainly designed to work in a distributed system (not as large as the Web) where trusted processes share a common memory. Any process can write any tuple in the tuplespace independently and any process can destroy any tuple from the tuplespace. To implement it on the open Web as a distributed system, we must consider the scale of the Web and aspects of security and authenticity. Unlike a closed small distributed system, the Web is not trusted.

3. RELATED WORK

3.1 Relay HTTP

The Relay HTTP draft specification [18] describes a way to overcome the CORS restriction posed by JavaScript in Ajax requests. A proxy service is built on the same domain to relay/replay HTTP requests between client and remote server. It uses message/http and application/http MIME types defined by [15] for tunneling HTTP traffic over HTTP. It requires additional setup on the Web server to host the proxy server.

Suppose that Alice wants to add a tasks block in her organization’s website example.org while still utilizing the services of Bob’s task manager hosted at example.com. She will fail to get data from or post data to the Bob’s server using Ajax, because of the cross-origin restriction posed by JavaScript. Modern Web browsers which support CORS require additional headers from the server. But if she does not have control of Bob’s server and if Bob’s server does not already support CORS, she will not be able to get the tasks data from Bob’s server. As a workaround, she may add an iframe in her website and embed Bob’s tasks manager web page but she will not have control of the design of the embedded web page.

To overcome her client side restriction, she might set up a Relay HTTP proxy server under her organization’s domain name (example.org) to delegate all cross origin requests to the proxy server to replay them on Bob’s server and get the response as if it came from the same domain.

Using Relay HTTP, Alice makes a POST request which encapsulates the desired PATCH request as an entity to the proxy service hosted under her organization’s domain hence avoiding any client side limitations as illustrated in Code 2 (Lines: 1-11). Proxy service then replays the encapsulated message/http entity Code 2 (Lines: 6-11) and forwards the response back to the client as Code 2 (Lines: 13-17). But Relay HTTP still can not solve the server side limitations.

Also, it is a synchronous system hence the client, the relay/proxy server, and the remote server must all meet in time.

**Code 2: Relay HTTP Communication**

```
> POST /proxy/example.com HTTP/1.1
> PATCH /tasks/1 HTTP/1.1
> Status=Done
> Content-Type: text/task-patch
> Host: example.org

< Content-Type: text/task
> Host: example.com
> Content-Length: 11

< (Done) [HIGH] Write a paper.
```

3.2 Bleeps

Bleeps is a live messaging system that is inspired by tweets. It uses Push style communication [5] to broadcast small messages using relay channels. Anyone can subscribe to one or more such channels to receive live message feeds. We explored Bleeps for the ResourceSync project [51, 27]. Bleeps support hashtags and mentions for discovery and searching. It can be configured to support a variety of message formats for parsing message attributes easily using a language identifier. Messages are pushed to various channels for broadcasting which can be captured by consumer applications or other services. Bleeps are intended to be compact in length, preferably one-liners but structured enough to make the parsing easy according to the attached language descriptor. An example bleep looks like this:

```
from=alice to=http://example.com/tasks/1 change status #done #bob $task
```

In this example, “$task” at the end of the message is the language descriptor which defines the template for the message. Fields “from” and “to” can be used to query the message store. Similarly, “#done” hashtag is there to help grouping the messages with the same status. Bob is being mentioned with the help of “@bob” which will cause the message to appear in Bob’s stream. The remaining free text is the message which can also be a URL of a long message hosted elsewhere to keep the size of the message small. The message format is completely up to the attached language descriptor which can be defined by anyone.

3.3 Enterprise Messaging System

Apache Qpid [47] is an implementation of platform agnostic Advanced Message Queuing Protocol (AMQP) [36]. Java Message Service (JMS) [44] defines reliable enterprise messaging standard. It is an integral part of J2EE platform. These messaging systems allow various modes of digital communication including point-to-point, peer-to-peer, pub-sub and other forms of individual and group messaging.

JMS is limiting as it is only for Java applications while Apache Qpid has servers (also called Message Brokers) written in C++ and Java, along with clients for C++, Java JMS, .Net, Python, and Ruby. However there is no easy way to interact with these messaging services using a web browser. There are plugins available for RabbitMQ [52] (a message broker implementation for AMQP) that enable web
communication (e.g., RabbitMQ-Web-Stomp [29] that utilizes STOMP [43] protocol and WebSockets [13] to enable browser based interaction with RabbitMQ server).

4. HTTP MAILBOX MESSAGING

HTTP Mailbox messaging is a fusion of Linda style open access message storage and a traditional email system using HTTP as transport to embrace REST style asynchronous HTTP communication on the open Web. An HTTP Mailbox serves as a Linda style tuplespace for HTTP Messages.

In HTTP Mailbox messaging, HTTP requests are encapsulated inside another HTTP Message entity to form an envelope request. A client makes an HTTP POST Request to the HTTP Mailbox irrespective of the method of the encapsulated HTTP message. HTTP Mailbox then stores the encapsulated HTTP Request along with various message metadata in a persistent storage. Later, to retrieve those stored messages, a client makes an HTTP GET request to the HTTP Mailbox. Figure 1 shows a typical HTTP Request and Response cycle. Figure 2 illustrates how the same objective can be achieved using HTTP Mailbox while avoiding some of the issues of HTTP communication such as client or server side limitations and time coupling.

4.1 Linda in HTTP

One of the major advantages of HTTP Mailbox messaging is making HTTP communication asynchronous so that both the parties involved in the communication (typically known as “client” and “server”) are time-uncoupled and do not need to meet in time for a successful HTTP communication (or a complete “request” and “response” cycle).

This asynchronous nature of communication is a good fit when a response from the recipient(s) is not necessary or not immediately needed. Hence we borrowed the “store and forward” model from Linda and transform it into a form that is suitable in HTTP environment on the scale of the Web. A URI or any other identifier of the recipient(s) can be used to query messages from the distributed message store similar to the expressions used in “rd” and “in” functions of Linda to query the tuplespace.

The “in” function of Linda may be redefined as a soft-delete in the HTTP environment. We may not want to allow true deletion of messages because of the lack of trust on open Web and authentication challenges. Instead, flagging messages as deleted (and keeping a history of actions) may be a better choice because storage is not as limited as in case of pure Linda shared memory.

The “out” function may exist without any modification and returning the message as many times and to as many clients as requested repeatedly. To facilitate additional functionalities, an access log may also be maintained.

The “rd” function of Linda may refer to the action of preparing desired HTTP Request by a client and encapsulating it in another HTTP Request to send it to the message store.

The “eval” function of Linda may refer to the action of unpacking a stored HTTP request by a server, performing the desired task and writing the HTTP response in to the message store if necessary.

4.2 HTTP Message

“HTTP Request” and “HTTP Response” [15] both translate to a unified term “HTTP Mailbox Message”. In order to complete the “HTTP Request” or “HTTP Response” transaction, both require a complete HTTP Mailbox messaging lifecycle.

From the HTTP Mailbox perspective, the restrictive terms “client” and “server” posed by “HTTP Request” (from client to server) and “HTTP Response” (from server to client) have disappeared and been replaced by the general terms “sender” and “recipient”. But concepts of “client”, “server”, “request”, and “response” continue to live inside the message body of the “HTTP Mailbox Message”. To understand the differences at the encapsulated message level, “HTTP Mailbox Message” can further be subdivided into two categories, “Indirect HTTP Request” and “Indirect HTTP Response”, in accordance with RFC 2616 “HTTP Request” and “HTTP Response”.

4.3 Indirect HTTP Request

Suppose Alice is using an HTTP Mailbox service hosted on example.net to communicate with Bob’s task manager service hosted on example.com from her organization’s website hosted on example.org in a RESTful style.

Code 3 (Lines: 7-12) is a typical HTTP PATCH request that she would send in order to mark the completed task done. Due to client or server side limitations (as discussed in section 1), a PATCH request may not be possible. Hence clients encapsulate the desired HTTP PATCH request in another HTTP POST request as illustrated in Code 3 (Lines: 1-12). This POST request is made to HTTP Mailbox on a different domain, it has a different path, Content-Type and Content-Length as illustrated in Code 3 (Lines: 1-5).

On a successful POSTing, HTTP Mailbox responds with a 201 Created status code and provides a Location header with the URI of the resulting message as illustrated in Code 3 (Lines: 14-16).

The request has not reached to Bob’s server yet but now it is the responsibility of the HTTP Mailbox to deliver it when requested by Bob’s server (in other words, when Bob’s server pulls). Hence Alice’s client is not blocked. In terms of Linda, thus far only the “out” function has been performed.

A client on behalf of http://example.com/tasks can then perform an HTTP GET request to the HTTP Mailbox as illustrated in Code 4 (Lines: 1-2) and get an HTTP response as illustrated in Code 4 (Lines: 4-21). This process is similar to the “rd” function of Linda.
Two complete HTTP Request and HTTP Response cycles between a client and HTTP Mailbox, and a server and HTTP Mailbox respectively make one Indirect HTTP Request as illustrated in Code 3 and Code 4 and shown in Figure 2.

**4.4 Indirect HTTP Response**

After fetching messages from HTTP Mailbox and with the help of `Content-Type` and `Content-Length` headers as illustrated in Code 4 (Lines: 13-14), the server can parse the encapsulated HTTP PATCH request as illustrated in Code 4 (Lines: 16-21). The extracted HTTP PATCH Request can then be transformed (if necessary), executed on the task manager server and (if necessary,) a response may be sent to Alice using HTTP Mailbox as illustrated in Code 5. This process is similar to the “eval” function of Linda.

Later, Alice wants to check to see if her change was made, so she queries the HTTP Mailbox as illustrated in Code 6. If Bob’s server updated Alice’s task list and sent a response to the HTTP Mailbox then Bob’s server response will be included in the HTTP Mailbox response to Alice’s query as illustrated in Code 6 (Lines: 15-19).

**4.5 Message Lifecycle**

A complete HTTP Mailbox messaging lifecycle consists of two phases, 1) Send, and 2) Retrieve. Each phase is further divided in two parts: request, and response. Each phase corresponds to one complete HTTP Request and Response cycle. Figure 3 summarises the HTTP Mailbox process on both sender and recipient ends. C1) A generic HTTP Message (Request or Response), C2) An HTTP Message encapsulated in an HTTP POST Request to HTTP Mailbox

---

**Figure 2: HTTP Mailbox store and on demand delivery scenario.**
4.5.1 Send Request

In the first phase of the HTTP Mailbox messaging a message is sent from the client (by or on behalf of a message sender) to the HTTP Mailbox server. This contains identifier of the recipient(s), some extra metadata and message body.

To send a message, an HTTP POST Request is made to the HTTP Mailbox server with recipients’ identifier appended to the HM-Base of the mailbox as advertised by the HTTP Mailbox service on a well known URL (or root URL of the service), HTTP Mailbox service host as Host header, other extra metadata should go in HTTP headers (as discussed in section 4.6.2). The entity must be a valid message/http or application/http request and appropriate Content-Type header must be present in the request headers (Figure 3 [C2, C2’]).

4.5.2 Send Response

This is a feedback message from the HTTP Mailbox server to the message sender after receiving the “send request” message.

A status code 201 Created will be returned along with the URI of the message in the Location header or an error code (e.g., 4xx/5xx) in case of failure (Figure 3 [C3]).

A success response (status code 201 Created) from the HTTP Mailbox server is a confirmation that the message has been stored and a promise that the message will be delivered whenever requested on behalf of the recipient(s).

4.5.3 Retrieve Request

The second phase of the HTTP Mailbox messaging begins with a message retrieval request from a client (by or on behalf of the recipient(s)). This request is made to the HTTP Mailbox server along with the identifier of the recipient(s) or direct URI of the message (if known), MIME type, and extra headers if necessary.

To retrieve the most recent message for a recipient, an HTTP GET request is made to the HTTP Mailbox server with recipients’ identifier appended to the HM-Base of the mailbox as advertised by the HTTP Mailbox service, HTTP Mailbox service host as Host header, other extra metadata should go in HTTP headers (if necessary) while the entity must be empty. To retrieve an arbitrary message from the HTTP Mailbox server, an HTTP GET request must be made to the unique URI of the message (Figure 3 [S1]).

4.5.4 Retrieve Response

This is the final stage of the HTTP Mailbox messaging lifecycle. It is the response message from HTTP Mailbox server to the client when a “retrieve request” is made. It contains the message in the response body, MIME type and several other essential or optional headers in the header section of the response.

If the retrieval query was successful, a status code of 200 OK should be returned along with Via, Link, Memento-Datetime, Content-Type, Content-Length and other optional headers (if necessary) followed by the message in the HTTP response body. The Memento-Datetime header contains the datetime when the message was first seen by the HTTP Mailbox. The Link header is used to provide navigational links to traverse the message chain back and forth (as discussed in section 4.6.4), identified by recipients’ identifier. In case of success, the entity will be a valid message/http or application/http response and the appropriate Content-Type header will be present in the response headers. If the query does not match any messages or any other error occurred, an appropriate status code (like 4xx/5xx) should be returned (Figure 3 [S2, S2’]).

4.6 HTTP Mailbox API

One of the REST principles is Hypermedia as the Engine of Application State (HATEOAS) [16, 2]. According to this a client needs no prior knowledge about how to interface with a RESTful service, except the generic understanding of relation types [35] and MIME types [23]. Client begins
interaction with the service from a fixed URL and discovers future actions within the resource representations returned from the server. Clients should not rely on out-of-band information to interact with the RESTful service [17].

4.6.1 HM-Request-Path

An HTTP Mailbox service will advertise its base path (or base URL) for messaging called HM-Base (see appendix). This HM-Base will be used to construct request path (or request URI) at the time of sending or retrieving messages to or from HTTP Mailbox. In our examples, HM-Base is /hm/.

HM-Request-Path consists of two parts, HM-Base followed by recipients’ identifier. Code 3 and 4 have http://example.com/tasks as recipient identifier in their first lines. Recipients’ identifier can be a URI or any URL-encoded string token (like “everyone” for broadcasting or “friends/alice” for multicasting among Alice’s friends).

Recipients’ identifier may or may not match the path (or URI) in the Request-Line of the enclosed entity body. This is particularly important, because HTTP Mailbox is an on-demand message delivery service and it does not allow wildcard searching.

For example, if Alice’s PUT request is to be sent to Bob’s server to create a new resource at a non-existing URI, the HM-Request-Path may never be queried by Bob’s server and the message will remain unread forever. In our examples, we have used http://example.com/tasks as recipient identifier while the enclosed entity has http://example.com/tasks/1 as its URI.

4.6.2 HM-Headers

Apart from general HTTP-Headers, some headers are significant to the HTTP Mailbox. In a Send Request, the HM-Sender generic header should be sent, because the client sending the message may be sending it on behalf of someone else. On the other hand, in a Retrieve Response (in case of success) a Via [15] header is returned containing the identifier of the sender and the hostname or IP address of the sender client. A Retrieve Response (in case of success) must also return a Link [34] header containing self, current, first, last, next, and previous message URIs as applicable. HTTP Mailbox will also return a Memento-Datetime header to report the time when the enclosed message was first seen by the HTTP Mailbox.

In a Send Response a Location header containing the URI of newly sent message will be returned from HTTP Mailbox along with status code 201 if the message was successfully stored.

More HTTP Mailbox specific headers can be added later to extend the features of HTTP Mailbox like security, privacy, and message state.

4.6.3 HM-Body

All Send Requests and successful Retrieve Responses must contain HM-Body as entity body. HM-Body is a Request or Response HTTP-Message in one of the message/http (single) and application/http (pipeline) media types defined in [15]. Corresponding Content-Type header must be present in the HM-Headers. In other cases, there can be no entity body or any generic entity body with appropriate Content-Type header and Status-Code. In case of Retrieve Request, there must not be any entity body because it is an HTTP GET request.

4.6.4 Message Chain

The HTTP Mailbox query mechanism using HM-Request-Path allows the retrieval of the single “most recent” message sent to the corresponding recipient (if any). Every message also has a unique URI that can be used to Retrieve the message. By using the Link header of the response from HTTP Mailbox, an arbitrary number of messages or the entire message chain for the recipient(s) can be retrieved in either chronological or reverse chronological order, one message at a time. In a successful Retrieve Response, the Link header will contain the URI of the most recent message based on HM-Request-Path for the recipient as rel=current. It must also return unique URIs of self, first, and last messages with corresponding rel attributes. HTTP Mailbox will also return unique URIs of previous and next messages if present with corresponding rel attributes. Multiple rel attributes can be put together separated by a space if they point to the same URI. Absence of next relation and same values of self and last, both indicate the end of the message chain. Similarly, absence of previous relation and same values of self and first, both indicate the beginning of the message chain. Clients may use these indicators to detect either end of the message chain at the time of retrieval. Usually beginning of the message chain remains the same while end of the chain keeps changing over the time as more and more messages arrive for the same recipient. Figure 4 shows a typical message chain retrieval scenario where the most recent message is retrieved first then it follows the previous link from the header until the first message in the chain is retrieved.

4.6.5 Accessibility

An HTTP Mailbox service should provide full CORS support so that restricted clients (like web browsers) can allow message sending and retrieval to and from the HTTP Mailbox while avoiding the JavaScript same-origin policy.

5. REFERENCE IMPLEMENTATION

A reference implementation of an HTTP Mailbox server was written in Ruby [31] using the Sinatra [33] Web framework running on a Thin [30] Web server. Fluidinfo [24] was used to store messages and other metadata associated with them and it was accessed using fluidinfo.rb [42] Ruby library. A copy of this code can be found on GitHub [1].

5.1 Benchmarking

ApacheBench [46] was used for stress testing of the reference implementation. We took digits of π to generate payloads of various sizes ranging from 1 byte to 100,000,000 bytes (≈ 100 MB). Figure 5 shows the benchmark results of the HTTP Mailbox server (our reference implementation) on various concurrency levels and data sizes for Send and Retrieve requests respectively. The Y-axis shows the value of mean time per request (MTPR) in ms. Each data point was generated by issuing total number of requests 10 times the concurrency level. The time taken by each request includes round trip time of the network time from benchmarking machine to HTTP Mailbox and message processing (which includes several HTTP connection between HTTP Mailbox and Fluidinfo server).

Graphs in Figure 5(a) and 5(b) show that the MTPR in both the cases decreases as concurrency increases. For
Figure 4: Message Chain Retrieval.
smaller payloads (below 1 MB) the variation is not distinguishable, but when the payloads increase from 1 MB to 10 MB, MTTP roughly doubles. After analyzing data, we picked the 100 KB file and performed the stress testing up to a concurrency level of 1,000 and observed a gradual decrease in MTTP. With 1,000 concurrent requests, MTTP was 46 ms for POST requests and 34 ms for GET requests, while on concurrency level 100, these values were 118 ms and 83 ms respectively. ApacheBench socket did not allow more than a thousand open files for concurrent posting. Our implementation of HTTP Mailbox did not allow posting 10 MB (or larger) messages. We have also observed 0.0144% (12/83,300) unexpected non-2xx responses in our benchmarking. While valid 404 Not Found responses took 53 ms MTTP (values ranging from 200 to 20 ms depending upon the concurrency level).

5.2 Preserve Me! Application

A web preservation application called “Preserve Me!” uses the services of the HTTP Mailbox heavily to fulfill its communication needs. This application is a JavaScript add-on utility that can be added in any web page. This will add a small “Preserve Me!” icon somewhere in the web page similar to several sharing icons (e.g., Tweet, Like, and +1).

When that icon is clicked, it looks for one or more link tags with rel=resourcemap in the page. If the href attribute of those links points to a valid Atom ResourceMap [28] files then it pops up a window as shown in Figure 6. This window gives insight into the ResourceMap and aggregated resources [28] and also allows users to exchange messages with other ResourceMaps and connect them via “family and friends” relationships. These message exchanges and relationships allow human assisted preservation of aggregated resources and aids in their long-term preservation.

Code 8: Retrieving Add Friend Request

The Preserve Me! application sends various messages including: friendship request, copy request, and copy service announcements. Code 7 and 8 illustrate a typical friendship request message completing its lifecycle using HTTP Mailbox service. An aggregation (collection of resources) represented by a ResourceMap goes through a process of unsupervised creation of Small World Networks [8, 9] to select various other aggregations as friends. To complete the friendship it requires the other aggregation to add a link tag in its ResourceMap, pointing back to the requester. To
Figure 6: “Preserve Me!” Application window.
Table 3: Feature Comparison of Various Messaging Systems.

| Feature        | HTTP | Linda | Relay HTTP | Bleebs | AMQP | HTTP Mailbox |
|----------------|------|-------|------------|--------|------|--------------|
| Multicast      | No   | Yes   | No         | Yes    | Yes  | Yes          |
| Non-Blocking   | No   | Yes   | No         | Yes    | Yes  | Yes          |
| Reliability    | Yes  | Yes   | Yes        | No     | Yes  | Yes          |
| Scale          | Web  | Small | Web        | Web    | Web  | Web          |
| Message Size   | Any  | Any   | Any        | Short  | Any  | Any          |

make that change, it prepares an XML-Patch [49] file as illustrated in Code 7 (Lines: 13-20) and sends an HTTP PATCH request as illustrated in Code 7 (Lines: 8-20) to the chosen friend. Because of the client and server side limitations, this request might not be directly possible so it wraps the HTTP PATCH Message in an HTTP POST Message and sends it to the HTTP Mailbox as illustrated in Code 7 (Lines: 1-20).

When the “Preserve Me!” window of the receiving aggregation is opened, it checks its Mailbox as illustrated in Code 8. It then shows the friendship request (and any other messages, if available) as shown in Figure 6. If that message is applied then the friendship link will be added in the ResourceMap of the receiving aggregation.

6. EVALUATION

Table 3 gives a quick overview of various features among various communication systems discussed in sections 2, 3, and 4. AMQP and the HTTP Mailbox are the two overall winners over the set of features listed in the table. While AMQP is a general purpose enterprise communication system, it is not very friendly for web communication especially using web browsers. On the other hand, the HTTP Mailbox is primarily made with RESTful web communication in mind.

6.1 Availability

Suppose that a sender has to send HTTP requests to $R$ number of recipients where an immediate response from the recipients is not required but the sender has to make sure that every recipient will eventually get the message. At any given time, a random subset of total recipients are unreachable but every recipient is mostly available over a period of time $T$.

In the HTTP communication, it may take a period of time as long as $T$ to successfully communicate with all the recipients and the sender has to make frequent attempts over time $T$. On the other hand, in the HTTP Mailbox communication, sender can send the message(s) to the HTTP Mailbox whose availability is much higher (assumed to be highly reliable) than individual recipients. The responsibility of eventual delivery of messages is then off loaded to the HTTP Mailbox and sender can proceed without being blocked.

6.2 Network Usage

The total number of HTTP cycles $C$ (where a cycle is combination of HTTP Request and Response) required to send $M$ messages to a group of $R$ recipients, assuming that there is no transient failure (or equally probable in all cases): HTTP messaging:

$$C = M \times R \tag{1}$$

HTTP Mailbox messaging (where recipients only make attempts after sender has successfully sent the message to HTTP Mailbox.)

$$C = M \times (R + 1) \tag{2}$$

If the $M$ messages are grouped in $N \leq M$ message pipelines using application/http [15] MIME type then the cost of HTTP Mailbox communication will reduce further while cost of HTTP communication will remain the same.

$$C = N \times (R + 1) \tag{3}$$

In the worst case, individual unicast messages will cost twice for HTTP Mailbox communication as compared to HTTP. For larger group messaging scenarios it will cost roughly the same as HTTP while message pipelining will drop the cost of communication by a factor derived from the ratio of number of messages to the number of message pipelines. For simplicity, we have ignored the communication cost introduced by Pull [5] attempts made by recipients before a new message arrived in the HTTP Mailbox for them, which is likely to happen because recipients are unaware of the sender’s state.

6.3 Response Pagination

The HTTP Mailbox server itself can handle various concurrent requests but the design of HTTP Mailbox restricts individual message recipients from accessing their messages concurrently. As discussed in section 4.6.4, a recipient can not access arbitrary messages unless the URIs of those messages are known. After fetching a message a recipient only has URIs of first, last, next, and previous messages in the chain as applicable.

In our reference implementation the round-trip time of a single GET request with usual payload is between 300 to 400 ms as shown in Figure 5(b) at concurrency level 1. As a result, a recipient can fetch roughly 3 subsequent messages per second from the message chain.

This issue is only limited to message retrieval (or GET requests). To overcome this problem, HTTP Mailbox can paginate responses. Every page can have a configurable number of subsequent messages that can be batched together along with the links to first, last, previous, and next pages to navigate through the chain of pages. To limit the scope, we have deferred the API definition for response pagination as future work.
7. FUTURE WORK
We are considering specifications for batch message retrieval (discussed in section 6.3) and adding more query mechanisms like retrieving messages after a given timestamp. For access control, security, privacy, integrity, and authenticity [26], we are planning to use techniques like OAuth [10], public key encryption [25], and hashing [48, 41]. Data storage services other than Fluidinfo should also be evaluated to compare robustness and response time in each case. We are also planning to add message access log feature in the HTTP Mailbox and evaluate how it affects the utility and performance of the system.

8. CONCLUSIONS
In an effort of preserving web objects, we needed a messaging system that can be used reliably on the scale of the Web. We explored various possibilities including Linda, HTTP, and Bleeps but none of them fit our needs. Hence we have developed a store and forward model of HTTP messaging called HTTP Mailbox that remains RESTful and provides asynchronous (non-blocking) message sending and on-demand message retrieval facility between sender and recipients. It also provides message pipelining and group messaging (broadcast and multicast) facilities that save network usage and time.

Based on our model, we have implemented an HTTP Mailbox and tested its robustness and performance. Benchmarking our reference implementation gave us very reliable and time efficient results even on high concurrency levels within a data size limit. Unexpected failure rate was as low as 0.0144% over more than 83,000 send and retrieve requests in our benchmarking.

We have successfully removed the client and server side barriers in using full range of HTTP methods in REST style. We have utilized our implementation of the HTTP Mailbox in the “Preserve Me!” application. We have also made the code of our implementation available on GitHub [1].

9. ACKNOWLEDGEMENTS
This work was supported in part by the NSF, Project 370161.

10. REFERENCES
[1] S. Alam. HTTPMailbox. https://github.com/ibnesayeed/HTTPMailbox, 2013.
[2] R. Alarcon, E. Wilde, and J. Bellido. Hypermedia-driven RESTful Service Composition. Service-Oriented Computing, pages 111–120, 2011.
[3] J. Aubourg, J. Song, and H. R. M. Steen. XMLHttpRequest. http://www.w3.org/TR/2012/ WD-XMHttpRequest-20121206/, 2012.
[4] R. Battle and E. Benson. Bridging the Semantic Web and Web 2.0 with Representational State Transfer (REST). Web Semantics: Science, Services and Agents on the World Wide Web, 6(1):61 – 69, 2008.
[5] E. Bozdag, A. Mesbah, and A. van Deursen. A Comparison of Push and Pull Techniques for AJAX. In Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on, pages 15–22. IEEE, 2007.
[6] CakePHP Cookbook. Creating Forms. http://book.cakephp.org/2.0/en/core-libraries/helpers/form.html#creating-forms, 2013.
[7] N. Carriero and D. Gelernter. Linda in Context. Communications of the ACM, 32(4):444 — 458, 1989.
[8] C. L. Cartledge and M. L. Nelson. Unsupervised Creation of Small World Networks for the Preservation of Digital Objects. In Proceedings of the 9th ACM/IEEE-CS joint conference on Digital libraries, pages 349–352. ACM, 2009.
[9] C. L. Cartledge and M. L. Nelson. Analysis of Graphs for Digital Preservation Suitability. In Proceedings of the 21st ACM conference on Hypertext and hypermedia, pages 109–118. ACM, 2010.
[10] E. D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, Oct. 2012.
[11] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Dec. 1998.
[12] L. Dusseau and J. Snell. PATCH Method for HTTP. RFC 5789, Mar. 2010.
[13] I. Fette. The WebSocket Protocol. RFC 6455, Dec. 2011.
[14] R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, 2000.
[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.
[16] R. Fielding and R. Taylor. Principled design of the modern web architecture. ACM Transactions on Internet Technology (TOIT), 2(2):115–150, 2002.
[17] R. T. Fielding. REST APIs must be hyperxtext-driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hyertext-driven, 2008.
[18] T. Garnock-Jones. Relay HTTP. http://reversehttp.netrelay-http-spec.html, 2009.
[19] D. Gelernter. Generative Communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS), 7(1):80–112, 1985.
[20] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). http://www.w3.org/TR/2007/REC-soap12-part1-20070427/, 2007.
[21] hawkeye. HttpMethodsMiddleware. http://djangosnippets.org/snippets/174/, 2007.
[22] M. Hossain. USING CORS. http://www.html5rocks.com/en/tutorials/cors/#toc-creating-the-xmlhttprequest-object, 2012.
[23] IANA. MIME Media Types. http://www.iana.org/assignments/media-types, 2013.
[24] T. Jones and E. Fernandez. Fluidinfo. http://fluidinfo.com/, 2009.
[25] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, Feb. 2003.
[26] C. Kaufman, R. Perlman, and M. Speciner. Network security: private communication in a public world. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2002.
[27] M. Klein, R. Sanderson, H. Van de Sompel, S. Warner, B. Hashhofer, C. Lagoze, and M. L. Nelson.
A Technical Framework for Resource Synchronization. *D-Lib Magazine*, 19(1), 2013.

[28] C. Lagoze, H. Van de Sompel, P. Johnston, M. Nelson, R. Sanderson, and S. Warner. ORE User Guide - Resource Map Implementation in Atom. http://www.openarchives.org/ore/1.0/atom, 2008.

[29] M. Majkowski. RabbitMQ-Web-Stomp plugin. https://github.com/rabbitmq/rabbitmq-web-stomp, 2012.

[30] Marc-André Cournoyer. Thin - yet another web server. http://code.macournoyer.com/thin/, 2008.

[31] Y. M. Matsumoto. Ruby programming language. http://www.ruby-lang.org/, 1995.

[32] S. Michelotti. Implementing a delete link with mvc 2 and httpprotoveride. http://vhlo.gs/Zk6, 2012.

[33] B. Mizerany. Sinatra. http://www.sinatrarb.com/, 2007.

[34] M. Nottingham. Web Linking. RFC 5988, Oct. 2010.

[35] M. Nottingham, J. Reschke, and J. Algermissen. Link relations. http://www.iana.org/assignments/link-relations/link-relations.xml, 2013.

[36] J. O’Hara. Advanced Message Queuing Protocol. http://www.amqp.org/, 2003.

[37] J. Postel. Internet Protocol. RFC 791, Sept. 1981.

[38] Rails Guides. How do forms with PUT or DELETE methods work? http://guides.rubyonrails.org/form_helpers.html#how-do-forms-with-put-or-delete-methods-work, 2013.

[39] J. Reschke. consider adding support for PUT and DELETE as form methods. https://www.w3.org/Bugs/Public/show_bug.cgi?id=10671, 2010.

[40] J. Reschke. Enhance http request generation from forms. https://www.w3.org/html/wg/tracker/issues/195, 2012.

[41] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, Apr. 1992.

[42] E. Seidel. fluidinfo.rb. https://github.com/gridaphobe/fluidinfo.rb, 2010.

[43] STOMP. Simple Text Oriented Messaging Protocol. http://stomp.github.io/, 2012.

[44] Sun Microsystems. Java Message Service (JMS). http://www.oracle.com/technetwork/java/jms/index.html, 2001.

[45] The Apache Software Foundation. Apache HTTP Server. http://projects.apache.org/projects/http_server.html, 1995.

[46] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html, 2005.

[47] The Apache Software Foundation. Apache Qpid. http://qpid.apache.org/, 2009.

[48] S. Turner. Using SHA2 Algorithms with Cryptographic Message Syntax. RFC 5754, Jan. 2010.

[49] J. Urpalainen. An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors. RFC 5261, Sept. 2008.

[50] H. Van de Sompel, M. L. Nelson, R. Sanderson, L. L. Balakireva, S. Ainsworth, and H. Shankar. Memento: Time Travel for the Web. arXiv preprint arXiv:0911.1112, 2009.

[51] H. Van de Sompel, R. Sanderson, M. Klein, M. L. Nelson, B. Haslohofer, S. Warner, and C. Lagoze. A Perspective on Resource Synchronization. *D-Lib Magazine*, 18(9), 2012.

[52] VMware. RabbitMQ. http://www.rabbitmq.com/, 2010.

**APPENDIX**

### A. ENHANCED BNF

```
HM-Request-Path = HM-Base ( http_URL | token )
HM-Base = absoluteURI | abs_path
HM-Body = HTTP-message
Send-Request = "POST" SP HM-Request-Path SP HTTP-Version CRLF*( HM-req-header CRLF ) CRLF HM-Body
Send-Response = Response
Retrieve-Request = "GET" SP HM-Request-Path SP HTTP-Version CRLF*( HM-req-header CRLF ) CRLF
Retrieve-Response = Status-Line*( HM-res-header CRLF ) CRLF ( HM-Body | message-body )
HM-req-header = ( Sender-header | general-header | request-header | entity-header )
HM-res-header = ( Via-header | Link | general-header | Memento-Datetime | response-header | entity-header )
Via-header = "Via" SP "sent by" SP (IP | IPv6 | Host) SP "on behalf of" SP absoluteURI SP "delivered by" SP absoluteURI
Sender-header = "HM-Sender" SP absoluteURI
```

Link is defined in RFC 5988 [34], Memento-Datetime is defined in [50], IP is defined in RFC 791 [37], IPv6 is defined in RFC 2460 [11], and remaining terms are inherited from RFC 2616 [15] unless defined here.