Summer 2013

HTTP Mailbox - Asynchronous Restful Communication

Sawood Alam
Old Dominion University

HTTP MAILBOX - ASYNCHRONOUS RESTFUL COMMUNICATION

by

Sawood Alam
B.Tech. May 2008, Jamia Millia Islamia, India

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 2013

Approved by:

Michael L. Nelson (Director)

Michele C. Weigle (Member)

Ravi Mukkamala (Member)

ABSTRACT

HTTP MAILBOX - ASYNCHRONOUS RESTFUL COMMUNICATION

Sawood Alam
Old Dominion University, 2013
Director: Dr. Michael L. Nelson

Traditionally, general web services used only the GET and POST methods of HTTP while several other HTTP methods like PUT, PATCH, and DELETE were rarely utilized. Additionally, the Web was mainly navigated 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. Recently, several web frameworks/libraries have started supporting RESTful web services through APIs. To support HTTP methods other than GET and POST 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. Middleware between the web server and the application may override the HTTP method based on special hidden form field values. 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. Browser-based communication also suffers from cross-origin restrictions for security reasons.

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 also allows for 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!" and a visualization application called "Preserve Me! Viz".

Beneath my mother's feet...

ACKNOWLEDGEMENTS

All praises be to the Almighty, who has brought me into existence to make unnoticeable subtle differences in my surroundings.

If I had the restriction to acknowledge only one person for this work, then that would be none other than my advisor, Dr. Michael L. Nelson. I would like to thank him for his continuous guidance, patience, support, and the time he spent reading and correcting my writing. It has always been interesting to see ideas spark during discussions with him and continue to burn afterwards. I often come back from my cubicle to his office immediately after our regular meetings to say “one more thing”. I enjoy listening to him even when he talks about music, old cars, Star Wars, and sports although it does not make a lot of sense to me. I look forward to continuing my future research under the guidance of this awesome adviser.

I would like to thank my committee members for their great support and feedback. Dr. Michele C. Weigle has provided valuable comments and corrected tons of grammatical mistakes in my writing. It was always a great pleasure to talk to Dr. Ravi Mukkamala on research topics and beyond. I would also like to thank Mr. Charles L. Cartledge for all his inputs and support. He has proved to be a wonderful research mate. He has not only contributed greatly to my academic growth but also encouraged and helped me to learn about the English language and culture. Thanks to the Google Hangouts for providing such a platform that enabled frictionless video conferencing and screen sharing that I utilized several hours a week to talk to Mr. Cartledge in pursuit of our research work. I would like to acknowledge that utilizing the HTTP Mailbox in the Preserve Me! Viz application for live visualization was his idea. I wish the best for soon to be Dr. Cartledge. I would also like to thank all the WSDL members for their feedback.

I would like to acknowledge that the idea of Bleeps came from early discussions about the ResourceSync project with Herbert Van de Sompel of Los Alamos National Laboratory.

I would like to thank professor M. M. Sufyan Beg, Professor Mohammad Zubair, and Professor Hussein Abdel-Wahab who were the primary reasons why I am at Old Dominion University. I would also like to thank Dr. Manish Wadhwa and his wife Komalpreet Kaur for kidnapping me regularly and making sure that I have sufficient groceries in my kitchen.

I would like to thank the Urdu community on the Web, especially the Mehfilians, for their patience on my repeated words, “I am too busy to do it right now, but I am putting it in my priority list”.

Last, but not least, I would like to take this moment to extend my gratitude towards my mother Sitara Begum, my elder brother Masood Alam, my maternal uncle Dr. Shabbeer Ahmad Khan, and my “one-and-a-half mother” Naseema Khatoon who played very important roles in my life and they had special expectations from me. I renew my dedication to strive to make their dreams about me come true. I would like to thank my parents, my grandparents, my brothers, my sisters, other family members, relatives, teachers, and my friends for playing such a wonderful role in my life. Although I am not mentioning their individual names but they know if they are in this list. Ammi Jaan and Bhai Jaan, may Almighty reward you the best for supporting my wife and my daughter during this period. Abida, Raihana, Sumbul, Fareeha, and everyone else, you mean a lot to me and I love you all.

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

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Alice would like 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 I) and started working on the highest priority task. Once it was finished, she wanted to mark that task as done. Hence, Alice made a Hypertext Transfer Protocol (HTTP) PATCH request [1] to the specific task's Uniform Resource Identifier (URI) on Bob's server to modify the task partially. PATCH is a method of HTTP that is used to partially update an existing resource. Unfortunately the server was down so this communication failed. Alice tried again after some time when the server was up but received a `501 Not Implemented` HTTP Response [2] 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 (where REST stands for REpresentational State Transfer). They wished there was an extra layer of indirection to provide a RESTful interface to Bob's server.

TABLE I
Alice's Tasks

ID	**Description**	**Priority**	**Status**
1	Write a paper.	HIGH	Pending
2	Go on vacation.	LOW	Pending

The 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 messaging capabilities, an enhancement not possible using HTTP.

In past years, general web services used only the GET and POST methods of HTTP while several other HTTP methods like PUT, PATCH, and DELETE were rarely utilized. Until recently, the Web was mainly navigated 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 [3, 4]. Recently, several web frameworks/libraries (like Ruby on Rails [5], CakePHP [6], Django [7], and .NET [8]) have started supporting RESTful web services through an Application Programming Interface (API). To support HTTP methods other than GET and POST 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. Middleware between the web server and the application may override the HTTP method based on special hidden form field values. On one hand, this unsupported method limitation is present only in HTML and not in Ajax. On the other hand, Ajax requests suffer from cross-origin restrictions. JavaScript has a security policy called the “same-origin” policy which restricts Ajax requests from a web page from communicating with a domain other than the origin domain of the web page. Support for Cross-Origin Resource Sharing (CORS) [9] is in the working draft of XMLHttpRequest [10]. While modern web browsers have recently started supporting cross-origin Ajax requests [11], this feature is not available in older browsers. Also, CORS support is server dependent. The web server needs to explicitly send specific headers in order to allow the browser to communicate with the server using Ajax.

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.

We describe the HTTP Mailbox, a mechanism to store HTTP messages (requests/responses) and deliver them on demand. The HTTP Mailbox makes HTTP communication asynchronous. It enables RESTful HTTP communication with the full range of HTTP methods otherwise unavailable to standard clients and servers. It enables cross-domain communication in Ajax with the help of standard CORS headers. The HTTP Mailbox is a store and delivery protocol that allows retrieval of the same message multiple times by any number of recipients on demand. It also provides multicast semantics otherwise unavailable in standard HTTP communication.

We evaluate a reference implementation in Ruby using ApacheBench [12] (a server stress testing tool). Our test demonstrates high throughput (on 1,000 concurrent requests) and a systemic error rate of 0.01%. Finally, we demonstrate our HTTP Mailbox implementation being utilized in a human-assisted Web preservation application called “Preserve Me!”. We have also utilized our HTTP Mailbox to visualize the graph of “Preserve Me!” in real time using an application called “Preserve Me! Viz”.

Our contribution in this thesis can be summarized as follows.

- Enabling asynchronous communication in HTTP.
- Enabling CORS support in restricted environments.
- Enabling REST with full method support in all web servers.
- Enabling indirect and group communication in HTTP.
- Implementation of the HTTP Mailbox system.
- Benchmarking of our implementation.
- Quantitative and qualitative evaluation of the system.
- Brief description and evaluation of several related communication systems.
- Utilization of the HTTP Mailbox implementation in various applications.

CHAPTER 2

BACKGROUND

The HTTP Mailbox is a store and on-demand delivery protocol for HTTP messages. It is inspired by the concept of shared memory in Linda [13] but implements the concept using web semantics. It uses RESTful HTTP communication to transport HTTP Messages between client/server and the HTTP Mailbox server.

2.1 LINDA

Linda [13] is a model based on generative communications [14] 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 is a pre-web model from 1980s that works only for machines connected to a shared memory.

Linda provides a means for asynchronous (time-uncoupled) communication in which the 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.

Linda implements CRUD with four basic operations or functions:

- “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.

```
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.

```
in("task", ?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.

```
eval("log", 1, changeStatus("Done"))
```

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.

```
("log", 1, "Done")
```

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.

```
rd("log", ?id, ?status)
```

We took the simplicity of this model and implemented it for storing and forwarding 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. (See chapter 7 for discussion on attacks and prevention.)

2.2 REST

REpresentational State Transfer (REST) [15, 16] is a software architecture for large-scale distributed systems which has emerged as the preeminent design pattern. It utilizes existing HTTP methods to generalize the interfaces of a web service by mapping resource actions like Create, Read, Update, and Delete (CRUD) [17] 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 and does not rely on HTTP methods for CRUD. REST encourages the use of nouns in the URI instead of verbs and hides the implementation details from the URI. Code 1 illustrates few RPC-style URIs and their corresponding RESTful URIs. A typical implementation of RPC on the Web is Simple Object Access Protocol (SOAP) [18] that allows querying available procedures and associates arguments on a remote server. A client can then invoke those procedures remotely using XML as the medium of exchange.

Code 1. RPC-Style vs. RESTful URIs

```
RPC  > GET /list_all_tasks.php
REST > GET /tasks

RPC  > GET /show_task_details.php?id=3
REST > GET /tasks/3

RPC  > POST /create_new_task.php
REST > POST /tasks

RPC  > POST /update_task_status.php
REST > PATCH /tasks/3

RPC  > GET /delete_task.php?id=3
REST > DELETE /tasks/3
```

Code 2. RESTful Communication

```
 1 > PATCH /tasks/1 HTTP/1.1
 2 > Host: example.com
 3 > Content-Type: text/task-patch
 4 > Content-Length: 11
 5 >
 6 > Status=Done
 7
 8 < HTTP/1.1 200 OK
 9 < Content-Type: text/task
10 < Content-Length: 28
11 <
12 < (Done) [HIGH] Write a paper.
```

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 2 Lines 1-6 and received the modified task resource in response as Code 2 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 the server will send the updated resource in the response. Media types `text/task` and `text/task-patch` are not defined. They are used here for illustration purpose only.

Unfortunately, many web services are not fully REST compliant. Hence, a PATCH request as in Code 2 (or other methods like PUT or DELETE) may cause the server to respond with `501 Not Implemented` or other failure responses. For example, the default Apache [19] web server setup returns `405 Method Not Allowed` in response to a PUT request. Another issue is if Bob's server is not available then Alice has to wait and keep sending the request periodically until the service comes back online and completes the request.

TABLE II
HTTP Method Support

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%

Table II lists common HTTP methods and their support in web browsers and LAMP[1] servers. It shows that Apache web server requires extra configuration in order to support PUT, DELETE and PATCH methods. Also, pure HTML has no interface to issue these methods from the browser except by using Ajax requests.

Table II also gives statistical distribution of support of various HTTP methods on the live Web. This statistical distribution was calculated from 40,902 random live URIs from DMOZ [20]. DMOZ is an Open Directory Project that maintains the curated directory of World Wide Web URIs, currently listing over 5 million sites. We have selected the only URIs that return 200 OK response on GET request out of 100,000 initial set of URIs. Then we issued an OPTIONS request on those 40,902 live URIs to collect data about supported methods from the "Allow" response header as illustrated in Code 3. Only 55% of live URIs responded to the OPTIONS request

[1] Linux, Apache, MySQL, and PHP, Perl or Python.

and only 1.16% URIs returned all the methods listed in Table II in their “Allow” response header. It shows the limited utilization of HTTP methods other than GET and POST on the web.

Code 3 illustrates four OPTIONS requests with different responses. First request returned `501 Not Implemented` response, which means it does not recognise OPTIONS method, although the URI associated with this request supports POST, GET, HEAD, PUT, and DELETE methods (according to its documentation). Second request does recognize the OPTIONS method, but returned `405 Not Allowed` response, hence we cannot query supported methods. Third request returned limited method support (only GET and POST methods listed in Table II). Finally, fourth request returned support for all the methods listed in Table II. We did not check to see if the URIs respond to the methods returned in the “Allow” header.

Code 3. OPTIONS Request to Retrieve Allowed Methods

```
$ curl -I -X OPTIONS http://fluiddb.fluidinfo.com/about
HTTP/1.1 501 Not Implemented
Server: nginx/1.1.19
Date: Wed, 07 Aug 2013 21:09:15 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 150
Connection: keep-alive
X-Fluiddb-Request-Id: API-9006-20130807-210915-18792385
X-Fluiddb-Error-Class: UnsupportedMethod

$ curl -I -X OPTIONS http://dev.bitly.com/
HTTP/1.1 405 Not Allowed
Content-Type: text/html
Date: Wed, 07 Aug 2013 22:24:05 GMT
Server: nginx
Content-Length: 166
Connection: keep-alive

$ curl -I -X OPTIONS http://www.cs.odu.edu/
HTTP/1.1 200 OK
Date: Wed, 07 Aug 2013 23:11:04 GMT
Server: Apache/2.2.17 (Unix) PHP/5.3.5 mod_ssl/2.2.17 OpenSSL/0.9.8q
Allow: GET,HEAD,POST,OPTIONS
Content-Length: 0
Content-Type: text/html

$ curl -I -X OPTIONS http://www.parasitesandvectors.com/
HTTP/1.1 200 OK
Set-Cookie: UUID=6818dd14-085e-4a50-806a-deab9b907585; Path=/
Allow: GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, PATCH
Content-Type: text/html
X-Cacheable: NO
Server: BioMed Central Web Server 1.0
Content-Length: 0
Accept-Ranges: bytes
Date: Wed, 07 Aug 2013 23:02:22 GMT
Connection: keep-alive

```

CHAPTER 3

RELATED WORK

Web messaging has various forms including store and forward, point-to-point, peer-to-peer, and publish-subscribe. Based on these messaging forms there are various communication protocols, tools and platforms like Email, Skype, Twitter, NNTP [21], IRC [22], and XMPP [23]. Many of these were built to allow humans to communicate with each other over the Internet. In contrast we are interested in protocols that allow web objects to communicate easily with each other over the Internet, preferably using HTTP. Here we will first describe cross-domain communication. Then we will discuss some techniques and protocols that were especially built to enable communication among applications. Finally, we will evaluate if those protocols can be used for web object communication.

3.1 CROSS-DOMAIN COMMUNICATION

When a web page from domain "A" is loaded in a web browser and communicates with another domain "B", it is called cross-domain communication. Fig. 1 illustrates cross-domain communication. Cross-domain communication is not limited to browser-based communication only, but when two servers from different domains communicate with each other, they usually do not have cross-domain restrictions.

In browser-based cross-domain communication, embedded resources (like image, CSS, JavaScript) and HTML Forms are generally allowed, but Ajax requests are restricted by default. Also, a browser cannot enable cross-domain communication in Ajax without the help of the remote server.

There are legitimate usage of cross-domain communication, for example, utilizing Content Distribution Networks (CDN) or consuming third-party web service APIs. There can be misuses of cross-domain communication, for example, consuming unauthorized third-party web service APIs. There can also be some unwanted implications of cross-domain communication, for example, sending browser cookies containing user's session information to a different domain. Luckily, web browsers prevent cookies from being sent to a different domain for security reasons.

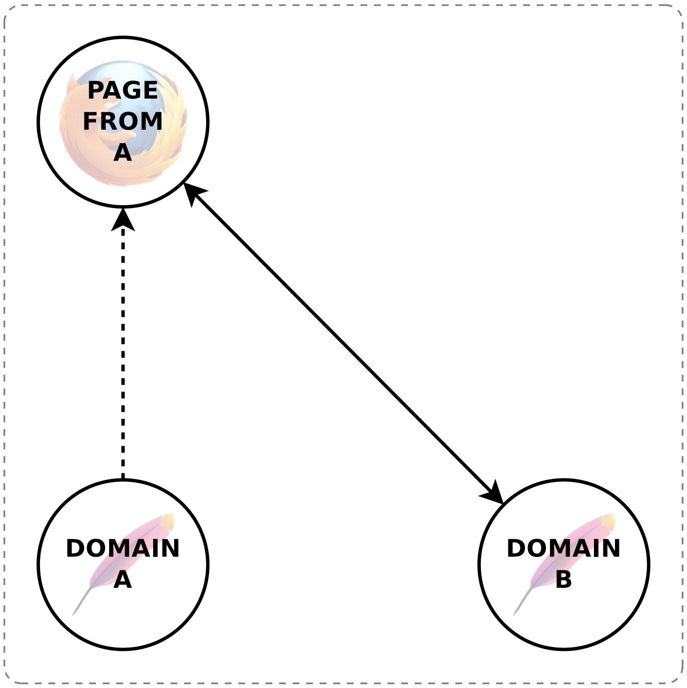


Fig. 1. Cross-Domain Communication.

3.2 AJAX AND HTML FORMS

Ajax or an HTML form can be used to communicate to a web server from a web browser that has a web page loaded in it. But both are limited in some ways especially when communicating to a domain other than the origin of the currently loaded web page.

Code 4. Cross-Origin Ajax Communication

```
1 var req = new XMLHttpRequest();
2 req.open("PATCH", "http://example.com/tasks/1", true);
3 req.setRequestHeader("Content-Type", "text/task-patch");
4 req.setRequestHeader("Origin", "example.org");
5 req.send("Status=Done");
6
7 ERROR: XMLHttpRequest cannot load http://example.com/tasks/1.
8        Origin http://example.org is not allowed by Access-Control-Allow-Origin.
```

Suppose that Alice has a web page loaded in her browser from her organization's website `example.org` and she tries to send a PATCH request to Bob's task manager service hosted on `example.com` domain to update the status of her first task. Also, suppose that Bob's server does not implement CORS headers. She may issue an Ajax request as illustrated in Code 4 Lines 1-5. This request will not be completed because of the cross-origin restriction imposed by the JavaScript, and the web browser will throw an error as illustrated in Code 4 Lines 7-8.

Alice can also try using an HTML form to make a cross-origin request but she will have many issues. If Bob's server has some protection against Cross-Site Request Forgery (CSRF) [24] her form submission may be rejected. CSRF is a type of malicious exploit of a website whereby malicious requests are sent by a malicious website on behalf of a user that the website trusts. Suppose that Bob's server does not protect against CSRF and Alice can submit HTML forms there. But the HTML form will not allow her to set the `Content-Type` header to an arbitrary value like `text/task-patch`. Another limitation of an HTML form is that it only supports GET and POST HTTP methods. Suppose Bob's server provides a non-standard workaround for the later issues with the help of a pre-defined request parameter "_method". Now, Alice may add a form in her organization's website as illustrated in Code 5 and submit it to make a POST HTTP request to Bob's server. If she does not set the proper value to the "target" attribute of the form element, then the browser will load the response from the URL as specified in the "action" attribute of the form element and the currently loaded page will be lost. While she can open the response in an iframe inside the current web page or in a new browser window/tab, the origin web page will not be able to read the response.

Code 5. Cross-Origin HTML Form

```
<form method="post" action="http://example.com/tasks/1" target="_blank">
  <input type="hidden" name="_method" value="patch">
  <select name="status">
    <option value="Done" selected="selected">Done</option>
    <option value="Pending">Pending</option>
  </select>
  <input type="submit" value="Submit">
</form>
```

3.3 RELAY HTTP

The Relay HTTP draft specification [25] describes a way to overcome the CORS restriction imposed by the 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 for tunneling HTTP traffic over HTTP [2]. 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 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. An iframe is an inline HTML element that allows embedding other web pages in a frame inside an HTML document. The primary HTML document can style the iframe element in limited ways (e.g., border, margins and dimensions) but has no control over the styling of the embedded document especially if the embedded document is from another domain.

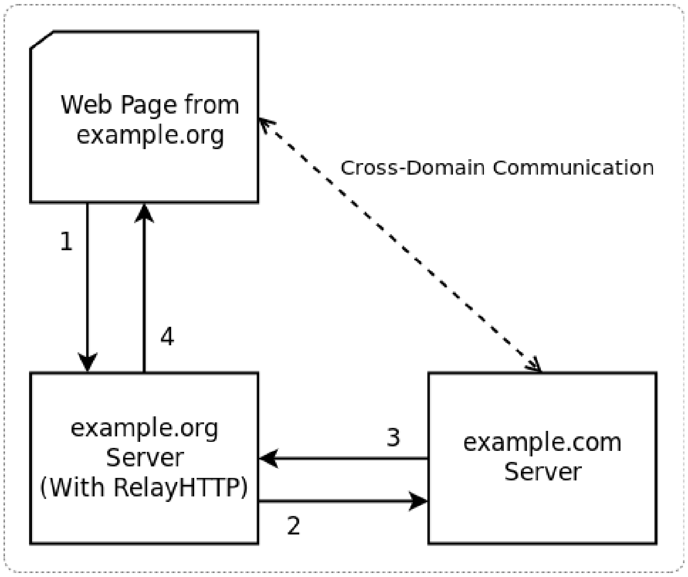

Web Page from example.org
Cross-Domain Communication
1
4
example.org Server (With RelayHTTP)
3
2
example.com Server

Fig. 2. Relay HTTP Cross-domain Communication.

Fig. 2 shows how a browser can overcome the cross-domain restriction imposed by the JavaScript security model using Relay HTTP proxy. If a page from domain A tries to access resources from domain B from within a browser window using Ajax, it needs to tunnel the HTTP Request through a proxy server hosted under domain A. The server of domain A can then relay that HTTP Request on the domain B server and return the HTTP Response to the browser window.

To overcome her client side restriction, Alice 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 6 Lines 1-11. The proxy service then replays the encapsulated `message/http` entity Code 6 Lines 6-11 and forwards the response back to the client as Code 6 Lines 13-17. But Relay HTTP still cannot 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 6. Relay HTTP Communication

```
 1 > POST /proxy/example.com HTTP/1.1
 2 > Host: example.org
 3 > Content-Type: message/http
 4 > Content-Length: 108
 5 >
 6 > PATCH /tasks/1 HTTP/1.1
 7 > Host: example.com
 8 > Content-Type: text/task-patch
 9 > Content-Length: 11
10 >
11 > Status=Done
12
13 < HTTP/1.1 200 OK
14 < Content-Type: text/task
15 < Content-Length: 28
16 <
17 < (Done) [HIGH] Write a paper.
```

3.4 ENTERPRISE MESSAGING SYSTEMS

An Enterprise Messaging System (EMS) is a platform-agnostic message queuing system to allow computer systems to communicate asynchronously. It uses enterprise-wide published standards and structured data to communicate semantic messages.

Apache Qpid [26] is an implementation of the platform agnostic Advanced Message Queuing Protocol (AMQP) [27]. Java Message Service (JMS) [28] defines reliable enterprise messaging standard. It is an integral part of the Java Platform, Enterprise Edition (Java EE) [29]. These are examples of enterprise messaging systems that allow various modes of digital communication including point-to-point, peer-to-peer, publish-subscribe, and other forms of individual and group messaging.

Code 7 illustrates a typical AMQP message that Alice will send to change the status of her first task to "Done". This message needs to be sent to an AMQP

Message Broker that will hold it as a persistent message and deliver it to the recipient identified by “to” field as shown in Code 7 Lines 12-14.

Code 7. AMQP Message Example

```
 1 <type name="header" class="composite" source="list" provides="section">
 2   <descriptor name="amqp:header:list" code="0x00000000:0x00000070"/>
 3   <field name="durable" type="boolean" default="true"/>
 4   <field name="delivery-count" type="uint" default="0"/>
 5 </type>
 6 <type name="properties" class="composite" source="list" provides="section">
 7   <descriptor name="amqp:properties:list" code="0x00000000:0x00000073"/>
 8   <field name="message-id" type="string" requires="message-id">
 9     a9168c52-26a2-43aa-b6a3-329ac39c1154
10   </field>
11   <field name="user-id" type="binary"/>
12   <field name="to" type="string" requires="address">
13     http://example.com/tasks/1
14   </field>
15   <field name="subject" type="string">
16     Change Task Status
17   </field>
18   <field name="content-type" type="symbol">text/task-patch</field>
19   <field name="creation-time" type="timestamp">1371606448</field>
20 </type>
21 <type name="amqp-value" class="restricted" source="*" provides="section">
22   <descriptor name="amqp:amqp-value:*" code="0x00000000:0x00000077"/>
23   Status=Done
24 </type>
```

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 some plugins available for RabbitMQ [30] (a message broker implementation for AMQP) that enable web communication (e.g., RabbitMQ-Web-Stomp [31] that utilizes STOMP [32] protocol and WebSockets [33] to enable browser-based interaction with RabbitMQ server).

3.5 SUMMARY

In this chapter we described cross-domain communication, then we discussed various existing techniques and protocols to enable web object communication. An HTML Form allows the user to make GET or POST requests only. Other HTTP methods are not supported in HTML Form element. When an HTML form is submitted to a domain other than the origin domain, response data cannot be read by the origin page. Ajax allows all the HTTP methods, but it is sever-dependent when it comes to cross-domain communication. If the domain that is being contacted is different from the origin, then the remote domain must explicitly return specific headers in order to allow cross-domain communication. CORS support is not enabled in the web servers by default. Relay HTTP solves the cross-domain problem in Ajax requests, but it requires the origin domain to have the extra relay service running. None of these techniques allow time-uncoupling or group communication. EMS provides time-uncoupling and group communication, but it is an RPC system and does not allow browser-to-server communication over HTTP. To enable web browser-based HTTP communication in EMS, we need an extra HTTP endpoint that communicates with the Message Broker and serves as an intermediate proxy.

CHAPTER 4

PRELIMINARY WORK

We were working on a human-assisted decentralized Web preservation system where we needed a messaging system to allow web objects to communicate with each other. Our primary goal was to leverage existing freely available Web infrastructure maintained by others while still keeping the system portable and independent of the underlying services so that services can be switched easily. We were also trying to allow interoperability among various instances of the preservation system irrespective of their chosen underlying communication system. We decided to develop an abstraction layer of communication system that can utilize MediaWiki, Blogger, Tumblr, Twitter, Gmail, Dropbox, and many other freely available content storage services to store messages and expose a uniform API to send and retrieve messages.

We tried various possibilities and realized that none were able to function at our desired level of scale. We tried the following:

- HTTP Communication
- Bleeps
- Micro-blogging (Tweet-like services)
- Decentralized Mailbox

4.1 HTTP COMMUNICATION

RESTful HTTP communication is a good choice to allow web objects to communicate with each other as it requires no special intermediate services to stablish communication. Web objects can utilize the already established infrastructure of the Internet to communicate. But HTTP has certain limitations as well. It requires the client and server to meet in time in order to communicate. Many web services do not support REST hence we cannot leverage various methods of HTTP. Group messaging that we needed in our preservation system was also not possible in plain

HTTP communication. It may also cause Denial of Service (DoS) attack for web servers due to heavy HTTP communication load.

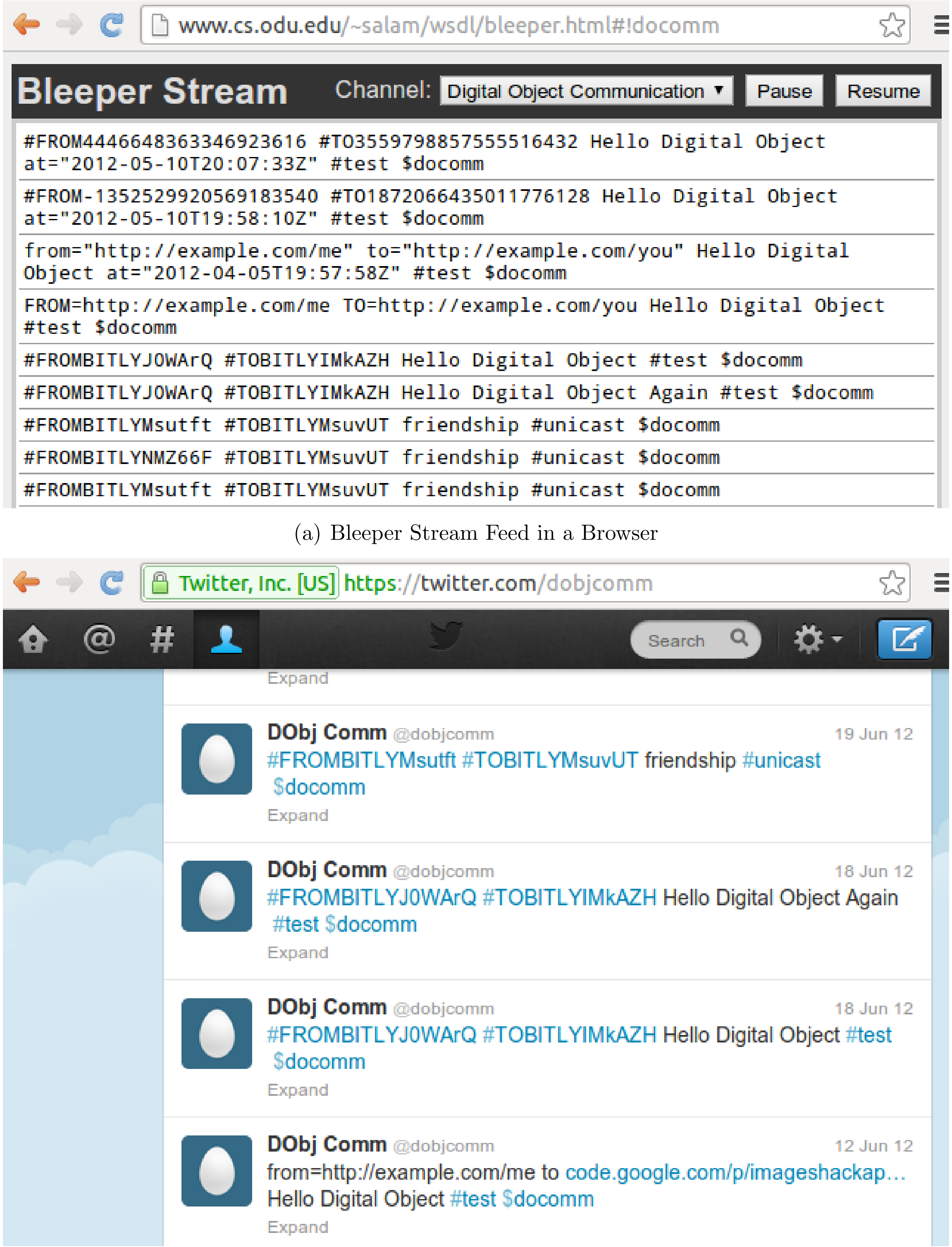
4.2 BLEEPS

To avoid the blocking nature of HTTP, we thought of having publish-subscribe style push notification system so that once a client sends a message on behalf of a web object to another web object hosted on a server, it may go on without worrying about the delivery of the message. This system requires the recipients to be available and listening to the push notification feed when there is any message for them. If a recipient was not available or a message was dropped for some reason, it will lose that message forever unless there is a message archiving service.

Bleeps is a live messaging system that is inspired by Twitter. It uses Push style communication [34] to broadcast small messages called Bleeps using relay channels. Anyone can subscribe to one or more such channels to receive live message feeds. We first explored Bleeps for the ResourceSync project [35, 36]. The message used in the Bleeps messaging system is called a Bleep. Bleeps supports Twitter-style 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 and can be captured by consumer applications or other services. A Bleep message is intended to be transported over Twitter (or like services) infrastructure, hence the length cannot be more than 140 characters as this is the limit imposed by Twitter on the maximum length of a tweet. Also, the message needs to be structured enough to make the parsing easy according to the attached language descriptor.

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

In this example above, “$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.



www.cs.odu.edu/~salam/wsdl/bleeper.html#!docomm
Bleeper Stream
Channel: Digital Object Communication ▼
Pause
Resume

```
#FROM4446648363346923616 #TO3559798857555516432 Hello Digital Object
at="2012-05-10T20:07:33Z" #test $docomm
#FROM-1352529920569183540 #TO1872066435011776128 Hello Digital Object
at="2012-05-10T19:58:10Z" #test $docomm
from="http://example.com/me" to="http://example.com/you" Hello Digital
Object at="2012-04-05T19:57:58Z" #test $docomm
FROM=http://example.com/me TO=http://example.com/you Hello Digital Object
#test $docomm
#FROMBITLYJ0WArQ #TOBITLYIMkAZH Hello Digital Object #test $docomm
#FROMBITLYJ0WArQ #TOBITLYIMkAZH Hello Digital Object Again #test $docomm
#FROMBITLYMsutft #TOBITLYMsuvUT friendship #unicast $docomm
#FROMBITLYNMZ66F #TOBITLYMsuvUT friendship #unicast $docomm
#FROMBITLYMsutft #TOBITLYMsuvUT friendship #unicast $docomm
```

(a) Bleeper Stream Feed in a Browser

DObj Comm @dobjcomm 19 Jun 12
#FROMBITLYMsutft #TOBITLYMsuvUT friendship #unicast $docomm
Expand

DObj Comm @dobjcomm 18 Jun 12
#FROMBITLYJ0WArQ #TOBITLYIMkAZH Hello Digital Object Again #test $docomm
Expand

DObj Comm @dobjcomm 18 Jun 12
#FROMBITLYJ0WArQ #TOBITLYIMkAZH Hello Digital Object #test $docomm
Expand

DObj Comm @dobjcomm 12 Jun 12
from=http://example.com/me to code.google.com/p/imageshackap… Hello Digital Object #test $docomm
Expand

(b) Bleeper Stream Stored in Twitter

Fig. 3. Bleeper Stream: Live and Stored.

We implemented a Bleeps server using a messaging system called Faye [37] that uses the Bayeux protocol [38]. We called our implementation Bleeper. Bleeper had various push notification channels including “Digital Object Communication”, “DBpedia Live”, and “News Feeds”. Fig. 3(a) shows a web application that is listening to the Bleeper notifications over “Digital Object Communication” channel.

To make these Bleeps persistent, we stored them in Twitter as shown in Fig. 3(b). We tried various encoding methods in the Bleeps messages to enable easy searching. But Twitter has indexed only few initial messages then categorized our account as a bot and removed it from search index.

4.3 MICRO-BLOGGING

To allow interoperability, we considered defining some standard but flexible message formats which are compact in size and extensible to adopt any type of messages. We utilized the message format used in Bleeps, which contains a language identifier in the message itself that describes the template of the message for easy parsing. We were able to fit most of the messages used in our web preservation application in very small text strings (less than 140 characters) so we thought we could use Twitter as the communication medium.

To use Twitter or like services, we designed and evaluated following three different modes of operation.

- Hashtag Model
- Scribe Model
- Hashtag Model With Add-Ons

4.3.1 HASHTAG MODEL

Suppose that Alice and Bob are web objects and they are trying to communicate using this hashtag-based Twitter communication model. Alice and Bob will chose unique hashtags or their URIs that can identify them. If the URIs are long then a Uniform Resource Locator (URL) shortening service (e.g., Bitly [39]) can be used, otherwise Twitter will shorten the URL itself. The search capability of Twitter (or like platforms) is utilized to facilitate asynchronous communication with multicast.

Now if Alice wants to send a message to Bob, she will post a tweet utilizing $docomm (a well-defined tweet format). Bob can search for $docomm messages in which Bob's identifier (i.e., hashtag or URI) has appeared as the recipient. Fig. 4 illustrates the work-flow of the hashtag model.

We have implemented this model using Twitter but could not succeed for more than a few days. Twitter prunes its search index after 6 to 9 days. This means the messages will be there but will not appear in the search results after a week of posting. Also Twitter has identified our test accounts as bots and excluded them from the search index. This exclusion made this model unusable as it relies on the search capability. According to the Twitter search rules and restrictions [40] they may automatically remove accounts from search or suspend it if they detect some robotic activities like repeatedly posting duplicate or near duplicate content, automated tweets, or posting similar messages over multiple accounts.

After facing restrictions imposed by Twitter we tried to implement this model using StatusNet [41] software. StatusNet is an open source micro-blogging software that can be installed on a web server as an alternative to Twitter. StatusNet allowed us to increase the message length and keep the search index for long time. But the search capabilities of StatusNet were limited as it does not allow expanded URL search if the URI was shortened in the message.

4.3.2 SCRIBE MODEL

Scribe is a service that utilizes Twitter (or like platforms) to facilitate asynchronous communication with multicast facility. It works similar to the hashtag

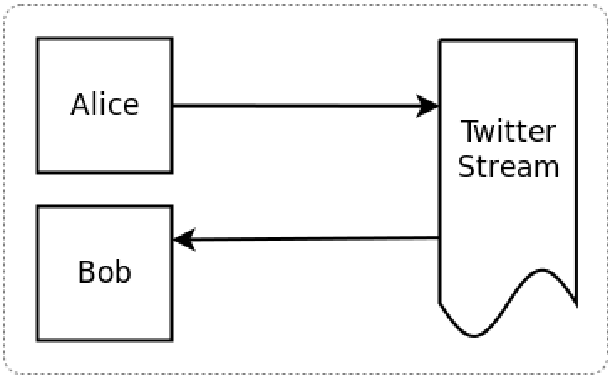


Fig. 4. Twitter-based Communication Using Hashtags or URIs.

model but the sender and recipients do not communicate directly. The scribe service processes the messages and sends alert messages to its subscribers. The scribe service works as the trusted intermediate to allow safe communication among web objects.

Now if Alice wants to send a message to Bob, she will post a tweet using a Bleep format and mention the accounts of one or more well-known scribe services in the tweet. These scribe services watch for $docomm (a well-defined Bleep message format) messages in which they are mentioned. Alice will use the unique identifier (hastag or URI) of Bob in the message as described in the $docomm format. Scribe services perform filtering, spam cleaning, and other tasks on those tweets then post a $docommalert (another well-defined Bleep format that contains reference to a corresponding $docom message) messages using one of their several subordinate Twitter accounts. They use multiple subordinate accounts (e.g., ScrOut1 and ScrOut2 as illustrated in Fig. 5) to bypass the limits imposed by Twitter on number of tweets

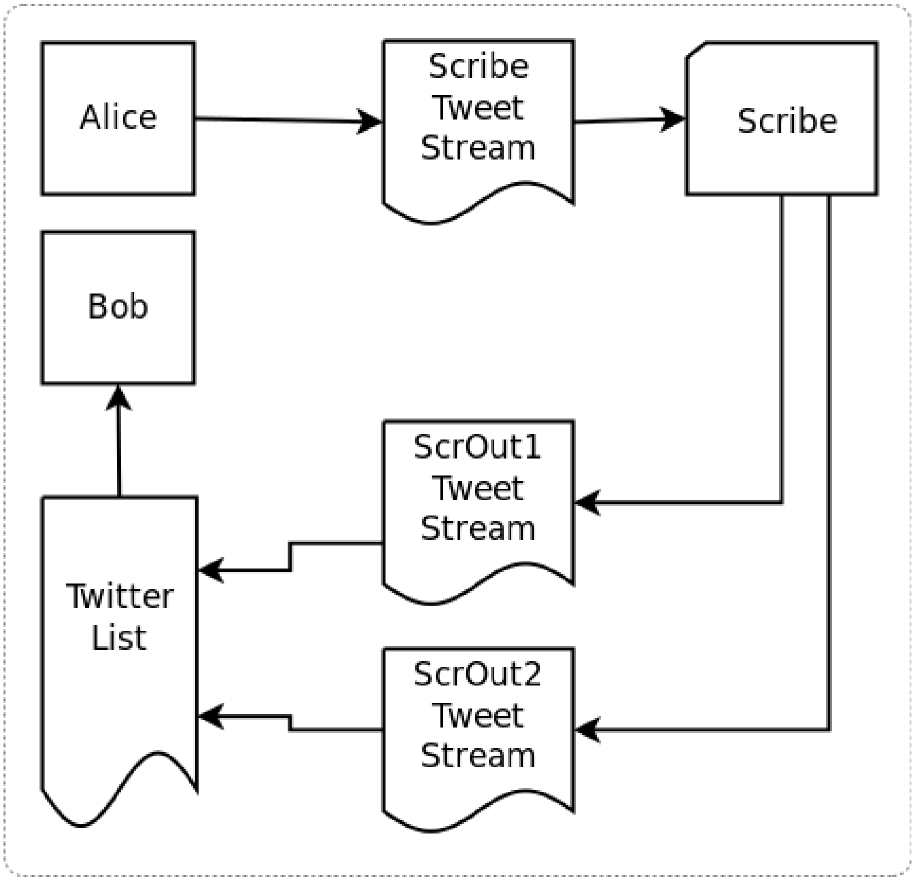


Fig. 5. Twitter-based Communication Using Scribe Service.

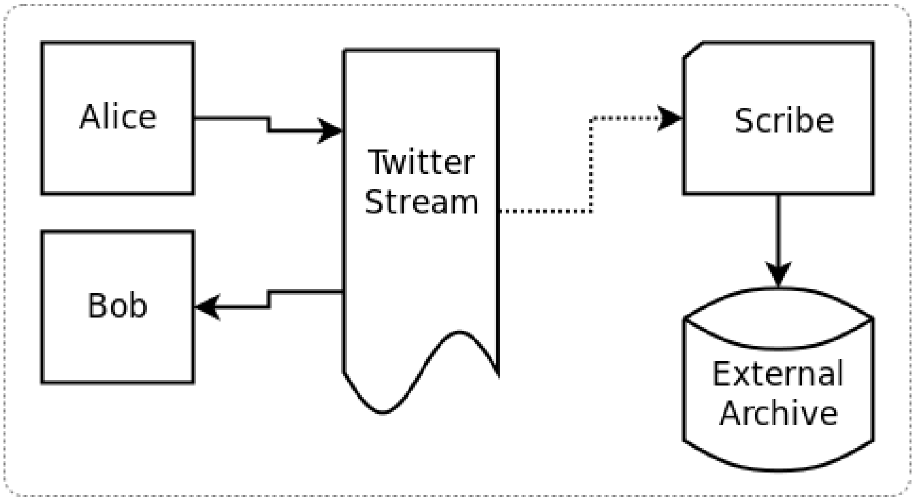


Fig. 6. Twitter-based Communication Using Hashtags and Add-on Services.

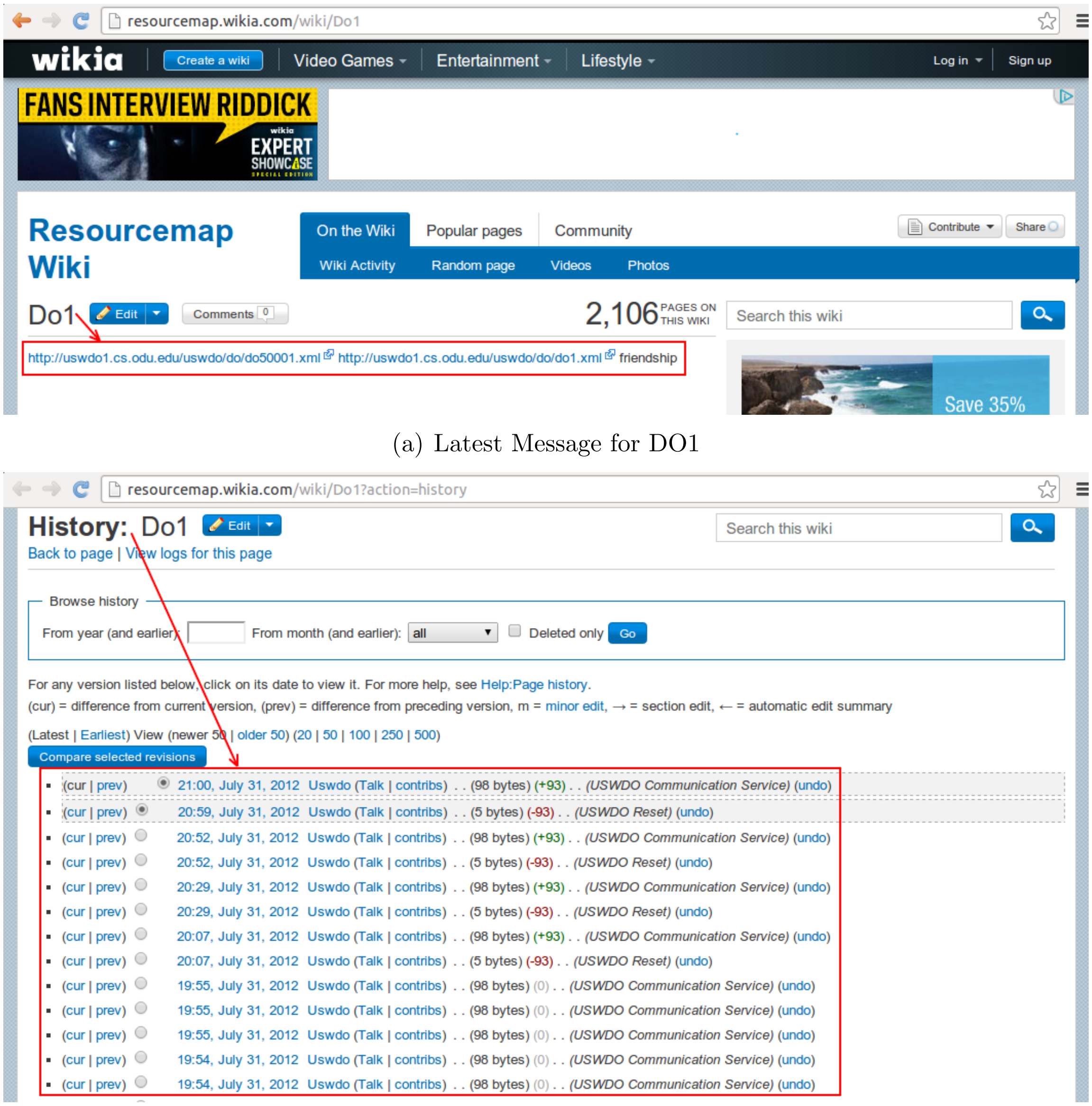
that an account can post in a certain period of time. All these accounts are then aggregated together in a “Twitter List” that is followed by various web objects. If a $docommalert corresponding to the $docomm message sent by Alice was posted in the Twitter list that Bob follows then Bob can access Alice’s message using the reference available in the $docommalert message. Fig. 5 illustrates the work-flow of the scribe model.

We have faced similar issues from Twitter as in the case of hashtag model. While this model facilitates centralized spam filtering, it creates a bottleneck. Failure of the scribe service will cause the failure of communication among the web objects subscribed to the scribe service.

4.3.3 HASHTAG MODEL WITH ADD-ONS

In this model, web objects are allowed to communicate using the hashtag model but there are add-on scribe services that can perform spam filtering, message archiving, and indexing to provide search facility later when Twitter fails to search the messages. Fig. 6 illustrates the work-flow of the this model.

These add-on services solve the problem of Twitter pruning the search index periodically but these services rely on the search capability of the Twitter. If Twitter has marked an account as bot or spam and does not index its tweets then add-on services will not be able to discover them to process, index, and archive.



Do1

http://uswdo1.cs.odu.edu/uswdo/do/do50001.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship

(a) Latest Message for DO1

resourcemap.wikia.com/wiki/Do1?action=history
Edit
Search this wiki
Browse history
From year (and earlier):
From month (and earlier): all
Deleted only
Go
Compare selected revisions

History: Do1

Back to page | View logs for this page

For any version listed below, click on its date to view it. For more help, see Help:Page history.

(cur) = difference from current version, (prev) = difference from preceding version, m = minor edit, → = section edit, ← = automatic edit summary

(Latest | Earliest) View (newer 50 | older 50) (20 | 50 | 100 | 250 | 500)

- (cur | prev) ● 21:00, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (+93) . . (USWDO Communication Service) (undo)
- (cur | prev) ● 20:59, July 31, 2012 Uswdo (Talk | contribs) . . (5 bytes) (-93) . . (USWDO Reset) (undo)
- (cur | prev) ○ 20:52, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (+93) . . (USWDO Communication Service) (undo)
- (cur | prev) ○ 20:52, July 31, 2012 Uswdo (Talk | contribs) . . (5 bytes) (-93) . . (USWDO Reset) (undo)
- (cur | prev) ○ 20:29, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (+93) . . (USWDO Communication Service) (undo)
- (cur | prev) ○ 20:29, July 31, 2012 Uswdo (Talk | contribs) . . (5 bytes) (-93) . . (USWDO Reset) (undo)
- (cur | prev) ○ 20:07, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (+93) . . (USWDO Communication Service) (undo)
- (cur | prev) ○ 20:07, July 31, 2012 Uswdo (Talk | contribs) . . (5 bytes) (-93) . . (USWDO Reset) (undo)
- (cur | prev) ○ 19:55, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (0) . . (USWDO Communication Service) (undo)
- (cur | prev) ○ 19:55, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (0) . . (USWDO Communication Service) (undo)
- (cur | prev) ○ 19:55, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (0) . . (USWDO Communication Service) (undo)
- (cur | prev) ○ 19:54, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (0) . . (USWDO Communication Service) (undo)
- (cur | prev) ○ 19:54, July 31, 2012 Uswdo (Talk | contribs) . . (98 bytes) (0) . . (USWDO Communication Service) (undo)

(b) Revision History of DO1's Mailbox Page

Fig. 7. Wikia Page as the Mailbox for the Web Object DO1.

4.4 DECENTRALIZED MAILBOX

To have persistent storage of messages, we thought of having decentralized personal and group mailboxes for every web object chosen by web object owners independently and advertising those mailboxes through the object's ResourceMap [42].

A ResourceMap is a file that describes a collection of resources known as an aggregation (see appendix B for an example ResourceMap file). A web object owner may choose Wiki, Blog, Micro-blog or virtually any web service that allows storage and retrieval of text data.

We have implemented this using `wikia.com` which hosts MediaWiki instances for public usage. We created individual Wiki pages for each web object that was working as a mailbox for the associated web object. Anyone can rewrite the Wiki page (using the Wiki API) with a message. To access all the messages written on that page one can retrieve the history of that Wiki page (using the Wiki API). Fig. 7(a) shows a Wikia-based mailbox page for a web object DO1, Fig. 7(b) shows the revision history of the same page, and Code 22 in appendix B illustrates how to retrieve the revisions of this page to consume them as messages. We realized that there were two major challenges in this approach. The first challenge was the scale, as it was difficult to create individual mailboxes for every ResourceMap. And the second challenge was interoperability. There was no uniform interface which allows every web object to communicate with a variety of mailboxes hosted on different public web services. There was also a chance that these web services might detect robotic activities and block the service.

4.5 SUMMARY

In this chapter we described various communication techniques and models that we built in order to enable browser-based cross-domain web communication. We started with a push-style messaging system called Bleeps, but it failed due to its volatile nature. Messages in Bleeps were not persistent, hence they were lost if a recipient failed to capture them in time. Then we tried to store these Bleeps in various web services like Wiki, Twitter, and StatusNet for persistence. Some of these experiments failed due to the scalability problem and some failed because Twitter did not index the messages for searching.

CHAPTER 5

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.

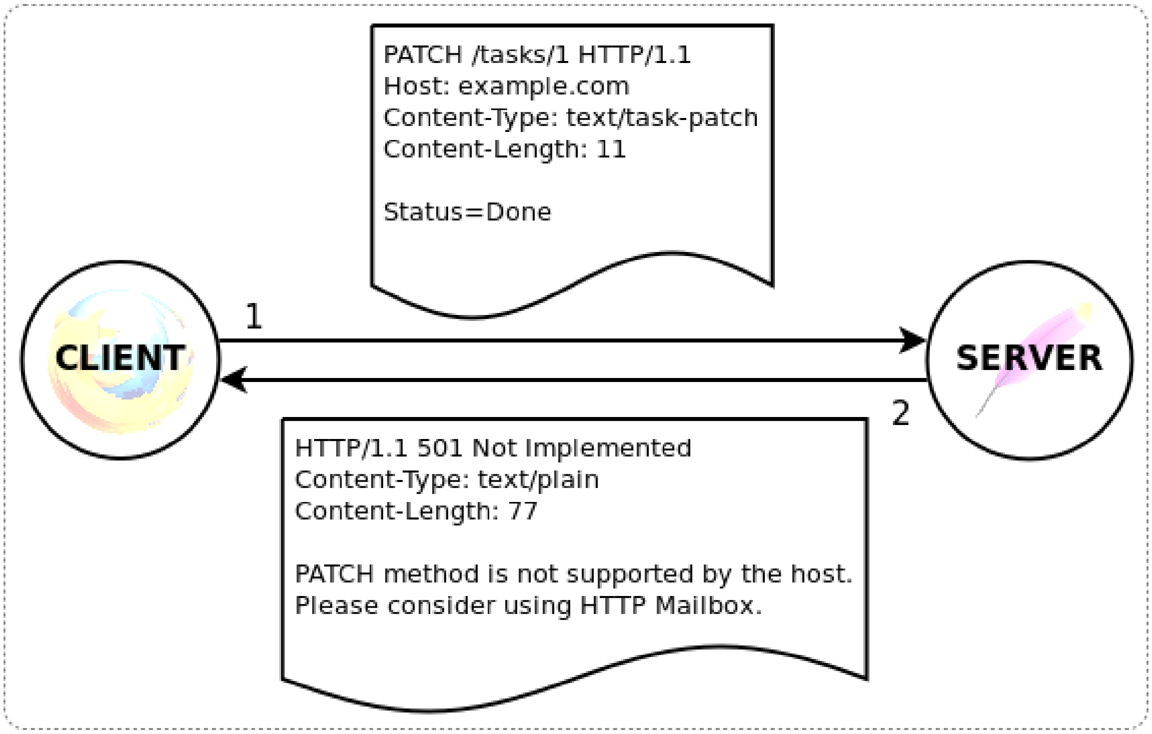
In HTTP Mailbox messaging, HTTP requests are encapsulated inside another HTTP Message entity to form an envelope request. A client makes an HTTP POST Request to the HTTP Mailbox irrespective of the method of the encapsulated HTTP message. HTTP Mailbox then stores the encapsulated HTTP Request along with various message metadata in a persistent storage. Later, to retrieve those stored messages, a client makes an HTTP GET request to the HTTP Mailbox. Fig. 8 shows a typical HTTP Request and Response cycle. Fig. 9 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.

An initial description of the HTTP Mailbox was published as a technical report [43]. Since then it has significantly advanced in functionality. Some of the features described in the future work section of the technical report have also been implemented.

5.1 MAPPING LINDA TO 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.



```
PATCH /tasks/1 HTTP/1.1
Host: example.com
Content-Type: text/task-patch
Content-Length: 11

Status=Done
```

```
HTTP/1.1 501 Not Implemented
Content-Type: text/plain
Content-Length: 77

PATCH method is not supported by the host.
Please consider using HTTP Mailbox.
```

Fig. 8. Typical HTTP Messaging Scenario.

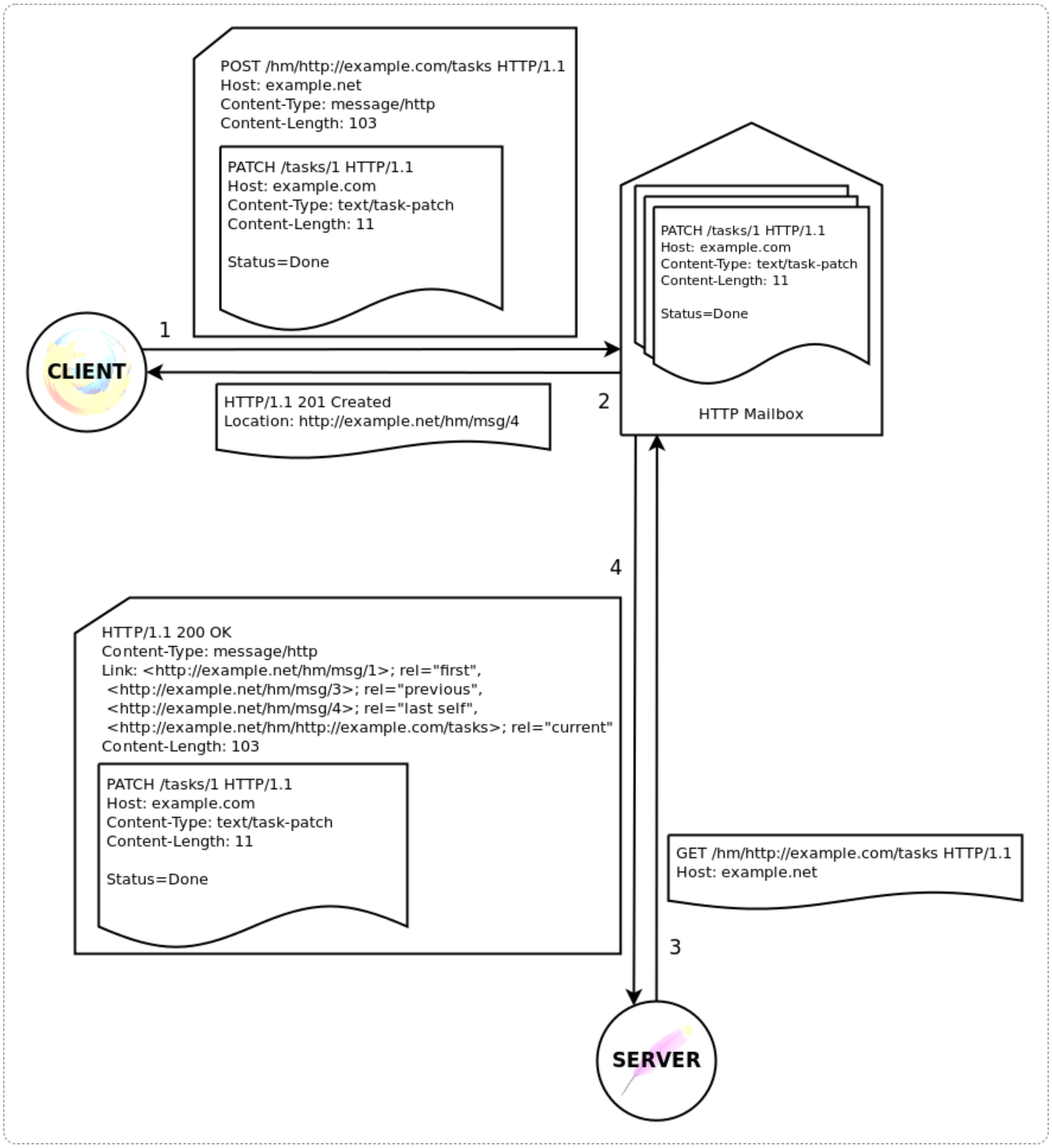
A URI or any other identifier of the recipient(s) can be used to query messages from the distributed message store similar to the expressions used in “rd” and “in” functions of Linda to query the tuplespace.

Table III summarizes the transformation of Linda functions into their HTTP equivalents.

TABLE III
HTTP Equivalents to Linda Functions

Linda	HTTP Equivalent
in()	GET followed by DELETE
rd()	GET
out()	POST
eval()	Execute Request followed by optional POST (Response)

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 (see chapter 7). Instead, flagging messages as deleted (and keeping a history of actions) may be a better choice because



```
POST /hm/http://example.com/tasks HTTP/1.1
Host: example.net
Content-Type: message/http
Content-Length: 103

PATCH /tasks/1 HTTP/1.1
Host: example.com
Content-Type: text/task-patch
Content-Length: 11

Status=Done
```

```
PATCH /tasks/1 HTTP/1.1
Host: example.com
Content-Type: text/task-patch
Content-Length: 11

Status=Done
```

```
HTTP/1.1 201 Created
Location: http://example.net/hm/msg/4
```

```
HTTP/1.1 200 OK
Content-Type: message/http
Link: <http://example.net/hm/msg/1>; rel="first",
 <http://example.net/hm/msg/3>; rel="previous",
 <http://example.net/hm/msg/4>; rel="last self",
 <http://example.net/hm/http://example.com/tasks>; rel="current"
Content-Length: 103

PATCH /tasks/1 HTTP/1.1
Host: example.com
Content-Type: text/task-patch
Content-Length: 11

Status=Done
```

```
GET /hm/http://example.com/tasks HTTP/1.1
Host: example.net
```

Fig. 9. HTTP Mailbox Store and On-demand Delivery Scenario.

storage is not as limited as in case of pure Linda shared memory.

The “rd” 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 “out” function of Linda may refer to the action of preparing the 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.

5.2 HTTP MESSAGE

In HTTP communication, an HTTP Message [2] is either an HTTP Request or an HTTP Response. An HTTP Request is a message that is sent from the client to the server. It contains a mandatory request line followed by optional headers and optional body. The request line contains the HTTP method used in the request, the URI of the resource, and the protocol name and its version. Headers are name-value pairs separated by a colon (:). The body of the message can be any type of data which has a Media type. An HTTP Response is returned from the server to the client in response to an HTTP Request. It contains a mandatory status line followed by optional headers and optional body. The status line contains the protocol name and version followed by a response code (a three digit number) and a response message (explanation of the response code). The headers and the body of the HTTP Response have same formats as the HTTP Request.

“HTTP Request” and “HTTP Response” 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”.

5.3 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 REST style.

Code 8 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 8 Lines 1-12. This POST request is made to the HTTP Mailbox on a different domain. It also has a different `path`, `Content-Type` and `Content-Length` as illustrated in Code 8 Lines 1-5.

Code 8. POST HTTP Mailbox Request

```
 1 > POST /hm/http://example.com/tasks HTTP/1.1
 2 > Host: example.net
 3 > HM-Sender: http://example.org/alice
 4 > Content-Type: message/http; msgtype: request
 5 > Content-Length: 108
 6 >
 7 > PATCH /tasks/1 HTTP/1.1
 8 > Host: example.com
 9 > Content-Type: text/task-patch
10 > Content-Length: 11
11 >
12 > Status=Done
13
14 < HTTP/1.1 201 Created
15 < Location: http://example.net/hm/id/5ecb44e0
16 < Date: Thu, 20 Dec 2012 02:22:56 GMT
```

On a successful POST operation, 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 8 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.

Code 9. GET HTTP Mailbox Request

```
> GET /hm/http://example.com/tasks HTTP/1.1
> Host: example.net

< HTTP/1.1 200 OK
< Date: Thu, 20 Dec 2012 02:10:22 GMT
< Link: <http://example.net/hm/id/aebed6e9>; rel="first",
<  <http://example.net/hm/id/5ecb44e0>; rel="last self",
<  <http://example.net/hm/id/85addc19>; rel="previous",
<  <http://example.net/hm/http://example.com/tasks>;rel="current"
< Via: Sent by 127.0.0.1
<  on behalf of http://example.org/alice
<  delivered by http://example.net/
< Content-Type: message/http; msgtype: request
< Content-Length: 108
<
< PATCH /tasks/1 HTTP/1.1
< Host: example.com
< Content-Type: text/task-patch
< Content-Length: 11
<
< Status=Done
```

A client on behalf of `http://example.com/tasks` can then perform an HTTP GET request to the HTTP Mailbox as illustrated in Code 9 Lines:1-2 and get an HTTP response as illustrated in Code 9 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 8 and Code 9 and shown in Fig. 9

5.4 INDIRECT HTTP RESPONSE

After fetching messages from HTTP Mailbox and with the help of `Content-Type` and `Content-Length` headers as illustrated in Code 9 Lines 13-14, the server can parse the encapsulated HTTP PATCH request as illustrated in Code 9 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 10. This process is similar to the "eval" function of Linda.

Code 10. POST HTTP Mailbox Response

```
> POST /hm/http://example.org/alice HTTP/1.1
> Host: example.net
> HM-Sender: http://example.com/tasks
> Content-Type: message/http; msgtype: response
> Content-Length: 93
>
> HTTP/1.1 200 OK
> Content-Type: text/plain
> Content-Length: 28
>
> (Done) [HIGH] Write a paper.

< HTTP/1.1 201 Created
< Location: http://example.net/hm/id/32ab1ce2
< Date: Thu, 20 Dec 2012 02:31:12 GMT
```

Code 11. GET HTTP Mailbox Response

```
> GET /hm/http://example.org/alice HTTP/1.1
> Host: example.net

< HTTP/1.1 200 OK
< Date: Thu, 20 Dec 2012 02:42:03 GMT
< Link: <http://example.net/hm/id/26d1a9c2>;rel="first previous",
<  <http://example.net/hm/id/32ab1ce2>; rel="last self",
<  <http://example.net/hm/http://example.org/alice>;rel="current"
< Via: Sent by 127.0.0.2
<  on behalf of http://example.com/tasks
<  delivered by http://example.net/
< Content-Type: message/http; msgtype: response
< Content-Length: 93
<
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 28
<
< (Done) [HIGH] Write a paper.
```

Later, Alice wants to check to see if her change was made, so she queries the HTTP Mailbox as illustrated in Code 11. If Bob's server has updated Alice's task list and sent a response to the HTTP Mailbox for Alice, then the response of Bob's server will be included in the HTTP Mailbox response to Alice's query as illustrated in Code 11 Lines 15-19.

5.5 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 Request and Response cycle of the HTTP messaging.

Fig. 10 summarizes the process of the HTTP Mailbox communication on both sender and recipient ends. On the client (sender) end the HTTP cycle completes in the following steps:

- A generic HTTP Message (Request or Response) (C1) is to be sent,
- The HTTP Message is encapsulated in an HTTP POST Request to the HTTP Mailbox (using `message/http` Media type) (C2) or a Pipeline of one or more HTTP Message(s) is encapsulated in an HTTP POST Request to the HTTP Mailbox (using `application/http` Media type) (C2'), and
- An HTTP Response is received from HTTP Mailbox (C3).

On the server (recipient) end the HTTP cycle completes in the following steps:

- An HTTP GET Request to the HTTP Mailbox (S1) is made to retrieve an HTTP Message,
- The HTTP Message is encapsulated in an HTTP Response from the HTTP Mailbox (using `message/http` Media type) (S2) or a Pipeline of one or more HTTP Message(s) encapsulated in an HTTP Response from the HTTP Mailbox (using `application/http` Media type) (S2'), and
- A generic HTTP Message (Request or Response) is extracted from the HTTP Mailbox Response (S3).

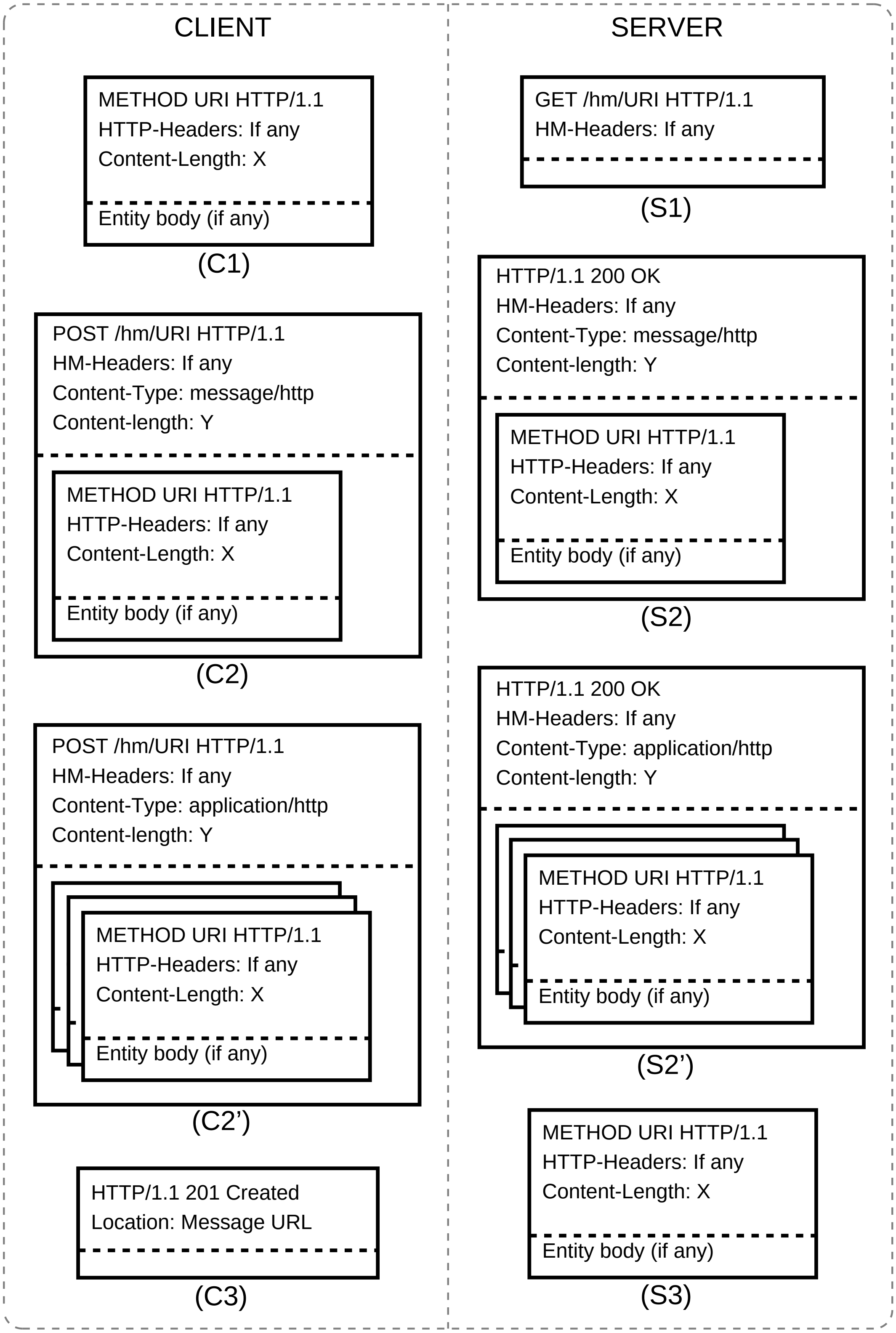


Fig. 10. HTTP Mailbox Lifecycle on Client (Sender) and Server (Recipient) Sides.

5.5.1 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 an 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 the recipients' identifier appended to the `HM-Base` of the mailbox as advertised by the HTTP Mailbox service on a well-known URI (or root URI of the service). The HTTP Mailbox service host as `Host` header and other extra metadata should go in the HTTP headers (to be described in section 6.5). The entity must be a valid `message/http` or `application/http` request and an appropriate `Content-Type` header must be present in the request headers (portions C2 and C2' of Fig. 10).

5.5.2 SEND RESPONSE

Send Response 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 (portion C3 of Fig. 10). 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).

5.5.3 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` (to be described in section 6.2) of the mailbox as advertised by the HTTP Mailbox service. The HTTP Mailbox service host as `Host` header and 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 (portion S1 of Fig. 10).

5.5.4 RETRIEVE RESPONSE

Retrieve Response 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, and the 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` [44] 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 (to be described in section 6.7), 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 (i.e., `4xx/5xx`) should be returned (portions S2 and S2' of Fig. 10).

5.6 SUMMARY

In this chapter we described the HTTP Mailbox. After experimenting with various communication models, we realized that if we can enable time-uncoupling and group messaging in HTTP communication, then we can easily utilize it in a browser-based web communication. We created a mailbox for HTTP messages that is inspired by the tuplespace of the Linda model. We discussed encapsulation of an HTTP Message in another HTTP Message and various phases of HTTP Mailbox communication.

The HTTP Mailbox not only enabled time-uncoupling and group communication, but it has also removed the cross-domain restriction (to be discussed in section 6.8) and enabled the full range of HTTP methods. The HTTP Mailbox has accumulated all the advantages of Linda, REST and Relay HTTP.

CHAPTER 6

MAILBOX API

One of the REST principles is Hypermedia as the Engine of Application State (HATEOAS) [16, 45]. According to this, a client needs no prior knowledge about how to interface with a RESTful service, except the generic understanding of relation types [46] and MIME types [47]. A client begins interaction with the service from a fixed URI 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 [48].

6.1 RECIPIENT

In the HTTP Mailbox communication model, the recipient identifier is flexible. It can be a URI, string literal, or a mix of the two, as opposed to the synchronous HTTP communication where the target should always be a URI or path (of a single resource). This flexibly enables group communication (multicast). To illustrate various ways to indicate the recipients in the HTTP Mailbox requests, consider the following scenarios.

Suppose that Alice and Bob were classmates in year 2000. A few years later Alice wants to send a message to Bob, and Bob can be reached at `http://example.com/bob`. She will send an HTTP Mailbox request as follows.

```
POST /hm/http://example.com/bob HTTP/1.1
```

Alice now wants to send a message to all of her classmates. Suppose that there was a home page for her class in her school's website at `http://example.edu/class/2000`. If she chooses to take this URI as an identifier for the group of her class mates, she can send an HTTP Mailbox request as follows.

```
POST /hm/http://example.edu/class/2000 HTTP/1.1
```

Now consider that a sub-group of Alice's class mates were known as "lazy-geeks" and she wants to send message to only this sub-group. She can send an HTTP Mailbox request as follows.

```
POST /hm/lazy-geeks HTTP/1.1
```

We have also explored another possibility of group messaging based on relationships. In this type of group messaging we can utilize well-known relations like “friends” or “family” (defined somewhere in a Resource Description Framework (RDF) [49] vocabulary) to infer the recipients. For example if Alice wants to send a message to all of the friends of Bob, she can use the literal relationship “friends” followed by the identifier of Bob (i.e., `http://example.com/bob`). She can send an HTTP Mailbox request as follows.

```
POST /hm/friends/http://example.com/bob HTTP/1.1
```

The recipients’ identifier is same for both POST and GET requests. A message can be retrieved by making a GET request to the exact same recipient identifier as it was used at the time of sending the message using POST request, if it is the most recent message for that recipient, otherwise URI of the message will be required to retrieve it.

6.1.1 RELATIONSHIP RESOLUTION

A ResourceMap can be used to resolve the relationship in a recipients’ identifier. A web object can have links in its ResourceMap to all the resources that are friends, family members, or related to the web object with any other relationship as illustrated in appendix B Code 21 Lines 78 and 101. Individual recipients can be identified using the definition of the relationship and the ResourceMap. It is similar to the relationship found in the Friend of a Friend (FoaF) [50] or social networks.

A relationship can be resolved at the time of message sending or at the time of retrieval. If there is significant time delay between sending and retrieving messages then there might be significant difference between the two resolution scenarios. The choice of one of the two resolution scenarios is application dependent.

A relationship can be resolved by the HTTP Mailbox service or it can be offloaded to the client. For the sake of simplicity, we have chosen the latter option in our reference implementation. Adding this knowledge in the HTTP Mailbox itself will affect the performance of the system significantly, and it will add several external dependencies in the HTTP Mailbox.

6.2 HM-REQUEST-PATH

An HTTP Mailbox service will advertise its base path (or base URI) for messaging called `HM-Base` (see appendix A). This `HM-Base` will be used to construct the 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/`.

The `HM-Request-Path` consists of three parts, the `HM-Base`, followed by an optional parameter (used in GET requests), followed by a recipients' identifier. We will discuss the optional parameter in sections 6.3 and 6.4. The recipients' identifier can be a URI or any URL-encoded string token. Codes 8 and 9 have `http://example.com/tasks` as the recipient identifier in their first lines. A typical `HM-Request-Path` is illustrated below.

```
/hm/http://example.com/tasks
```

The corresponding request URI is illustrated below:

```
http://example.net/hm/http://example.com/tasks
```

The 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 the recipient identifier while the enclosed entity has `http://example.com/tasks/1` as its URI.

6.3 TIME BASED ACCESS

A GET request to `HM-Request-Path` as illustrated in section 6.2 returns the most recent message from the message chain. The most recent message gives the URIs of the first and last messages in the chain which allows the traversal of the message chain from either direction.

Suppose Bob's task manager service has been using the HTTP Mailbox for a long time, and it has a large number of messages in its message chain. Bob's server likes to retrieve and process messages in chronological order (i.e., from the first message to the most recent message). Bob's server has recently checked his mailbox and read all the messages in its message chain. Later, if Bob's server checks its mailbox again then it can either traverse through all the earlier messages it has already processed to reach to the place it has left the message chain last time, or it can retrieve only the unprocessed messages in reverse-chronological order and reverse their order locally before processing. To avoid this extra work, Bob's server can store the URI of the last processed message to start traversal again from there.

The HTTP Mailbox provides a way to access the message chain based on the time when a message was first seen by the HTTP Mailbox (nearly when the message was sent). In this case an optional time parameter is passed between the `HM-Base` and the recipients' identifier. This will return the earliest message from the message chain that was added in the chain after the given time, if any. The time parameter needs to be a 14 digit integer in `YYYYMMDDHHMMSS` format where symbols correspond to year, month, day, hour, minute, and second respectively. This time needs to be in UTC time zone.

If Bob's task server wants to retrieve the earliest message after 2013-02-17 02:46:05 then it can make a GET request to the following `HM-Request-Path`.

```
/hm/20130217024605/http://example.com/tasks
```

This request format with optional time parameter is only available to GET requests.

6.4 RESPONSE PAGINATION

So far we have only discussed retrieval of one message from a message chain at a time. This retrieval mechanism is simple but time consuming because every message requires a round trip HTTP communication. This is impractical if large number of messages are to be retrieved from the mailbox.

To overcome this problem, the HTTP Mailbox facilitates response pagination. Every message in the message chain gets a sequence number starting from 0 for the first message and incremented by 1 till the last message of the chain. To retrieve a page of multiple consecutive messages from the message chain with a single request, the client needs to send the starting and the ending sequence numbers separated by a minus sign (-) as an optional parameter between the `HM-Base` and the recipients' identifier. The two numbers need to be positive integers or zero, and the starting

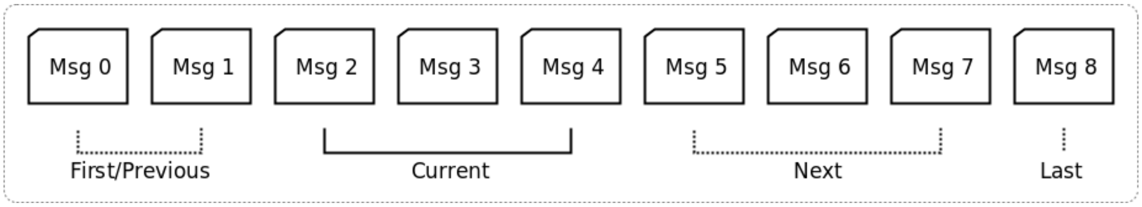


Fig. 11. Pagination in the Message Chain.

sequence number must not be greater than the ending sequence number. The HTTP Mailbox will determine the page size based on the given range. The HTTP Mailbox will also determine the links of `first`, `last`, `next` and `previous` pages of the same size based on the given range in the current request. The following equations can be used to determine appropriate page navigation links.

$$\begin{aligned}
LS &= \text{Sequence number of the last message in the message chain} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Last Sequence number)} \\
BC &= \text{Starting sequence number from the request parameter} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Beginning of the Current page)} \\
EC &= \text{Ending sequence number from the request parameter} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(End of the Current page)} \\
PS &= EC - BC + 1 && \text{(Page Size)} \\
BP &= \max(0, BC - PS) \ \{\text{if } BC > 0\} && \text{(Beginning of the Previous page)} \\
EP &= BC - 1 \ \{\text{if } BC > 0\} && \text{(End of the Previous page)} \\
BN &= EC + 1 \ \{\text{if } EC < LS\} && \text{(Beginning of the Next page)} \\
EN &= \min(LS, EC + PS) \ \{\text{if } EC < LS\} && \text{(End of the Next page)} \\
BF &= 0 && \text{(Beginning of the First page)} \\
EF &= \min(LS, PS - 1, (BC + PS - 1) \bmod PS) && \text{(End of the First page)} \\
BL &= \max(LS - ((LS - BC) \bmod PS)) && \text{(Beginning of the Last page)} \\
EL &= LS && \text{(End of the Last page)}
\end{aligned}$$

Code 12. Paginated GET HTTP Mailbox Request

```
> GET /hm/2-4/http://example.com/tasks HTTP/1.1
> Host: example.net

< HTTP/1.1 200 OK
< Date: Thu, 20 Dec 2012 02:10:22 GMT
< Link: <http://example.net/hm/2-4/http://example.com/tasks>; rel="self",
<  <http://example.net/hm/0-1/http://example.com/tasks>; rel="first previous",
<  <http://example.net/hm/8-8/http://example.com/tasks>; rel="last",
<  <http://example.net/hm/5-7/http://example.com/tasks>; rel="next"
< Content-Type: application/http
< Content-Length: 1985
<
< HTTP/1.1 200 OK
< Memento-Datetime: Tue, 18 Dec 2012 05:15:55 GMT
< Link: <http://example.net/hm/id/d1472c17>; rel="self",
<  <http://example.net/hm/id/aebed6e9>; rel="first",
<  <http://example.net/hm/id/5ecb44e0>; rel="last",
<  <http://example.net/hm/id/25ad905c>; rel="next",
<  <http://example.net/hm/id/85addc19>; rel="previous",
<  <http://example.net/hm/http://example.com/tasks>;rel="current"
< Via: Sent by 127.0.0.1
<  on behalf of http://example.org/alice
<  delivered by http://example.net/
< Content-Type: message/http; msgtype: request
< Content-Length: 108
<
< PATCH /tasks/1 HTTP/1.1
< Host: example.com
< Content-Type: text/task-patch
< Content-Length: 11
<
< Status=Done
<
< HTTP/1.1 200 OK
< Memento-Datetime: Tue, 18 Dec 2012 01:24:13 GMT
< Link: <http://example.net/hm/id/25ad905c>; rel="self",
<  <http://example.net/hm/id/aebed6e9>; rel="first",
<  <http://example.net/hm/id/5ecb44e0>; rel="last",
<  <http://example.net/hm/id/7a150ade>; rel="next",
<  <http://example.net/hm/id/d1472c17>; rel="previous",
<  <http://example.net/hm/http://example.com/tasks>;rel="current"
< Via: Sent by 127.0.0.1
<  on behalf of http://example.org/alice
<  delivered by http://example.net/
< Content-Type: message/http; msgtype: request
< Content-Length: 42
<
< DELETE /tasks/2 HTTP/1.1
< Host: example.com
<
< HTTP/1.1 200 OK
< Memento-Datetime: Wed, 19 Dec 2012 21:21:25 GMT
< Link: <http://example.net/hm/id/7a150ade>; rel="self",
<  <http://example.net/hm/id/aebed6e9>; rel="first",
<  <http://example.net/hm/id/5ecb44e0>; rel="last",
<  <http://example.net/hm/id/15a3dd12>; rel="next",
<  <http://example.net/hm/id/25ad905c>; rel="previous",
<  <http://example.net/hm/http://example.com/tasks>;rel="current"
< Via: Sent by 127.0.0.1
<  on behalf of http://example.org/alice
<  delivered by http://example.net/
< Content-Type: message/http; msgtype: request
< Content-Length: 106
<
< PATCH /tasks HTTP/1.1
< Host: example.com
< Content-Type: text/task
< Content-Length: 31
<
< (Pending) [LOW] Go on vacation.
```

In a paginated response, the HTTP Mailbox concatenates all the individual message responses in the given range as a pipeline. This pipeline is then wrapped in another HTTP response as an entity. If the given range does not exist in the message chain then the HTTP Mailbox will return a `404 Not Found` response. Suppose that Bob's server has 9 messages in its message chain starting from sequence number 0 to 8 as illustrated in Fig. 11 and it wants to access them three consecutive messages at a time (or page size of 3). Suppose that it requests "2-4" as the range to start the retrieval. Code 12 illustrates the request and response of this scenario. In Code 12

Lines 1-2 represent the page request, Lines 4-70 represent the paginated response. Lines 4-11 represent the headers related to the paginated response while Lines 13-32, Lines 34-49, and Lines 51-70 are the three individual response messages that the HTTP Mailbox would have been returned if those were requested individually. Lines 6-9 represent the navigation URIs of pages. Page related headers do not have `Via` and `Memento-Datetime` headers (as these belong to the individual messages) while message related headers do not have a `Date` header because it belongs to the current request.

The pagination option is only available in GET requests (message retrieval). If client wants to send multiple messages to a single recipient, it can send them all as a single pipeline message. But those messages will be counted as one message and their individual retrieval will not be possible.

The parameters for pagination and time based access both have the same place in the `HM-Request-Path` hence they cannot work together in conjunction. If this optional parameter is a 14 digit integer then the HTTP Mailbox recognizes it as the time-based access parameter. If the optional parameter is a combination of two integers separated by a minus sign (-) then the HTTP Mailbox recognizes it as the pagination parameter. In all other cases it will be considered to be part of the recipients' identifier.

6.5 HM-HEADERS

In addition to standard HTTP headers, some headers are defined by the HTTP Mailbox or being utilized from various HTTP extensions. In a Send Request, the `HM-Sender` request header is sent to indicate the original sender because the client sending the message may be sending it on behalf of someone else (see appendix A for the format of the `HM-Sender` header). On the other hand, in a Retrieve Response (in case of success) a `Via` [2] header is returned containing the identifier of the sender and the hostname or IP address of the client. A typical `Via` header is illustrated below.

```
Via: Sent by 127.0.0.1 on behalf of http://example.org/alice delivered by http://example.net/
```

A Retrieve Response (in case of success) must also return a `Link` [51] header containing `self`, `current`, `first`, `last`, `next`, and `previous` message URIs as applicable. The `Link` header contains a comma separated list of URIs and their respective relations with the response message. URIs are enclosed in angular brackets and relations are enclosed by quotes while the two are separated by a semi-colon as illustrated below. The relation indicated by `rel` can have multiple values separated by spaces.

```
Link: <http://example.net/hm/id/aebed6e9>; rel="first", <http://example.net/hm/id/5ecb44e0>; rel="last self"
```

The HTTP Mailbox will also return a `Memento-Datetime` header (described in appendix A as Memento-header) to report the time when the enclosed message was first seen by the HTTP Mailbox. For the sake of simplicity, this header was excluded from various examples.

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

Headers that are introduced by the HTTP Mailbox and have no prior reference will have a prefix of "HM-" for example `HM-Sender`. There is another class of headers that have been introduced by the HTTP Mailbox but are not consumed by the HTTP Mailbox directly. These headers are generated by the message sender and forwarded to the recipient(s) of the message. We call them "HM-Forward" headers. The HTTP Mailbox will store those headers separate from the entity as metadata. These headers are particularly useful if the header section inside the encapsulated HTTP message itself is not the best place to add them. For example if the HTTP message is encrypted and the sender wants to inform the recipient how to decrypt the

TABLE IV
Categories of HTTP Mailbox Headers

Category	Example	Description
HTTP Header	Host, Date, Link, Via, Content-Length	It includes HTTP headers defined in RFC 2616 or its extensions.
HM-*	HM-Sender	These request or response headers are introduced in the HTTP Mailbox and the HTTP Mailbox understands them.
HM-Forward-*	HM-Forward-Encoding, HM-Forward-Content-MD5	These headers will not be used by the HTTP Mailbox directly but they will be forwarded to the recipient(s) as received from the sender. Sender and Recipient(s) may agree on any general header of this type to pass extra data that cannot fit in the message body.

TABLE V
Headers Used in HTTP Mailbox

Header	Type	Description
Content-Type	Entity	It indicates the media type of the entity.
Content-Length	Entity	It indicates the size (octet count) of the entity.
Link	Entity	It is utilized in responses from the HTTP Mailbox to list various links related to the returned message.
Host	Request	It indicates the hostname and port number of the server. It is not needed if full URI is given in the request line.
Origin	Request	It is being utilized in CORS to indicate the hostname and port number of the request originating server.
Location	Response	When a message is sent to the HTTP Mailbox, it returns the URI of the newly created message to the sender using this header.
Memento-Datetime	Response	It indicates the date and time when the message was first seen by the HTTP Mailbox.
Access-Control-Allow-Origin	Response	It gives the list of allowed origins for CORS.
Access-Control-Expose-Headers	Response	It gives the list of headers that are exposed to the client in CORS.
Access-Control-Allow-Methods	Response	It gives the list of allowed methods for CORS.
Date	General	It indicates the time when message was originated.
Via	General	It is utilized in the HTTP Mailbox response to indicate the sender and the client used to send the message.
HM-Sender	HM	This is a request header that client needs to send in order to indicate the sender of the message.
HM-Forward-Encoding	HM-Forward	This header is used to encode the message and forward the related information to the recipient.
HM-Forward-Content-MD5	HM-Forward	This header contains the MD5 digest of the entity to verify the integrity.

message, such an indication can be put in the "HM-Forward" header. HM-Forward header is a generic header that has HM-Forward-* format. Sender and recipient(s) may agree upon any number of such headers to communicate extra information. Table IV gives an overview of various categories of headers and their examples.

Table V lists the headers that are being utilized in the HTTP Mailbox. More HTTP Mailbox specific headers can be added later to extend the features of HTTP Mailbox.

6.6 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 [2]. The corresponding Content-Type header must be present in the HM-Headers. In other cases, there may be no entity body or any generic entity body with an appropriate Content-Type header and Status-Code. In the case of Retrieve Request, there must not be any entity body because it is an HTTP GET request.

6.7 MESSAGE CHAIN

The HTTP Mailbox query mechanism using HM-Request-Path as illustrated in section 6.2 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 the 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. The absence of the next relation and same values of self and last, both indicate the end of the message chain. Similarly, the absence of the 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

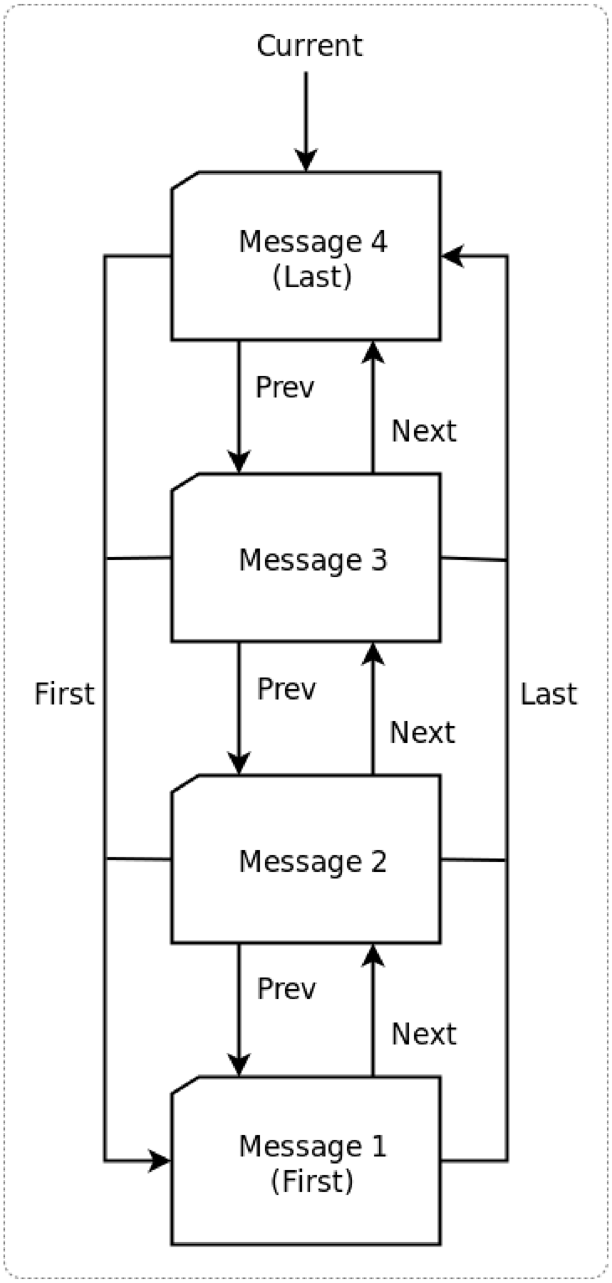
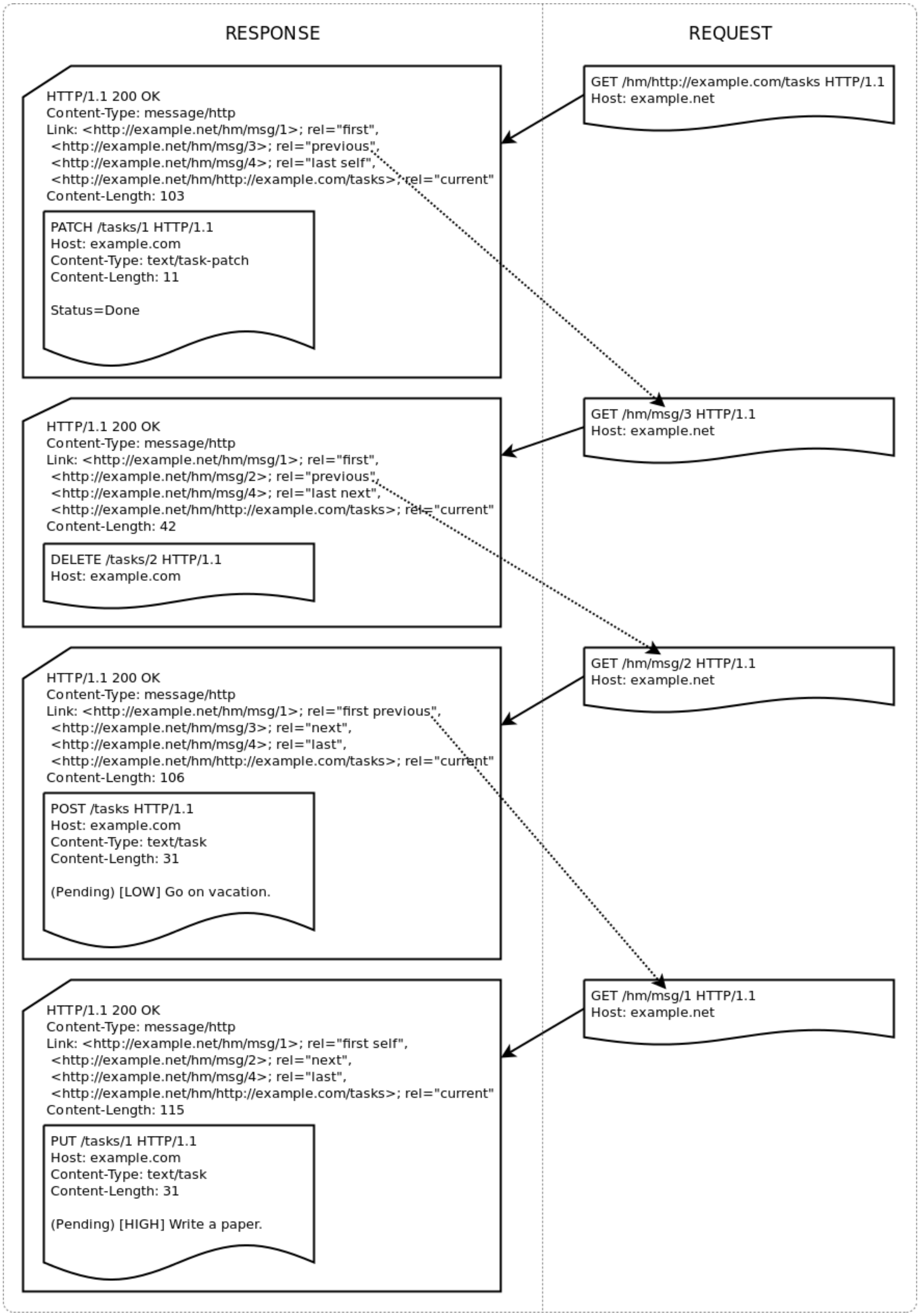


Fig. 12. Message Chain.

of retrieval. Usually the 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. Fig. 12 shows how messages are arranged in chronological order and linked together to form a chain of messages. Fig. 13 shows a typical message chain retrieval scenario where the most recent message is retrieved first then it follows the



```
GET /hm/http://example.com/tasks HTTP/1.1
Host: example.net
```

```
HTTP/1.1 200 OK
Content-Type: message/http
Link: <http://example.net/hm/msg/1>; rel="first",
 <http://example.net/hm/msg/3>; rel="previous",
 <http://example.net/hm/msg/4>; rel="last self",
 <http://example.net/hm/http://example.com/tasks>; rel="current"
Content-Length: 103

PATCH /tasks/1 HTTP/1.1
Host: example.com
Content-Type: text/task-patch
Content-Length: 11

Status=Done
```

```
GET /hm/msg/3 HTTP/1.1
Host: example.net
```

```
HTTP/1.1 200 OK
Content-Type: message/http
Link: <http://example.net/hm/msg/1>; rel="first",
 <http://example.net/hm/msg/2>; rel="previous",
 <http://example.net/hm/msg/4>; rel="last next",
 <http://example.net/hm/http://example.com/tasks>; rel="current"
Content-Length: 42

DELETE /tasks/2 HTTP/1.1
Host: example.com
```

```
GET /hm/msg/2 HTTP/1.1
Host: example.net
```

```
HTTP/1.1 200 OK
Content-Type: message/http
Link: <http://example.net/hm/msg/1>; rel="first previous",
 <http://example.net/hm/msg/3>; rel="next",
 <http://example.net/hm/msg/4>; rel="last",
 <http://example.net/hm/http://example.com/tasks>; rel="current"
Content-Length: 106

POST /tasks HTTP/1.1
Host: example.com
Content-Type: text/task
Content-Length: 31

(Pending) [LOW] Go on vacation.
```

```
GET /hm/msg/1 HTTP/1.1
Host: example.net
```

```
HTTP/1.1 200 OK
Content-Type: message/http
Link: <http://example.net/hm/msg/1>; rel="first self",
 <http://example.net/hm/msg/2>; rel="next",
 <http://example.net/hm/msg/4>; rel="last",
 <http://example.net/hm/http://example.com/tasks>; rel="current"
Content-Length: 115

PUT /tasks/1 HTTP/1.1
Host: example.com
Content-Type: text/task
Content-Length: 31

(Pending) [HIGH] Write a paper.
```

Fig. 13. Message Chain Retrieval.

`previous` link from the header until the first message in the chain is retrieved.

6.7.1 CHAIN IMPLEMENTATION

There can be different ways to implement the message chain depending on the storage service. In our reference implementation we used a linked list like structure to form the message chain and retrieve it efficiently.

If every message stores the references of previous, next, first, and last messages locally then the retrieval efficiency will be good but posting a message will cause all the previous messages in the chain to be modified in order to update the last message reference. This approach has a linear complexity with respect to the message chain size.

An efficient approach could be to have first and last message references stored in the metadata of the message chain rather than in each individual message. At the time of message retrieval, particular message and message chain metadata needs to be fetched in order to prepare the response that contains references to previous, next, first, and last messages. When a new message is sent to be added at the end of the message chain, it only requires modifications in two places. The message that was previously the last message to reference next message and the message chain metadata to reference new message as the last message. This approach has a constant complexity independent of the size of the message chain.

6.8 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.

Code 13. CORS Response Headers

```
> HEAD /hm/http://example.com/tasks HTTP/1.1
> Host: example.net
> Origin: example.org

< HTTP/1.1 200 OK
< Access-Control-Allow-Origin: example.org
< Access-Control-Allow-Methods: GET, POST, OPTIONS
< Access-Control-Expose-Headers: Link, Via, Date, Memento-Datetime
< Date: Thu, 20 Dec 2012 02:10:22 GMT
< Link: <http://example.net/hm/id/aebed6e9>; rel="first",
<  <http://example.net/hm/id/5ecb44e0>; rel="last self",
<  <http://example.net/hm/id/85addc19>; rel="previous",
<  <http://example.net/hm/http://example.com/tasks>;rel="current"
< Via: Sent by 127.0.0.1
<  on behalf of http://example.org/alice
<  delivered by http://example.net/
< Content-Type: message/http; msgtype: request
< Content-Length: 108
```

To support CORS, the HTTP Mailbox needs to return CORS related headers (that are prefixed with “Access-Control-”) like `Access-Control-Allow-Origin`, `Access-Control-Expose-Headers`, and `Access-Control-Allow-Methods` (all CORS related headers are described in CORS specifications [9]). Code 13 Lines 6-8 illustrate the typical CORS related response header in the HTTP Mailbox. The `Access-Control-Allow-Origin` header gives a comma separated list of origins that are allowed to make CORS requests. If an HTTP Mailbox allows every domain to perform CORS requests, it can either echo back the value of `Origin` [52] header or return a wildcard character “*”. The `Access-Control-Allow-Methods` header gives a comma separated list of HTTP methods that are allowed for CORS. The `Access-Control-Expose-Headers` header gives a comma separated list of headers that should be exposed to the client. This header plays an important role in navigating through message chain. If `Link` header is not exposed to the client the JavaScript-based clients will not be able to access the vale of the `Link` header hence they will fail to discover the URI of the other messages in the message chain.

Some browsers have a bug that prevents them from exposing response headers in CORS although the `Access-Control-Expose-Headers` header is setup correctly. See appendix C for a detailed description of the bug and its workaround.

6.9 SUMMARY

In this chapter we described various components of HTTP Mailbox messaging. We described how the various forms of recipients’ identifier enables group communication, how time-based message retrieval and pagination is enabled, and how message chain works. We discussed various headers utilized in the HTTP Mailbox communication. Finally, we discussed the importance of CORS support in the HTTP Mailbox and how it is enabled.

CHAPTER 7

ATTACKS AND PREVENTION

HTTP Mailbox communication provides open access to the mailbox. This means anyone can send and receive messages on behalf of anyone. Although it makes the design of the HTTP Mailbox simple, it raises concerns related to security, privacy, authentication, and spamming. In some applications this may not be a problem; for wider applicability the HTTP Mailbox must allow ways to deal with these things. The design of the HTTP Mailbox is flexible enough to allow clients and servers to devise their own mechanisms to deal with these concerns. This way the responsibility is offloaded to the senders and recipients.

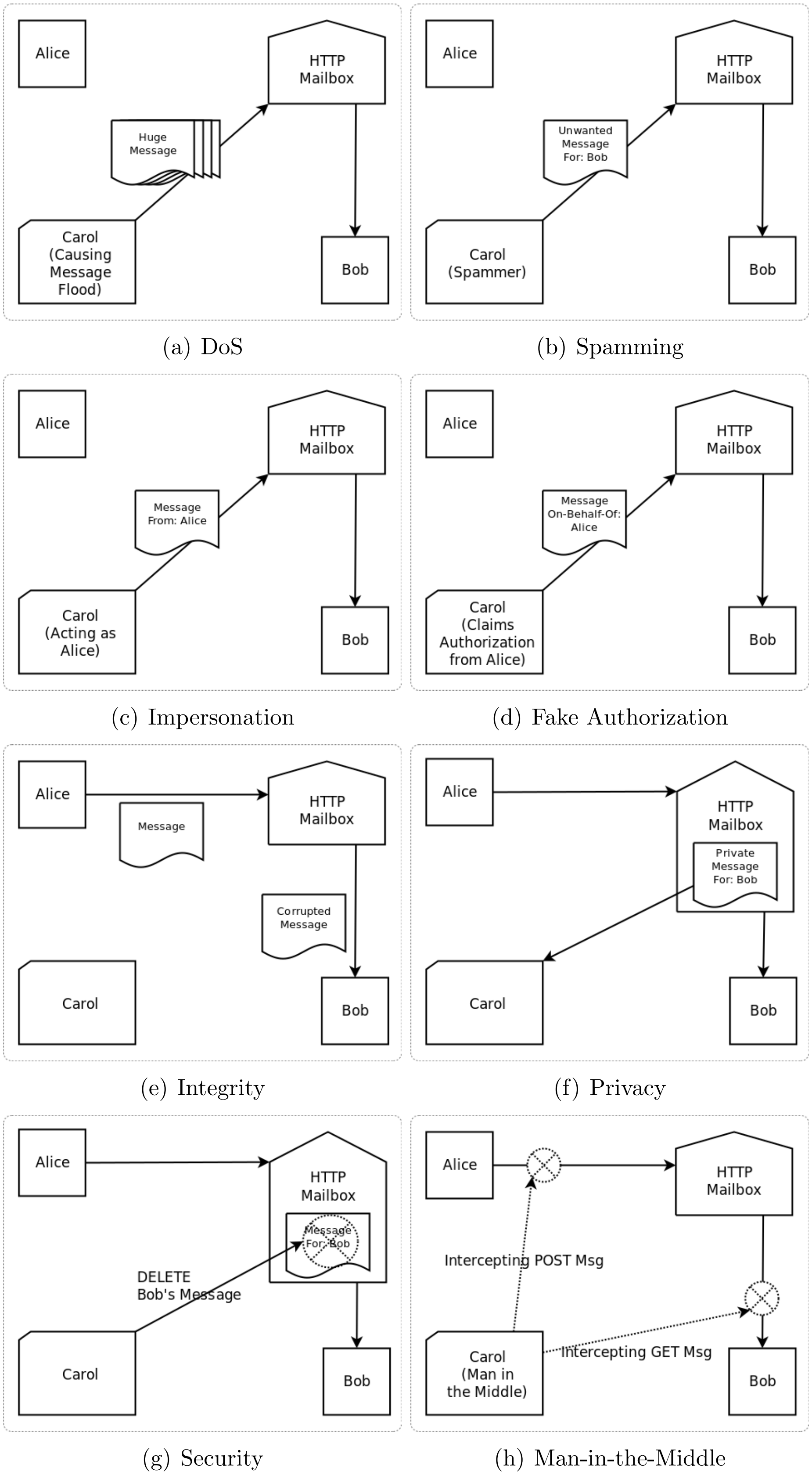
In HTTPS (Hypertext Transfer Protocol Secure) [53] communication, security is delegated to a separate layer in which HTTP traffic is protected by SSL/TLS [54]. The HTTP Mailbox itself can operate on HTTPS but it will not prevent attacks over the complete lifecycle of the HTTP Mailbox communication.

When deciding on a mechanism to ensure security, privacy, or authenticity, one should be aware that HTTP Mailbox is an asynchronous store and forward system. Hence protocols that involve multiple steps back and forth between sender and recipient to set up encryption mechanism are not suitable. Senders and recipients either need to agree upon an encryption key in advance or they need to use a protocol that does not require multiple message exchanges to setup the key. In this chapter we will describe how public key encryption can be used to achieve security, privacy and authentication.

7.1 ATTACKS

Here we will examine various possible attacks in an HTTP Mailbox lifecycle. We will also look briefly into possible attack prevention techniques that are feasible in the HTTP Mailbox assisted communication.

Assume that Alice is the sender of the message, Bob is the recipient, and Carol is an intruder. Also assume that they are using the HTTP Mailbox to exchange messages. Table VI briefly describes some of the possible attacks. These attacks are also illustrated in Fig. 14.



(a) DoS

(b) Spamming

(c) Impersonation

(d) Fake Authorization

(e) Integrity

(f) Privacy

(g) Security

(h) Man-in-the-Middle

Fig. 14. Various Attacks.

TABLE VI
VARIOUS ATTACKS AT A GLANCE

Attack	Victim	Example	Prevention
DoS	HTTP Mailbox	Carol can flood with heavy volume of GET/POST requests and cause DoS attack on HTTP Mailbox service.	Limiting access
Spam	Recipient	Carol can send unwanted/spam messages to Bob.	Spam filtering
Impersonation	Sender	Carol can impersonate as Alice and send messages to Bob as if those were sent by Alice.	Signing
Fake Authorization	Sender	Carol can send messages on Alice's behalf and maliciously claim the authorization to send messages on her behalf.	OAuth [55] or access token
Integrity	Message	Bob may retrieve a message that is not exactly the same as it was sent from Alice.	Hash digest
Privacy	Sender and Recipient	Carol can read messages sent by Alice that are stored in the HTTP Mailbox.	Encryption
Security	Sender and Recipient	Carol can edit, delete, or change the state of the stored messages.	Authentication
Man-in-the-middle	Sender and Recipient	Carol can eavesdrop POST HTTP traffic between Alice and HTTP Mailbox or GET HTTP traffic between Bob and HTTP Mailbox and change the message packets.	HTTPS

7.1.1 DENIAL OF SERVICE ATTACK

An attacker can send a large number of GET/POST requests with huge payloads to the HTTP Mailbox that can cause DoS (Fig. 14(a)). At HTTP Mailbox this type of attack can be prevented by limiting access in various ways. An HTTP Mailbox may have limit on the maximum size of the data that can be sent in a single POST request. It may also limit the maximum number of requests over a period of time from one IP address. Maintaining white/black lists may also help blocking attackers while allowing legitimate access. A white list contains the list of approved users and a black list contains the list of blocked users.

7.1.2 SPAM

Sending unwanted messages to the recipients' mailbox is called spamming (Fig. 14(b)). This is a serious problem that can cause an increase in the time to access all unread messages. A recipient may safely discard any unwanted messages, but it will require more HTTP GET requests to access entire thread of unread messages because HTTP Mailbox only allows sequential (chronological) access from either direction. Like traditional email services, the main victim of this attack is the recipient. To avoid this issue, HTTP Mailbox needs to have some ways to identify spam messages and prevent them from being saved in the Mailbox.

Another approach could be to use third party services to identify potential spammers and limit the POST requests to only authentic clients. For example, if the HTTP Mailbox requires authentication/authorization from some OAuth [55] providers like Twitter, Facebook, Google, or OpenID before it lets someone POST anything then spam identification will be off-loaded to these third party services indirectly. This approach is based on sender's credibility instead of the content of the posted message.

7.1.3 IMPERSONATION

The HTTP Mailbox is an open mailbox where any one can store and retrieve messages. This open nature enables the possibility of sending and retrieving someone else's messages by faking the identity (Fig. 14(c)). To avoid this issue, a private-key signature can be put in place in every POST request. The sender and the recipient(s) need to deal with the signing of the message using private-key of the sender and decrypting it back using public-key of the sender. The public-key should be hosted in a place which is owned by the sender and the location of the public-key should be provided in the HTTP headers.

7.1.4 FAKE AUTHORIZATION

In this attack an attacker does not impersonate but falsely claims the authorization on someone else's behalf (Fig. 14(d)). It may affect both the sender and recipient but main victim of this type of attack are the senders responsible for sending messages. While fake authorization may affect the recipient as well if retrieving a message also changes the state of the message. In most of the cases message retrieval can be kept open to reduce the complexity of the system.

An HTTP Mailbox can use OAuth or other mechanisms of access tokens to put authorization in place. Senders can register their authorized clients with the HTTP Mailbox and set up access tokens that clients can use at the time of sending messages. This way the HTTP Mailbox can ensure that only authorized clients can send messages on a particular sender's behalf.

7.1.5 INTEGRITY

If a retrieved message is not exactly the same as it was sent then it has lost its integrity (Fig. 14(e)). There could be many reasons why this had happened. It may be caused by network issue, corrupted message store, man-in-the-middle or unintended modification of the stored message (possibly by an attacker).

A hash digest of the message can be added in the HTTP headers to check the integrity. This may not be helpful in detecting the integrity if it was caused by an attacker. But those attacks can be avoided by other means as described in the coming sections of security and man-in-the-middle attacks.

7.1.6 PRIVACY

Due to the open nature of HTTP Mailbox anyone can read the stored messages of anyone (Fig. 14(f)). If this is not in the best interests of the sender and recipient, they can encrypt the messages before sending POSTing with appropriate HTTP headers in place to indicate the encryption. Only the legitimate recipient(s) should then be able to decrypt the messages. If it is a unicast message then public private-key encryption can be used and in case of group messaging, shared key encryption can be used.

7.1.7 SECURITY

Gaining access to someone else's content and making unauthorized changes like editing or deleting a message or changing the state of the message is considered as security attack (Fig. 14(g)). To avoid this threat the HTTP Mailbox may require locally hosted or third party authentication (for example OAuth) for all the requests that may cause changes in a message or its state.

7.1.8 MAN-IN-THE-MIDDLE

An attacker may intercept GET or POST HTTP requests or responses between clients (sender/recipient) and the HTTP Mailbox (Fig. 14(h)). In this case the attacker can modify the message (request/response) before it reaches the destination. Man-in-the-middle attack can succeed only when the attacker can impersonate both client and server.

The HTTP Mailbox can use Secure Socket Layer (SSL) [56] to prevent from the man-in-the-middle attack.

7.2 PUBLIC-KEY CRYPTOGRAPHY

Public-key cryptography [57] is an asymmetric key cryptography in which there are two keys. If a message is encrypted using one of the two keys, it can only be decrypted using the other key as illustrated in Fig. 15. Ciphering using public-key is called encryption and ciphering using private-key is called signing. RSA [58] is a good example of asymmetric key based cryptography. This technique will be utilized in the following sections 7.3, 7.4, and 7.5.

In practice, a key pair is generated, one of the keys is kept secret as private-key while the other one is advertised as public-key and is kept in a place where anyone can get it. There are couple of ways to ensure that the public-key actually belongs to whom it is claimed. One of those ways is to keep the public-key in a place which is owned by the entity claiming as the owner of the public-key. Another approach is to host the public-key in public-key registries and get it signed by several trusted parties.

7.3 ENCRYPTING THE HTTP MESSAGE

Alice wants to send an HTTP Request (as illustrated in Code 8) to Bob's task service using HTTP Mailbox but she does not want to expose the contents of her request to anyone except Bob's server. She can encrypt her HTTP Request using the public-key of Bob's task server then send it to the HTTP Mailbox. This way anyone can retrieve that message but cannot read the content meaningfully unless has access to the private-key of Bob's task server to decrypt it. An indication of this encryption needs to be recorded in the HM-Headers in order to recognize that the message was encrypted using a public-key. The URI of the public-key used in encryption needs to be there in the header in order to recognize the private-key counter part for decryption.

The HTTPsec [59] protocol has a similar approach to add authentication headers in the HTTP request but it cannot be used here because it involves initial set up in order to client and server to agree on a shared token.

Code 14. Encrypting HTTP Message

```
1 > POST /hm/http://example.com/tasks HTTP/1.1
2 > Host: example.net
3 > HM-Sender: http://example.org/alice
4 > Content-Type: message/http
5 > HM-Forward-Encoding: rsa-encrypt certificate=http://example.com/bob.pub
6 > Content-Length: 39
7 >
8 > NOTMEANINGFULLYREADABLEENCRYPTEDMESSAGE
```

Code 14 illustrates how an encrypted HTTP message will be sent to the HTTP Mailbox. Code 14 Line 5 is an HM-Forward header that will be forwarded to the message recipient(s). It contains the signal that the encapsulated HTTP message is encrypted using the RSA encryption algorithm with the public-key available at the URL identified by the `certificate` attribute. The corresponding private-key should be used by the recipient to decrypt the message in order to make it meaningful.

7.4 SIGNING HTTP MESSAGE

Suppose that Bob's task server has received the encrypted HTTP request from Alice, decrypted it using its private-key and performed the request. Now Bob's server

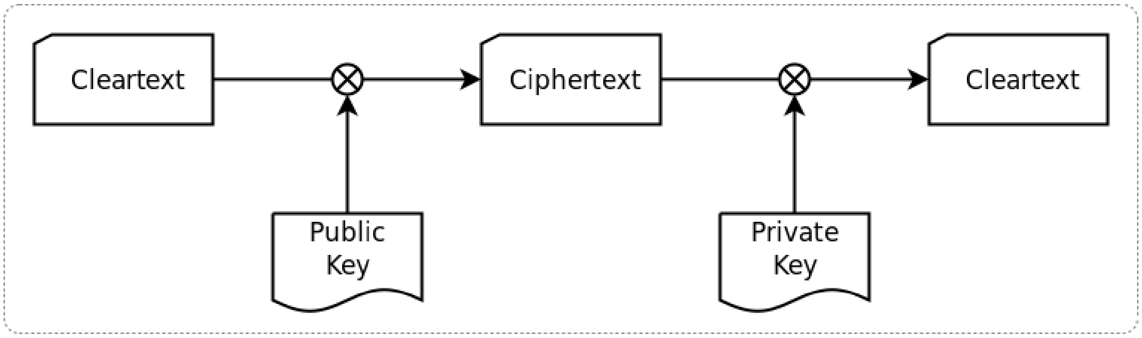


Fig. 15. Public-Key Encryption

is willing to send the response back to Alice using the HTTP Mailbox as illustrated in Code 10. But Bob's server wants to sign the response so that Alice is sure that the response indeed came from Bob's server. In order to do so, Bob's server will use its private-key to sign the HTTP response message and advertise corresponding public-key.

Code 15. Signing HTTP Message

```
1 > POST /hm/http://example.org/alice HTTP/1.1
2 > Host: example.net
3 > HM-Sender: http://example.com/tasks
4 > Content-Type: message/http
5 > HM-Forward-Encoding: rsa-sign certificate=http://example.com/bob.pub
6 > Content-Length: 19
7 >
8 > SIGNEDMESSAGESTREAM
```

Code 15 illustrates how a signed HTTP message will be sent to the HTTP Mailbox. Code 14 Line 5 indicates that the entity is signed using the RSA and can be decrypted using the public-key identified by the following `certificate` attribute.

7.5 SIGNING AND ENCRYPTING HTTP MESSAGE

Now suppose Alice wants to send an encrypted messages (as discussed in section 7.3) but also wants to sign the HTTP message. This way she will make sure that only Bob's server will be able to meaningfully read the message and Bob's server will be sure that the request is indeed made by Alice. In order to do so, she will first sign the HTTP message using her private-key then she will encrypt the signed message using Bob's public-key. Both the public-keys needs to be advertised in the HM-Headers to indicate the signing and encryption.

Code 16. Signing and Encrypting HTTP Message

```
1 > POST /hm/http://example.com/tasks HTTP/1.1
2 > Host: example.net
3 > HM-Sender: http://example.org/alice
4 > Content-Type: message/http
5 > HM-Forward-Encoding: rsa-sign certificate=http://example.org/alice.pub
6 >  rsa-encrypt certificate=http://example.com/bob.pub
7 > Content-Length: 48
8 >
9 > NOTMEANINGFULLYREADABLESIGNEDANDENCRYPTEDMESSAGE
```

Code 16 illustrates signing and encryption both together. Code 16 Lines 5-6 indicates the operations performed on the message. Order of the signing and encryption is important here because exact reverse operation required in order to meaningfully retrieve the message. Bob's server will first decrypt the message using its private-key then decrypt it again using Alice's public-key to gain access to the original HTTP message.

7.6 HTTP MESSAGE DIGEST

To detect the damage in the integrity of the message, a digest of the message can be added in the HM-Headers. This digest will correspond to the content of the entity sent to the HTTP Mailbox. If the entity is encrypted or signed then the digest will be generated from the encrypted/signed message not from the original message.

Code 17. HTTP Message Digest

```
 1 > POST /hm/http://example.com/tasks HTTP/1.1
 2 > Host: example.net
 3 > HM-Sender: http://example.org/alice
 4 > Content-Type: message/http
 5 > HM-Forward-Encoding: rsa-sign certificate=http://example.org/alice.pub
 6 >  rsa-encrypt certificate=http://example.com/bob.pub
 7 > HM-Forward-Content-MD5: 4ada22b76d5dece41619f49eb97ec216
 8 > Content-Length: 48
 9 >
10 > NOTMEANINGFULLYREADABLESIGNEDANDENCRYPTEDMESSAGE
```

Code 17 Line 7 holds the MD5 digest of the message in Code 17 Line 10. The recipient(s) can verify the integrity of the message by generating the MD5 sum of the retrieved message and comparing it with the one that was sent in the `HM-Forward-Content-MD5` header.

7.7 SUMMARY

In this chapter we discussed various attacks that may occur in HTTP Mailbox messaging. We briefly talked about the victims of those attacks and possible prevention mechanisms. We described the use of asymmetric keys for signing and encrypting messages, then we discussed the use of a hash digest to ensure the integrity of the message. We illustrated the use of signing, encryption, and hash digests with code examples.

CHAPTER 8

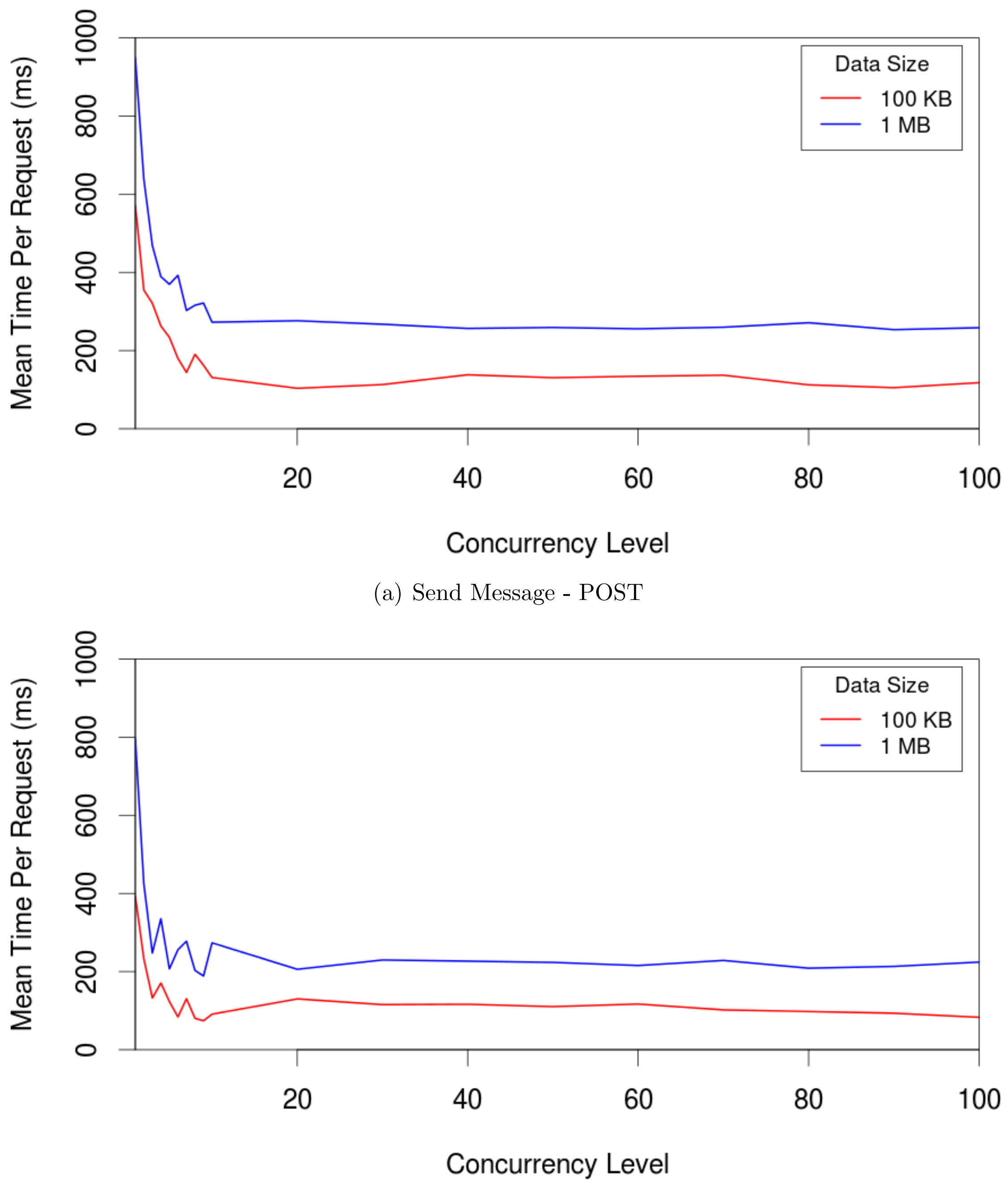
REFERENCE IMPLEMENTATION BENCHMARKING

A reference implementation of an HTTP Mailbox server was written in Ruby [60] using the Sinatra [61] Web framework running on a Thin [62] Web server. Fluidinfo [63] was used to store messages and other metadata associated with them and it was accessed using fluidinfo.rb [64] Ruby library. A copy of this code can be found on GitHub [65].

8.1 STRESS TEST ANALYSIS

ApacheBench [12] 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 ($\approx$ 100 MB). Fig. 16 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 concurrency level is the number of concurrent requests made by ApacheBench to the server at a given time. 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 connections between HTTP Mailbox and Fluidinfo server).

Graphs in Fig. 16(a) and 16(b) show that the MTPR in both the cases decreases as concurrency level increases. For smaller payloads (below 100 KB) the variation is not distinguishable, but when the payloads increase from 100 KB to 1 MB, MTPR roughly doubles. After analyzing data, we picked the 100 KB file and performed further stress testing up to 1,000 concurrency level and observed a gradual decrease in MTPR. With 1,000 concurrent requests, MTPR 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 observed 0.0144% (12/83,300)



(a) Send Message - POST

(b) Retrieve Message - GET

Fig. 16. Stress Test Analysis of HTTP Mailbox Using ApacheBench.

non-2xx responses (transient failures) in our benchmarking. We have also measured the time taken by GET requests returning valid `404 Not Found` responses. These responses took 53 ms MTPR (values ranging from 200 to 20 ms depending upon the concurrency level).

8.2 PAGINATION ANALYSIS

The basic method of retrieving messages from the HTTP Mailbox is to start form the most recent message in the message chain, discover the URIs of the first and last messages in the chain, and traverse the message chain from either direction, sequentially, one message at a time. Arbitrary message URIs are not discoverable except from their adjacent messages in the message chain. This method may not be suitable if there are too many messages to retrieve from a message chain as it will take too long to access the chain sequentially, one message at a time.

The HTTP Mailbox introduces pagination in message retrieval as discussed in section 6.4 which allows retrieval of an arbitrary batch of consecutive messages called a page from the message chain. In this retrieval method, the client requests for a range of messages from the message chain using the message sequence numbers. This approach minimizes the network usage in terms of number of requests and responses, while the amount of data transferred is roughly the same over the same number of messages retrieved.

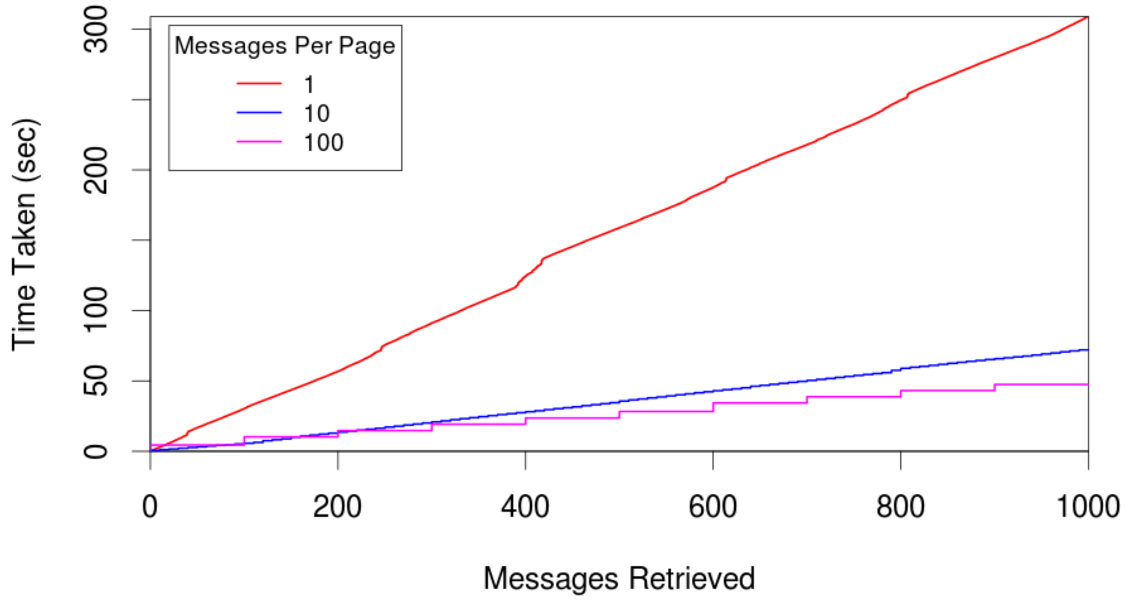


Fig. 17. Response Time for Various Page Sizes.

Fig. 17 is a plot of time taken to retrieve 1,000 messages, each of size 10 KB using basic method (one message at a time), pages of 10 messages each, and pages of 100 messages each. The basic method requires 1,000 requests to retrieve all the message while other two pagination methods require 100 requests and 10 requests respectively. In our reference implementation, it took an average of 309 seconds to retrieve 1,000 messages using basic retrieval method, while pages of 10 messages took 72 seconds and pages of 100 messages took 47 seconds to complete the retrieval of the same 1,000 messages.

Apart from the number of retrieval requests, there are following three other factors that affect the total time of retrieval: 1) The mechanisms of query in the Fluidinfo store we used in our reference implementation are different for basic retrieval and paginated retrieval, 2) The HTTP Mailbox processes the messages differently to prepare the responses for the two retrieval methods, and 3) The size of the response payload per request is different in the three test cases.

Basic message retrieval method does not allow parallel retrieval because the URIs of the messages in the chain are not known in advance. While in case of paginated retrieval, the client can request multiple pages of the chain in parallel to avoid data size limit from the HTTP Mailbox.

8.3 SEGMENT ANALYSIS

Our reference implementation takes a round trip time of about 300 milliseconds on average to complete one message retrieval using basic retrieval method. As illustrated in Fig. 18, in our test on average 1.33% of the total time is consumed in the network between the client and the HTTP Mailbox server, 1.89% is consumed in message processing in the HTTP Mailbox, and the remaining 96.78% of the total time in

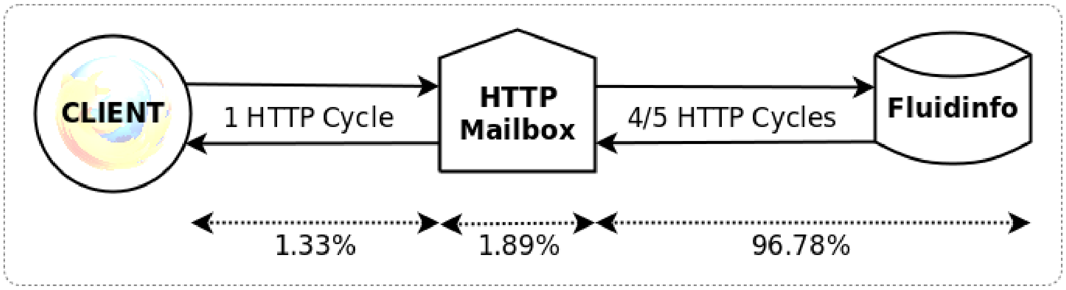


Fig. 18. Time Distribution in the HTTP Mailbox Messaging Retrieval.

communication between the HTTP Mailbox and the Fluidinfo servers. Our naive reference implementation makes five GET requests to the Fluidinfo server in order to serve one basic message retrieval response. Some of these requests were required in order to easily reset our testbed during the evaluation. These five GET requests to Fluidinfo server can be minimized to just two GET requests that will reduce the overall time significantly as these constitute the majority of the response time.

Fluidinfo is a third party service that requires the HTTP Mailbox test server to make multiple API requests over HTTP. If a locally hosted data storage service is used, it will reduce the total time significantly in both of message sending and retrieval requests.

8.4 SUMMARY

In this chapter we described our reference implementation. We discussed various benchmarking results including stress test of the HTTP Mailbox server for GET and POST requests, the change in the accumulated retrieval time with pagination, and the time taken by various segments of our implementation.

Our implementation did not show any decrease in performance under high concurrent request rate (tested upto a concurrency level of 1,000 requests at the same time). It shows a sudden decrease in MTPR initially and a gradual decrease after concurrency level reaches ten requests at a time. In paginated requests, retrieval time decreases significantly for large page sizes (large number of messages per page) unless the data size of the page exceeds the allowed limit of the HTTP Mailbox implementation. In our implementation maximum time was spent in communicating with the third party data storage service called Fluidinfo.

CHAPTER 9

APPLICATIONS

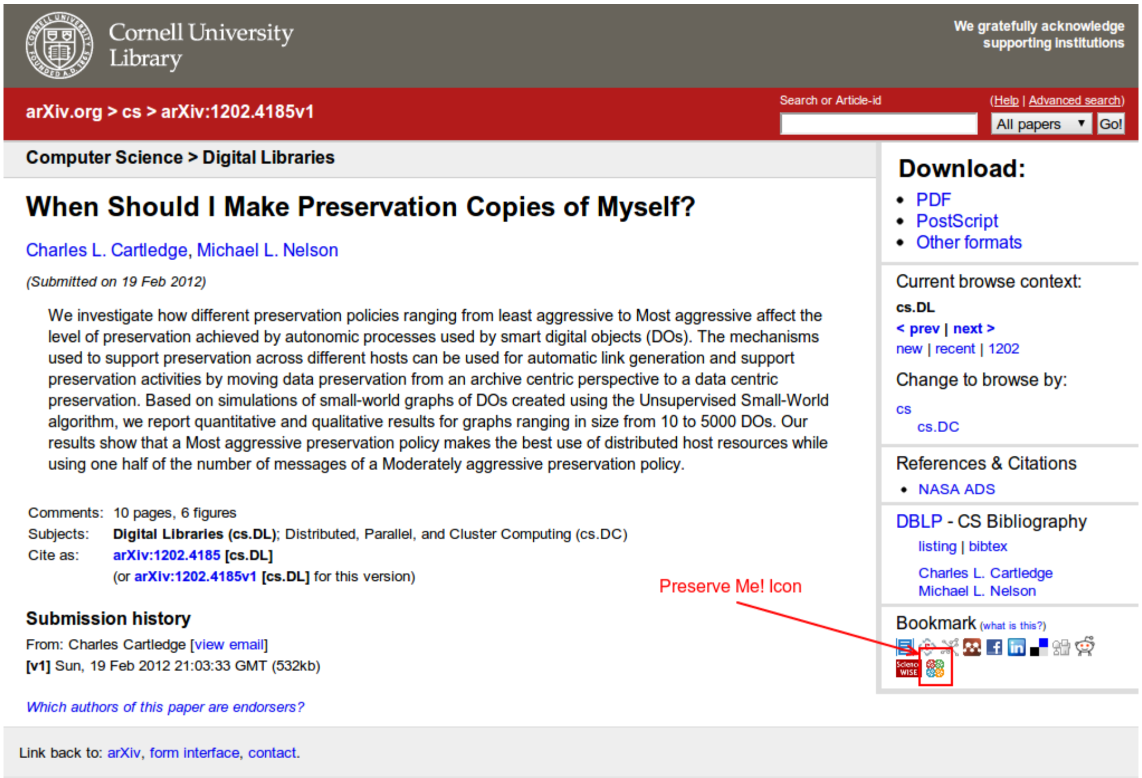
Our implementation of the HTTP Mailbox is being utilized in some applications for their asynchronous web communication needs. We will discuss two such applications, their goals, their working, and how they are utilizing the HTTP Mailbox for messaging.

9.1 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) as illustrated in Fig. 19.

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 [42] files then it pops up a window as shown in Fig. 20. This window gives insight into the ResourceMap and aggregated resources [42] and also allows users to exchange messages with other ResourceMaps and connect them via family and friend relationships. These message exchanges and relationships allow human-assisted preservation of aggregated resources and aids in their long term preservation.

The Preserve Me! application sends various messages including: friendship request, copy request, and copy service announcements. Codes 18 and 19 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 [66, 67] to select various other aggregations as friends. To complete the friendship it requires the other aggregation to add a `link` element in its ResourceMap, pointing back to the requester. To make that change, it prepares an XML-Patch [68] file as illustrated in Code 18 Lines 12-19 and sends an HTTP PATCH request as illustrated in Code 18 Lines 7-19 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



Cornell University Library

We gratefully acknowledge supporting institutions

arXiv.org > cs > arXiv:1202.4185v1

Search or Article-id (Help | Advanced search)

All papers ▼ Go!

Computer Science > Digital Libraries

When Should I Make Preservation Copies of Myself?

Charles L. Cartledge, Michael L. Nelson

(Submitted on 19 Feb 2012)

We investigate how different preservation policies ranging from least aggressive to Most aggressive affect the level of preservation achieved by autonomic processes used by smart digital objects (DOs). The mechanisms used to support preservation across different hosts can be used for automatic link generation and support preservation activities by moving data preservation from an archive centric perspective to a data centric preservation. Based on simulations of small-world graphs of DOs created using the Unsupervised Small-World algorithm, we report quantitative and qualitative results for graphs ranging in size from 10 to 5000 DOs. Our results show that a Most aggressive preservation policy makes the best use of distributed host resources while using one half of the number of messages of a Moderately aggressive preservation policy.

Comments:	10 pages, 6 figures
Subjects:	**Digital Libraries (cs.DL)**; Distributed, Parallel, and Cluster Computing (cs.DC)
Cite as:	arXiv:1202.4185 [cs.DL]
	(or arXiv:1202.4185v1 [cs.DL] for this version)

Submission history

From: Charles Cartledge [view email]
[v1] Sun, 19 Feb 2012 21:03:33 GMT (532kb)

Which authors of this paper are endorsers?

Link back to: arXiv, form interface, contact.

Download:
- PDF
- PostScript
- Other formats

Current browse context:
cs.DL
< prev | next >
new | recent | 1202

Change to browse by:
cs
cs.DC

References & Citations
- NASA ADS

DBLP - CS Bibliography
listing | bibtex
Charles L. Cartledge
Michael L. Nelson

Bookmark (what is this?)

Fig. 19. "Preserve Me!" Icon in a Splash Page.

Message in an HTTP POST Message and sends it to the HTTP Mailbox as illustrated in Code 18 Lines 1-19.

Code 18. Sending Add Friend Request

```
 1 > POST /hm/http://flickr.cs.odu.edu/rems/flickr-adittel-8162004738.xml HTTP/1.1
 2 > Host: hm.cs.odu.edu
 3 > Content-Type: message/http
 4 > HM-Sender: http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml
 5 > Content-Length: 412
 6 >
 7 > PATCH /rems/flickr-adittel-8162004738.xml HTTP/1.1
 8 > Host: flickr.cs.odu.edu
 9 > Content-Type: application/patch-ops-error+xml
10 > Content-Length: 270
11 >
12 > <?xml version="1.0" encoding="UTF-8"?>
13 > <diff>
14 >   <add sel="entry">
15 >     <link rel="http://wsdl.cs.odu.edu/uswdo/terms/friend"
16 >       href="http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml"
17 >       title="Automatic Text Area Segmentation in Natural Images"/>
18 >   </add>
19 > <diff>
20
21 < HTTP/1.1 201 Created
22 < Server: HTTP Mailbox
23 < Location: http://hm.cs.odu.edu/hm/id/5ecb44e0-859c-403f-9184-65e3a086ea2b
```

When the "Preserve Me!" window of the receiving aggregation is opened, it checks its Mailbox as illustrated in Code 19. It then shows the friendship request (and any other messages, if available) as shown in Fig. 20. If that message is applied then the friendship link will be added in the ResourceMap of the receiving aggregation.

Code 19. Retrieving Add Friend Request

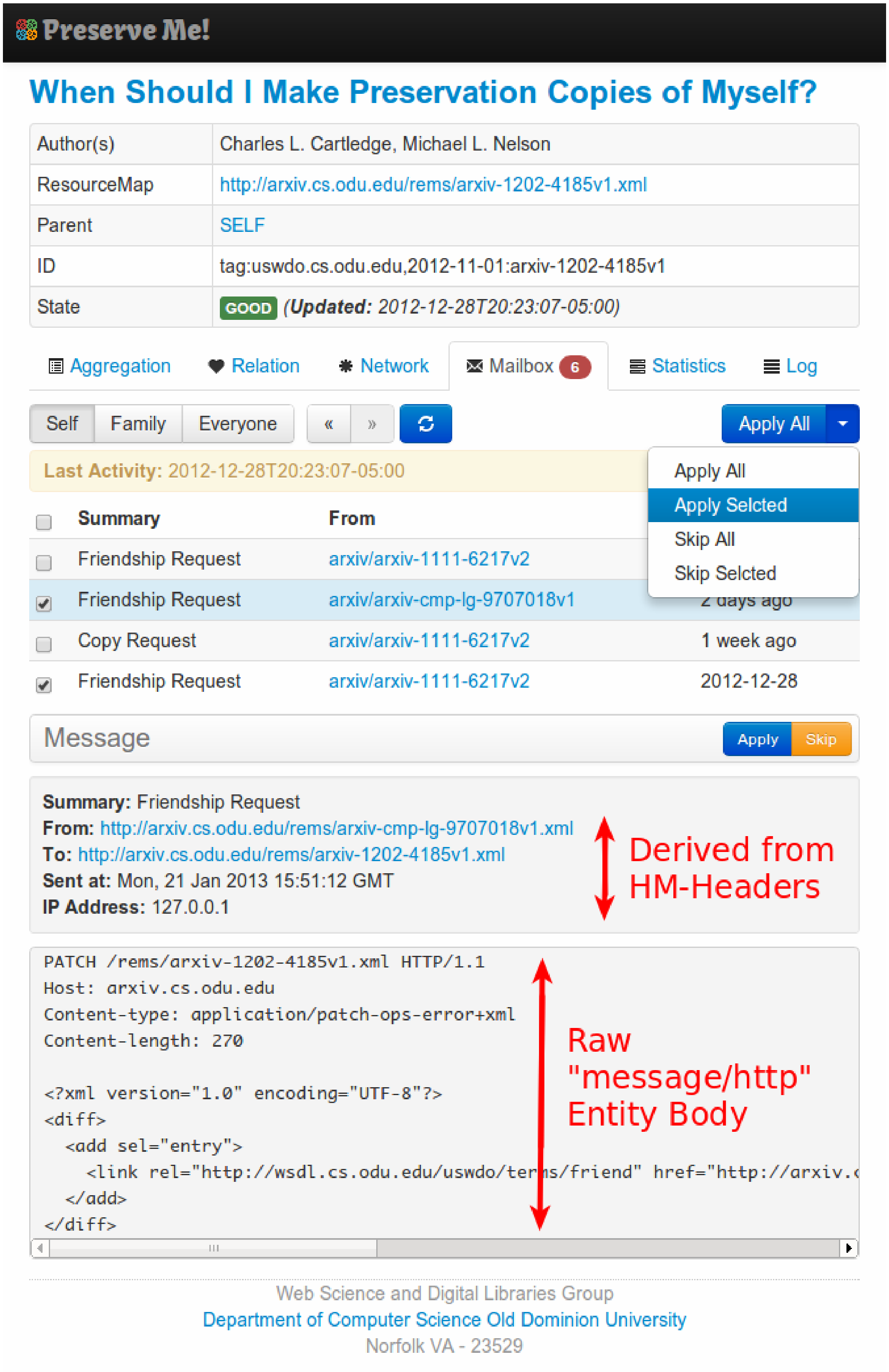
```
> GET /hm/http://flickr.cs.odu.edu/rems/flickr-adittel-8162004738.xml HTTP/1.1
> Host: hm.cs.odu.edu
> Content-Type: message/http

< HTTP/1.1 200 OK
< Server: HTTP Mailbox
< Content-Type: message/http
< Date: Thu, 13 Dec 2012 15:34:24 GMT
< Memento-Datetime: Thu, 13 Dec 2012 05:15:55 GMT
< Via: sent by 68.225.179.9
<  on behalf of http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml,
<  delivered by http://hm.cs.odu.edu/hm/
< Link: <http://hm.cs.odu.edu/hm/http://flickr.cs.odu.edu/rems/
    flickr-adittel-8162004738.xml>; rel="current",
<  <http://hm.cs.odu.edu/hm/id/aebed6e9-e8ac-4051-9970-cc87fde2a549>;
    rel="first",
<  <http://hm.cs.odu.edu/hm/id/5ecb44e0-859c-403f-9184-65e3a086ea2b>;
    rel="last self",
<  <http://hm.cs.odu.edu/hm/id/85addc19-9358-46c7-a836-74b5161b2986>;
    rel="previous"
< Content-Length: 412
<
< PATCH /rems/flickr-adittel-8162004738.xml HTTP/1.1
< Host: flickr.cs.odu.edu
< Content-Type: application/patch-ops-error+xml
< Content-Length: 270
<
< <?xml version="1.0" encoding="UTF-8"?>
< <diff>
<   <add sel="entry">
<     <link rel="http://wsdl.cs.odu.edu/uswdo/terms/friend"
<       href="http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml"
<       title="Automatic Text Area Segmentation in Natural Images"/>
<   </add>
< <diff>
```

9.2 PRESERVE ME! VIZ APPLICATION

To visualize the network of web objects and working of the “Preserve Me!” application, we have created another tool called “Preserve Me! Viz” (Fig. 21). This interactive tool allows the user to visualize relationships and communication among various digital objects in real time or replay recorded sessions.

Digital objects are represented as nodes, their relationships are represented as color-coded directed edges, and communication or data flow is animated from the origin node the target node then disappears. Every node contains a thumbnail of the digital object (splash page) that has a color-coded stroke indicating the host it belongs to. Hovering over a node highlights the related nodes and connections while clicking on a node reveals further details of the digital object (Fig. 21(b)).

The “Preserve Me! Viz” tool starts with a screen having video player-like canvas and controls on it. It allows the user to select the live stream or a pre-recorded session stream. Then it gives options to filter various types of nodes, edges, or messages from animation (Fig. 21(a)). Once the play button is clicked, it starts rendering every event sequentially. A progress bar is shown in recorded sessions that allows



When Should I Make Preservation Copies of Myself?

Author(s)	Charles L. Cartledge, Michael L. Nelson
ResourceMap	http://arxiv.cs.odu.edu/rems/arxiv-1202-4185v1.xml
Parent	SELF
ID	tag:uswdo.cs.odu.edu,2012-11-01:arxiv-1202-4185v1
State	GOOD *(**Updated:** 2012-12-28T20:23:07-05:00)*

Last Activity: 2012-12-28T20:23:07-05:00

□	Summary	From	
□	Friendship Request	arxiv/arxiv-1111-6217v2	
☑	Friendship Request	arxiv/arxiv-cmp-lg-9707018v1	2 days ago
□	Copy Request	arxiv/arxiv-1111-6217v2	1 week ago
☑	Friendship Request	arxiv/arxiv-1111-6217v2	2012-12-28

Message

Summary: Friendship Request
From: http://arxiv.cs.odu.edu/rems/arxiv-cmp-lg-9707018v1.xml
To: http://arxiv.cs.odu.edu/rems/arxiv-1202-4185v1.xml
Sent at: Mon, 21 Jan 2013 15:51:12 GMT
IP Address: 127.0.0.1

```
PATCH /rems/arxiv-1202-4185v1.xml HTTP/1.1
Host: arxiv.cs.odu.edu
Content-type: application/patch-ops-error+xml
Content-length: 270

<?xml version="1.0" encoding="UTF-8"?>
<diff>
  <add sel="entry">
    <link rel="http://wsdl.cs.odu.edu/uswdo/terms/friend" href="http://arxiv.c
  </add>
</diff>
```

Web Science and Digital Libraries Group
Department of Computer Science Old Dominion University
Norfolk VA - 23529

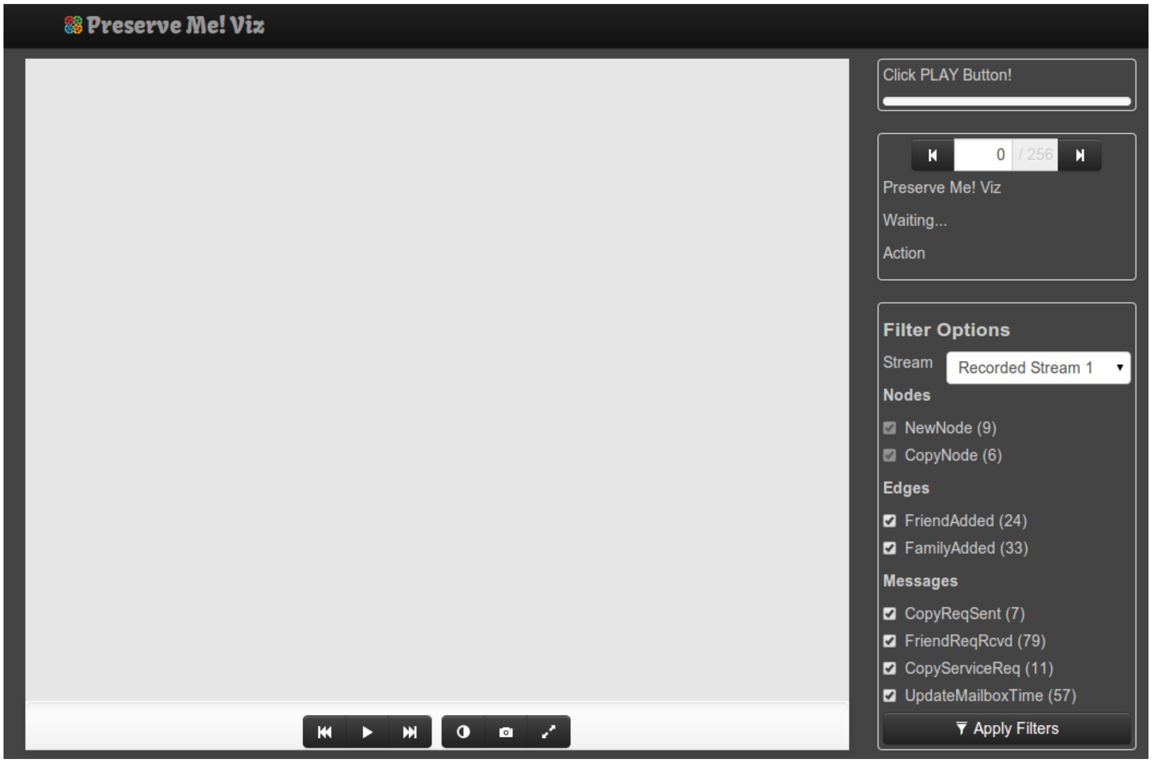
Fig. 20. “Preserve Me!” Application Window.

seeking at any frame and play the animation from there. The canvas is interactive which allows zooming, panning, and relocating nodes. This tool also allows the user to change the background of the canvas, making the canvas full-screen, and capture the current state of the graph and save it as an Scalable Vector Graphics (SVG) [69] file.

Code 20. JSON Data for Visualization

```
[{
  "timestamp": 1360795466.123,
  "type": "Node",
  "id": "http://radiolab.cs.odu.edu/rems/radiolab-adding-memory.xml",
  "title": "Adding Memory",
  "link": "http://radiolab.cs.odu.edu/radiolab-adding-memory.html",
  "group": "radiolab",
  "role": "parent",
  "family": "tag:uswdo.cs.odu.edu,2012-11-01:radiolab-adding-memory",
  "beta": 0.3,
  "gamma": 0.2,
  "aggregationSize": 3,
  "cc": 0.35,
  "apl": 1.27,
  "category": "NewNode"
}, {
  "timestamp": 1360795469.132,
  "type": "Edge",
  "id": "info:radiolab.cs.odu.edu/radiolab-adding-memory/friend/flickr.cs.odu.edu/flickr-clas-8161932383",
  "from": "http://radiolab.cs.odu.edu/rems/radiolab-adding-memory.xml",
  "to": "http://flickr.cs.odu.edu/rems/flickr-clas-8161932383.xml",
  "cc": 0.77,
  "apl": 1.49,
  "category": "FriendAdded"
}, {
  "timestamp": 1360795475.234,
  "type": "Message",
  "id": "info:AddFamilySent/at/1360795475.234",
  "from": "http://radiolab.cs.odu.edu/rems/radiolab-adding-memory.xml",
  "to": "tag:uswdo.cs.odu.edu,2012-11-01:radiolab-adding-memory",
  "via": "http://ws-dl-02.cs.odu.edu:10101/hm",
  "category": "AddFamilySent"
}]
```

To record the sessions, we use a JavaScript Object Notation (JSON) [70] encoded file as illustrated in Code 20 and record every node, edge, and message sequentially as they occur. To visualize the “Preserve Me!” live, we utilize the HTTP Mailbox. We send each event as it occurs to the HTTP Mailbox with the URI of the “Preserve Me! Viz” application as the recipient. When the live stream is selected (as opposed to the recorded event-log files) in the “Preserve Me! Viz”, it reads the events from the HTTP Mailbox and animates them on the canvas. Capturing only the recent messages from the HTTP mailbox is not sufficient to bring the canvas to a stage where all the nodes and edges are rendered to represent the current state of the testbed of the “Preserve Me!” application. “Preserve Me! Viz” needs to retrieve all the messages from the start in the same order to render them properly. This means if the “Preserve Me!” application is running from sometime before “Preserve Me! Viz” was started, then it will take some time before it reaches to the current state of the testbed. To overcome this delay, we send batched messages periodically to another recipient using the HTTP Mailbox. We realized that only nodes and



Preserve Me! Viz

Click PLAY Button!

0 / 256

Preserve Me! Viz

Waiting...

Action

Filter Options

Stream: Recorded Stream 1

Nodes

- ☑ NewNode (9)
- ☑ CopyNode (6)

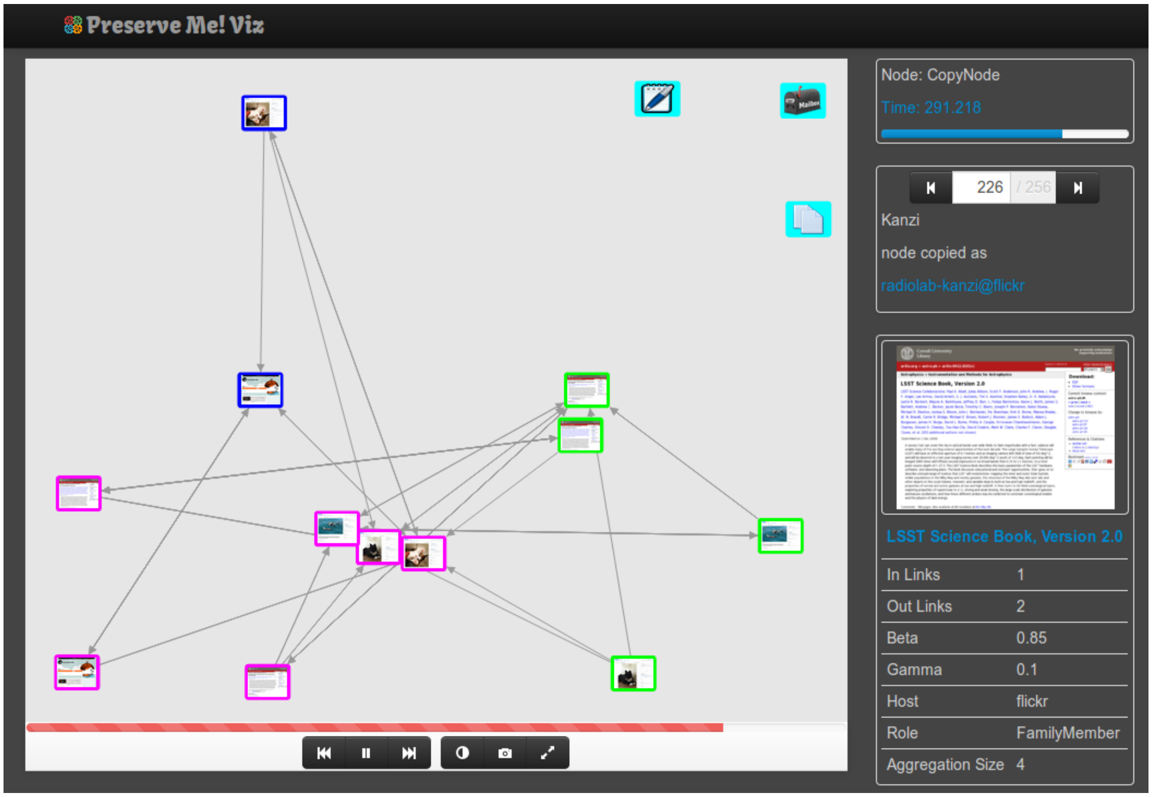
Edges

- ☑ FriendAdded (24)
- ☑ FamilyAdded (33)

Messages

- ☑ CopyReqSent (7)
- ☑ FriendReqRcvd (79)
- ☑ CopyServiceReq (11)
- ☑ UpdateMailboxTime (57)

(a) Start by Selecting a Stream and Applying Filters



Preserve Me! Viz

Node: CopyNode

Time: 291.218

226 / 256

Kanzi

node copied as

radiolab-kanzi@flickr

LSST Science Book, Version 2.0

In Links	1
Out Links	2
Beta	0.85
Gamma	0.1
Host	flickr
Role	FamilyMember
Aggregation Size	4

(b) Reveal Details of a Node

Fig. 21. "Preserve Me! Viz" Application Window.

edges are relevant to construct the graph, messaging among the nodes is not (as it is volatile). Hence we only include node and edge related events in the batched message. Now, when live stream is selected in the “Preserve Me! Viz” application, it first fetches all the batched messages sequentially from the HTTP Mailbox and sets the canvas then starts fetching the live messages from there.

To visualize a network in real-time, push notifications can be used. But the HTTP Mailbox has a significant advantage over push notification here. The HTTP Mailbox keeps the record of all the events (in the form of messages) in chronological order that can be replayed later. Also, when the visualizer starts, it needs the current state of the graph that may not be available in push notifications, unless it sends the entire state of the graph every time instead of individual events. On the other hand, using the HTTP Mailbox service, we can fetch all the previous events from the beginning and create the network graph in the visualizer that represents the current state, when the visualizer starts. From that point, it keeps checking the HTTP Mailbox periodically for any new messages. As soon as a new event message arrives in the HTTP Mailbox, the visualizer can reflect that on the canvas.

The “Preserve Me! Viz” application uses VivaGraphJS [71], a JavaScript-based graph drawing library. We developed the timeline animation, player interface, and specified the data format used to animate the visualization.

9.3 SUMMARY

In this chapter we described our two web applications, “Preserve Me!” and “Preserve Me! Viz” that are using the HTTP Mailbox service. We described the working of these applications and how they are using the HTTP Mailbox for their communication needs.

“Preserve Me!” is a human-assisted web preservation application. This application was the reason that caused us to explore various browser-based web communication options and finally gave birth to the HTTP Mailbox.

“Preserve Me! Viz” is a real-time interactive network visualization tool that shows nodes of a network, their connections, and communications. It reflects the changes in the network as they occur. This tool was built to visualize the network of web objects created by the “Preserve Me!” application. It uses the HTTP Mailbox for live visualization, where all the events are being sent from “Preserve Me!”.

CHAPTER 10

EVALUATION

We perform both qualitative and quantitative evaluations of the HTTP Mailbox against various other web communication options. In qualitative evaluation we compare various features of communication like reliability and scale. In quantitative evaluation we formulate various quantities like availability and network usage.

10.1 FEATURE COMPARISON

Table VII gives a quick overview of various features among various communication systems discussed in chapters 2, 3, and 5. 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 suitable for web communication especially using web browsers. On the other hand, the HTTP Mailbox is primarily made with RESTful web communication in mind.

TABLE VII
Feature Comparison of Various Messaging Systems

Feature	**Linda**	**HTTP**	**Relay HTTP**	**EMS**	**Bleeps**	**HTTP Mailbox**
Multicast	Yes	No	No	Yes	Yes	Yes
Non-Blocking	Yes	No	No	Yes	Yes	Yes
Reliability	Yes	Yes	Yes	Yes	No	Yes
Message Size	Any	Any	Any	Any	Short	Any
Browser Support	No	Limited	Full	No	Full	Full
Transport	Shared Memory	Web	Web	Web	Web	Web

10.2 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 the sender can proceed without being blocked.

10.3 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:

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

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

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

If these M messages are sent in N ($\leq M$) batches using `application/http` [2] MIME type then the cost of HTTP Mailbox communication will reduce further while cost of HTTP communication will remain the same.

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

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 batches. For simplicity, we have ignored the communication cost introduced by Pull [34] 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.

10.4 SUMMARY

In this chapter we evaluated the HTTP Mailbox qualitatively and quantitatively against various other communication systems. These evaluations show that the HTTP Mailbox is the best option for browser-based non-blocking RESTful communication. Our evaluation also demonstrates that the HTTP Mailbox can save significant amount of network usage and time in group communication and batched messaging from sender's perspective as it shifts the responsibility of message retrieval to the recipient.

CHAPTER 11

FUTURE WORK

For access control, security, privacy, integrity, and authenticity [72], we are considering techniques like OAuth [55], public-key encryption [73], and hashing [74, 75]. We have discussed these techniques in chapter 7 but did not implement and evaluate them. 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 destructive-read (cf. Linda's "in" function) and message access log features in the HTTP Mailbox and evaluate how they affect the utility and performance of the system.

We would like to explore the possibility of how multiple mailboxes can work together to improve fault-tolerance and availability. For this purpose, shared message store, peer-to-peer message store, or hybrid storage system can be used.

In the case of shared message store, there will be one message store and several instances of the HTTP Mailbox will be connected to it. All instances of the HTTP mailbox will be storing their messages in (and accessing them from) the same shared store. In this centralized message store system, the message store can be the bottleneck and single point of failure.

We would also like to explore the possibility of a globally distributed peer-to-peer key-value based data storage system to power the HTTP Mailbox. This will allow the HTTP Mailbox to utilize a single shared data store across all the instances of the HTTP Mailbox rather than having a private message store per instance. In this case the HTTP Mailbox will work as an API gateway for the shared message storage to store and retrieve messages. To optimize the search time and storage space, these message stores may archive old messages and only maintain recent messages over the peer-to-peer network.

A hybrid message storage system can be designed by allowing individual HTTP Mailbox instances to have their own independent storage and periodically exchange messages in bulk with other message stores to synchronize.

CHAPTER 12

CONCLUSIONS

Originally the Web was envisioned as a read-write system for the web resources. But this vision was only partially embraced after the popularity of Web 2.0. Traditionally, general web services did not utilize REST and mainly relied on the GET and POST methods of HTTP in an RPC style to perform all the CRUD operations. Until recently, the Web was mainly navigated 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. Recently, several web frameworks/libraries have started supporting RESTful web services through APIs. To support HTTP methods other than GET and POST in browsers, these frameworks have used hidden HTML form fields as a workaround to convey the desired HTTP method to the server application. The server-side web application then infers the desired HTTP method based on special parameter values. 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. Browser-based communication also suffers from cross-origin restrictions for security reasons.

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 (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!" and the "Preserve Me! Viz" applications. We have also made the code of our implementation available on GitHub [65].

REFERENCES

[1] L. Dusseault and J. Snell, “PATCH Method for HTTP.” RFC 5789, Mar. 2010.

[2] 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.

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

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

[5] 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.

[6] CakePHP Cookbook, “Creating Forms.” `http://book.cakephp.org/2.0/en/core-libraries/helpers/form.html#creating-forms`, 2013.

[7] hawkeye, “HttpMethodsMiddleware.” `http://djangosnippets.org/snippets/174/`, 2007.

[8] S. Michelotti, “Implementing a delete link with mvc 2 and httpmethodoverride.” `http://wblo.gs/Zk6`, 2012.

[9] Anne van Kesteren, “Cross-Origin Resource Sharing.” `http://www.w3.org/TR/2013/CR-cors-20130129/`, 2013.

[10] J. Aubourg, J. Song, and H. R. M. Steen, “XMLHttpRequest.” `http://www.w3.org/TR/2012/WD-XMLHttpRequest-20121206/`, 2012.

[11] M. Hossain, “USING CORS.” `http://www.html5rocks.com/en/tutorials/cors/#toc-creating-the-xmlhttprequest-object`, 2012.

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

[13] N. Carriero and D. Gelernter, “Linda in Context,” *Communications of the ACM*, vol. 32, no. 4, pp. 444 — 458, 1989.

[14] D. Gelernter, “Generative Communication in Linda,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

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

[16] R. Fielding and R. Taylor, “Principled design of the modern web architecture,” *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[17] 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*, vol. 6, no. 1, pp. 61 – 69, 2008.

[18] 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.

[19] The Apache Software Foundation, “Apache HTTP Server.” `http://projects.apache.org/projects/http_server.html`, 1995.

[20] Netscape, “DMOZ Open Directory Project.” `http://www.dmoz.org/`, 1998.

[21] B. Kantor and P. Lapsley, “Network News Transfer Protocol.” RFC 977, Feb. 1986.

[22] J. Oikarinen and D. Reed, “Internet Relay Chat Protocol.” RFC 1459, May 1993.

[23] P. Saint-Andre, “Extensible Messaging and Presence Protocol (XMPP): Core.” RFC 6120, Mar. 2011.

[24] A. Barth, C. Jackson, and J. C. Mitchell, “Robust Defenses for Cross-Site Request Forgery,” in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 75–88, ACM, 2008.

[25] T. Garnock-Jones, “Relay HTTP.” `http://reversehttp.net/relay-http-spec.html`, 2009.

[26] The Apache Software Foundation, “Apache Qpid.” `http://qpid.apache.org/`, 2009.

[27] J. O’Hara, “Advanced Message Queuing Protocol.” `http://www.amqp.org/`, 2003.

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

[29] Oracle, “Java Platform, Enterprise Edition (Java EE).” `http://www.oracle.com/technetwork/java/javaee/overview/index.html`, 2012.

[30] VMware, “RabbitMQ.” `http://www.rabbitmq.com/`, 2010.

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

[32] STOMP, “Simple Text Oriented Messaging Protocol.” `http://stomp.github.io/`, 2012.

[33] I. Fette, “The WebSocket Protocol.” RFC 6455, Dec. 2011.

[34] 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*, pp. 15–22, IEEE, 2007.

[35] H. Van de Sompel, R. Sanderson, M. Klein, M. L. Nelson, B. Haslhofer, S. Warner, and C. Lagoze, “A Perspective on Resource Synchronization,” *D-Lib Magazine*, vol. 18, no. 9, 2012.

[36] M. Klein, R. Sanderson, H. Van de Sompel, S. Warner, B. Haslhofer, C. Lagoze, and M. L. Nelson, “A Technical Framework for Resource Synchronization,” *D-Lib Magazine*, vol. 19, no. 1, 2013.

[37] J. Coglan, “FAYE - Simple pub/sub messaging for the web.” `http://faye.jcoglan.com/`, 2009.

[38] A. Russell, G. Wilkins, D. Davis, and M. Nesbitt, "Bayeux Protocol." `http://svn.cometd.com/trunk/bayeux/bayeux.html`, 2007.

[39] Bitly, Inc., "Bitly." `http://en.wikipedia.org/wiki/Bitly`, 2008.

[40] Twitter, "Twitter search rules and restrictions." `https://support.twitter.com/forums/10713/entries/42646`, 2012.

[41] Evan Prodromou, "StatusNet." `http://en.wikipedia.org/wiki/StatusNet`, 2010.

[42] 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.

[43] S. Alam, C. L. Cartledge, and M. L. Nelson, "HTTP Mailbox - Asynchronous RESTful Communication," tech. rep., 2013.

[44] H. Van de Sompel, M. L. Nelson, R. Sanderson, L. L. Balakireva, S. Ainsworth, and H. Shankar, "Memento: Time Travel for the Web," tech. rep., 2009.

[45] R. Alarcon, E. Wilde, and J. Bellido, "Hypermedia-driven RESTful Service Composition," *Service-Oriented Computing*, pp. 111–120, 2011.

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

[47] IANA, "MIME Media Types." `http://www.iana.org/assignments/media-types`, 2013.

[48] R. T. Fielding, "REST APIs must be hypertext-driven." `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`, 2008.

[49] RDF Working Group, "Resource Description Framework (RDF)." `http://www.w3.org/RDF/`, 2004.

[50] D. Brickley and L. Miller, "Foaf vocabulary specification 0.98," *Namespace Document*, vol. 9, 2010.

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

[52] A. Barth, “The Web Origin Concept.” RFC 6454, Dec. 2011.

[53] E. Rescorla, “HTTP Over TLS.” RFC 2818, May 2000.

[54] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2.” RFC 5246, Aug. 2008.

[55] E. D. Hardt, “The OAuth 2.0 Authorization Framework.” RFC 6749, Oct. 2012.

[56] A. Freier, P. Karlton, and P. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0.” RFC 6101, Aug. 2011.

[57] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2007.

[58] B. Kaliski, “PKCS #1: RSA Encryption Version 1.5.” RFC 2313, Mar. 1998.

[59] Stephan Fowler, “HTTPsec.” `http://en.wikipedia.org/wiki/HTTPsec`, 2006.

[60] Y. M. Matsumoto, “Ruby programming language.” `http://www.ruby-lang.org/`, 1995.

[61] B. Mizerany, “Sinatra.” `http://www.sinatrarb.com/`, 2007.

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

[63] T. Jones and E. Fernandez, “Fluidinfo.” `http://fluidinfo.com/`, 2009.

[64] E. Seidel, “fluidinfo.rb.” `https://github.com/gridaphobe/fluidinfo.rb`, 2010.

[65] S. Alam, “HTTPMailbox.” `https://github.com/ibnesayeed/HTTPMailbox`, 2013.

[66] 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*, pp. 349–352, ACM, 2009.

[67] 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*, pp. 109–118, ACM, 2010.

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

[69] W3C SVG Working Group, “Scalable Vector Graphics (SVG).” `http://www.w3.org/Graphics/SVG/`, 2001.

[70] D. Crockford, “The application/json Media Type for JavaScript Object Notation (JSON).” RFC 4627, July 2006.

[71] A. Kashcha, “VivaGraphJS.” `https://github.com/anvaka/VivaGraphJS`, 2011.

[72] C. Kaufman, R. Perlman, and M. Speciner, *Network security: private communication in a public world*. Prentice Hall Press, 2002.

[73] J. Jonsson and B. Kaliski, “Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1.” RFC 3447, Feb. 2003.

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

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

[76] J. Postel, “Internet Protocol.” RFC 791, Sept. 1981.

[77] S. Deering and R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification.” RFC 2460, Dec. 1998.

[78] N. H. Tollervey, “Bug 608735 - XMLHttpRequest’s getAllResponseHeaders() returns null if the request is the result of CORS.” `https://bugzilla.mozilla.org/show_bug.cgi?id=608735`, 2010.

[79] Stuart Ng, “Bug 41210 - Cross Origin XMLHttpRequest can not expose headers indicated in Access-Control-Expose-Headers HTTP Response Header.” `https://bugs.webkit.org/show_bug.cgi?id=41210`, 2010.

[80] jQuery Team, “jQuery.” `http://jquery.com/`, 2012.

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-Header          = HM-res-header | HM-req-header
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" ":" "sent by"
                     SP (IP | IPv6 | Host)
                     SP "on behalf of" SP absoluteURI
                     SP "delivered by" SP absoluteURI
Sender-header      = "HM-Sender" ":" absoluteURI
Memento-header     = "Memento-Datetime" ":" HTTP-date
```

`Link` is defined in RFC 5988 [51], `Memento-Datetime` is defined in [44] which uses preferred fixed-width format of `HTTP-date`, `IP` is defined in RFC 791 [76], `IPv6` is defined in RFC 2460 [77], and remaining terms are inherited from RFC 2616 [2] unless defined here.

APPENDIX B

CODE SAMPLES

Code 21. Atom XML Based Sample ResourceMap

```xml
<?xml version="1.0" encoding="UTF-8"?>
<entry xmlns="http://www.w3.org/2005/Atom"
  xmlns:oreatom="http://www.openarchives.org/ore/atom/"
  xmlns:dcterms="http://purl.org/dc/terms/"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:ore="http://www.openarchives.org/ore/terms/"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:grddl="http://www.w3.org/2003/g/data-view#"
  xmlns:relationship="http://purl.org/vocab/relationship/"
  xmlns:usw="http://wsdl.cs.odu.edu/uswdo/terms/"
  grddl:transformation="http://www.openarchives.org/ore/atom/atom-grddl.xsl">
  <id>tag:uswdo.cs.odu.edu,2012-11-01:arxiv-0801-4807v1</id>
  <link rel="alternate"
    type="text/html"
    href="http://arxiv.cs.odu.edu/arxiv-0801-4807v1.html" />
  <link rel="self"
    type="application/atom+xml"
    href="http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml" />
  <link rel="edit"
    type="application/atom+xml"
    href="http://cetus.cs.odu.edu:10123/remmod/http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml" />
  <link rel="http://wsdl.cs.odu.edu/uswdo/terms/copy"
    type="application/atom+xml"
    href="http://cetus.cs.odu.edu:10123/remcopy/http://arxiv.cs.odu.edu/" />
  <link rel="http://wsdl.cs.odu.edu/uswdo/terms/httpmailbox#self"
    href="http://cetus.cs.odu.edu:10123/ms/http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml"
    usw:last-checked="" />
  <link rel="http://wsdl.cs.odu.edu/uswdo/terms/httpmailbox#all"
    href="http://cetus.cs.odu.edu:10123/ms/all"
    usw:last-checked="" />
  <link rel="http://wsdl.cs.odu.edu/uswdo/terms/httpmailbox#family"
    href="http://cetus.cs.odu.edu:10123/ms/tag:uswdo.cs.odu.edu,2012-11-01:arxiv-0801-4807v1"
    usw:last-checked="" />
  <link rel="http://www.openarchives.org/ore/terms/describes"
    href="http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml#aggregation" />
  <source>
    <author>
      <name>ODU WSDL ReM Generator</name>
      <uri>http://ws-dl-02.cs.odu.edu/</uri>
    </author>
  </source>
  <published>2012-12-09T23:19:38-05:00</published>
  <updated>2012-12-09T23:19:38-05:00</updated>
  <link
    rel="license"
    type="application/rdf+xml"
    href="http://creativecommons.org/licenses/by-nc/2.5/rdf" />
  <rights>
    This Resource Map is available under the Creative Commons Attribution-Noncommercial 2.5 Generic license
  </rights>
  <title>Automatic Text Area Segmentation in Natural Images</title>
  <author>
    <name>Syed Ali Raza Jafri</name>
  </author>
  <author>
    <name>Mireille Boutin</name>
  </author>
  <author>
    <name>Edward J. Delp</name>
  </author>
  <category term="http://www.openarchives.org/ore/terms/Aggregation"
    label="Aggregation"
    scheme="http://www.openarchives.org/ore/terms/" />
  <category term="2008-01-31T01:46:32+00:00"
    scheme="http://www.openarchives.org/ore/atom/created" />
  <category term="2008-01-31T01:46:32+00:00"
    scheme="http://www.openarchives.org/ore/atom/modified" />
  <category term="3"
    scheme="http://wsdl.cs.odu.edu/uswdo/terms/preservationCopiesMinimumNumber" />
  <category term="5"
    scheme="http://wsdl.cs.odu.edu/uswdo/terms/preservationCopiesMaximumNumber" />
  <category term="0.5"
    scheme="http://wsdl.cs.odu.edu/uswdo/terms/beta" />
  <category term="0.5"
    scheme="http://wsdl.cs.odu.edu/uswdo/terms/gamma" />
  <link rel="http://wsdl.cs.odu.edu/uswdo/terms/family#parent"
    type="application/atom+xml"
    title="Automatic Text Area Segmentation in Natural Images"
    href="http://arxiv.cs.odu.edu/rems/arxiv-0801-4807v1.xml" />
  <link rel="http://www.openarchives.org/ore/terms/aggregates"
    type="text/html"
    title="Automatic Text Area Segmentation in Natural Images"
    href="http://arxiv.cs.odu.edu/arxiv-0801-4807v1.html" />
  <link rel="http://www.openarchives.org/ore/terms/aggregates"
    type="text/html"
    title="[0801.4807v1] Automatic Text Area Segmentation in Natural Images"
    href="http://arxiv.org/abs/0801.4807v1"
    usw:preservation-mode="skip" />
  <link rel="http://www.openarchives.org/ore/terms/aggregates"
    type="application/pdf"
    title="[PDF] Automatic Text Area Segmentation in Natural Images"
    href="http://arxiv.org/pdf/0801.4807v1"
    usw:preservation-mode="skip" />
  <link rel="http://www.openarchives.org/ore/terms/aggregates"
    type="application/x-tgz"
    title="[Source] Automatic Text Area Segmentation in Natural Images"
    href="http://arxiv.org/e-print/0801.4807v1"
    usw:preservation-mode="skip" />
  <link rel="http://wsdl.cs.odu.edu/uswdo/terms/friend"
    href="http://arxiv.cs.odu.edu/rems/arxiv-0704-3647v1.xml"
    title="Evaluating Personal Archiving Strategies for Internet-based Information" />
</entry>
```

Code 22. Revision History of Wiki-based Mailbox of DO1

```
$ curl -i -G -d "format=xml&action=query&titles=Do1&prop=revisions&rvprop=ids|timestamp|content&rvlimit=20" \
> http://resourcemap.wikia.com/api.php

HTTP/1.1 200 OK
Server: Apache
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
Cache-Control: max-age=3600, s-maxage=3600, public
Content-Type: text/xml; charset=utf-8
X-Cacheable: YES
Content-Length: 4175
Accept-Ranges: bytes
Date: Thu, 01 Aug 2013 15:19:36 GMT
Connection: keep-alive
X-Served-By: cache-s24-SJC, cache-at52-ATL
X-Cache: HIT, HIT
X-Cache-Hits: 1, 1
X-Timer: S1375369946.303683996,VS0,VS35,VE36,VE430236
Vary: Accept-Encoding
Set-Cookie: wikia_beacon_id=3zwg7OUEb6; domain=.wikia.com; path=/; expires=Tue, 28 Jan 2014 15:19:36 GMT;
Set-Cookie: Geo={%22city%22:%22FIXME%22%2C%22country%22:%22US%22%2C%22continent%22:%22NA%22}; path=/
X-Age: 430

<?xml version="1.0"?>
<api>
  <query>
    <pages>
      <page pageid="2040" ns="0" title="Do1">
        <revisions>
          <rev revid="139816" parentid="139815" timestamp="2012-07-31T21:00:14Z" xml:space="preserve">
            http://uswdo1.cs.odu.edu/uswdo/do/do50001.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139815" parentid="139813" timestamp="2012-07-31T20:59:56Z" xml:space="preserve">BEGIN</rev>
          <rev revid="139813" parentid="139809" timestamp="2012-07-31T20:52:56Z" xml:space="preserve">
            http://uswdo1.cs.odu.edu/uswdo/do/do50001.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139809" parentid="139804" timestamp="2012-07-31T20:52:20Z" xml:space="preserve">BEGIN</rev>
          <rev revid="139804" parentid="139801" timestamp="2012-07-31T20:29:53Z" xml:space="preserve">
            http://uswdo1.cs.odu.edu/uswdo/do/do50001.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139801" parentid="139797" timestamp="2012-07-31T20:29:33Z" xml:space="preserve">BEGIN</rev>
          <rev revid="139797" parentid="139796" timestamp="2012-07-31T20:07:51Z" xml:space="preserve">
            http://uswdo1.cs.odu.edu/uswdo/do/do50001.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139796" parentid="139794" timestamp="2012-07-31T20:07:05Z" xml:space="preserve">BEGIN</rev>
          <rev revid="139794" parentid="139793" timestamp="2012-07-31T19:55:37Z" xml:space="preserve">
            http://uswdo1.cs.odu.edu/uswdo/do/do50009.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139793" parentid="139792" timestamp="2012-07-31T19:55:28Z" xml:space="preserve">
            http://uswdo4.cs.odu.edu/uswdo/do/do50008.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139792" parentid="139791" timestamp="2012-07-31T19:55:09Z" xml:space="preserve">
            http://uswdo3.cs.odu.edu/uswdo/do/do50007.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139791" parentid="139790" timestamp="2012-07-31T19:54:40Z" xml:space="preserve">
            http://uswdo2.cs.odu.edu/uswdo/do/do50006.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139790" parentid="139789" timestamp="2012-07-31T19:54:24Z" xml:space="preserve">
            http://uswdo1.cs.odu.edu/uswdo/do/do50005.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139789" parentid="139788" timestamp="2012-07-31T19:54:16Z" xml:space="preserve">
            http://uswdo4.cs.odu.edu/uswdo/do/do50004.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139788" parentid="139787" timestamp="2012-07-31T19:54:05Z" xml:space="preserve">
            http://uswdo2.cs.odu.edu/uswdo/do/do50002.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139787" parentid="139786" timestamp="2012-07-31T19:53:49Z" xml:space="preserve">
            http://uswdo3.cs.odu.edu/uswdo/do/do50003.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139786" parentid="139784" timestamp="2012-07-31T19:53:33Z" xml:space="preserve">
            http://uswdo1.cs.odu.edu/uswdo/do/do50001.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139784" parentid="139775" timestamp="2012-07-31T19:51:46Z" xml:space="preserve">BEGIN</rev>
          <rev revid="139775" parentid="139773" timestamp="2012-07-31T19:31:11Z" xml:space="preserve">
            http://uswdo1.cs.odu.edu/uswdo/do/do50001.xml http://uswdo1.cs.odu.edu/uswdo/do/do1.xml friendship
          </rev>
          <rev revid="139773" parentid="139764" timestamp="2012-07-31T19:29:57Z" xml:space="preserve">BEGIN</rev>
        </revisions>
      </page>
    </pages>
  </query>
  <query-continue>
    <revisions rvstartid="139764" />
  </query-continue>
</api>
```

APPENDIX C

CORS RELATED BROWSER BUG

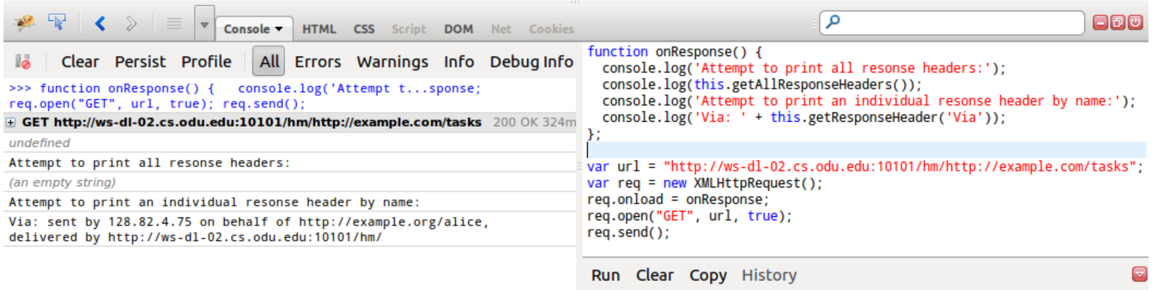
In order to allow the clients to access response headers in a CORS request, the remote server must send `Access-Control-Expose-Headers` header in the response. This header contains a coma separated list of response headers that will be exposed to the client. If a browser makes an Ajax request, the "XMLHttpRequests" object has a method "getAllResponseHeaders()" that should return all the exposed headers. At the time of writing many popular web browsers have buggy implementation of this method and they do not honor `Access-Control-Expose-Headers` [78, 79]. These browsers still return expected result if "getResponseHeader(name)" method is called to query individual response headers.

Code 23. CORS Response Headers Using cURL

```
$ curl -I -H "Origin: example.com" http://hm.cs.odu.edu/hm/http://example.com/tasks
HTTP/1.1 200 OK
Server: HTTP Mailbox
Content-type: message/http
Date: Mon, 27 May 2013 16:35:24 GMT
Memento-Datetime: Mon, 27 May 2013 16:32:17 GMT
Via: sent by 128.82.4.75 on behalf of http://example.org/alice, delivered by http://hm.cs.odu.edu/hm/
Link: <http://hm.cs.odu.edu/hm/http://example.com/tasks>; rel="current",
 <http://hm.cs.odu.edu/hm/id/4a571ae3-970e-4226-9ad6-3d4cbd02be3a>; rel="self",
 <http://hm.cs.odu.edu/hm/id/3475e58a-9458-496d-a90e-f31b24ef8e04>; rel="first",
 <http://hm.cs.odu.edu/hm/id/4a571ae3-970e-4226-9ad6-3d4cbd02be3a>; rel="last",
 <http://hm.cs.odu.edu/hm/id/3475e58a-9458-496d-a90e-f31b24ef8e04>; rel="previous"
Content-Length: 43
Access-Control-Allow-Origin: example.com
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Expose-Headers: Link, Via, Date, Memento-Datetime
Access-Control-Max-Age: 1728000
Access-Control-Allow-Credentials: true
Vary: Origin
Connection: keep-alive

$
```

To understand this browser bug and its consequences we will see an example. Code 23 shows CORS response headers of a URI from the HTTP Mailbox using cURL. In this illustration `Access-Control-*` headers are CORS specific. The `Access-Control-Expose-Headers` header tells that `Link`, `Via`, `Date`, and `Memento-Datetime` response headers should be accessible to the clients.

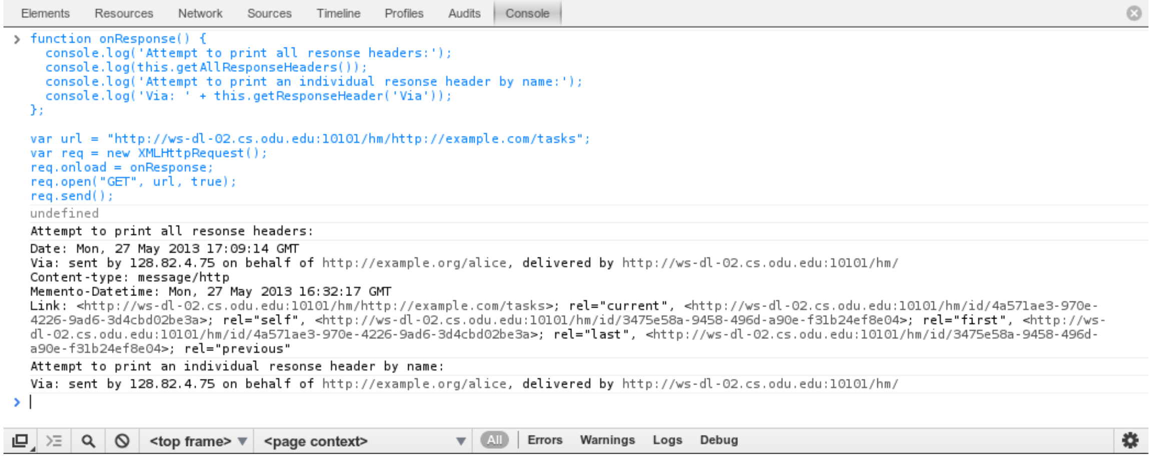
Now we will attempt to access the headers of this resource from Firefox browser (version 20.0) using Firebug (a developer utility add-on for Firefox). Fig. 22(a) illustrates an Ajax GET request on the same URI as in the cURL example above.



```
>>> function onResponse() {   console.log('Attempt t...sponse;
req.open("GET", url, true); req.send();
GET http://ws-dl-02.cs.odu.edu:10101/hm/http://example.com/tasks   200 OK 324m
undefined
Attempt to print all resonse headers:
(an empty string)
Attempt to print an individual resonse header by name:
Via: sent by 128.82.4.75 on behalf of http://example.org/alice,
delivered by http://ws-dl-02.cs.odu.edu:10101/hm/
```

```
function onResponse() {
  console.log('Attempt to print all resonse headers:');
  console.log(this.getAllResponseHeaders());
  console.log('Attempt to print an individual resonse header by name:');
  console.log('Via: ' + this.getResponseHeader('Via'));
};

var url = "http://ws-dl-02.cs.odu.edu:10101/hm/http://example.com/tasks";
var req = new XMLHttpRequest();
req.onload = onResponse;
req.open("GET", url, true);
req.send();
```

(a) Firefox (Version 20.0)



```
> function onResponse() {
    console.log('Attempt to print all resonse headers:');
    console.log(this.getAllResponseHeaders());
    console.log('Attempt to print an individual resonse header by name:');
    console.log('Via: ' + this.getResponseHeader('Via'));
  };

  var url = "http://ws-dl-02.cs.odu.edu:10101/hm/http://example.com/tasks";
  var req = new XMLHttpRequest();
  req.onload = onResponse;
  req.open("GET", url, true);
  req.send();
  undefined
  Attempt to print all resonse headers:
  Date: Mon, 27 May 2013 17:09:14 GMT
  Via: sent by 128.82.4.75 on behalf of http://example.org/alice, delivered by http://ws-dl-02.cs.odu.edu:10101/hm/
  Content-type: message/http
  Memento-Datetime: Mon, 27 May 2013 16:32:17 GMT
  Link: <http://ws-dl-02.cs.odu.edu:10101/hm/http://example.com/tasks>; rel="current", <http://ws-dl-02.cs.odu.edu:10101/hm/id/4a571ae3-970e-
  4226-9ad6-3d4cbd02be3a>; rel="self", <http://ws-dl-02.cs.odu.edu:10101/hm/id/3475e58a-9458-496d-a90e-f31b24ef8e04>; rel="first", <http://ws-
  dl-02.cs.odu.edu:10101/hm/id/4a571ae3-970e-4226-9ad6-3d4cbd02be3a>; rel="last", <http://ws-dl-02.cs.odu.edu:10101/hm/id/3475e58a-9458-496d-
  a90e-f31b24ef8e04>; rel="previous"
  Attempt to print an individual resonse header by name:
  Via: sent by 128.82.4.75 on behalf of http://example.org/alice, delivered by http://ws-dl-02.cs.odu.edu:10101/hm/
> |
```

(b) Google Chrome(Version 26.0)

Fig. 22. Accessing CORS Response Headers.

Once the response from the server is loaded we attempt to print all the response headers. We expect to see the name value pairs of all the exposed response headers. Firefox returns an empty string because of the bug in the implementation. Next, we attempt to access an individual response header (in this illustration it is `Via` header) and we get expected response.

Now we will run the same experiment in Google Chrome browser (version 26.0) which has got the bug fixed. Fig. 22(b) illustrates the successful attempt to print all the exposed response headers at once as well as individual response headers by name.

Retrieving the chain of messages from the HTTP Mailbox is dependent upon `Link` header. If for some reason a client cannot access that, it will not be able to identify the next (or previous) message in the chain hence, it will fail to fetch the messages.

A workaround to this bug is to override the “getAllResponseHeaders()” method for CORS requests. First of all use `getResponseHeader(Access-Control-Expose-Headers)` and parse the returned value to get a list of all exposed headers. Add this list to the simple response headers and then query for each of the headers from the list using `getResponseHeader(name)` and concatenate the result. return this concatenated result from overridden “getAllResponseHeaders()” method.

As we have seen in the illustrations that exposed headers can always be retrieved individually by name then this bug should not be an issue. It turns out that jQuery [80] overrides the default “getAllResponseHeaders()” and “getResponseHeader(name)” methods in a way that “getResponseHeader(name)” method is derived from the outcome of “getAllResponseHeaders()”. This is an issue if the client is making Ajax CORS requests using jQuery in buggy browsers. In that case the client will not get the appropriate values of individual headers by name.

A rather simpler but not generic patch to jQuery (tested in version 1.8.1) is illustrated below which assumes that the client knows a list of expected response headers that are exposed.

```
// Firefox CORS bug fix in jQuery-1.8.1
var _super = $.ajaxSettings.xhr;
$.ajaxSetup({
  xhr: function() {
    var xhr = _super();
    var getAllResponseHeaders = xhr.getAllResponseHeaders;
    xhr.getAllResponseHeaders = function() {
      var allHeaders = getAllResponseHeaders.call(xhr);
      if (allHeaders) {
        return allHeaders;
      }
      allHeaders = "";
      var conHeader = function(i, header_name) {
        if (xhr.getResponseHeader(header_name)) {
          allHeaders += header_name + ": " + xhr.getResponseHeader( header_name ) + "\n";
        }
      };
      $(["Cache-Control", "Content-Language", "Content-Type", "Expires", "Last-Modified", "Pragma"]).each(conHeader);
      $(["Location", "Link", "Via", "Date", "Memento-Datetime"] ).each(conHeader);
      return allHeaders;
    };
    return xhr;
  }
});
```

VITA

Sawood Alam
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

EDUCATION

M.S.	Computer Science, Old Dominion University, 2013
B.Tech.	Computer Science, Jamia Millia Islamia, 2008

EMPLOYMENT AND FELLOWSHIPS

2009 - Present	Research Assistant, Old Dominion University, USA
2008 - 2009	Computer Scientist, Belzabar Software, India
2006 - 2008	Research Assistant, CIT Jamia Millia Islamia, India
2006 and 2008	Research Fellow (Two Urdu Related Projects), Sarai CSDS, India

PUBLICATIONS AND PRESENTATIONS

HTTP Mailbox - Asynchronous RESTful Communication
Document Compression and Ciphering Using Pattern Matching Technique
WIDE - Web-based Integrated Development Environment
Various Workshops on Software Localization and Rapid Web Development Using Ruby on Rails
A complete list is available at `http://www.cs.odu.edu/~salam/pubs.html`

AFFILIATIONS AND ACTIVITIES

Active Administrator, UrduWeb - A mission to bring Unicode Urdu language on the Web
Vice Chair, Computer Society of India - Student Chapter JMI
Founding President and Co-founder of ODU and JMI Linux Users Groups respectively
Various Workshops on Rapid Web Development Using Ruby on Rails
Chief Editor and Co-editor of Izhar-e-Haq and Samt Urdu Magazines respectively

CONTACT

Email	salam@cs.odu.edu	ibnesayeed@gmail.com
Homepage	`http://www.cs.odu.edu/~salam/`	`http://www.ibnesayeed.com/`
Mobile	+1 757 606 0011	
Lexical Signature	Web, Digital Library, Web Archiving, Ruby on Rails, PHP, HTML, CSS, JavaScript, ExtJS, Urdu, RTL, and Linux	

Typeset using LaTeX.