PURIC: A Multimedia Uniform Resource Identifier Management System

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Flavio Okuhara Ishii

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

The types of content being transferred over the Internet are getting richer and larger; the number of social media channels users have to sift through to publish and find content is also increasing. Average users are uploading and downloading richer and larger media files as they feel the urge to share their content with others. This work explores a novel process for publishing personal media files on social applications, where the publisher retains control over the media, while the implementation follows the principles of the WWW. The Personal URI Channel (PURIC) system is introduced as a process that can take place along side social applications like email clients, social networking sites (i.e. Twitter and Facebook), and emerging decentralized social networking sites. The PURIC system is a media resource link management tool used for publishing and maintaining the links published on social applications. This work explores the feasibility, benefits, and drawbacks of the PURIC system. It reveals the modularity and scalability of the system, and how it compliments social applications without placing too much load on network traffic and server-side cpu processing.

Contents

List of Tables

List of Figures

List of Abbreviations

ACL	Access Control List
API	Application Programming Interface
AWS	Amazon Web Services
CCS	Cloud Computing Service
DSN	Decentralized Social Networking
GAE	Google App Engine
FOAF	Friend of a Friend
HTTP	HyperText Transfer Protocol
HTML	HyperText Markup Language
IaaS	Infrastructure as a Service
ISP	Internet Service Provider
JSON	Javascript Object Notation
P2P	Peer To Peer
PaaS	Platform as a Service
PURIC	Personal Universal Resource Identifier Channel
REST	REpresentational State Transfer
RPC	Remote Procedure Call
RSS	Rich Site Summary
SaaS	Software as a Service
SN	Social Networking
SNS	Social Networking Site
SOAP	Simple Object Access Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator or Universal Resource Locator
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	eXtensible Markup Language

Chapter 1

Introduction

Tim Berners-Lee created the internet in 1990 as a place where researchers could exchange structured data in a universal fashion. It has since been able to scale significantly due to improvements in the HyperText Transfer Protocol (HTTP) along with the guidance of the REpresentational State Transfer (REST) architectural style [44]. The World Wide Web (WWW) has become the largest and successful software system as a result of these technologies. The RESTful design principles support the changing nature and massive scalability potential of an online system.

Web 2.0 emerged as a way for people to easily contribute content to different Social Networking sites like Twitter [54] and Facebook [55], or other social applications such as YouTube [23] and WordPress[1] blogs. Web users now rely on social applications to stay current, viewing updates and trends from family, friends, businesses, and other groups that may interest them. The data published by a social application user and his online activities can become enclosed in the social application silo, as property of the social application or service provider. Some Social Networking (SN) sites provide access to the published data via an Application Programming Interface (API), but they are still behind walled gardens since the services have policies in place to regulate the data accessibility and it is difficult for the general public to make use of such technical services. It is difficult to export published content from one social app to another, and things could get complicated if the link has been already published in different media channels. A number of factors are calling for a new method of publishing and consuming rich media in the Web:

- The World Wide Web Consortium (W3C) and Berners-Lee have recognized that the closed concept of popular SN sites do not follow the open and scalable nature

[1] http://wordpress.com

of the WWW design. The stewardship of this idea has gone public with the W3C Workshop on the Future of Social Networking [72] and with Tim Berners-Lee speaking out on Linked Data and against the walled gardens of SN sites [29, 30]. Such movements play key roles in addressing the issues of SN silos and opening the discussion over who owns and controls the user data.

- Decentralized Social Networking (DSN) sites have emerged as an alternative to SN sites for many users. DSN sites may offer the same functionalities of existing SN sites but also support more freedom for the user to take his/her account credentials and published content from one DSN host to another. DSN sites have not been very successful as they are competing against very well established SN sites and there remains many technical design challenges to be solved in the decentralized arena.

- Internet traffic is increasing significantly as more and more people become connected and contribute to the social Web. This also means that there is an increase in HTTP requests that need to be handled by a Web server. There is also an abundance of client platforms making HTTP requests, with different cpu power, memory, screen size, and capabilities. As this internet usage across client platforms rises including in the mobile arena, the need for client platform specific content and experience (responsive content) will also rise.

- Personal Web content ownership and control will become more relevant and desired as more and more people use the internet to publish, share, store, and control their data that exist in various social networks. A different way of sharing social content is needed and it must be designed to integrate well with the different types of social applications. The sharing process should support the publishing of Web resources in a more user-controlled, robust, flexible, and responsive manner.

The next chapter discusses the problem definitions that takes in the four factors above. Chapter 3 breaks down the history of HTTP, REST, the emergence of responsive content, Web resource versions, and previous works. Chapter 4 identifies a set of guidelines and details the design approach to attain the goals outlined in chapter

2. Chapter 5 discusses the implementation of the approach and chosen technologies. Chapter 6 evaluates the system implemented and mentions some of the future work. Lastly Chapter 7 provides conclusions of this research.

Chapter 2

Problem Definition

This thesis focuses on a media resource publishing scheme that supports the sustainable growth for the WWW along side social applications. Sustainable growth in the sense that content can be added and controlled by the publisher in a way that does not go against the open design principles of the WWW, and that does not degrade the internet with extra CPU processing, memory usage, or network traffic congestion.

This Chapter describes in more detail all the issues associated with the topic of social media publishing. Social applications need a new publishing model that provide ownership and flexibility of published content, as well as: (1) maintaining low CPU, (2) memory, and (3) network traffic loads. The following sections will go over the feasibility of implementing the solution.

2.1 Thesis Motivation

Print publishing and Web publishing need to be looked at differently, as there are more options in the Web publishing medium. The option of granting the ability for changing a published content can be extremely valuable in some cases. In print, once an author publishes his/her content, it cannot be modified or retrieved, only referenced to from a later publication. This can serve a good purpose for formal documents and news articles, but in the Web 2.0 arena there is the option to change what has been published and people know that it can be done, so it may as well be made accessible for social media users.

Social media applications, or SN sites, have the problem of being or becoming silos, or walled gardens as termed in [68]. The papers and summary report of the W3C Workshop of 2009 [72] served as the basis for the motivation behind this thesis.

Section 3.1 goes into detail as to its position on the state of SN and where it should go. This thesis takes some of the challenges discussed in [72] and handles them in the media publishing use case. SN sites lack the scalability found in the Web, as the data becomes enclosed in a place where only members can view them and where in some cases the published media becomes property of the SN site.

The current process for publishing video and image media files to social applications can negatively impact how the Web grows as a shared public space. A media file that is uploaded on Facebook becomes the property of Facebook and is accessible only within its silos, or network of users. Another negative impact arises when a published media file is updated; it will require a separate Facebook posting with the updated version re-uploaded, resulting in additional server processing and network loads.

When a YouTube link is published in a social application like Twitter, the published link remains public even if the video resource is updated as a new YouTube video [1], or if it is removed from YouTube and hosted in a different service. This results in the consumer user being redirected to an error page. If the Twitter posting is deleted it can still live in other users' Twitter public timelines, if that same link had been retweeted (reposted on Twitter) by someone else. Similar occurrences of incorrect or bad links can happen with other social applications like email clients, where messages that include media links can be forwarded to multiple recipients infinitely. The publisher has no control over the links published.

Three problems motivated the work for this thesis:

- Control and Robustness of Public Media Content - the control of shared data in social applications.
- Ownership and Hosting Flexibility - the challenges exposed for supporting ownership and hosting flexibility of the media file content.
- Processing Resources and Network Loads - The unnecessary CPU and memory usage, and network traffic for publishing and consuming content on social applications.

[1] YouTube videos may not be edited once published.

A social application problem needs to be quickly explored and solved before it escalates to a more severe problem, because of the masses of users in the system. The following sub sections describe each of these problems or motivations in detail.

2.2 Media Publishing Limitations in Social Apps

Most applications are now social applications, either by nature or by having a social component add-on. A social app should make it more effective to share and manage information among its readership, but that is not always the case. It is in the best interest of social applications to not only engage members to be active but also to grant them with more ownership and control over their own published content. This will aid in keeping the membership content (happy).

Email, forums, blogs, YouTube, Facebook, Twitter, and other social applications are all channels that people use to publish and share the links of their own media resource like documents, photographs, videos, and audio files. Once the content is published to a particular network (or email recipients) the target audience can follow the link, view the resource, and share it with others by re-publishing the link. Published resources are conveniently shared and re-published by others via a unique identification string, in Web terms it's called the Uniform Resource Identifier (URI).

Users have minimal control over what has been published, the user is dependent on the functionality exposed by the publication service. For example, YouTube currently does not allow publishers to edit their existing videos; if an update is needed. YouTube forces publishers to either remove the video or leave the video as is, and upload the updated video as a new video with a new URI.

Status updates are re-posted (or retweeted) on an SN site like Twitter depending on many factors. A retweet is the power given to people to selectively keep spreading an existing tweet (posting) on Twitter. There is no relationship between the number of followers (subscribers) and retweets a user gets, based on [54]. However, the act of re-posting information, which often contains links, occurs often in all social applications. Considering this, it is important to realize that when information and links are shared, they are likely to be re-published by others.

Similar patterns are followed by similar applications and services limiting the

control over the resources. The following is a scenario depicting the problems raised due to such constraints:

A videographer publishes a video on YouTube and shares the link on Twitter and Facebook. Later he realizes an important footage is missing from the video. While he edits the video on his computer his colleagues have re-tweeted and shared the original video link respectively on their Twitter status timeline and Facebook walls. Once the video is updated the videographer has to now remove the video from YouTube, upload the new one to YouTube, and once more share the new link on Twitter and Facebook.

The above situation causes the following problems:

- orphan links - the original URL is broken if the resource is removed from the host;
- clarification needed - an explanation and instruction must be posted by the publisher to guide users to the new Web resource;
- false/unintentional publication - the original URI was shared among many friends that may have also shared with their friends/followers, inheriting the previous problems;
- tedious management - with different versions of a resource circulating on the internet there is no simple way to control who sees which version.
- temporary incorrect search engine caching - search results will show the original URI until it is no longer being cached;

The media resource hosting application needs to provide a way to modify the published content [2], in such a way that an update or replacement the media will not require a new Uniform Resource Link (URL) to be published.

The problem becomes even more complex when the URLs get shared by other users, after the root URI of the main content has changed. Depending on the social application, published posts may not always be removed or replaced easily, cleanly, promptly, or at all; the post can still live on and can be found without the change

[2]Blogs and forum apps allow users to edit their posts. Facebook and Twitter allow the user to remove a post or tweet but not to edit it. Typical email clients are less forgiving, functioning more like the traditional print media, where it is nearly impossible to retract something published once it is released to the masses.

explanation trail of posts. In the case of an email message with a link, the problem is more severe and requires a follow up email explaining the changes, which would have to be sent to all these who received the link originally.

Removing any published media resource results in a 404 HTTP status error code if not handled properly. Seeing this on a Web browser is a bad example of user experience. The author of the published content is dependent on the data host serving up the data to handle these errors adequately. And although it is possible, there are no services in social applications that allow the publisher to redirect old links to a new link. Services can attempt to handle these cases appropriately, but as mentioned, if the linked data has been published and re-published by others the complexity of the case rises and it is not handled.

The other limitation only happens with Peer to Peer (P2P) based social applications. P2P applications require the users to be network connected in order for them to find each other and share their content. Applications like the Locker Project [17] and P2Pme [75] are examples of recent P2P social applications, which foster data ownership and control but tend to target the more tech-savvy users as oppose to the general public.

Data storage cloud services like DropBox may not have the problems mentioned above as files may be renamed while its published URL may remain the same. But the ideal social app must support resources that are published in all types of services in order to provide the user with the desired flexibility of moving resources from one host to another.

2.3 Publishing Decentralized Social Networking (DSN) Resources

DSN implementations [7, 75, 22] have emerged as an alternative to SN sites. But they have not been successfully embraced by the general public mostly due to its infancy stage and niche target audience. The general characteristics of a DSN site help sum up the need for a new publishing process:

- User can be a member of multiple DSN sites, and DSN sites can share users and

their content. There is cross site user authenticity (sign in) and authorization (access permission).

- User credentials and resources are not required to be centrally stored.
- User can migrate from one DSN site to another, taking all his/her social capital; also meaning that if postings used public URLs and URIs as opposed to uploading the file to the Site (like Facebook supports) for media files it would facilitate the migration.
- Open standards like HTTP, OAuth, and OpenID are followed to help with the connectivity across sites.
- Niche, non-mainstream, or a special interest groups are targeted as potential user groups.

Having to compete against well established technologies bring forth many challenges besides the marketing aspect of converting users to switch over. Popular SN sites have a large following, making it challenging for DSN Sites to compete. They also have the resources to make changes or integrate new features or revert old ones based on user demand, as seen with Facebook [42].

There are many challenges brought up in [72] that remain to be solved in order for DSN applications to survive as an integrated component of SN sites. It still inherits the limitation and SN site problems introduced in the previous section. Another design challenge of DSN sites is on how to share Web resources such as image and video files. DSN sites need a content publishing process that supports control, ownership, robustness, and flexibility to switch between DSN sites or share across DSN sites.

Controlling a resource shared across DSN sites is a challenge. The publisher may want to share a resource across two DSN sites, and not share or publish it at all with other DSN sites. Even more of a challenge is the access control for privately shared resources. A publicly shared resource has a URI that can be accessed by any user without the need for authentication and authorization. A privately shared resource is a URI that is only accessed by those that have been granted authorization to view its content by the publisher and have been authenticated by the host or third

party service to access it. Sharing a private URI across multiple social applications require an authentication and authorization module to be in place for each social application. This is not an ideal situation for developers as it would require much effort to implement and maintain all of the modules for each of the many social applications in existence.

2.4 Retaining Low Traffic and CPU Loads

A lot of content is publicly and privately published online. Facebook has 800 million active users with over 250 million photos uploaded per day [2, 41]. In 2011, Twitter had 490 million unique users worldwide per month, out of those there was an average of 400 tweets per minute containing a YouTube link [37]. Facebook has over 150 years of YouTube videos watched per day [37] and out of those there are no available stats for how many are retracted, replaced, removed, or simply left up there as invalid resources. Such scenarios can be accumulated causing an abundance of unnecessary network traffic and CPU processing. Furthermore, mobile users expect faster website loads [34] or nearly match the speed on their desktop computers in the near future. Low traffic loads and CPU loads must decrease and be kept low. This assumption is the basis for the thesis sub-problems for when resource links are published on social apps.

Sharing documents on social applications play a big role in the day to day communication between parties. People share links in emails, post links on Facebook and Twitter, publish and share YouTube videos, share files from DropBox, and re-post other people's posts. Cloud storage services are becoming a popular way to backup and share files online privately or publicly. The pattern of uploading data to one of these services and then sharing the link via email, Twitter, or another social app is also becoming a common publishing practice. An afterthought for these actions is that once a link to a content or file is published it can not, or at least should not, be changed to a different link without the cascading negative consequences.

The scenario described in 2.1.1 in addition to the growth in requests for Web resources from clients also implies a growth in failed and unnecessary resource traffic and computations. Unnecessary traffic can happen based on the following scenarios:

- A bad or dead link, where the content has been moved and it is not found - 404 HTTP Status Code.
- Incorrect or outdated resource returned and displayed.
- Small devices with low resolution screens downloading large resolution content.
- Lack of client-side caching where the e-tag is not utilized.

2.5 Thesis Goals

A system must be created to solve the problems mentioned in Section 2.1 or at least find compromises to diminish the problems, while still being able to integrate with existing and future social applications. More specifically the system needs to support the following scenarios for its users:

- Provide the ability to change resource hosts, and ultimately the root URI, without affecting the published URL due to a change in host, update in media file, or provide different versions for different groups.
- Retain the control for the accessibility of a resource published and hosted on a general purpose SN site (Twitter, Facebook, LinkedIn) or social application (email, blog, project management system).
- Provide a visual display of the resource based on the client application or platform requesting it.

The last scenario should be noted as a problem that occurs when a URL can be used for pointing to different formats of the same resource. What is presented depends on whether the user is using a mobile device with a small screen or a desktop browser on a larger screen. If the resource was to be an image, it would be sized according to the client application displaying it so that it fits appropriately on a small or large screen resolution.

Two thesis goals are formulated based on the motivations, problems, and scenarios mentioned earlier in this chapter. This thesis ultimately focuses on designing, implementing, and evaluating a system that resolves the following goals and subgoals:

- Evaluate the challenges of implementing a novel way to publish media in social applications, such that the link is robust and flexible, allowing the host or root URI to change while the published link stays the same.

  Modularity - integrate with existing social apps.

  Data control and ownership - publisher owns and controls the root URI of each URL published and re-published by others in any social application. Full control is retained by the publisher over the root resource to be consumed.

  Cross platform support - support a myriad of client platforms.

- Evaluate the implementation and find its drawbacks and limitations.

  Low Resource Usage - the system serves up Web content that is generic and does not result in extra CPU, memory, and traffic loads.

  Scalability - perform server-side load tests by measuring how well the system under high requests and traffic.

This thesis presents the design, implementation, and analyzes of a URI management tool called PURIC to uncover its benefits and drawbacks. PURIC adds an ownership and control layer to published media content on social applications, while providing the necessary flexibility to switch the whereabouts the content is hosted without breaking existing published links, or URLs - essentially creating a Personal URI Channel (PURIC) that is a suitable method for publishing media content to existing and future social applications. PURIC supports changes made to the origin of a published media content without the typical side effects of changing published links.

Chapter 3

Literature Review

The aforementioned PURIC system needs to provide users with a way to control their social Web resources, which integrates well with and not necessarily compete with existing social applications. This chapter points out existing studies building up to and supporting the need for a system like PURIC.

PURIC is a unique solution for a specific set of use cases in social media content publishing that overcomes the limitations of existing works. Its approach uses standards and popular technologies. This chapter introduces the concept of social applications, presents the current issues with SN and the recent attempts at decentralized solutions, and discusses the technological options chosen for the foundation of the PURIC system. Potential improvements in the end-user's experience is also discussed. A literature review table is provided in Section 3.3, listing all the references related to this chapter's topics and sections.

3.1 Background - Social Applications

Social applications are browser based tools with a communication and relationship management component. Social applications have become more and more utilized for personal, entertainment, and professional use cases. Examples of social Web apps are sites like Facebook, YouTube [23], Flickr [1], blogs, and even email clients, all of which allow the user to publish content or links to content for their subscribers or the public to see. Utility applications like DropBox [2], Zoho Docs [3], and Google Drive [4] are used to store, share, and collaborate on documents and media files for SN sites.

[1] http://flickr.com
[2] http://dropbox.com
[3] http://zoho.com
[4] http://drive.google.com

More recent social apps like the video collaboration site HitRecord [5] or the unique bookmark management and sharing app PearlTrees [6] have emerged providing unique experiences for people to manage their Web resources, share them, and collaborate.

Social applications fall under the Web 2.0 branded websites. [58] highlights the importance and requirements of Web 2.0 apps: use it as a platform for developing software in; support lightweight and loose coupling in the programming models (i.e., REST); outsource and mash up content via contribution and collaboration; provide ownership and control for published content. These points fall very much in line with the thesis motivations in Section 2.1.

Table 3.1 shows what each social application can host, and what type of media access and authentication schema each supports. For the most part privately shared media files requires a membership with the host, the exception being for Email which requires a service account. Diaspora [8] promises to support content import export across other Diaspora pods (hosts).

Data aggregation applications have emerged on the market to narrow the gap between different SN sites. Aggregation solves some of the user's burden of publishing content in multiple SN sites by having one central management location; however, it does not provide the user with complete control and ownership of the resources published. Facebook [43] has the right of data ownership until the account is deleted, but if content was shared by others, the content still exists even after the account is deleted. Social sites like YouTube grant the publisher with more options to control their content; for instance, if the content (video) is deleted, everything associated with it and linked to it is removed and the video becomes inaccessible [24].

3.1.1 Social Networking Status Quo

This section focusses on the background of how Web resources (or files) are stored and published and the issues with the status quo. Social applications have certain limitations like how content is published and shared, specifically on SN sites. SN sites have a tendency of becoming silos of published data, which degrades the value of the WWW by segregating data. Only members of a SN site have access to the

[5] http://hitrecord.org
[6] http://pearltrees.com

data published within it, forcing people to join that SN site. The W3C Workshop Report [72] discusses the issues and the future of SN, it generalizes all of the position papers submitted for the workshop into the following topics:

- Decentralized Architecture for Social Networks and Data Portability
- Business Considerations
- Privacy and Trust
- Context Sensitivity
- Adapted User Experiences

The W3C Workshop provided the basis of what needs to be done. This thesis covers a little on each topic, except for Context Sensitivity. DSN sites have yet to gain momentum partly because there needs to be a migration path for it to first work along side the popular SN sites. A user that is a member of multiple SN sites is asked to perform a lot of redundant and tedious activities, such as maintaining their profile and published content in multiple locations. A DSN architecture can dissolve this issue, aggregating all the SN sites into one location and interoperating with others regardless of the SN they belong to.

SN sites could "open" themselves by implementing open protocols. This may make the case for achieving greater usage numbers as more potential users would gain trust, sign up and be retained if SN sites were to do that. This would transform existing SN sites into an open Social Web, which is more in line with the principles of DSN sites and of the WWW. OpenSocial [56] a standard that allowed SN sites to take the initiative to interoperate amongst each other did attempt just that, but without the key players taking part in it, it was not possible to make the ecosystem work.

The privacy risks and lack of ownership are overlooked or do not yet make it worth the effort to join yet another social application. Decentralization is something very foreign to general users, and only attracts a very minute number of them comparatively speaking. A privacy scenario that has not been studied much is when a user knowingly or unknowingly publishes sensitive or damaging content about another

user. Published content can easily be re-published by others, or even "go viral", making it difficult to control. The valuable data that the popular SN sites have will continue to be locked down within themselves, it is up to the users to initiate and start publishing their content in a more open schema. Privacy risks such as a damaging media file going viral or showing up in the wrong stream of data needs to be addressed so that it can be avoided or quickly reacted upon.

The last W3C Workshop topic is the cross platform (mobile, desktop, tv, large billboards) support. The Workshop includes papers on better accessibility to include all types of users, which could be covered in a future case. It is important that the concrete solutions of the topics brought up in the workshop can be used across platforms.

3.1.2 Previous Works

There has been many previous studies that have attempted to resolve some of the issues highlighted by the W3C Workshop. Studies such as [77, 17, 56] look at interoperability of the SNS communication strategies via different protocols and frameworks, while PURIC looks after the interoperability for the media file sharing using existing means. The other studies focus on sharing private data, while avoiding redundancy. PURIC does the same except the data is not relationship data but PURIC looks at public media files for massive teach, and control over their many versions.

[59] offers a survey on different approaches for decentralizing SN sites. It discusses the general SN site functionalities that can be broken down into social relationships (networking) and user content (data) functions. PURIC focusses on components for the data functions. According to [59] DSN sites can be developed as a Web-based or Peer-to-peer system. It also points out that cloud storage integration (i.e., DropBox, Box.com) for DSN sites may not be ideal since they are paid services after a certain amount of data is stored in them; however the cloud storage service provides access to the content whether the publisher is online or not and the publisher has control over the files.

The [36] work mentions the synchronization of messages (cached in the proxies) but it does not consider larger data types like media files. They have a strong case

for extending the browser and creating a Browser-Server, or Browserver solution, that can act as a Proxy server offering services accessed by a Gateway and other Browservers. The Browserver advocates the usage of public FOAF and micro formats for data to be publicly accessible and search engine friendly. This can be used to find new users and their published data. The Gateway server can be accessed when the user is offline. Other Browservers can be accessed directly from a search engine search result or through a local directory which adds the perception of a P2P system. This study offers an off the shelf solution that could require the addition of a plugin or way to extend a browser to offer services, this may complicate things as we move from a desktop browser to a mobile device browser or smartTV browser in which a user could be publishing their content from. This solution also does not discuss media files sharing in which case it could take some points from PURIC or even integrate a similar solution for sharing media links via the brow server or gateway.

Similar to the [36] study, PURIC is more about a shift towards a new way of controlling published media on social applications as oppose to implementing a new technology. It does not require a costly and complicated infrastructure as it is a simple gateway. It is easy to integrate as all that is needed is the URL generated. Some social activities should be public as it can be of benefit to the general public but they should also remain controlled.

A much discussed issue with SN and DSN sites is privacy which can be sectioned off to controlled and uncontrolled data. Controlled data is defined as all data that is published by the user, which the user may edit or remove. Uncontrolled data is defined as all data that is republished by friends of the publisher, which the publisher has no control over. While most studies like [26] focus on privacy in general, which is authentication and authorization to content, PURIC focusses on acquiring control over both published (controlled) and re-published (uncontrolled) data so that privacy issues can be effectively dealt with. If something was published that should not have been published because it could cause some privacy issues PURIC can effectively deal with it. P2P works like LotusNet [26] may be more ideal for private DSN sites. PURIC allows users to publish public media with complete control over wherever service is used and taking advantage of the different network of friends for greater

exposure and feedback. This allows the Web to grow. P2P solutions are still walled gardens in a sense, just smaller gardens more spread apart.

The Locker Project, previously known as Lockr [69, 64], is an open source system that provides social privacy and SN Application Programming Interface (API) aggregation layers for centralized and decentralized SN services. It focusses on the privacy of user content across different SN sites. Similar to PURIC it allows the user to use any SN, it adds on content authorization support but it does not support other features. Locker Project has a different use case and targets those that are interested in managing private content through a hosted proxy or a P2Pmeans. Nevertheless, it is an intriguing take on adding a content management layer to existing SN services. Its design approaches based on [69] require some workarounds and custom schemes to interoperate with different SN services like Facebook, Flickr, and BitTorrent. The Locker Project has similar principles as PURIC; it allows users to choose content storage locations, but it still encourages a walled garden approach. Something that makes it difficult for growing the public internet, data discovery, and relationship discovery. The Locker Project could easily integrate PURIC features into its own set of features as PURIC's design is very pluggable. On the other hand, PURIC could add an ACL layer to the Proxy similar to how it's implemented in The Locker Project.

A recent number of works have been published on DSN but most have been a P2P infrastructure [17, 32, 74, 63, 31, 67, 27]. P2P seems like a logical solution but the main issue with them is content accessibility due to the unreliability of the network connected user-based hosts. Unless a caching server is used which makes the system more of a hybrid of P2P and server-client infrastructure, like the Browserver system. Also P2P systems are typically not as user friendly as centralized or hosted solutions, which is a big part in how Web 2.0 apps attract and retain all types of users and not just technically inclined users. P2P system may require a software to be installed by the user or the management of paired keys for security purposes, which degrades the user experience. The replication of media files that occurs in P2P systems is redundant and not as efficient and effective.

Existing solutions have their own set of use cases:

- Professional media publishers wanting control over their own commercial or entertainment content.
- Advanced, experts, managers of social media users that want more control.
- Advocates for the sustainable growth of the WWW.

What all the proposed solutions are missing out on except for Diaspora, is the ease of use by the developer, which is just as important as it is what will allow more startup companies to get their product and services out there. The easier and more abundant the technologies used in the solution, the faster and more implementations there will be. Whatever the set of technologies used to develop the idealistic SN system is, it must be chosen so that there is a low entry process by the developers. There is no killer social app of the future, whatever the new phase of SNS is it will be a slow transition to get there, as there are masses of users that would need to be "converted". Integration or interoperability of current SNS is what needs to happen as the first stepping stone.

3.2 System Design & Infrastructure Improvements

3.2.1 Modularity

The communication between documents occurs via a client and server architecture and the Hypertext Transfer Protocol (HTTP). It is via HTTP request methods and responses that a client like a Web browser can request resources from a server and the server provides the proper response back to the client. A Web content, or resource, is mapped to a unique URI or URL, which locates the resource on the network and returns it in an HTTP response. A Web resource has its own URI used to be located in the WWW. HTTP is the underlying communication technology supporting a client-server interaction where the client sends a request to be processed by the server who then returns the response to the client. HTTP will be used as the protocol for the system at hand as it will be a Web-based system. HTTP responses can be in different formats: HTML, JSON, XML, and streamed bytes. The server's response includes a header with all the meta-data necessary for the client to properly

parse or process the response type.

A communication layer is required between the client and server system components. There are namely two potential standards or design patterns that could be used for this communication layer, one being Simple Object Access Protocol (SOAP) and the other being REpresentational State Transfer (REST). Both of these methods could be used for setting the system protocol guidelines. These are two different but popular styles of providing Web services and exchanging structured data [61, 60].

SOAP

If a client and server are communicating with each other via SOAP [58], there exists a tighter coupling where the interaction and components are more dependent on each other and information shared between each application is on a need to know basis. SOAP is ideal for corporate Web services as it needs to be more secure and specific to the job, a SOAP Web service maps to custom use cases, or procedures, where the request may perform multiple actions on multiple resources in the backend that is blackboxed from the service consumer. Much of the SOAP process is abstracted in the Web service URL, which is ideal for enterprise and proprietary solutions, but not so much an ideal solution for interactions among social Web applications.

SOAP entails the invocation of Remote Procedure Calls (RPC's) that provide well formulated and specific responses. There is typically a specific procedure call for every business logic action the client requires from the server, if an action does not exist it must either be created or aggregated from multiple RPC's in the server, making it complex to change or extend services since the response is formed on the server side.

The realm of social application requires Web services that will suit a variety of needs, in order to do so it is best to provide basic actions to the rudimentary blocks of data, or Web resources, as oppose to supplying SOAP Web services for each action that the application wants to support. It also requires a more lightweight mean of interaction between components. At this point it is clear that although it is possible to use SOAP Web services for social applications, a different protocol may be better suited.

REST

REST is an architectural style used as a means to integrate software by abstracting away concerns and following certain constraints and standards. The REST architectural style and HTTP guidelines [40, 65] may be used to build a Web system much like it has helped build and shape the WWW. It provides the service granularity for accessing a *Web resource* of an application and manipulating it based on the requirements. The RESTful way of communicating between client and server is more transparent and fine grained. The client is aware of the exact resource or group of resources that is being manipulated and what action will be performed on it or the group. The intentions are clear and fine grained which makes the system as a whole more robust and modular.

The convention used in a RESTful interaction between client and server follows the standards posed by the HTTP request methods and status response structure. REST has become a defacto for designing an independent communication component between Web systems. REST is an ideal design pattern for providing a scalable, modular, and flexible platform to develop with for the social media domain, where a lot of times use cases emerge that had not been thought of yet. Public and social applications created with Web frameworks like [20, 9, 18, 6] have embraced REST. Social applications and services like Facebook, Twitter[70], OAuth, and others also make use of a RESTful API for interoperability.

Fielding's PhD dissertation [44] in 2000 brought about a change, in how distributed Web systems should be designed based with proper use of HTTP and REST to simplify the machine-to-machine and machine-to-human communication. REST allows system components such as server applications to scale naturally within and beyond itself. The RESTful approach simplifies the machine-to-machine interface from the ground up making it ideal for scalable and distributed (or decentralized) systems; whereas a traditional approach prioritizes the machine-to-human interface which later suffers from scalability issues.

RESTful services are used to design systems with accessible resources that can be created, read, updated, and deleted (CRUD). The WWW [28, 61, 45, 44] itself has proven to be the most successful and largest RESTful software due to its ability

to scale, discover linked data, and robustness and efficiency via caching mechanisms. RESTful APIs support UIs that are separated from the service provider, where developers can create custom data or resource displays while using the core functionalities and data of a system.

The benefit of using an API is that clients are developed as thin cross-platform applications. This lowers the barrier for many client application developers to get involved. Supporting the standard Web standard technologies (HTML, Javascript, and CSS) increases the chance of attracting more developers to create client applications. SN APIs provide the opportunity for innovative third party applications, which in return extend the services and value of the API provider.

A RESTful approach can help simplify the inter-connectivity between proprietary SN APIs, while also supporting the integration of new functionalities and heterogeneous client platforms (i.e., smartphones, PC's, TV's). A RESTful API can successfully scale an online system and succeed in promoting the fundamental principles of the open Web standards for social networks.

Providing an API for the internal and external consumption of services can improve the scalability and modularity of the system. The functionalities and components may be changed or added at a granular level without affecting other components or Web resources of the system.

Web Applications contain many Web resources like a photo, document, or other files. Web resources can be stored and served by a variety of hosts. Most content published on SN sites and other social applications are meant to be public but once published inside a SN site or targeted directly to certain recipients it causes a silos for data. Making it impossible for that content to be accessed by the general public. Although there are use cases where data needs to be kept private, the World Wide Web (WWW) was created to share documents and keep the World informed and connected regardless of who, where, and when. Hence, there is a need to make data more accessible and completely controlled by the publisher, similar to storing revisions of documents and sharing certain revisions with others as they get updated. There should be a more efficient way of doing this with Web resources but there are no existing solutions.

Systems built with HTTP and REST support the requirements for implementing current trends and technologies such as cloud computing, horizontal scalability, and popular social application APIs. The technologies used by the client and server has many different capabilities and it will continue to add more capabilities in the future; thus it is important that the communication approach is scalable and flexible.

A social application built on HTTP and a RESTful foundation can support the ability for content publishers to control the data they published with resource-based proxy capabilities. It is a suitable Web-based architectural style for a social application to scale well and succeed in promoting the fundamental principles of an open Web.

HTTP has a large specification that has been kept general to serve many purposes. This has not stopped the need by some systems to extend HTTP for their own practical purposes. Several works have looked at extending HTTP, adding Header fields, and extending REST, or even making REST more like SOAP Web services by using URIs as verbs rather than identifiers [62, 50, 57]. These works have proven to over-complicate the standards for implementation, let alone the re-implementation of so many existing applications.

HTTP extensions is one route that may be ideal within a private system scenario to make it run more efficiently, but they may be challenging for a public system as extensions are not well known even if added to the standards. It can be argued that an HTTP standard becomes usable only when it gains in popularity, as specific purpose standards can be deprecated, misused, or not used.

A more general route is to make use of popular HTTP and REST standards and adding an application layer over the system in place. This may not be as efficient but may prove to be more practical in order for existing RESTful applications to start adjusting to the needs of the internet usage, while not complicating HTTP implementation and forcing other application integrations to comply. The general REST communication process involves: HTTP method requests sent from the client application, processed by the server application (or resource owner) that returns an HTTP response to the client. Web systems like PURIC require the scalability and agility that REST can provide.

The REST architectural style has a more suitable and flexible approach for implementing Web services, which utilizes the set of standard HTTP methods [71] to be invoked by a server as an action to perform on an existing resource or to create a new resource. Table 3.2 outlines the different HTTP methods. These HTTP methods map to CRUD operations for different Web resources. The protocol makes use of the URI of the Web resource(s) being manipulated; HTTP request headers for passing various meta-data including the resource's caching information; and optionally the body content for creating or updating resources on the server.

3.2.2 Scalability

Social applications, specifically SN sites, are very unpredictable as topics can become popular at a minutes notice and content grows exponentially as more users join the network. In order to support such varying loads the social applications and their supporting applications need to scale horizontally. The scalability of social application can be easily outsourced to a Cloud Computing Service (CCS). CCS can be broken down into different layers of abstraction based on the services provided. Three layers are listed below:

- Infrastructure as a Service (IaaS) - The hardware is management is offered as a service [3, 19, 16]
- Platform as a Service (PaaS) - The OS and development platform is offered as a service [47, 14, 4]
- Software as a Service (SaaS) - The software and everything underneath it is offered as a service [21, 13, 25]. This can also entail the Data as a Service [10, 5].

Data Cloud Storage Services have made their way into mainstream usage. DropBox and Box.net became popular as a tool to backup and share data in 2010. ZippyShare and MediaFire are two more recent cloud storage websites that allow users to sign up for free and share their files with their friends by sending them a webpage link. It is becoming evident that such a need is unavoidable.

Storing sensitive data and applying a secure model for fine grained access control is a complex challenge as it involves encryption techniques that need to be efficient enough to support high loads. This topic, as discussed in [52, 73, 76], is left outside the scope of this thesis.

Data storage cloud services have their own proprietary method of granting data access, where the publisher shares a private link with any user or shares the resource with other users with accounts in the same system. This method is simpler and well established in Web applications, but the downside is that the link is still publicly accessed if published by others, or in the later case, users need to have accounts with the system. This thesis work assumes that the user will use whichever CCS for storing and sharing their resources and that the CCS is "good enough" for social apps, and that the content is not critical data or sensitive information as the main target application here is for general use social networking sites.

3.2.3 Load Testing a PaaS

PaaS has low maintenance from an IT's perspective and it is well suited for social applications since it has load balancers that automatically deploy new server instances as needed. Load testing PaaS systems can be tricky as it will adapt to the request handling and processing loads to keep the system balanced. This makes it difficult to measure the point at which it crashes but the tests can give back some projected costs for hosting the app and possibly see some other interesting patterns.

Several solutions were looked at for running the load test operations, as the load test server. There are commercial and Web-based load testing services, where the user creates an account and interacts with the tool in a Web interface. Alternative services were looked at since they had fees for different usage and needs. A GAE python app [48] as the load tester, but it was limiting in terms of the results produced. Stand-alone test applications are a great cost-effective method of performing load tests but not all of them do what they are set out to do, such as true concurrency. The Tsung load tester stands out from the rest as it is free to install and run on most platforms. Tsung can capture the data needed to create a number of charts. Tsung is discussed in more detail in Chapter 5, where it is used for testing the PURIC

System.

3.3 Literature Summary

This chapter looked at the issues with SN, the technologies and applications that have attempted to solve some of the issues, and the supporting technologies used for the PURIC system. The previous works mentioned have a lack of public control of published social content. The technologies highlighted in this chapter supports the goal requirements discussed in Chapter 2. HTTP and REST principles can help with its design constraints for implementing a solution that is compatible with existing social applications and scales as well as the internet has.

Many open source projects have sprouted like Diaspora and Status.net, which use existing protocols and technologies in order to fast track the movement of data control and ownership. But the continuous popularity growth of existing SN sites has proven to be very challenging for new competing DSN sites that solve some of the issues discussed. This calls for a new approach that is less forceful on users and SN sites, one that can benefit both and set a path toward solving the existing issues. The next two chapters will focus on the design details and implementation of the PURIC.

Table 3.3 lists the literature used in this thesis and what concepts each paper supports. This literature review provides the foundation and guidelines for designing and implementing a basic proxy. Its design considers user data ownership, privacy control, and scalability in the foundation.

App	Description	Media Access Supported	Auth Schema
DropBox	Hosts small and large files.	Private to members	Proprietary
Facebook	Hosts images and videos.	Private to members	Proprietary (Facebook Connect)
Twitter	Uses 3rd party hosting services and shares the link.	Public	OAuth
Diaspora	Hosts image files, open source.	Private to members	None
YouTube	Hosts the published videos.	Public, or private to members	GoogleID (proprietary)
Email System	Sends files to service provider.	Private to other email addresses.	Proprietary
Blog Site	Hosts any media file.	Meant to be publicly accessed.	Proprietary
Google App Engine Apps	Hosts any media file, size dependant on cost.	Supports public, private.	GoogleID and OAuth

Table 3.1: Social Application Hosts

HTTP Method	Mapped Action
GET	Retrieve a set of resources or a single resource by specifying its ID
HEAD	Request resource's meta data without receiving the content of the resource.
POST	Create a single record or all records defined in the body, or/and replace single specified record or a set of matching records
PUT	Update a single specified record or all matching records
DELETE	Remove a resource or all resources.

Table 3.2: HTTP methods.

Section	Work	Support
3.1.1	[72]	SN status quo and paves the roadmap, challenges, and approaches for the future of SN sites.
3.1	[36]	SN, DSN - PURIC can potentially integrate to the solution of the [36] study for the storage and publication of larger media files.
3.1	[26]	A more ideal P2P DSN solution for sharing sensitive content within a private systems supporting social ACL and privacy.
3.1	[69]	DSN, P2P, privacy - Provides a solution for managing private content stored locally for a P2P-like case or hosted for a centralized case.

Table 3.3: Key Studies

Section	Paper	Support
3.1	[58]	Web 2.0 content participation
3.1	[23, 12]	Social apps, part of the Web 2.0 movement.
3.1.1	[54]	Study on the topology of SN site (Twitter), importance of Retweets, and usage statistics.
3.1.1	[72, 56, 77]	SN status quo and paves the roadmap, challenges, and approaches for the future of SN sites.
3.1.1	[43, 24]	Different terms of service for publishing media content.
3.1.1	[59]	walled garden, where user's activities and data are trapped, and how DSN fits in.
3.1.2	[8]	Most popular DSN project.
3.1.2	[74, 17, 69, 26, 32, 63, 31, 67]	P2P DSN projects and privacy.
3.2.1	[65, 40, 44]	HTTP Status Quo.
3.2.1	[61, 60]	REST and SOAP.
3.2.1	[45, 28]	RESTful Design patterns.
3.2.1	[49]	RESTful application.
3.2.1	[65, 71]	HTTP Reference.
3.2.1	[11, 70]	References for HTTP-based and RESTful APIs from SNS.
3.2.1	[35, 1]	Discussed data knowledge, manipulation, control, and aggregation. P2P. But none of these works were related to extending social URI.
3.2.1	[45, 44, 28, 62]	Definition and properties to be made use of.
3.2.1	[50, 57]	HTTP Extension.
3.2.1	[20, 9, 18, 6]	Web RESTful Frameworks.
3.2.2	[3, 19, 16]	IaaS providers.
3.2.2	[47, 14, 4]	PaaS providers.
3.2.2	[21, 13, 25, 10, 5]	SaaS providers.
3.2.2	[52, 76, 73]	Study for storing sensitive data in the cloud using efficient cryptography methods. Out of scope.
3.2.3	[48]	Load testing tips.

Table 3.4: Literature Review Papers.

Chapter 4

Thesis Solution - PURIC System Design

Social application users need to have more control over what they publish and share with others whether it is public or private content. This thesis explores the feasibility of implementing a management system for published resource links in SN sites like Facebook and Twitter, and social applications like email clients and Project Management Systems. The complete control over the origin of a resource and its aliases can provide valued features that would not have been possible otherwise, such as a higher level of version control and responsive Web.

A typical published URL is fragile, PURIC can add resilience and longevity to it, allowing the content's origin (Web resource) to change while not breaking the published link. PURIC provides control over which version of the content to return, based on the requesting client's platform. The most recent and proper content version can be returned to the user, as oppose to returning 404 pages, large and improper media files, or old content only for the user to have to re-fetch the latest content after the fact. PURIC adds a presentation layer, grants more ownership and control over a media file shared while also making the content more responsive. This chapter describes the details of the PURIC system design and the value it brings.

4.1 The PURIC System

PURIC takes a different approach where it is up to the users to add additional layer to publish and share media content among social applications. It is a difficult task for existing SN sites to come up with a solution where the public content that is shared within the network is accessible outside of its walled garden [68, 55], while there is also the private content that needs to be kept inside the walled garden. There is a tradeoff between data discovery and SNS interoperability versus privacy. For

instance, a user profile needs to be public for the systems that may come into play (i.e., search engines) in order to lure more people in. The more private its content is the more redundancy and inefficiency there will be. PURIC follows the approach and use cases that have resulted in the growth of the WWW, keeping its resources public and using URI aliases to redirect requests for content.

PURIC is not intended to solve privacy issues or the notion of "Big Brother" watching over all the social content, but rather allow more flexibility for the user to store their data anywhere they wish as a Web resource and have complete control over it after publishing it. It focuses on expanding an open public internet infrastructure, similar to PearlTrees or Instagram [15] where the links and media posted on these social applications are open to the public by default, but could also be private. PURIC encourages a sustainable growth of the WWW by keeping most content accessible to the public, while avoiding data replication as in P2P systems, and supporting an array of media storage and publication services such as YouTube, DropBox, and Flickr. The discoverability of PURIC content is an important design element, supporting a sustainable growth of the Web because the content is public and are linked to from a public source. Data storage services can be used in a private manner by using their own authentication and authorization mechanisms already in place, so the PURIC system does not need to handle this separately.

The PURIC system allows a user to own and control the content published on social applications overcoming two of the thesis goals described in Section 2.5. It also allows a Web resource publisher to create a URI alias or public channel, in which the publisher can show any content intended at any given time. The URI channel is flexible in terms of allowing changes to be made on the Web resource being served, or on the Web resource's origin URI. The system is also designed to manage different versions of the media file for different display sizes. As an example, multiple versions of a media file (Web resources) can be uploaded to a third party host service, such as DropBox, for the public to access in a manner that is completely controlled by the publisher.

A traditional method of publishing and consuming a media file (Web resource) in social apps is to upload it into a third party data storage service, post the link

in a social application, consume the media file by requesting (clicking on) that link, which displays the media. It is more effective and efficient to have the URL of a media resource (or file), and publish the link with a text summary than to upload and attach the media resource to every post or message in every social application. This process also allows for others to easily re-publish or share the same content link on their own networks and mediums.

PURIC is designed in such a way that avoids the unnecessary computing and network traffic loads that occurs in the traditional method, while granting more control over the media resources published on social applications. Figure 4.1 shows a comparison of the layers involved in publishing a Web resource in the traditional method versus the PURIC method. Note that the PURIC Architecture has an additional layer, necessary for the added control and ownership features offered to the publisher.

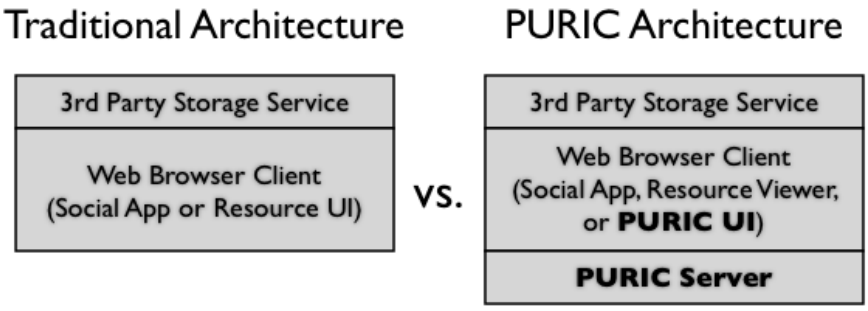

Figure 4.1: A comparison of the social resource publishing architectures.

Figure 4.2 shows the process of publishing with the PURIC system, and how each component is connected to each other. The PURIC Publisher Client is the interface for the user to create a URI alias and map a URI to an endpoint URI. The 3rd Party Storage Service (i.e., DropBox) is the place where the publisher uploaded the Web resource to. The PURIC Server handles all of the publisher client requests. The Social App is where the published content ends up, it represents email client, Twitter or other SN sites where links to media content can be included in messages or postings. The first step of the publish process is to upload the content to a 3rd Party Storage Service and retrieving its public URI. The second step in the process

is to map and store the public URI with a URI alias. The third step is to publish the URI alias to a social application. Note that uploading to a 3rd Party Storage Service and publishing to a Social App can be done via their respective client interfaces, or if they have an API via the PURIC Publisher Client and Server.

The following list walks through the scenario of publishing media content for a PURIC (see also Figure 4.2):

1. Upload media file to a third party data storage app.
2. Using the PURIC system, create a URL which points to the root URI of where the media was uploaded.
3. Repeat above two steps if there are different versions of the Web resource that will be shared to different social networks or for different screen size resolutions (i.e., mobile phone, desktop screen, TV, large screens).
4. Publish the appropriate URL (root URI alias) to the social application(s). Subscribed users or recipients of the published media URL will follow the link and the most appropriate (for screen resolution, or if a version was specified in the URL) will be shown.

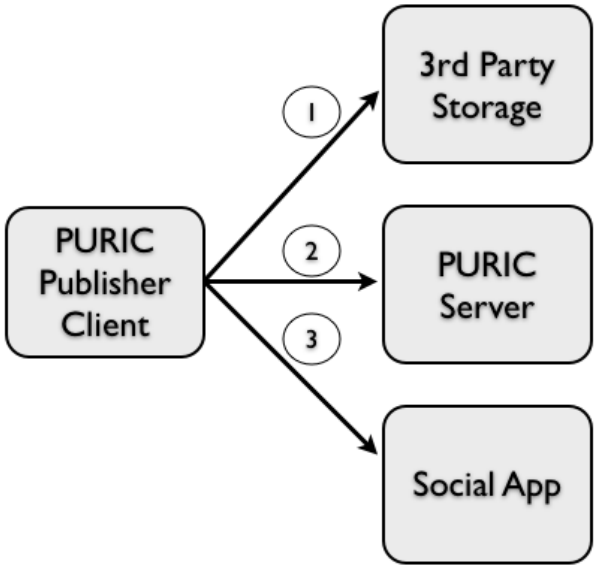

Figure 4.2: PURIC Publish Process.

Figure 4.3 shows the Web resource consumption process and its components.

The first step of the consumer process starts with (1) the consumer requesting and displaying published content in the Social Application; (2) when the user clicks on a post's link, or a PURIC link, the PURIC Media Viewer (HTML content) is returned along with a JSON containing a list of the actual URI's representing the location for each version of the content); (3) the remaining step involves processing the logic in the client to select and request the proper content version (i.e. small resolution image) from the 3rd Party Storage Service, and displaying it in the Media Viewer.

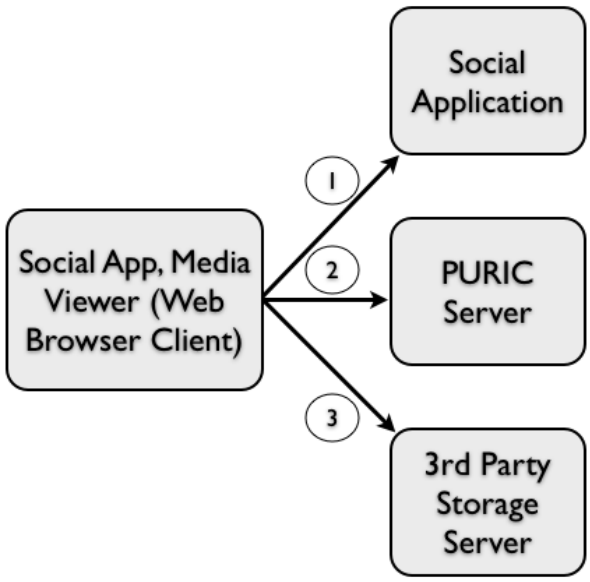

Figure 4.3: PURIC Consumption Process.

4.1.1 System Design

The general architecture of PURIC is illustrated by the components and process of Figures 4.2 and 4.3. The decrease in unnecessary computing and network loads can be resolved by providing extra data and doing the processing logic in the client as oppose to the server. Publishers can manage the version and origin of the published content by utilizing a Uniform Resource Locator (URL) pointing to the PURIC server which maps to one or many Uniform Resource Indentifier(s) (URI) where the actual media file is hosted. A PURIC URL is manually shared on social application streams and threads, while its mapping to the media content URI(s) is controlled by the publisher.

The PURIC system is created to give the publisher control over which URI the client will fetch from a third party storage service. Actions like posting, reposting, or forwarding a PURIC URL are not affected when the URI alias mapping is changed, it is resolved in the backend by the PURIC server. The server determines which URI(s) to send back to the client based on the most recent and active URI(s) and the client's platform display size value. This process avoids fetching dead links or over-sized content, especially significant for mobile client platforms. Web savvy end-users can reverse engineer the origin URI of the media file, but this is not a major concern as the aim of this work is to add resilience and longevity to published URLs.

It is important to consider what each system component supports to find the common ground for a reliable, scalable, and secure method of communication between all entities and future entities. Figure 4.4 illustrates the PURIC process flow chart.

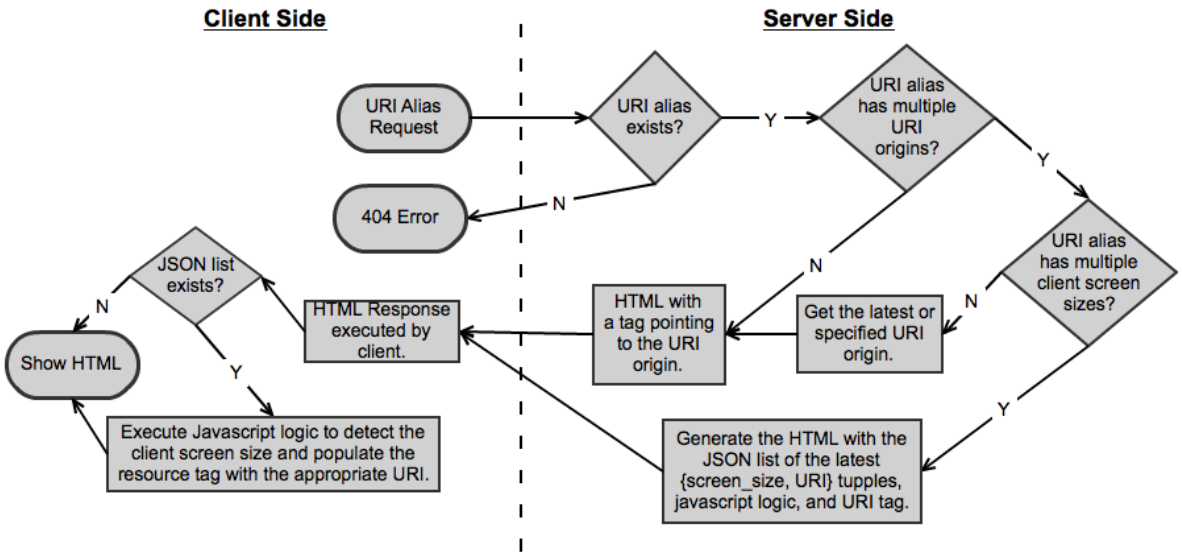

Figure 4.4: Proxy Flow Chart.

There is no authorization aggregator service solution in existence that is simple and popular enough for the content publisher to select a private group of users that would work across client applications. For instance, a link to a video can be separately published to a Facebook group and private Twitter feed, and authorization can be applied to open the link based on where it's published, a third component is required to manage and authenticate the users that have the authorization. Table 3.1

shows the lack of standards across the SN sites, DSN sites, and other social apps. This component would have to be implemented by the PURIC system but it is outside the scope for this thesis. It would require a lot of effort to develop and maintain such a component that supports the creation of groups with users from different social applications. Particularly, since each social application could follow a different type of open or proprietary authentication method.

To make matters more complicated for the above scenario, if the publisher wanted to export the media file to a non-PURIC server then both system would need to support the same data format standard for export and import. This component is not implemented due to the limited scope of this thesis and because SN service providers have not yet agreed upon a unified and ubiquitous solution to manage, control, and import content access across social applications. Thus, the private method of sharing media in a social application is used, and it requires the media consumer to have an account. However, the PURIC system is modular by nature and supports either method, since its design advocates the accessible data principles of the WWW.

4.1.2 User Experience

PURIC considers three topics in regards to user experience, which have become standard for social applications:

- Cross-platform support - the features run in multiple client operating systems and device types (i.e., desktop, mobile, tv, billboards) via standard Web technologies - HTML, Javascript, and CSS.
- Responsive experience - provide the best suited media files and visual layouts based on client side properties.
- Data availability and speed - always available data with low network latency.

Different paths can be taken when making a software that works in all major platforms and devices. The most time consuming option would be to build a version of the application for each platform. Another option would be to use cross-platform development tools such as Titanium, PhoneGap, and standard Web technologies (HTML, Javascript, and CSS). Since PURIC is targeting all platforms the simplest

solution is to choose the browser as the client and the Web as the platform. In order to create the best user experience in the Web, PURIC needs to have some responsive Web components.

Responsive Web is about enriching the user's experience [46], it is a Web design decision that can have some positive implications in terms of proper visual layout and using the proper amount of network bandwidth for a given client platform and specifications. It can also affect the features the user sees based on what the device can handle. Typically the topic of responsive Web focuses on layouts that can dynamically adjust to the client's display properties but in this thesis we take it one step up and include the need for the proper version of the content based on the client's display properties. This also results in more efficient use of CPU, Memory, and network traffic in the server and client.

Metadata supplied by the client request can be used as parameters for querying a client platform registry (either local or external) [66]. Javascript and Web browser technologies are constantly transforming, becoming more efficient, and adding new features as client computer capabilities change over time. One Javascript library called ıscreen contains the properties width and height for the screen's resolution and can be used as a responsive content means.

Existing public and commercial services offer basic device detection and content adaptation solutions. These solutions require a private server installation or are provided as paid cloud services, offering a lot of information about the mobile client mostly used for analytical purposes, and more practical purposes like dynamically adjusting the app's layout or functionality.

Authors in [38, 39, 33] looked at the topic of resource versioning. PURIC supports resource versioning as a solution to improve responsiveness. A published Web resource may change overtime or the resource publisher may want to show a different version based on who or what is accessing it. This requires that the URI of each version of the resource still remains accessible. The version control for such a system does not require version control on the origin resource, but on the origin link to the resource.

Transcoding [51], or converting content formats, to suit the need of a client requesting a resource is not an optimal solution for SN services due to the sheer amount of data exchanged between server and client. A proxy gateway solution is more optimal, where content version or type is selected and served based on the client attributes.

4.2 PURIC and REST

The constraints imposed by the REST architectural style helps create a scalable WWW. Resources are all linked to each other and they have standard HTTP methods (i.e., GET, PUT, POST...) that can be applied to them. The same approach taken by the WWW can be used to reach the scalability goals of a RESTful Proxy. This work does not impose on creating extensions to HTTP or adding custom Header fields since these approaches lead to a solution that does not follow the current standards, which would make clients and third party applications add or handle customized Header fields. Instead it focuses on using the existing infrastructure URL parameters and existing Header fields for solving the problems outlined.

Figure 4.5 shows a more detail view of the consumer process of Figure 4.3; the flow from clicking on a URI alias in a social app posting to displaying the Web resource associated with that URI alias.

HTTP and REST can provide the guidelines for modelling the PURIC system. The server side makes use of the REST architectural constraints and two HTTP methods from Table 3.2. They provide the design pattern for developers to benefit from the horizontal scalability and modular design attributes for changes that the system may require in the future.

Figure 4.6 shows a sequence diagram for the URI alias request from the client for the media file (origin URI), the server analyses and chooses the proper single or set of origin URIs and dynamically generates the HTML wrapper, the client then does any necessary logic to select, request, and show the origin URI media. The processing work is split between the client and server.

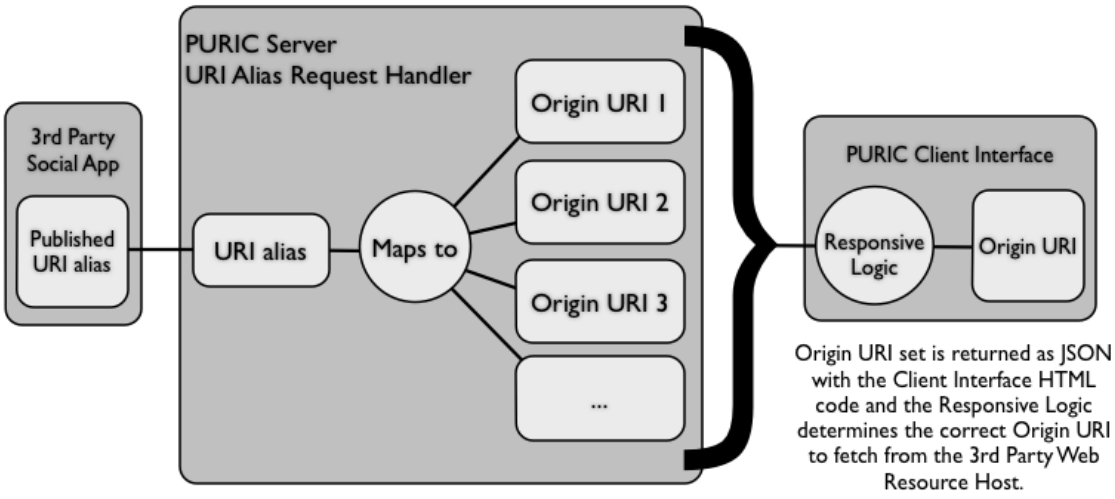

Figure 4.5: URI alias to Origin URI Flow.

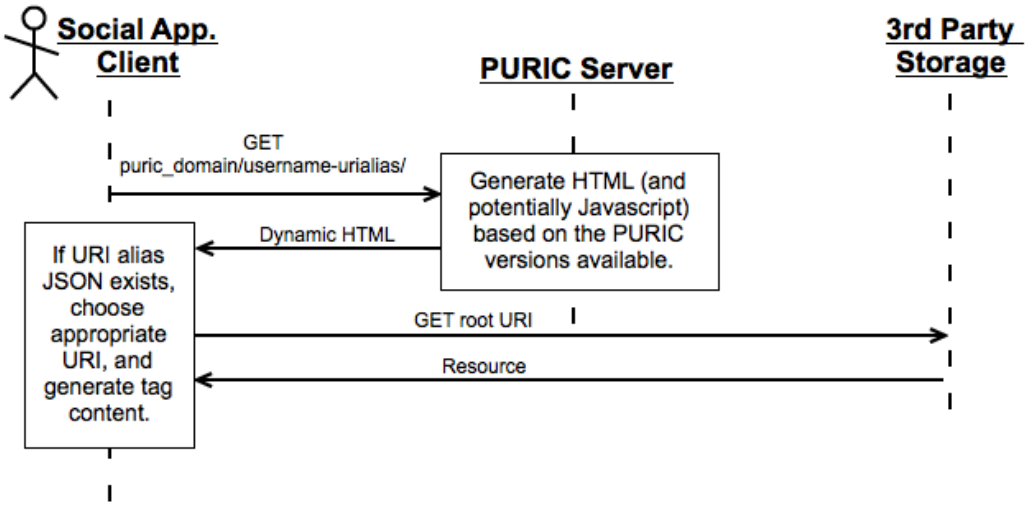

Figure 4.6: GET request for Web resource data.

4.3 The Value of PURIC

The PURIC system can improve the internet traffic by sending properly sized media files to clients. Although the PURIC system is built as a separate standalone system for the purposes of this thesis, it could prove to be of value to existing social applications as an added feature for their users to help them manage their published links. Facebook and Google+ have features for the publisher to specify groups that will see the content published. This type of publishing control feature can be complimented by PURIC, as different versions of the same public media file can be seen by specific groups of subscribers.

Another valuable aspect of PURIC is the ability for content publishers to change the actual URI the PURIC URL is mapped transparently for their audiences. A PURIC is responsive enough to show content based on which medium or platform the end-user is requesting the resource from. It can link to the same media content over time or to a media content that changes over time, in which case the publisher has a new origin URI and a new version of the aliased URI channel link. This case demonstrates how the PURIC system can support ownership and control; allowing the user to store the content wherever they wish. A new version of the origin URI does not have to result in republishing the new link as long as the PURIC is made aware of it, the change is invisible from the end-user's perspective.

Having a single URL for media content published that can map to different versions of itself allows a more effective way to manage and control it. The publisher can remove or replace the actual URI of the published URL, even if it is re-published by others. When necessary the publisher can still state the change in the case of a URL published inside an ebook. Scenario:

User A publishes an inappropriate video of user B. User B contacts publisher, so that appropriate measures may be taken to remove or replace the published content swiftly. If the content was republished by others prior to the removal of it, it can still be acted upon since the URL now offers a different content. User B could also flag the URL as inappropriate, which would trigger a notification to be sent out to the content's publisher and system administrator. By replacing the root URI, which was

wrapped in the page of the URL, all the re-published URLs will now show the new resource, thus granting more control for the users.

As a side effect, a different use case arises with the ability to reuse a long living URI for content that changes regularly such as a daily image or video webpage. You can essentially publish a link that will live for as long as the PURIC Server lives with an ever changing resource origin. Instead of sharing a webpage with changing content, a PURIC can be shared as a single link to a specific media content that changes over time. The long lived URI acts like a channel displaying scheduled media content. The URI can be included in an HTML tag as part of a webpage. In some level we are already doing this through blogs, timelines, streams, RSS, but this is taking it to a lower level where the resource origin itself is shared.

The need for social features in any application accessed by a group of people is on the rise; resulting in more use cases for PURIC to emerge as this happens. The popular ePub[1] and Kindle[2] eBook formats are becoming more and more like apps as eBook readers adopt HTML5 features. There is great value to the eBook product as an author, reader or publisher company to add dynamic content from its readership and it is only a matter of time before something like PURIC is adopted in the digital publishing industry [53]. When a PURIC link is published in an eBook, referencing a Web resource, the root Web resource may still change after the eBook is published without any negative effect, or calls for user action, the change occurs transparently.

4.4 Goals - Overcoming the Challenges

The approaches discussed in this chapter help overcome the challenges associated with the goals discussed in Section 2.5. Table 4.1 summarizes the goals and their respective approaches.

The system design has good scalability with the RESTful pattern focussing on resources as oppose to customized Web services. A PURIC system also has several features and requirements that need to be implemented in order to reach the goals listed in Section 2.5. The next chapter goes over the details for the implementation

[1] http://idpf.org/epub
[2] http://www.amazon.com/gp/feature.html/189-1124120-3806941?ie=UTF8&docId=1000234621

of the PURIC system.

Goal	Solution
Modularity	A REST API foundation.
Data control and ownership	Choice of a resource storage service for the media files and a way to control its publication even after it has been republished by others.
Cross platform support	The use of standard Web technologies for a responsive system that selects the resource requested based on certain client properties exposed on the request parameters and later computed on the client side. In PURIC the client and server split this task. The publisher is given a management interface to edit and publish different versions for the URL of the PURIC. The server returns the latest or specified version of the resource requested.
Scalability	Load balance work is outsourced to Cloud Computing Services for horizontal scalability.
Low Resource Usage	Server side CPU and memory usage for the SN server is lowered since the resource is hosted elsewhere. Client side CPU and memory usage is lowered since the media is not re-scaled in the client. Memory used can be lowered since content is uniquely hosted in one place as oppose to being hosted in many social applications.Network traffic can be lowered when the content pulled by the client has the appropriate small size which is more adequate for small devices.

Table 4.1: Achieved system design goals.

Chapter 5

PURIC System Implementation

One of the challenges from the previous studies mentioned in Chapter 3 was attracting users to either switch over to a an app like a DSN site and stick with it. Such users typically need to be developers or have a keen understanding of social media and software deployment to take part in. Some solutions mentioned also require the installation of standalone applications others require the management of login and profile credentials for all SN services. PURIC takes a simpler approach that is not so much about the lack of technology that impedes its usage, SN users can start using it now and it's a process SN service providers can easily integrate into their systems as a feature. This makes PURIC appealing and differentiates from the previous attempts to solve some of the challenges pointed out in Section 2.5.

The PURIC system implementation includes: a Web browser (client) for displaying the HTML wrapper code with the proper media resource showing; a third party storage service for hosting the media resource; and an administration interface for managing URI aliases and origin URIs. Processing is shared between the Google App Engine (GAE) server and the end-user's Web browser, the clients and servers communicate via RESTful patterns supported by the GAE platform, sending HTML and JSON data back and forth. This chapter will go over the PURIC system implementation on the client and server sides.

5.1 Implementation Choices

There is a myriad of technology options for implementing Web-based systems. The options have benefits and drawbacks, there is no correct design pattern or development platform to choose. Choices must be looked at pragmatically, from the perspective of the system requirements. Sometimes compromises must be made in

order to fulfil the job. The design patterns and development platforms chosen to fulfil the requirements of PURIC without too many compromises for the purpose of this thesis. A production version of this implementation may use different technologies based on the requirements of the development team and the system's usage, but its system design remains as its core.

The system is designed to support a number of technologies for the communication between the system components and the user experience. The technologies explored here are based on the industry's status quo and the low cost and access barriers for a development team; they include: GAE's Python Platform as a Service, HTML5, and Javascript. HTML5 features were chosen as there is an emerging movement to make it a viable option for the development on different browsers and platforms, specifically for supporting multiple mobile platforms.

Two options were explored for implementing the PURIC Server but only one was chosen. Option A, parses and transforms the resource content on the server and returns the HTML, so its origin is completely masked as if it is entirely served by the PURIC server. Option B includes an HTML code wrapper with the origin URI and some logic in Javascript to request the appropriate image, video (for video or audio resources), or iframe tag (for HTML, pdf, and other documents). Option B was chosen because it is a simpler solution, and results in lower network traffic and server side processing loads. The core code written for this implementation, excluding third party libraries, sum up to 988 lines of code.

5.2 PURIC Administration UI

There are three requirements that the PURIC user interface should follow in order to be satisfactory:

- Intuitive and simple interface and vocabulary.
- Low effort to join and leave.
- Easy to integrate with existing social apps.

The publisher's PURIC administration UI requires the user to login, lists all the URI aliases that has been created (Figure 5.1), and for each URI alias there may

be a set of origin URIs or versions of the resource (Figure 5.2). Origin URIs can be added or removed as a different link to a version of the resource.

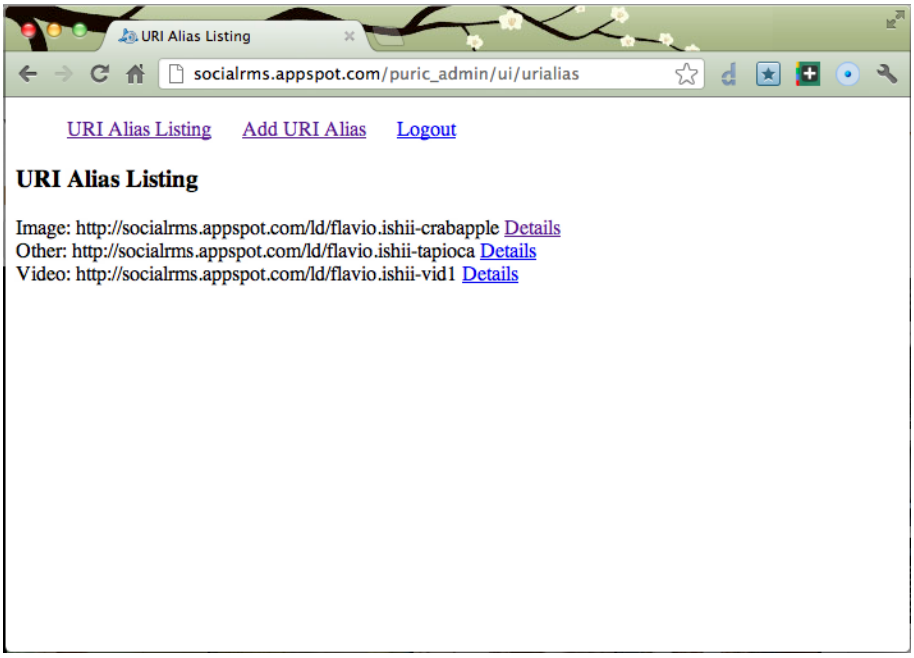

Figure 5.1: Publisher UI showing all of the URI alias the user has created that are ready for publishing.

The social resource consumption UI is simply a webpage with HTML and Javascript that shows the media appropriately chosen. Media file show depends on attributes from the client platform displaying it. Figures 5.3 and 5.4 show how an image retrieved from a PURIC is displayed differently depending on the resolution of a desktop browser's window. Figures 5.5 and 5.6 show how an image retrieved from a PURIC is displayed differently depending on the orientation of the mobile devices. Note that the resolution size is displayed below the image on each screen for test purposes. The image file pulled is the best sized image for the display without having to resize it. In other words, if the screen resolution is 800x600, the 800p sized image is displayed if there is one; if the screen resolution is 1024x768, the 1024p sized image is displayed.

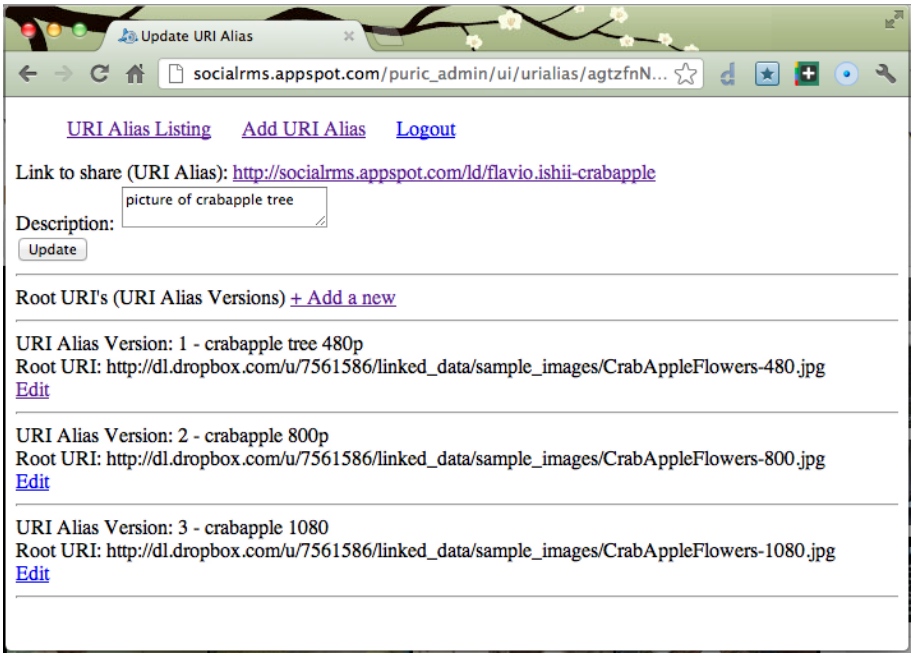

Figure 5.2: Publisher UI showing URI alias details and origin URI list (URI alias versions).

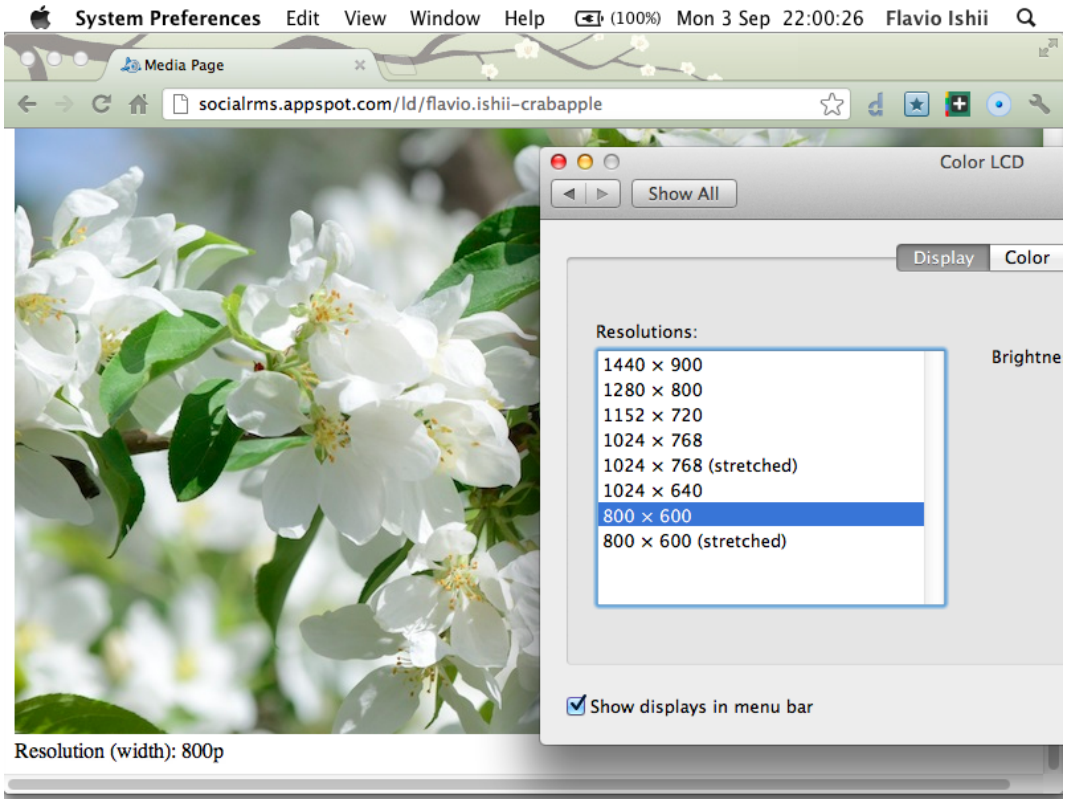

Figure 5.3: Desktop browser at 800p resolution displaying 800p image.

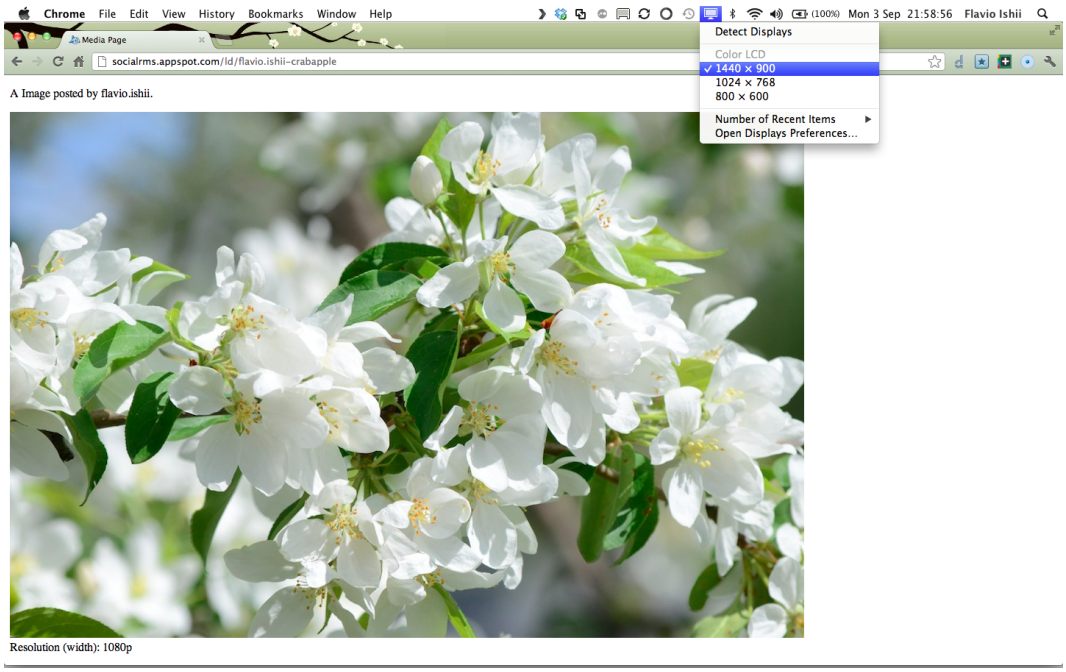

Figure 5.4: Desktop browser at 1440p resolution displaying 1080p image.

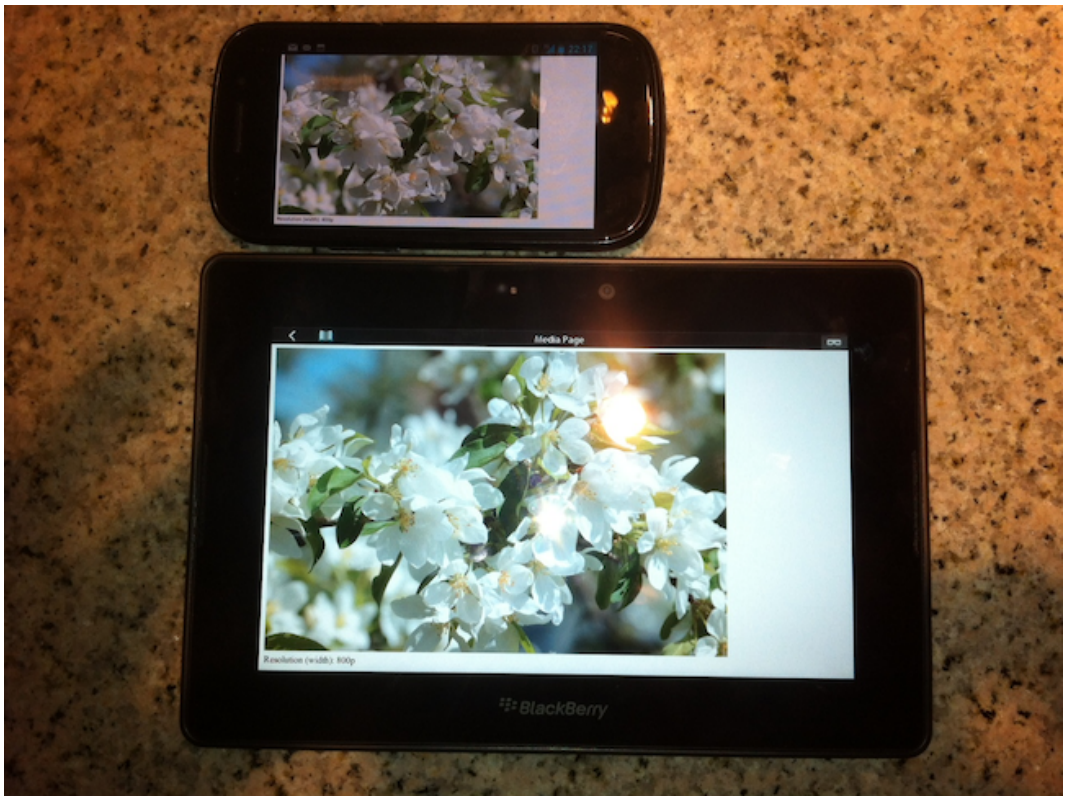

Figure 5.5: Mobile devices in landscape orientation mode displaying larger 800p resolution than in portrait.

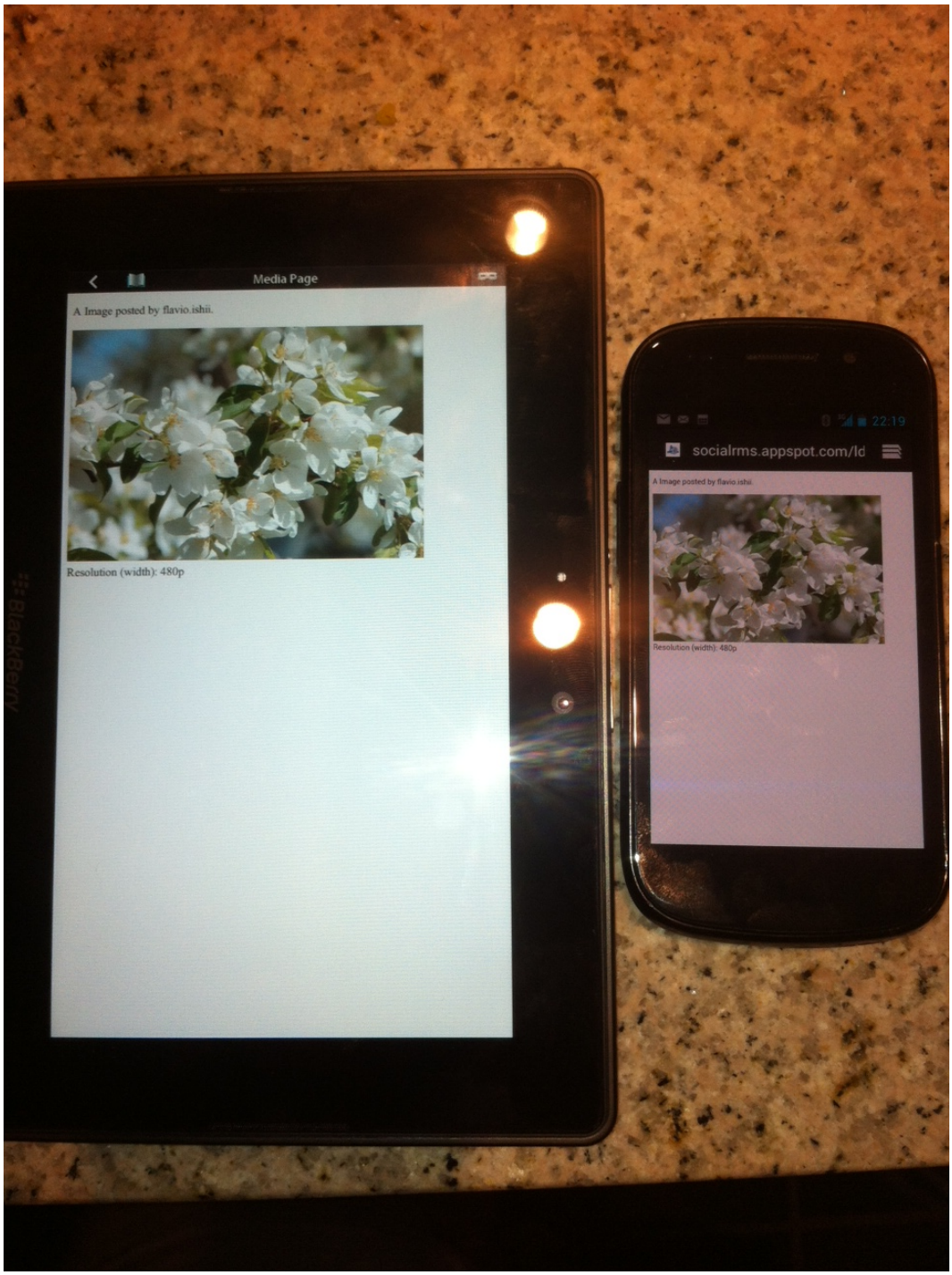

Figure 5.6: Mobile devices in portrait orientation mode displaying smaller 480p resolution than in landscape.

5.3 Client Side

The Web browser (user-agent client) was chosen to run the end-user interface for three system components: the PURIC Admin, Social App, and Media Viewer. Other than a modern Web browser there is no need to install another client software as part of the PURIC system. The client technologies are readily available for all users.

The PURIC system needs to work with different types of hosts services that can serve Web resources such as an image file or dynamic content such as a YouTube video page. This leads to a landing webpage design that is flexible enough to display an array of resources, or media types. The HTML document can be linked to from any social application as a URL. The publisher copies the link provided in PURIC's Administration Interface and pastes it on an email, Twitter posting, Facebook posting and so forth. When a PURIC (URI alias or URL) is published on Twitter and a user clicks on it, the client receives an HTML document that includes a JSON list of the possible origin URIs to show, plus javascript statements that selects the appropriate URI to request and then display it.

For the purposes of this thesis the PURIC Server accepts jpg/png, webm, mp3, html, pdf, and xml content types for the actual URI. The HTML5 Video tag may only support certain encodings and will play .webm videos and .mp3 or .mp4 audio resource. The image tag is used for any image resource. The iframe tag is used for external or 3rd Party dynamic resources like YouTube, Flickr, and Wordpress webpages.

The responsive Web component of the PURIC system relies on the ability to detect the client's screen width (resolution) and request the most appropriately sized resource if it is an image, video, or a dynamically sized document. As mentioned in Chapter 3, there are two possible methods to detect the client's screen size, via a cloud service such as WURFL or via Javascript properties screen.width and screen.height.

The WURFL Cloud service currently supports three client platforms (Java, ASP .NET, and PHP) with plans to support Python and Ruby. However it's no longer a freely available service as of recent times, and since this work does not require all

the features offered by the service the Javascript method is more suitable. In order to keep the implementation simple the screen size detection is accomplished with Javascript. The public interface does allow the overriding of the origin URI if the client supplies a version and screen size on the HTTP GET request. This is useful for testing purposes or in the case where the publisher has a reason to specify the one version of the Web resource to publish.

The client side Javascript can detect the screen width and height resolutions in pixels for the current device. Five different client devices (BlackBerry Curve with OS5, BlackBerry Playbook with OS7, iPhone 4G, iPad, Samsung Nexus S with Android) were used to test the support for the Javascript for detecting the screen width and height resolutions in pixels. The devices with accelerometers have two different resolutions, depending on its orientation (portrait or landscape) as discussed in the previous section. All devices tested but the BlackBerry Curve use an accelerometer to change the orientation. Screen orientation is taken under account for selecting a visual resource with the lowest supported width and height. The system will select the smallest dimension of the screen resolution to match the largest dimension in the visual media file, and serve the matching file.

Listing 5.1 shows the dynamic HTML code used to generate the media consumption webpage. It includes the JSON with a list of possible root URIs for the media and the Javascript logic to determine which root URI to fetch based on the screen resolution, and the dynamically loaded img tag requesting the image source (URI). The client side executes the consumption webpage code synchronously, but if the webpage contained many elements an asynchronous solution could be implemented. The webpage source code is generated by the server and changes based on the media type requested. In this case it loads the HTML for displaying an image file but, as mentioned, the server also handles the case for audio or video files. Some of the code has been left out and replaced with "..." to focus on the points discussed here.

```
<script language="JavaScript">
        var linked_data_list = [
                { "size": 1080, "url": "http://dl.dropbox.com/u/7561586/linked_data
                    /sample_images/CrabAppleFlowers-1080.jpg", },
                { "size": 800, "url": "http://dl.dropbox.com/u/7561586/linked_data/
                    sample_images/CrabAppleFlowers-800.jpg",},
```

```
                { "size": 480, "url": "http://dl.dropbox.com/u/7561586/linked_data/
                    sample_images/CrabAppleFlowers-480.jpg",},
        ];
        function getBestFitURI() {
                var largestWidthResource;
                var screen_width = screen.width;
                var screen_height = screen.height;
                if(navigator.userAgent.match(/(iPhone|iPod|iPad)/i)) {
                        if (orientation == 90 || orientation == -90) {
                                screen_width = screen.height;
                                screen_height = screen.width;
                        }
                }
                for(var i = 0; i < linked_data_list.length; i++) {
                        if(screen_width >= linked_data_list[i].size) {
                                if((largestWidthResource == null) || (
                                    largestWidthResource != null && linked_data_list
                                    [i].size > largestWidthResource.size))
                                {
                                        largestWidthResource = linked_data_list[i];
                                }
                        }
                }
                document.getElementById('media_size').innerHTML='Resolution (width):
                     '+largestWidthResource.size+'p';
                return largestWidthResource;
        }
</script>
...
<div id="resource"></div>
...
<script language="JavaScript">
        var ld = getBestFitURI();

        document.getElementById('resource').innerHTML = '<img src="'+ld.url+'" alt="
            Resource" />';
</script>
```

Listing 5.1: Client side Javascript source wrapper for fetching and displaying the media content.

5.4 Server Side Implementation

The PURIC server is in charge of storing and handling the different types of requests for all the URI alias links. The server groups a set of Web resources which in some sense is virtually the same resource. The latest version of the virtual resource is selected and sent to the client based on the client's screen resolutions. The difference being slight modifications made to them (versions) for possibly different display resolutions or content personalization.

Cloud computing services and adoption in standard Web technologies, specifically in the mobile devices sector, have grown significantly as more people are using them. Novel systems can now be built where it would have been much more complex and limited for a software developer to build in previous times. The system processes proposed in the previous chapter allows computations to be transmitted and performed on the thin client as well as on a cloud server.

5.4.1 Cloud Computing Services (CCS)

Social applications can quickly grow in terms of stored data related to a person; by keeping the published content where they already publicly exist – in cloud hosted services, there is no need to duplicate and complicate the entire system. A cloud-based server is well suited for social applications since they do not have mission critical transactions. The general requirements for choosing a cloud-based implementation are security, high accessibility, transaction management, and load balancing. CCS can handle all of these concerns behind the scenes, they are outsourced to the host service provider.

The nature of the PURIC system does not require long running large tasks, this is economical in terms of the cost for scalability and concurrently running instances, or resident instances in GAE terms. The entire PURIC URL request processing is not very intensive, the server is only querying for the mapped root URIs of the URL requested. Making GAE an ideal candidate for hosting the PURIC server for the purpose of this research. GAE has built in authentication, and it uses Python a dynamic scripting language ideal for developing Web 2.0 applications [58]. It also

has good support for RESTful Web application. Local caching of the resource has to be implemented on the third party storage server that hosts the resource origin. It cannot be done on the PURIC server since the origin URI is requested by and sent to the client and the client sends the request from the browser to the origin server. If the resource origin server supports caching then the resource will only be transferred to the client after the first time if it has expired on the client, or if it has changed on the server.

Software as a Service (SaaS) technologies like DropBox may be used to store public and private media files, people's Web resources. Social media sites like YouTube may be used to host different versions of the same video, say with different endings, with one URI alias and the PURIC server determining the origin URIs to send. The client determines the origin URI, calls it, and displays it.

Availability and accessibility are very important when it comes to social networks since the user needs to build a trust with the system. However, warnings and default webpages are shown when issues arise. The PURIC server hosts default webpages that are shown in case there are any issues with the origin URI connection, whether it's an image, video, document, or dynamic content.

5.4.2 Communication Layer

REST patterns are used in the communication layer components to pass information. It is the most standard method of communicating between the HTTP clients and server. The information is modelled in such a way that it is flexible and does not require a stateful communication. Meaning any event message contains all the information required in order for it to be executed.

GAE has been known to scale well under many requests, thus the PURIC request handlers will trigger new server instances to run on when overwhelmed. It is an elastic system that scales up and down depending on demand. It is also known to measure well under very high loads of reads, or GET requests, as the datastore works on a hash principle, it is optimized for reads, which is the bulk of the requests handled by the PURIC server.

The GAE's webapp2 framework used to develop the PURIC server supports han-

dlers for the popular HTTP methods for building a RESTful app. REST is used in the admin side in order to facilitate the creation of a PURIC API and integration of its CRUD with existing social applications. The public or client component of the PURIC system displays the origin resource and makes use of the GET method to retrieve the HTML with the Javascript logic and JSON. Figure 4.6 showed the communication between the client, server, and third party storage service.

The social resource publisher component makes use of the GET method for retrieving the list of origin URIs and retrieving the list of URI aliases a publisher user has, and the POST method for submitting the forms to create and update URI aliases and origin URI pointers. The RESTful design will give the PURIC system the potential to add a full API if necessary that makes use of the PUT and DELETE methods.

Listing 5.2 shows the PURIC server-side source for generating the wrapper code from an HTML template, and injecting the JSON of the potential root URI of the media content. Note that if there is only one version then the resource's html tag is included in the initial response as oppose to a list of URIs.

```
...
class LinkedDataUIHandler(BaseUIHandler):
        def get(self, **kwargs):

                alias = "http://socialrms.appspot.com/ld/"+kwargs['alias']
                urialias = db.GqlQuery('SELECT * FROM URIAlias WHERE alias = :1',
                    alias).get()

                ld_qry = db.GqlQuery('SELECT * FROM LinkedData WHERE ANCESTOR IS :1
                    ORDER BY version DESC', urialias)
                linkeddata = None
                ld_list = []
                single_url = None
                ld_list_json = None

                # If version is specified get the correct version.
                if self.request.GET.has_key('v'):
                        for ld in ld_qry:
                                if ld.version == int(self.request.GET['v']):
                                        ld_list.append({'size':ld.client_screen, '
                                            url':ld.origin_uri})
                                        break
```

```
                else: # Create a list with the latest ld version of each size.
                        for ld in ld_qry:
                                ld_list.append({'size':ld.client_screen, 'url':ld.
                                    origin_uri})

                if len(ld_list) == 1:
                        template_values = {
                                'file_type': urialias.type,
                                'single_ld': ld_list[0],
                                'publisher': urialias.publisher,
                        }
                else:
                        template_values = {
                                'file_type': urialias.type,
                                'ld_list': ld_list,
                                'publisher': urialias.publisher,
                        }

                if urialias.type == 'Video':
                        template_values['file_type'] = urialias.type
                        file_ext = os.path.splitext(ld_qry[0].origin_uri)[1][1:].
                            strip()
                        template_values['source_type'] = 'video/'+file_ext
                elif urialias.type == 'Audio':
                        template_values['source_type'] = 'audio/mpeg'

                return self.render_template('public/ld.html', **template_values)
...
```

Listing 5.2: Server side handler for returning the wrapper for the media content.

Chapter 6

Tests, Evaluation, and Future Work

This chapter discusses the tests and evaluation of the PURIC system, starting with the upper public client layer of the system down to the lower server layer. Four tests examined the feasibility of the PURIC system, and how well it reacts and performs under different circumstances. The tests proved that the system is feasible and that in a production environment such as GAE CCS it can withstand large loads with its fluctuating resource request load balancing. The client side tests examine the responsive Web design elements, as well as the perceived network latencies. Some usability testing may have to be done in a future work to make sure that the perceived latency values are acceptable.

The system drawbacks and limitations found from the tests and discussed in the evaluation, come of which are compromises for the system to function based on the requirements or use cases it needs to fulfil. The future work section shows how the modularity of the PURIC system can support a handful of interesting use cases.

6.1 Experiments

The tests performed on the system are listed below and the subsections that follow will go over each of them:

1. Responsive Web content test - tested the feasibility of how PURIC can supply responsive media content formats, that can decrease network traffic, CPU processing, and memory usage.
2. Client-side perceived network latency - tested if the response time for showing the content requested is acceptable.
3. GAE local server load test over varying request loads - tested the app perfor-

mance in a non-scalable (no load balancing) environment to find the point at which the app halts to a complete failure or an unacceptable performance level.

4. GAE cloud server load test over varying request loads - tested the load balancing in GAE's Cloud Computing Python platform, given the same code as in the previous test.

6.1.1 Test 1: Responsive Web Test

Description

This simple feasibility test looked into how the PURIC client reacted based on the screen resolution properties of a particular device. The PURIC URL of the image used for this test is mapped to three different image files, each being an image resource of different resolutions (width and height): 480p, 800p, and 1024p. The mobile device requesting the PURIC URL should request the best fit image based on its screen resolution and possibly active orientation (portrait or landscape). The Javascript that identifies the current screen size (and orientation) and selects the appropriate image URI is put to test here.

Results

Table 6.1 shows what is supported for each client platform. Testing for the support of different resolutions on the same device required that the device be in one orientation before loading the PURIC URL, then physically turning it to the other orientation and reloading the PURIC URL. The images (root URI) fetched should be different, being the best fit image for the particular resolution. As an example, when loading the PURIC URL of an image resource in the Samsung Nexus S in portrait mode (480 x 800) it fetches the 480p image; rotating it to landscape mode (800 x 480) fetches the 800p image.

Evaluation

Android Nexus S and BlackBerry Playbook browsers support the correct width resolution depending on the current orientation of the device. Whereas the iPhone and iPad browsers require some orientation detection and logic code to get the

Client Platform	Screen Resolution(s) Supported
BlackBerry Curve OS5	320 x 240
BlackBerry Playbook OS7	1024 x 600 (landscape) and 600 x 1024 (portrait)
iPhone 4G iOS 5.1	320 x 480
iPad iOS 5.1	768 x 1024
Samsung Nexus S w/ Android 4.0	480 x 800 (portrait) and 800 x 480 (landscape)
MacBook Air with Chrome	1440 x 900 and 1152 x 720

Table 6.1: Responsive Tests: Client Devices Used

correct screen width resolution. The BlackBerry Curve OS5 supports the screen width/height property as expected and since there is no accelerometer it does not require to have any logic for a change in orientation. Based on the results this feature is quite feasible , well supported, and simple to implement.

6.1.2 Test 2: Client-side Perceived Latency

Description

Web Browser	Timestamp	Page Transaction Times	Total PURIC Page Transaction
Chrome	2012/10/11 9:07am	4.74s + 0.216s + 1.53s	6.76s
Chrome	2012/10/11 9:08am	0.156s + 0.136s + 1.43s	1.84s
Chrome	2012/10/11 9:08am	0.173s + 0.190s + 1.65s	2.12s
Chrome	2012/10/11 9:09am	0.206s + 0.134s + 1.13s	1.58s
Chrome	2012/10/11 9:10am	0.521s + 0.125s + 1.21s	1.97s
Firefox	2012/10/12 8:15am	4.56s + 0.211s + 1.76s	6:59s
Firefox	2012/10/12 8:16am	0.399s + 0.146s + 1.32s	1.93s
Firefox	2012/10/12 8:17am	0.385s + 0.181s + 1.31s	1.98s
Firefox	2012/10/12 8:18am	0.386s + 0.146s + 1.49s	2.1s
Firefox	2012/10/12 8:19am	0.158s + 0.246s + 1.3s	1.78s

Table 6.2: Desktop Client Request Measurements

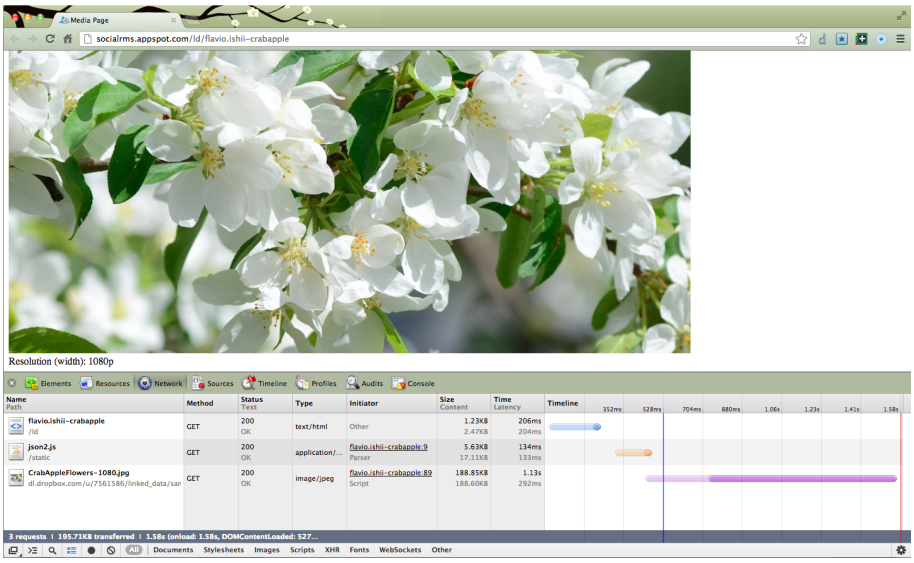

Figure 6.1: Measurement of one page load in the Chrome browser.

Three resource GET requests are sent from client: the PURIC URL followed by the json2.js file and the image link from DropBox. The latency time for each resource request has a waiting time and a receiving time, Table 6.2 shows the time for each request-response and the total time to request, parse, and render the PURIC page contents. Figure 6.1 shows the Chrome browser's Developer Tool, with the Network tab open to view the requests and response information captured. In all five test cases the same media URL was used which contained the same root URI of the media resource, so the download size for each of the three resource requests were kept the same. Five test runs were executed for each browser in each day, the exact time they were ran were off by one hour which could have caused a minor discrepancy depending on the network traffic at the time. Different days were used in order to show that the discrepancy among the first measurement and the consecutive measurements of the day were not affected by the Internet Service Providers' (ISPs) data caching or any internal cloud server caching. The Chrome browser was used on the first day and the Firefox browser on the second to make sure that the times for rendering the webpages in each browser was not significantly different.

The test procedure is as follows:

1. Process load for the media URL at 5 separate times for each Web browser (Chrome and Firefox)

2. Clear browser cache so the response always contains the resource.

3. Record the time it takes for each HTTP request to get a response

The client machine specifications: Mac OS 10.7.5, 1.8 GHz Intel Core i7 processor, 4 GB RAM, download speed caps at 20 Mbps based on optimal conditions offered by the ISP, connection speed at the resource host will differ depending on active loads, a wired connection is used by the client.

Client browsers have plugins that help developers to debug scripts and also measure all data transfer times. The Chrome browser has the Developer Tools and Firefox browser has Firebug. The client-side tests both browsers to make sure that the browser performance between the two popular browsers are not significantly different. Browsers have different parsing, rendering, and painting techniques.

Results

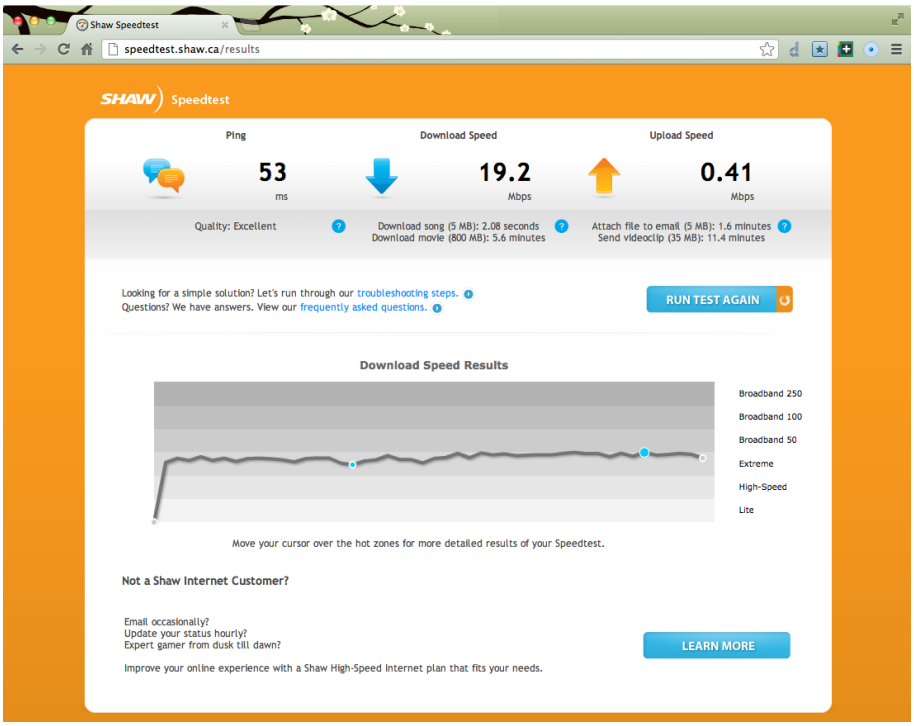

Figure 6.2: Shaw (cable ISP) speed tests results.

Shaw (cable ISP) Speed Tests were conducted prior to loading the first PURIC URL on each browser. Figure 6.2 shows a sample screen shot of the speed test for the Chrome browser.

Table 6.2 provides the timestamp of each request in Chrome and Firefox, the response times for each resource for each PURIC URL request, and the Page load time. The PURIC URL request pulls a dynamic HTML file from the GAE Production Cloud, which pulls two additional files – json2.js and the media content (given by the root URI).

Resource	Lowest	Highest	Mean
1. PURIC URL: dynamic HTML	0.156 sec	4.74 sec	1.168 sec
2. json2.js: static javascript library	0.125 sec	0.246 sec	0.173 sec
3. root media resource: image file	1.13 sec	1.76 sec	1.413 sec

Table 6.3: Client Mean Times for Resource Requests

Evaluation

Note the response time outliers in the first requests for each of the browsers each day. The discrepancy is mostly due to a new GAE instance having to be spawned to process the first request of the day. Part of the reason why json2.js and the media resources were mostly lower after each days' first request was due to the ISP caching that can occur between the host server and client. The mean for the entire page load was 2.865 seconds, which is an acceptable time for loading 1080pixel image.

6.1.3 Test 3: Local GAE Development Server Tests

Description

The tests conducted in this section show how the code reacts on the local GAE development server, a non-scalable and limited resource machine. It shows how many request/response transactions it can handle before slowing down to an unacceptable response rate. And ultimately shows the feasibility of running the system this type of environment.

The local GAE development server runs in a controlled environment setting, which is useful for finding out some limitations. The infrastructure in this environment does not scale up or down based on the changing needs, only one instance of the server will

run locally as oppose multiple instances in the actual GAE cloud server. The server machine specifications are as follows: Mac OS 10.7.5, 1.8 GHz Intel Core i7 Processor, 4 GB RAM, 25 Mbps download speed (measured at 14.12 Mbps before tests), max upload speed of 2.5 Mbps (measured at 0.34 Mbps before tests). The ISP provided a network connection speed test Web application (http://speedtest.shaw.ca/results) to extract these values.

A separate server was setup on the Rackspace IaaS to send the requests to the GAE application. The Rackspace server was on a Ubuntu 10.04 LTS (Lucid) machine with 1 vCPU, 256 MB of memory, 10GB of storage, 20 Mbps of public network bandwidth. The Tsung load testing tool and its dependancies were installed and used to run the load tests, capture the data, and graph it. The requests generated from Tsung simulate many concurrent users hitting the PURIC system, requesting a PURIC URL, which are logged by GAE and shown graphically and statistically in the application's Dashboard.

The Tsung open-source multi-protocol was used as the distributed load testing tool for the PURIC server. It is written in Erlang, which allows it to create hundreds and thousands of concurrent requests per second, while not consuming too much of its host's resources. Tsung outputs stats and graphical reports based on recorded samples of data taken as well as high and low values. 10 second samples are used and recorded to avoid affecting the load test and take up extra processing, slowing down the test machine. The following test runs were performed by the Tsung server to hit the Local GAE Development Server:

1. 5 requests per seconds (qps) over 5 minutes;
2. 50 qps over 5 minutes;
3. 100 qps over 5 minutes.

A test session consists of two requests (PURIC URL and json2.js), plus a simulated request to the root URI by adding its mean time to the session from Table 6.3. The code listing 6.1 shows a sample xml that was used to configure the Tsung load test for the 100qps over 5 minutes.

```
<tsung loglevel="notice" version="1.0">
  <clients>
```

```
    <client host="localhost" use_controller_vm="true" maxusers="10000"/>
  </clients>
  <servers>
    <server host="174.2.4.124" port="80" type="tcp"/>
  </servers>

  <load>
    <arrivalphase phase="1" duration="5" unit="minute">
      <users arrivalrate="100" unit="second"/>
    </arrivalphase>
  </load>

  <options>
    <option type="ts_http" name="user_agent">
      <user_agent probability="100">
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.8) Gecko/20050513 Galeon/1.3.21
      </user_agent>
    </option>
  </options>

  <sessions>
    <session name="http-example" probability="100" type="ts_http">
      <request>
        <http url="/ld/flavio.ishii-crabapple" method="GET" version="1.1"/>
      </request>

      <request>
        <http url="/static/json2.js" method="GET" version="1.1"/>
      </request>

      <thinktime value="1.413" random="true"/>
    </session>
  </sessions>
</tsung>
```

Listing 6.1: Tsung configuration XML.

Results

The results for the experiment's phase 3 (100 qps over a five minute duration) are shown below in Tables 6.4, 6.5, and 6.6; as it represents the case with the largest load tested on the local GAE development environment. The data collected for run 1 (5qps over 5 minutes) and run 2 (50 qps over 5 minutes) are found in the tables and graphs of Appendix B.1.1 and B.1.2 respectively.

Name	Highest 10sec mean	Lowest 10sec mean	Highest rate	Mean	Total
Connect	2.59 sec	39.51 msec	119.8 / sec	1.44 sec	21,291
Page	8mins 44sec	3.33 sec	42.3 / sec	37.85 sec	10,276
Request	7mn 32sec	1.70 sec	86.1 / sec	18.25 sec	21,290
Session	6mn 42sec	4.41 sec	83.4 / sec	59.60 sec	23,041

Table 6.4: Main Stats.

Name	Highest rate	Total
size_rcv	588.56 Kbits / sec	19.91 MB
size_sent	140.45 Kbits / sec	3.03 MB

Table 6.5: Network Throughput Stats

Name	Highest rate	Total
error_abort_max_conn_retries	64.2 / sec	12,765
error_connect_etimedout	5.2 / sec	677
error_connect_system_limit	271.1 / sec	61,860

Table 6.6: Errors

Evaluation

Tsung did not crash the PURIC server running on the local development environment, instead it times out after long requests and re-sends them. This accumulates the requests being handled and processed. Figure 6.3 shows how the transactions are still being completed after the 300 seconds (five minute) mark. The peaks in Figure 6.3 represent the times when the transaction rate is high, meaning the requests have accumulated and it starts to process and finishes them in the troughts. The cycles begin again as it starts to accumulate and queue up more requests. The graph starts to flatten out as the queue shortens; especially after the 300 seconds (5 minute) mark. When comparing the graphs from all three runs, the first run does not go much over the five minute mark, the second and third runs almost reach the 600 seconds (10 minute) mark in order to keep up with accumulated requests.

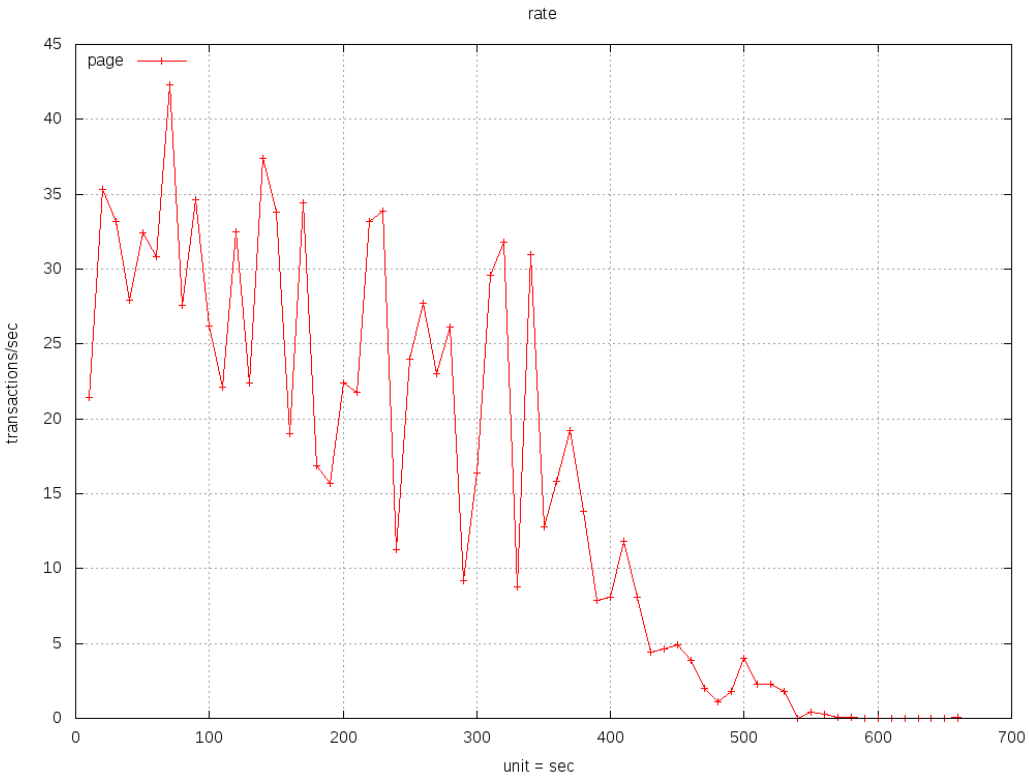

Figure 6.3: Tsung Experiment: Run 3 - 100qps for 5 minutes.

It also appears that the error rate is doubled from the 50 to 100 requests/second runs, implying that the error rate and the latency will keep increasing as the requests per second increase. The 9.12 seconds mean page load value shown in the main stats (Appendix B.1.1) for the 5 qps over 5 minutes test run is unacceptable, and it jumps to 32.87 seconds and 37.85 seconds (50 qps over 5 mins in Appendix B.1.2 and 100 qps over 5 mins in Table 6.7). The unacceptable 9.12 seconds mean may be due to a slow processor in the local host, the geographical distances between the Tsung server and the local PURIC server that causes long latency, but the likely culprit for the slow page load mean may simply be due to how the Local GAE Development Server is implemented and handles the requests in a not so timely manner, unlike GAE's production server seen in the next experiment. These tests concluded that the local GAE development server is not an ideal testing ground for handling massive requests above 5 qps. It is however great for initial tests to give an approximation of how the system is handling the requests and make sure there is an expectation of what the results will be in the production server.

6.1.4 Test 4: Cloud-based GAE Production Server Tests

Description

Cloud-based GAE Production Server tests demonstrate how well the PURIC system scales and performs in a production environment at a low cost. It is set out to prove the feasibility of the code implemented for handling massive requests. The following two test runs and their phases were performed by the Tsung server to hit the Cloud-based GAE Production Server:

1. 5 requests per seconds (qps) over 5 minutes;
2. 50 qps over 5 minutes;
3. 100 qps over 5 minutes;
4. A combination of five different phases all at once (25qps, 50qps, 100qps, 50qps, 25qps each ran over 5 minutes) lasting 25 minutes.

The last test run performs a load test by ramping up the load sent then decreasing it, in the attempts to show the elastic scalability of a PaaS like GAE.

Results

The same tests that were ran on the Local GAE server were conducted for the Cloud-based GAE environment with the results of the first three runs in this experiment found in tables and graphs of Appendix B.2. The last run is shown in Tables 6.7 and 6.8. GAE's cloud environment could have had more performance enhancement features enabled but it was kept to its basic/default configuration.

No errors were reported in the stats of any of this section's experiments, unlike with the local GAE development server. This is due to its scalability and lower latency that the Cloud GAE Production Server has; which allows plenty of time to recover and handle the requests accordingly.

The request rates and duration do not crash the cloud server as new instances are activated based on the increase in demand, and all requests started within the duration specified were completed no matter how long it takes. Instances are activated and de-activated as the demand increases and decreases respectfully. GAE's administration console for the PURIC app provided the server-side test result data

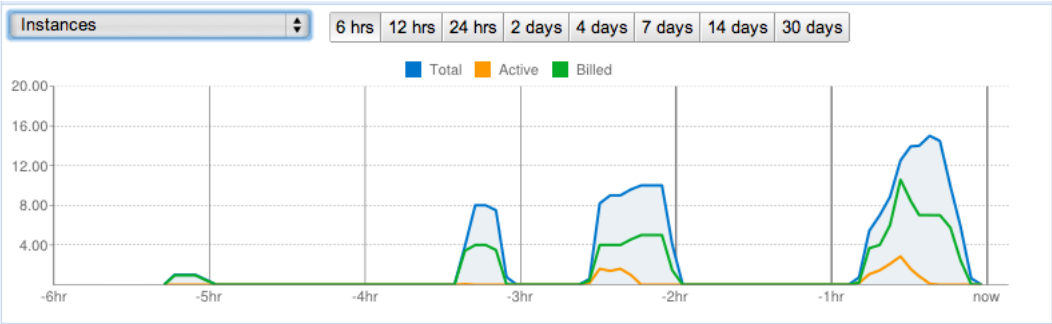

Figure 6.4: Instances spawned in the production server for all tests ran.

and graphs showing when new instances were activated (see Figure 6.4) for all four test runs, depending on how many requests were pending.

The requests per second values in the experiments were chosen so that the Tsung machine network bandwidth of 20MB/s was not surpassed, which is the limit set for the Rackspace Tsung VM. If a higher bandwidth was used by all system components the qps values could have been increased.

Name	Highest 10sec mean	Lowest 10sec mean	Highest rate	Mean	Count
Connect	1.17 sec	14.02 msec	91.5 / sec	40.75 msec	66,126
Page	2.45 sec	0.25 sec	113.4 / sec	0.46 msec	66,126
Request	1.22 sec	0.13 sec	227/ sec	0.23 msec	132,252
Session	3.74 sec	1.56 sec	117.7 / sec	1.88 msec	66,126

Table 6.7: Main Stats

Name	Highest rate	Total
size_rcv	17.39 Mbits/sec	1.24 GB
size_sent	253.91 Kbits/sec	20.05 MB

Table 6.8: Network Throughput Stats

The test phases happen in iterations, one after the other. Figure 6.5 shows the fluctuation happening at 300 seconds, 600 seconds, 900 seconds, 1200 seconds, and 1500 seconds; directly mapping to the start/end of each of the test run phases – the 300 seconds intervals.

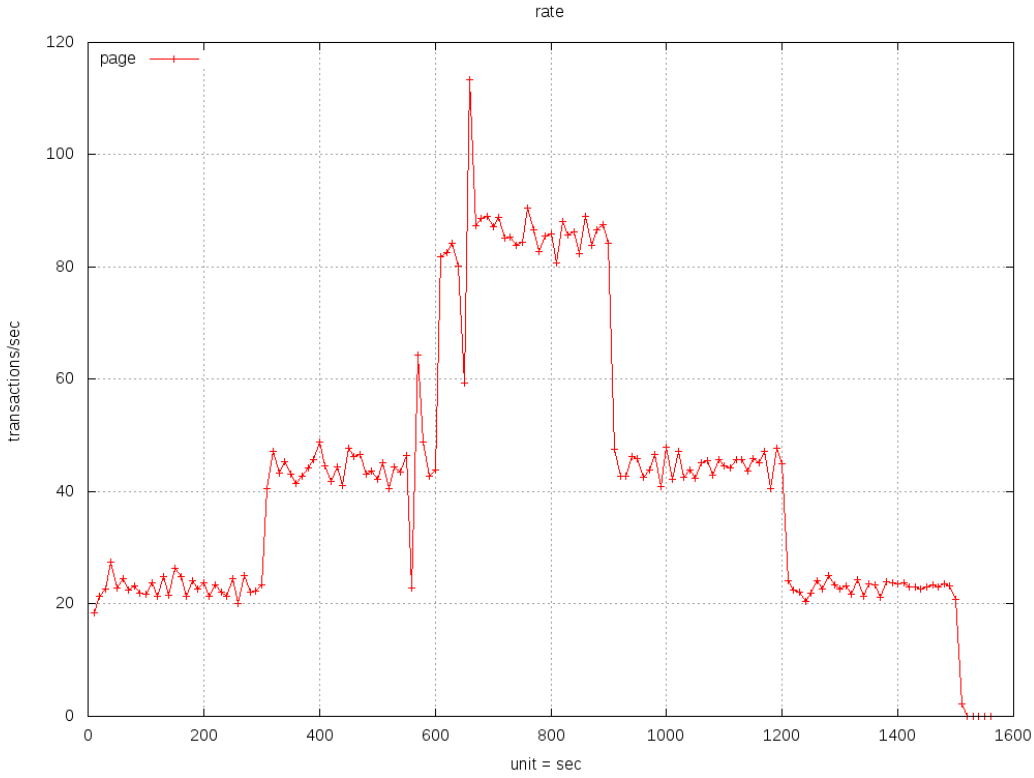

Figure 6.5: Tsung Experiment 4: 25qps for 5 minutes, 50qps for 5 minutes, 100qps for 5 minutes, 50qps for 5 minutes, 25qps for 5 minutes.

Evaluation

Unlike the Local GAE Development server, the Cloud GAE Production Server does handle the requests and has acceptable mean page load times: 0.23 seconds (5 qps over 5 minutes), 0.31 seconds (50 qps over 5 minutes), and 0.4 seconds (100 qps over 5 minutes).

Figure 6.5 shows that even though each phase should exponentially increase the number of requests queued the curve remained flat, and act like the 50qps or 100qps per 5 minute runs in the previous test runs, each 5 minute phase ran their respective 5 minute run. This occurred because GAE's production server scales according to demand by either spawning new instances or removing spawned instances. The transactions per second stays relatively constant for each of the five phases, increasing to a higher transactions/second rate only at the 5 and 10 minute marks, decreasing at the 15 and 20 minute mark, as the request demands increase and decrease. This concludes that it is feasible to implement the PURIC System in a PaaS like GAE and it would be able to handle the massive request loads tested.

6.2 PURIC Drawbacks and Limitations

There are several drawbacks and limitations that can be pointed out based on the design, technologies chosen, and system evaluations:

- Although PURIC provides a media URI sharing mechanism, it adds an extra step for publishing a media resource – the step to map the root URI with a PURIC URL.
- When posting a PURIC URL on social applications it will only show the URL and not a thumbnail on the post, specifically on Facebook. But if Facebook were to integrate a PURIC like module as a feature they could customize it to show a visual preview of the resource.
- The root URI for a media resource pointed by a PURIC page must be public. Using the media storing service's authorization mechanisms to grant access to the private resource is a remaining challenge. Such authorization mechanisms do lock the content to the particular storage service, because the user access

information cannot be easily transferred from one host to another as there is no industry standard for this in place. This is only of concern when the resource is not public and such use case does not comply with the needs of the sustainable Web growth this thesis focusses on.

- PURIC URLs will live for as long as the PURIC Server lives, meaning that if the server goes down or seizes to exist all the PURIC URLs published will return 404 HTTP status codes, as they will no longer be found. The PURIC server can however be moved to a different host without service interruption.
- Because not all browsers support the Javascript and phone capabilities the same, the screen width that is initially detected may not be correct, depending on the initial orientation (landscape or portrait) of the device. The screen width property and orientation detection code have not been tested on other platforms and thus there is no guarantee that it will work in a different way if supported at all. This is a matter of time before these properties are fully supported.

The adaptation of PURIC as part of existing popular SN sites can improve the user experience by tightly integrating the two systems. Facebook could support PURIC-like features and it could solve the first three points listed above, and as long as Facebook lived on the PURIC features would as well, handling the last point above.

6.3 Future Work

Additional functionalities, use cases, and challenges could be looked at, building on the core design and work presented in this thesis:

- Authentication and authorization layers could be added to the PURIC system as it is quite modular, via RESTful means. This would support data privacy control via authenticated linked data routing if the use case calls for a secure method of sharing resources. It can also add the possibility to personalization, where content can not only be specifically selected but also customized for specific users accessing it.
- An analytics module could be integrated that also uses REST via AJAX, send-

ing POST requests from the client to the user analytics module. The analytics can be used by the publisher to see which root URIs are being requested the most, and if some versions are not even being requested. Publishers can then determine things such as what screen sizes to support for future resources they publish.

- A client's contextual awareness module based on the surrounding environment conditions could be added as another level of responsiveness. Context based responsive Web does not only look at the client platform's properties but also the client's user's context. As an example that would entail this is when showing an image resource the PURIC client can choose the language of the text over it based on the city that the user is at. This would make use of existing client side features such as the geolocation javascript functions.
- Scheduled updates, where the root URI version being pointed to by a PURIC URL can change at a scheduled time. This could be useful for the true sense of a PURIC, which is a channel of changing media files for users to view periodically.
- Adding a Web crawl-friendly page based on tag words can make the published media SEO-friendly. A study can be conducted on how this publishing model impacts SEO. As the root URI changes, SEO effectiveness will depend on the content found in the webpage of the URL, and by also using friendly URLs. Pingback's, RDFs, and tagging could be used to improve SEO.
- PURIC could also have a notification mechanism such as RSS or even a realtime component to it, where if the root URI changes it notifies the subscriber.
- Accessibility versioning may be supported by the PURIC system, a responsive page based on accessibility settings.
- User experience is still a challenge for the PURIC system as it requires the user to manage URLs and their content in a cloud storage provider (i.e., DropBox). PURIC could be wrapped as a layer over DropBoxes functionalities, by using DropBox's API to simplify the publishing process.
- A PURIC feature providing the option for the publisher to show the Web resource embedded on a webpage or forward it to the root URL.

- Broken link detection and automatic re-routing of Web resource. If redirected publisher is notified by the PURIC system, for the publisher to react accordingly.

Chapter 7

Conclusion

The PURIC system has a simple design and implementation. It provides an innovative method of publishing and controlling the links to media files in such a way that does not require extra CPU processing, memory usage, and network traffic on the server and client. It ultimately allows the Web content publishers to:

- Host his/her public media resources in any data storage service.
- Control over the different versions of the media resources that are aliased by the PURIC URLs published.
- Control over what the published PURIC link displays, whether the link has been republished by others or not.
- Move the root media resources around without having to re-publish new URLs.
- Not have to store files in their own computer nor require them to keep their computer on with the server running as P2P solutions require.
- Not have to install any program on their computer, since PURIC runs on the browser.

The RESTful design of the PURIC system allows it to be easily integrated into existing systems. Its modularity also supports additional functionalities and technologies to be implemented in order to improve the system and support future use cases. The most complex challenge for this system is for users to adopt it as a de-facto since it requires an additional step to publish content along with other drawbacks and limitations. This could be easily overturned if the popular social applications like Facebook and Twitter integrate PURIC as a module.

PURIC may not be and does not have to be used in all situations of publishing media, there are certain use cases where it thrives such as updating a corporate

video or event video where its URL has already been published, or in the case of a URL inside an ebook where its source has changed location. Most SN site users may publish a URL and not worry about it once it has been published.

PURIC introduces a publishing schema that is different from others but yet it can be integrated into any existing social application because of its simple and light implementation. It utilizes standard Web technologies in order to target all platforms and devices in the market with a Web browser. And most importantly PURIC follows the same principles of the WWW, in terms of accessible data and sustainable growth.

The implementation and tests conducted demonstrate the feasibility of PURIC in a cloud-based platform. The implementation is simple and is able to handle massive amounts of resource requests originating from many types of client platforms. Many future use cases and work have been mentioned to demonstrate PURIC's usefulness and where its path is leading towards.

References

[1] Social networks and the semantic web. In *IEEE/WIC/ACM International Conference on Web Intelligence, 2004. WI 2004. Proceedings*, pages 285–291, September 2004.

[2] Facebook. "http://mildgreenhelpliquid.com/?p=393", April 2011.

[3] Amazon web services. "http://aws.amazon.com/", November 2012.

[4] Appfog. "https://www.appfog.com/", November 2012.

[5] Box.com. "http://box.com", November 2012.

[6] Cakephp. "http://cakephp.org/", November 2012.

[7] Diaspora. "http://en.wikipedia.org/wiki/Diaspora_(software)", October 2012.

[8] Diaspora. "http://diasporaproject.org/", November 2012.

[9] Django. "https://www.djangoproject.com/", November 2012.

[10] Dropbox. "http://dropbox.com/", November 2012.

[11] Facebook graph api. "https://developers.facebook.com/docs/reference/api/", November 2012.

[12] Flickr. "http://flickr.com", November 2012.

[13] Google drive. "http://drive.google.com/", December 2012.

[14] Heroku. "http://www.heroku.com/", November 2012.

[15] Instagram. "http://instagram.com/", December 2012.

[16] Joyent cloud. "http://joyent.com/products/joyent-cloud", November 2012.

[17] The locker project. "http://lockerproject.org", October 2012.

[18] Nitrogen web framework. "http://nitrogenproject.com/", November 2012.

[19] Rackspace cloud. "http://www.rackspace.com/cloud/", November 2012.

[20] Ruby on rails. "http://rubyonrails.org/", November 2012.

[21] Salesforce. "http://www.salesforce.com/", November 2012.

[22] Socialriver. "http://socialriver.org", November 2012.

[23] Youtube. "http://www.youtube.com"", November 2012.

[24] *YouTube - Terms of Service.* "https://www.youtube.com/static?gl=US&template=term", November 2012.

[25] Zoho. "http://www.zoho.com/", November 2012.

[26] L.M. Aiello and G. Ruffo. Lotusnet: tunable privacy for distributed online social network services. *Computer Communications*, 35(1):75–88, 2012.

[27] Luca Maria Aiello and Giancarlo Ruffo. Secure and flexible framework for decentralized social network services. In *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2010.

[28] Subbu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly Media, 2010.

[29] Tim Berners-Lee. Long live the web: A call for continued open standards and neutrality. "http://www.scientificamerican.com/article.cfm?id=long-live-the-web&print=true", November 2010.

[30] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann. World-wide web: the information universe. *Internet Research*, 20, Issue 4:461–471, 2010.

[31] S. Buchegger and A. Datta. A case for p2p infrastructure for social networks-opportunities & challenges. In *Wireless On-Demand Network Systems and Services, 2009. WONS 2009. Sixth International Conference on*, pages 161–168. IEEE, 2009.

[32] S. Buchegger, D. Schiöberg, L.H. Vu, and A. Datta. Peerson: P2p social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pages 46–52. ACM, 2009.

[33] Claudia Canali, Valeria Cardellini, Michele Colajanni, Riccardo Lancellotti, and Philip S. Yu. Web content caching and distribution. chapter Cooperative architectures and algorithms for discovery and transcoding of multi-version content, pages 205–221. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[34] Martin Clancy, Ronan Cremin, and John Leonard. Implementing your mobile strategy. Whitepaper, dotMobi, February 2012.

[35] Stefan Decker and Martin Frank. The social semantic desktop. Technical report, DERI - Digital Enterprise Research Institute, 2004.

[36] J. Delgado. Bridging provider-centric and user-centric social networks. *Handbook of Research on Business Social Networking: Organizational, Managerial, and Technological Dimensions*, pages 63–83, November 2012.

[37] Amy-Mae Elliot. Youtube facts. "http://mashable.com/2011/02/19/youtube-facts/", February 2011.

[38] G. Clemm et al. Versioning extensions to webdav (web distributed authoring and versioning). HTTP RFC, March 2002.

[39] G. Clemm et al. Web distributed authoring and versioning (webdav) access control protocol. HTTP RFC, May 2004.

[40] Roy T. Fielding et al. Hypertext transfer protocol – http/1.1. HTTP RFC, June 1997.

[41] Less Faber. Canadian social media statistics 2011. "http://www.webfuel.ca/canada-social-media-statistics-2011/", July 2011.

[42] Facebook. Facebook blog. "https://blog.facebook.com", November 2012.

[43] Facebook, "https://www.facebook.com/legal/terms". *Statement of Rights and Responsibility*, November 2012.

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

[45] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *ACM Transactions on Internet Technology*, volume 2, pages 115–150, May 2002.

[46] B. Gardner. Responsive web design: Enriching the user experience, 2011.

[47] Google. Google app engine, November 2012.

[48] Joe Gregorio. Getting started load testing your app engine application. Website, August 2009.

[49] Dominique Guinard. Towards the web of things: Web mashups for embedded devices. In *MEM 2009*, 2009.

[50] S. Lawrence H. Nielsen, P. Leach. An http extension framework. HTTP RFC, February 2000.

[51] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile web browsing. *Personal Communications, IEEE*, 5(6):8–17, 1998.

[52] W. Itani, A. Kayssi, and A. Chehab. Privacy as a service: Privacy-aware data storage and processing in cloud computing architectures. In *Dependable, Autonomic and Secure Computing, 2009. DASC'09. Eighth IEEE International Conference on*, pages 711–716. IEEE, 2009.

[53] Terry Jones. *Book: A Futurist's Manifesto*, chapter Why Digital Books Will Become Writable. O'Reily Media, 2012.

[54] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW 2010 Proceedings of the 19th international conference on World wide web*, 2010.

[55] Frank McCown and Michael L. Nelson. What happens when facebook is gone? In *JCDL 2009 Proceedings of the 9th ACM/IEEE-CS joint conference on Digital libraries*, 2009.

[56] J. Mitchell-Wong, R. Kowalczyk, A. Roshelova, B. Joy, and H. Tsai. Opensocial: From social networks to social ecosystem. In *Digital EcoSystems and Technologies Conference, 2007. DEST'07. Inaugural IEEE-IES*, pages 361–366. IEEE, 2007.

[57] Jeffrey C. Mogul. Clarifying the fundamentals of http, 2004.

[58] T. OReilly. What is web 2.0: Design patterns and business models for the next generation of software. *Communications & strategies*, (1):17, 2007.

[59] T. Paul, S. Buchegger, and T. Strufe. Decentralized social networking services. *Trustworthy Internet*, pages 187–199, 2011.

[60] Cesare Pautasso and Erik Wilde. Restful web services: principles, patterns, emerging technologies. In *WWW 2010 Proceedings of the 19th international conference on World wide web*, 2010.

[61] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big" web services: Making the right architectural decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*. ACM, 2008.

[62] Lucian Popa, Ali Ghodsi, and Ion Stoica. Http as the narrow waist of the future internet. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[63] J.A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H.J. Epema, M. Reinders, M.R. Van Steen, and H.J. Sips. Tribler: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*, 20(2):127–138, 2008.

[64] quartzjer. Next steps for the locker project. "http://blog.lockerproject.org/2012/05/01/next-steps-for-the-locker-project/", December 2012.

[65] et al. Roy T. Fielding. Http 1.1: Method definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html, March 2010.

[66] ScientialMobile. Wurfl website, April 2012.

[67] Seok-Won Seong, Jiwon Seo, Matthew Nasielski, Debangsu Sengupta, Sudheendra Hangal, Seng Keat Teh, Ruven Chu, Ben Dodson, and Monica S. Lam. Prpl: a decentralized social networking infrastructure. In *MCS 2010 Proceedings of the 1st ACM Workshop on Mobile Cloud Computing and Services: Social Networks and Beyond*, 2010.

[68] San-Tsai Sun, Kirstie Hawkey, and Konstantin Beznosov. Secure web 2.0 content sharing beyond walled gardens. In *Computer Security Applications Conference, ACSAC 2009.*, 2009.

[69] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: better privacy for social networks. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 169–180. ACM, 2009.

[70] Twitter. Rest api. http://apiwiki.twitter.com.

[71] W3C. Http hypertext transfer protocol. http://www.w3.org/Protocols/.

[72] W3C. W3c workshop on the future of social networking. "http://www.w3.org/2008/09/msnws/report.html", January 2009.

[73] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[74] C.A. Yeung, I. Liccardi, K. Lu, O. Seneviratne, and T. Berners-Lee. Decentralization: The future of online social networking. In *W3C Workshop on the Future of Social Networking Position Papers*, volume 2, 2009.

[75] Polychronis Ypodimatopoulos. Ego - decentralized social networking. "http://polychronis.gr/projects/ego-decentralized-social-networking/comment-page-1/", December 2009.

[76] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[77] B. Zhou and C. Wu. Social networking interoperability through extended foaf vocabulary and service. In *Information Sciences and Interaction Sciences (ICIS), 2010 3rd International Conference on*, pages 50–55. IEEE, 2010.

Appendix A

Implementation Source Code

A.1 Server-side

The files listed in this section consist of the code for the server app configuration, and client- and admin-side request handlers.

```
application: socialrms
version: 1
runtime: python27
api_version: 1
threadsafe: yes

inbound_services:
- warmup

skip_files:
- ^(.*/)?app\.yaml
- ^(.*/)?app\.yml
- ^(.*/)?index\.yaml
- ^(.*/)?index\.yml
- ^(.*/)?#.*#
- ^(.*/)?.*~
- ^(.*/)?.*\.py[co]
- ^(.*/)?.*/RCS/.*
- ^(.*/)?\..*

handlers:
- url: /robots\.txt
  static_files: static/robots.txt
  upload: static/robots\.txt

- url: /static
  static_dir: static

- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: .*
  script: main.app

libraries:
- name: jinja2
  version: "latest"
```

Listing A.1: Webapp2 application specific settings - app.yaml

```
"""
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: Webapp2 framework configuration file
"""

webapp2_config = {}
webapp2_config['webapp2_extras.sessions'] = {
        'secret_key': 'my secret key',
}

# Reference: http://webapp-improved.appspot.com/_modules/webapp2_extras/auth.html
# 86400 = seconds in a day
webapp2_config['webapp2_extras.auth'] = {
```

```
    'user_model': 'models.User',
    'cookie_name': 'my-session',
    'token_max_age':   86400 * 7 * 3,
    'token_new_age':   86400,
    'token_cache_age': 3600,
}
```

Listing A.2: Webapp2 framework configuration file - config.py

```
"""
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: Driver class for webapp2 app. Based on the boilerplate code from https
    ://github.com/coto/gae-boilerplate
"""

import webapp2
import routes
import config

app = webapp2.WSGIApplication(debug=True, config=config.webapp2_config)
routes.add_routes(app)
```

Listing A.3: Driver class for webapp2 app - main.py

```
'''
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: Model classes for URIAlias and LinkedData records.
Note: A resource has one alias but may have multiple origins. Its origins will
    depend on the platform chosen and/or versions for that platform.
'''

from datetime import datetime

from google.appengine.ext import db

class URIAlias(db.Model):

        TYPE_CHOICES = [
                (u'Image'),
                (u'Audio'),
                (u'Video'),
                (u'Html'), #URL (containing any of the above file URIs)
                (u'Flash'),
                (u'Other'),
        ]
        alias = db.StringProperty()
        publisher = db.UserProperty(required=True) #db.StringProperty(required=True)
        is_archived = db.BooleanProperty(default=False)
        latest_version = db.IntegerProperty(default=1)
        description = db.StringProperty(multiline=True)
        tags = db.ListProperty(unicode,default=None)
        type = db.StringProperty(choices=TYPE_CHOICES) # once created it may not
            differ from the LinkedData


# Uses URIAlias as its parent/ancestor
class LinkedData(db.Model):

        SCREEN_CHOICES = [
                ('480'),
                ('800'),
                ('1080')
        ]

        origin_uri = db.StringProperty(required=True)
        version = db.IntegerProperty(default=1)
```

```
        client_screen = db.StringProperty(choices=SCREEN_CHOICES)
        #authorized_usernames = db.StringListProperty(default=[])
        description = db.StringProperty(multiline=True)

        is_archived = db.BooleanProperty(default=False)
        updated = db.DateProperty(auto_now=True)
        created = db.DateProperty(auto_now_add=True)
```

Listing A.4: Model classes for URIAlias and LinkedData records - models.py

```
'''
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: Route configuration code for webapp2 app.
'''

import logging
from webapp2_extras import routes
from webapp2_extras.routes import RedirectRoute

from request_handlers.publicuihandlers import HomeUIHandler, LinkedDataUIHandler
from request_handlers.apihandlers import URIAliasAPIHandler
from request_handlers.adminuihandlers import URIAliasListUIHandler,
    URIAliasAddFormHandler, URIAliasUIHandler, LinkedDataAddFormHandler,
    LinkedDataEditFormHandler

# Everything is treated as a resource: alias, ui content, original resource content
# Using RedirectRoute instead of webapp2.Route since it supports strict_slash
_routes = [

        # These API handles return JSON
        RedirectRoute('/api/alias/<alias>', handler=URIAliasAPIHandler, name='
            versioned-alias-api', strict_slash=True),
        RedirectRoute('/api/alias/<alias>', handler=URIAliasAPIHandler, name='alias-
            api', strict_slash=True),

        # These admin handles return ui resources
        RedirectRoute('/puric_admin/ui/urialias/add', handler=URIAliasAddFormHandler
            , name='alias-addform', strict_slash=True),

        RedirectRoute('/puric_admin/ui/urialias/<ua_key>/linkeddata/add', handler=
            LinkedDataAddFormHandler, name='ld-addform', strict_slash=True),
        RedirectRoute('/puric_admin/ui/linkeddata/<ld_key>', handler=
            LinkedDataEditFormHandler, name='ld-editform', strict_slash=True),

        RedirectRoute('/puric_admin/ui/urialias/<ua_key>', handler=URIAliasUIHandler
            , name='alias-editform', strict_slash=True),
        RedirectRoute('/puric_admin/ui/urialias', handler=URIAliasListUIHandler,
            name='aliaslist-ui', strict_slash=True),

        RedirectRoute('/puric_admin/', handler=URIAliasListUIHandler, name='puric-ui
            ', strict_slash=True),
        RedirectRoute('/ld/<alias>', handler=LinkedDataUIHandler, name='versions-ui
            ', strict_slash=True),
        RedirectRoute('/ld/', handler=LinkedDataUIHandler, name='versions-ui',
            strict_slash=True),

        RedirectRoute('/', handler=HomeUIHandler, name='homepage', strict_slash=True
            ),
]

def get_routes():
    return _routes

def add_routes(app):
    for r in _routes:
        app.router.add(r)
```

Listing A.5: Route configuration code for webapp2 app. - routes.py

```
'''
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: Form definitions for creating and editing an ad campaign.
'''

from wtforms import validators, Form, TextField, TextAreaField, BooleanField,
    SelectField, HiddenField, RadioField

'''SCREEN_CHOICES = [
        (u'Any','Any'),
        (u'Small Mobile', 'Small Mobile Device (480px phones)'),
        (u'Large Mobile', 'Large Mobile Device (800px tablets)'),
        (u'Desktop','Desktop/Laptop (1280)'),
]'''

SCREEN_CHOICES = [
        ('480', 'Small Mobile Devices - 480p'),
        ('800', 'Medium to Large Mobile Devices - 800p'),
        ('1080', 'Laptop or Desktop - 1080p')
]

TYPE_CHOICES = [
        (u'Video','Video'),
        (u'Audio','Audio'),
        (u'Image','Image'),
        (u'Other','Other (HTML, Flash, Documents...)'),
]

class LinkedDataDetailForm(Form):
        origin_uri = TextField('Root URI', [validators.Required()])
        client_screen = SelectField('Screen Size', choices=SCREEN_CHOICES)
        is_archived = BooleanField('Archived', default=False)
        description = TextAreaField('Description')

class URIAliasDetailForm(Form):
        alias_uri = TextField('Link to share (URI Alias): ', [validators.Required()
            ])
        description = TextAreaField('Description:')
        is_archived = BooleanField('Archived', default=False)

# This form object is used to create both the URIAlias and LinkedData entities as
    the first step.
class NewURIAliasLinkedDataDetailForm(URIAliasDetailForm):
        origin_uri = TextField('Root URI', [validators.Required()])
        client_screen = SelectField('Screen Size', choices=SCREEN_CHOICES)
        file_type = SelectField('File Type', choices=TYPE_CHOICES, default='Any')
```

Listing A.6: Form definitions for creating and editing an ad campaign - ldmsforms/linkeddata.py

```
"""
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: Base request handling code, parent for other request handler codes/
    files.
"""

import os
import webapp2
import logging
from webapp2_extras import auth

from webapp2_extras import sessions
from webapp2_extras import jinja2

class BaseHandler(webapp2.RequestHandler):
        """
```

```
        BaseHandler for all requests
        Holds the auth and session properties so they are reachable for all requests
        """
        def dispatch(self):
                """
                        Save the sessions for preservation across requests
                """
                try:
                        response = super(BaseHandler, self).dispatch()
                        #self.response.write(response)
                finally:
                        self.session_store.save_sessions(self.response)

        @webapp2.cached_property
        def auth(self):
                return auth.get_auth()

        @webapp2.cached_property
        def session_store(self):
                return sessions.get_store(request=self.request)

        @webapp2.cached_property
        def auth_config(self):
                """
                Dict to hold urls for login/logout
                """
                return {
                        'login_url': self.uri_for('login'),
                        'logout_url': self.uri_for('logout')
                }


class BaseUIHandler(BaseHandler):
        @webapp2.cached_property
        def jinja2(self):
                return jinja2.get_jinja2(app=self.app)

        def render_template(self, filename, **template_args):
                self.response.write(self.jinja2.render_template(filename, **
                    template_args))
```

Listing A.7: Base request handling code, parent for other request handler codes/files - request_handlers/basehandlers.py

```
"""
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: Request handlers for UI screens of the administration side of app.
Notes: A URI Alias can have many different versions. Each version can be any linked
    data URI as the origin URI.
"""

from google.appengine.ext import db
from google.appengine.api import users

from request_handlers.basehandlers import BaseUIHandler
from ldmsforms.linkeddata import LinkedDataDetailForm, URIAliasDetailForm,
    NewURIAliasLinkedDataDetailForm

from models import LinkedData, URIAlias

import webapp2
import logging

def require_auth(handler_method):
        def check_login(self, *args, **kwargs):
                if users.get_current_user():
                        return handler_method(self, *args, **kwargs)
```

```
                # User is not signed in.
                try:
                        self.redirect(users.create_login_url("/puric_admin/"), abort
                            =True)
                except (AttributeError, KeyError), e:
                        self.abort(403) # Forbidden if handler has no login_url
                            specified

        return check_login


'''
        GET /puric_admin/ui/urialias shows the list of URIAlias
'''
class URIAliasListUIHandler(BaseUIHandler):
        # Alias Listing screen
        @require_auth
        def get(self):
                user = users.get_current_user()
                alias_qry = db.GqlQuery('SELECT * FROM URIAlias WHERE publisher =
                    :1', user)
                template_values = {
                        'test': 'Linked data alias listing page: ',
                        'alias_qry': alias_qry,
                        'logout_url': users.create_logout_url("/puric_admin/"),
                }
                return self.render_template('admin/urialias/list.html', **
                    template_values)


'''
        GET|POST /puric_admin/ui/urialias/add handles the form display/submission to
             add the URIAlias and the first LinkedData entities
'''
class URIAliasAddFormHandler(BaseUIHandler):
        @require_auth
        def get(self):
                user = users.get_current_user()
                template_values = {
                        'uri_alias_prefix': 'http://socialrms.appspot.com/ld/'+str(
                            user)+'-',
                        'form_action': self.request.url,
                        'form_method': 'POST',
                        'form': NewURIAliasLinkedDataDetailForm(self.request.POST),
                        'logout_url': users.create_logout_url("/puric_admin/")
                }
                return self.render_template('admin/urialias/add.html', **
                    template_values)

        @require_auth
        def post(self):
                user = users.get_current_user()

                form = NewURIAliasLinkedDataDetailForm(self.request.POST)
                del form.is_archived

                if form.validate():
                        alias_key = "http://socialrms.appspot.com/ld/"+str(user)
                            +"-"+form.alias_uri.data

                        alias = URIAlias(alias=alias_key, publisher=user,
                            description=form.description.data, type = form.file_type
                            .data)
                        alias.put()

                        # Very first ld for this uri_alias
```

```
                        ld = LinkedData(parent=alias.key(), version=1, origin_uri=
                            form.origin_uri.data, client_screen=form.client_screen.
                            data, description=form.description.data)
                        ld.put()

                        self.redirect('/puric_admin/ui/urialias')
                else:
                        return 'error'


'''
        GET /puric_admin/ui/urialias/<ua_key> for showing the URIAlias record form
            and its linked data records
        POST /puric_admin/ui/urialias/<ua_key> for handling the URIAlias update form
             submission
'''
class URIAliasUIHandler(BaseUIHandler):
        @require_auth
        def get(self, **kwargs):
                urialias = db.get(kwargs['ua_key'])
                if urialias:
                        form = URIAliasDetailForm(self.request.POST, obj=urialias)
                        ld_qry = db.GqlQuery('SELECT * FROM LinkedData WHERE
                            ancestor is :1 and is_archived = False', urialias)

                else:
                        urialias = None
                        ld_qry = None

                template_values = {
                        'urialias': urialias,
                        'alias_form_action': '/puric_admin/ui/urialias/'+str(
                            urialias.key()),
                        'alias_form_method': 'POST',
                        'form': form,
                        'ld_qry': ld_qry,
                        'logout_url': users.create_logout_url("/puric_admin/")
                }
                return self.render_template('admin/urialias/edit.html', **
                    template_values)

        @require_auth
        def post(self, **kwargs):
                user = users.get_current_user()
                if user:
                        uri_alias = db.get(kwargs['ua_key'])
                        uri_alias.description = self.request.POST['description']
                        uri_alias.put()

                        self.redirect(self.request.url)
                else:
                        return 'error'


# LinkedData Request Handlers ***************************************

'''
        GET|POST /puric_admin/ui/urialias/<ua_key>/linkeddata/add
'''
class LinkedDataAddFormHandler(BaseUIHandler):
        @require_auth
        def get(self, **kwargs):
                template_values = {
                        'test': 'alias detail ui for alias id: ',
                        'form_action': self.request.url,
                        'form_method': 'POST',
                        'form': LinkedDataDetailForm(self.request.POST),
                        'logout_url': users.create_logout_url("/puric_admin/")
```

```
                }
                return self.render_template('admin/linkeddata/add.html', **
                    template_values)

        @require_auth
        def post(self, **kwargs):
                form = LinkedDataDetailForm(self.request.POST)
                del form.is_archived
                if form.validate():
                        alias = db.get(kwargs['ua_key'])
                        ld = LinkedData(parent=alias, version=alias.latest_version +
                             1, origin_uri=form.origin_uri.data, client_screen=form.
                            client_screen.data, description=form.description.data)
                        ld.put()
                        #if success:
                        alias.latest_version = alias.latest_version+1
                        alias.put()

                        self.redirect('/puric_admin/ui/urialias/'+kwargs['ua_key'])
                else:
                        return 'error'

'''
Handles: GET|POST/puric_admin/ui/linkeddata/<id:\w+>
'''
class LinkedDataEditFormHandler(BaseUIHandler):

        @require_auth
        def get(self, **kwargs):
                linked_data = db.GqlQuery('SELECT * FROM LinkedData WHERE __key__ =
                    :1', db.Key(kwargs['ld_key'])).get()
                if linked_data:
                        form = LinkedDataDetailForm(self.request.POST, obj=
                            linked_data)
                        urialias = linked_data.parent()
                        ld_version = linked_data.version
                else:
                        urialias = None
                        form = None
                        ld_version = None
                template_values = {
                        'urialias': urialias,
                        'version': ld_version,
                        'form_action': self.request.url,
                        'form_method': 'POST',
                        'form': form,
                        'logout_url': users.create_logout_url("/puric_admin/")
                }
                return self.render_template('admin/linkeddata/edit.html', **
                    template_values)

        @require_auth
        def post(self, **kwargs):

                linked_data = db.GqlQuery('SELECT * FROM LinkedData WHERE __key__ =
                    :1', db.Key(kwargs['ld_key'])).get()
                urialias = linked_data.parent()
                if linked_data and urialias.publisher == users.get_current_user():
                        form = LinkedDataDetailForm(self.request.POST, obj=
                            linked_data)
                        linked_data.client_screen = linked_data.client_screen
                        form.populate_obj(linked_data)
                        linked_data.put()
                        urialias = linked_data.parent()
                        ld_version = linked_data.version
                        updated = "Updated Linked Data successfully."
                        error = None
                else:
```

```
                        urialias = None
                        ld_version = None
                        updated = None
                        error = "Did not update Linked Data."
                template_values = {
                        'error': error,
                        'urialias': urialias,
                        'version': ld_version,
                        'updated': updated,
                        'form_action': self.request.url,
                        'form_method': 'POST',
                        'form': form,
                        'logout_url': users.create_logout_url("/puric_admin/")
                }
                return self.render_template('admin/linkeddata/edit.html', **
                    template_values)
```

Listing A.8: Request handlers for UI screens of the administration side of app. - request_handlers/adminuihandlers.py

```
"""
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: Request handlers for UI screens of client side of app, showing the
    screens for consuming the root resource (ie. video, image, html...)
"""

from google.appengine.ext import db
from models import LinkedData, URIAlias
from request_handlers.basehandlers import BaseHandler, BaseUIHandler
import webapp2
import logging
import os
from webapp2_extras import json
from urllib import unquote

# Request Handlers ********************************************

class HomeUIHandler(BaseUIHandler):
        def get(self):
                template_values = {'test': 'homepage'}
                return self.render_template('public/home.html', **template_values)


class LinkedDataUIHandler(BaseUIHandler):
        """
                Return html template for displaying the resource along with a json
                    of the potential resource version depending on the screen
                    resolution.
                If there is only one version then the resource's html tag is
                    included in the initial response.
        """
        def get(self, **kwargs):

                alias = "http://socialrms.appspot.com/ld/"+kwargs['alias']
                urialias = db.GqlQuery('SELECT * FROM URIAlias WHERE alias = :1',
                    alias).get()

                ld_qry = db.GqlQuery('SELECT * FROM LinkedData WHERE ANCESTOR IS :1
                    ORDER BY version DESC', urialias)
                linkeddata = None
                ld_list = []
                single_url = None
                ld_list_json = None

                # if version is specified get the correct version otherwise get the
                    latest version
                if self.request.GET.has_key('v'):
```

```
                        for ld in ld_qry:
                                if ld.version == int(self.request.GET['v']):
                                        ld_list.append({'size':ld.client_screen, '
                                            url':ld.origin_uri})
                                        break
                else: # else create a list with the latest ld version of each size
                        for ld in ld_qry:
                                ld_list.append({'size':ld.client_screen, 'url':ld.
                                    origin_uri})

                if len(ld_list) == 1:
                        template_values = {
                                'file_type': urialias.type,
                                'single_ld': ld_list[0],
                                'publisher': urialias.publisher,
                        }
                else:
                        template_values = {
                                'file_type': urialias.type,
                                'ld_list': ld_list,
                                'publisher': urialias.publisher,
                        }

                if urialias.type == 'Video':
                        template_values['file_type'] = urialias.type
                        file_ext = os.path.splitext(ld_qry[0].origin_uri)[1][1:].
                            strip()
                        template_values['source_type'] = 'video/'+file_ext
                elif urialias.type == 'Audio':
                        template_values['source_type'] = 'audio/mpeg'

                return self.render_template('public/ld.html', **template_values)


        def proxy(self, linkeddata):
                # do proxy code here ...
                return str(linkeddata.__dict__)
```

Listing A.9: Request handlers for UI screens of client side of app, showing the screens for consuming the root resource - request_handlers/publicuihandlers.py

```
"""
Flavio Ishii - flavio.ishii@usask.ca
November 01, 2012
Description: API request handling code.
"""

from google.appengine.ext import db
from request_handlers.basehandlers import BaseHandler
from models import LinkedData, URIAlias

import logging

'''
Handles: GET /api/alias/<alias>
'''
class URIAliasAPIHandler(BaseHandler):
        # Alias Listing and Detail screens
        def get(self, **kwargs):
                alias = "http://socialrms.appspot.com/alias/"+kwargs['alias']
                urialias = db.GqlQuery('SELECT * FROM URIAlias WHERE alias = :1',
                    alias).get()
                ld_qry = db.GqlQuery('SELECT * FROM LinkedData WHERE ANCESTOR IS :1
                    ORDER BY version DESC', urialias)

                if self.request.GET.has_key('ssize'):
                        new_ld_qry = []
                        for ld in ld_qry:
```

```
                                if ld.client_screen == 'Any' or self.request.GET['
                                    ssize'] == 'Any' or ld.client_screen == self.
                                    request.GET['ssize']:
                                        new_ld_qry.append(ld)
                        ld_qry = new_ld_qry

                # if version is specified get the correct version otherwise get the
                    latest version
                if self.request.GET.has_key('v'):
                        for ld in ld_qry:
                                if ld.version == int(self.request.GET['v']):
                                        linkeddata = ld
                                        break
                                else:
                                        linkeddata = ld
                else:
                        for ld in ld_qry:
                                if urialias.latest_version == ld.version:
                                        linkeddata = ld
                                        break
                                else:
                                        linkeddata = ld
                response = self.proxy(linkeddata)
                return self.response.out.write(response)

        def proxy(self, linkeddata):

                # do proxy code here ...

                return str(linkeddata.__dict__)
```

Listing A.10: API request handling code - request_handlers/apihandlers.py

A.2 Administration Templates

The files listed in this section consist of the code for the administration side templates.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
        {% block head %}
                <!--link rel="stylesheet" href="admin.css" /-->
                <title>{% block title %}{% endblock %}</title>
        {% endblock %}
        {% block head_scripts %}
        {% endblock %}

        <style type="text/css">
        .menu li
        {
        display: inline;
        list-style-type: none;
        padding-right: 20px;
        }
        </style>
</head>
<body>

        <div id="menu">
                <ul class="menu" >
                        {% block menuitems %}
                                <li><a href="/puric_admin/ui/urialias">URI Alias
                                    Listing</a></li>
                                <li><a href="/puric_admin/ui/urialias/add">Add URI
                                    Alias</a></li>
```

```
                        <li><a href="{{logout_url}}">Logout</a></li>
                    {% endblock %}
                </ul>
        </div>
        <div id="header">{% block header %}{% endblock %}</div>
        <div id="content">{% block content %}{% endblock %}</div>


        <div id="footer">
                {% block footer %}
                {% endblock %}
        </div>
</body>
</html>
```

Listing A.11: Base HTML template used by the admin UI screens - templates/admin/base.html

```
{% extends "admin/base.html" %}
{% block title %}Root URI Listing{% endblock %}
{% block head_scripts %}
{% endblock %}
{% block content %}
        <h3>Root URI Listing for {{urialias.alias}}</h3>
        <div id="linkeddata_list">
        {% for ld in ld_qry %}
                <div class="{{ loop.cycle('odd', 'even') }}">
                        /alias/flavio-{{ld.alias_uri}} <a href="/puric_admin/ui/
                            linkeddata/{{ld.key()}}">Edit</a>
                </div>
        {% endfor %}
        </div>
{% endblock %}
```

Listing A.12: Admin UI for listing the URI Alias records- templates/admin/urialias/list.html

```
{% extends "admin/base.html" %}
{% block title %}New Root URI{% endblock %}
{% block head_scripts %}
{% endblock %}
{% block content %}
        {% if form %}
        <form action="{{form_action}}" method="{{form_method}}">
                <div>{{form.origin_uri.label}} {{ form.origin_uri(class="css_class")
                     }}</div>
                <div>{{form.description.label}} {{ form.description(class="css_class
                    ") }}</div>
                <div>{{form.client_screen.label}} {{ form.client_screen(class="
                    css_class") }}</div>
                <button>Add</button>
        </form>
        {% endif %}
{% endblock %}
```

Listing A.13: Admin UI for creating a URI Alias record - templates/admin/urialias/add.html

```
{% extends "admin/base.html" %}
{% block title %}Update root URI for {{urialias.alias}}{% endblock %}
{% block content %}
        <h3>Update Root URI (URI alias v{{version}}) for {{urialias.alias}}</h3>
        {% if error %}
                <div class="error">{{error}}</div>
        {% endif %}
        {% if updated %}
                <div class="updated">{{updated}}</div>
```

```
        {% endif %}
        {% if form %}
                <div>
                <form action="{{form_action}}" method="{{form_method}}">
                        <div>{{form.origin_uri.label}} {{ form.origin_uri(class="
                            css_class") }}</div>
                        <div>{{form.client_screen.label}} {{ form.client_screen(
                            class="css_class") }}</div>
                        <div>{{form.description.label}} {{ form.description(class="
                            css_class") }} </div>
                        <div>{{ form.is_archived(class="css_class") }} {{form.
                            is_archived.label}}</div>
                        <input type="submit" value="Update"/>
                </form>
                </div>
        {% endif %}
{% endblock %}
```

Listing A.14: Admin UI for editing a URI Alias record - templates/admin/urialias/edit.html

```
{% extends "admin/base.html" %}
{% block title %}Root URI Listing{% endblock %}
{% block head_scripts %}
{% endblock %}
{% block content %}
        <h3>Root URI Listing for {{urialias.alias}}</h3>
        <div id="linkeddata_list">
        {% for ld in ld_qry %}
                <div class="{{ loop.cycle('odd', 'even') }}">
                        /alias/flavio-{{ld.alias_uri}} <a href="/puric_admin/ui/
                            linkeddata/{{ld.key()}}">Edit</a>
                </div>
        {% endfor %}
        </div>
{% endblock %}
```

Listing A.15: Admin UI for listing the Linked Data records- templates/admin/linkeddata/list.html

```
{% extends "admin/base.html" %}
{% block title %}New Root URI{% endblock %}
{% block head_scripts %}
{% endblock %}
{% block content %}
        {% if form %}
        <form action="{{form_action}}" method="{{form_method}}">
                <div>{{form.origin_uri.label}} {{ form.origin_uri(class="css_class")
                     }}</div>
                <div>{{form.description.label}} {{ form.description(class="css_class
                    ") }}</div>
                <div>{{form.client_screen.label}} {{ form.client_screen(class="
                    css_class") }}</div>
                <button>Add</button>
        </form>
        {% endif %}
{% endblock %}
```

Listing A.16: Admin UI for creating a Linked Data record - templates/admin/linkeddata/add.html

```
{% extends "admin/base.html" %}
{% block title %}Update root URI for {{urialias.alias}}{% endblock %}
{% block content %}
        <h3>Update Root URI (URI alias v{{version}}) for {{urialias.alias}}</h3>
        {% if error %}
                <div class="error">{{error}}</div>
```

```
        {% endif %}
        {% if updated %}
                <div class="updated">{{updated}}</div>
        {% endif %}
        {% if form %}
                <div>
                <form action="{{form_action}}" method="{{form_method}}">
                        <div>{{form.origin_uri.label}} {{ form.origin_uri(class="
                            css_class") }}</div>
                        <div>{{form.client_screen.label}} {{ form.client_screen(
                            class="css_class") }}</div>
                        <div>{{form.description.label}} {{ form.description(class="
                            css_class") }} </div>
                        <div>{{ form.is_archived(class="css_class") }} {{form.
                            is_archived.label}}</div>
                        <input type="submit" value="Update"/>
                </form>
                </div>
        {% endif %}
{% endblock %}
```

Listing A.17: Admin UI for editing a Linked Data record - templates/admin/linkeddata/edit.html

A.3 Client Templates

The files listed in this section consist of the code for the client side templates, mainly html files with code injection.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
        {% block head %}
                <title>{% block title %}{% endblock %}</title>
                {% block cssheets %}{% endblock %}
                {% block jsscripts %}{% endblock %}
        {% endblock %}
</head>
<body>

        {% if error %}
                <div class="error">{{error}}</div>
        {% endif %}

        <div id="content">{% block content %}{% endblock %}</div>

        <div id="footer">
                {% block footer %}
                <span id="media_size"></span>
                {% endblock %}
        </div>

        {% block body_js %}
        {% endblock %}
</body>
</html>
```

Listing A.18: Base HTML template used by the client/public UI screen - templates/public/base.html

```
{% extends "public/base.html" %}
{% block title %}Media Page{% endblock %}

{% block jsscripts %}
        <script src="/static/json2.js"></script>
```

```
        <script language="JavaScript">
                var linked_data_list = [
                        {% if single_ld %}
                                {
                                "size": {{single_ld.size}},
                                "url": "{{single_ld.url}}",
                                }
                        {% else %}
                                {% for ld in ld_list %}
                                        {
                                        "size": {{ld.size}},
                                        "url": "{{ld.url}}",
                                        },
                                {% endfor %}
                        {% endif %}
                ];

                function getBestFitURI() {
                        var largestWidthResource;
                        var screen_width = screen.width;
                        var screen_height = screen.height;

                        // Fix for Safari browser in iOS, since it will not change
                            the screen width based on orientation.
                        if(navigator.userAgent.match(/(iPhone|iPod|iPad)/i)) {
                                if (orientation == 90 || orientation == -90) {
                                        //landscapeMode
                                        screen_width = screen.height;
                                        screen_height = screen.width;
                                }
                        }

                        for(var i = 0; i < linked_data_list.length; i++) {
                                if(screen_width >= linked_data_list[i].size) {
                                        if((largestWidthResource == null)
                                                || (largestWidthResource != null &&
                                                    linked_data_list[i].size >
                                                    largestWidthResource.size))
                                        {
                                                        largestWidthResource =
                                                            linked_data_list[i];
                                        }
                                }
                        }
                        document.getElementById('media_size').innerHTML='Resolution
                            (width): '+largestWidthResource.size+'p';
                        return largestWidthResource;
                }
        </script>
        {% if file_type == "Video" %}
                <link href="http://vjs.zencdn.net/c/video-js.css" rel="stylesheet">
                <script src="http://vjs.zencdn.net/c/video.js"></script>
        {% endif %}
{% endblock %}

{% block content %}
        <p>A {{file_type}} posted by {{publisher}}. </p>

        <div id="resource"></div>

{% endblock %}

{% block body_js %}
        <script language="JavaScript">
                var ld = getBestFitURI();

                {% if file_type == "Image" %}
```

```
                        document.getElementById('resource').innerHTML = '<img src
                            ="'+ld.url+'" alt="Resource" />';

                {% elif file_type == "Audio" %}
                        document.getElementById('resource').innerHTML = '<video id="
                            my_video_1" class="video-js vjs-default-skin" controls \
                          preload="auto" width="'+ld.size+'" height="'+(ld.size/1.5)
                              +'" \
                          data-setup="{}"> \
                          <source src="'+ld.url+'" type="{{source_type}}"> \
                        </video>';

                {% elif file_type == "Video" %}
                        document.getElementById('resource').innerHTML = '<video id="
                            my_video_1" class="video-js vjs-default-skin" controls \
                          preload="auto" width="'+ld.size+'" height="'+(ld.size/1.5)
                              +'" \
                          data-setup="{}"> \
                          <source src="'+ld.url+'" type="{{source_type}}"> \
                        </video>';

                {% else %}
                        document.getElementById('resource').innerHTML = '<iframe
                            width="420" height="315" src="'+ld.url+'" frameborder
                            ="0" allowfullscreen></iframe>';

                {% endif %}
        </script>
{% endblock %}
```

Listing A.19: HTML template used by the client/public UI screen to consume the web resource requested - templates/public/ld.html

```
<!DOCTYPE html>
<html>
<head>
        <title>Home screen</title>
</head>
<body>
        {{ test }}
        <ul>
                <li><a href="/puric_admin/ui/urialias">Linked Data Admin</a></li>
            <!--li><a href="/login/">Login</a></li>
            <li><a href="/sadmin/">Admins login with Google App Account</a></li>
            <li><a href="/passwordreset">Forgot Password</a></li-->
        </ul>
</body>
</html>
```

Listing A.20: Landing page for web resource publishers and administrators - templates/public/home.html

Appendix B

Data Results

B.1 Local GAE Development Server Test Results

B.1.1 Load Test Data for 5 qps over 5 minutes

Name	Highest 10sec mean	Lowest 10sec mean	Highest rate	Mean	Count
Connect	2.13 sec	48.86 msec	14.2 / sec	1.01 sec	2,960
Page	23.73 sec	2.45 sec	9.9 / sec	9.12 sec	1,480
Request	22.34 sec	1.13 sec	16.8 / sec	4.46 sec	2,960
Session	26.01 sec	3.42 sec	10.6 / sec	10.56 sec	1,480

Table B.1: Main Stats

Name	Highest rate	Total
size_rcv	570.02 Kbits/sec	13.83 MB
size_sent	16.47 Kbits/sec	430.70 KB

Table B.2: Network Throughput Stats

Name	Highest rate	Total
error_connect_etimedout	0.4 / sec	13

Table B.3: Errors

B.1.2 Load Test Data for 50 qps over 5 minutes

Name	Highest 10sec mean	Lowest 10sec mean	Highest rate	Mean	Count
Connect	3.29 sec	35.93 msec	87.5 / sec	1.43 sec	19,308
Page	5mins 22sec	4.09 sec	49.8 / sec	32.87 sec	9,451
Request	3mn 48sec	1.92 sec	95.7 / sec	15.92 sec	19,188
Session	5mn 32sec	4.95 sec	49.5 / sec	50.68 sec	12,509

Table B.4: Main Stats

Name	Highest rate	Total
size_rcv	723.61Kbits/sec	17.68 MB
size_sent	102.08 Kbits/sec	2.75 MB

Table B.5: Network Throughput Stats

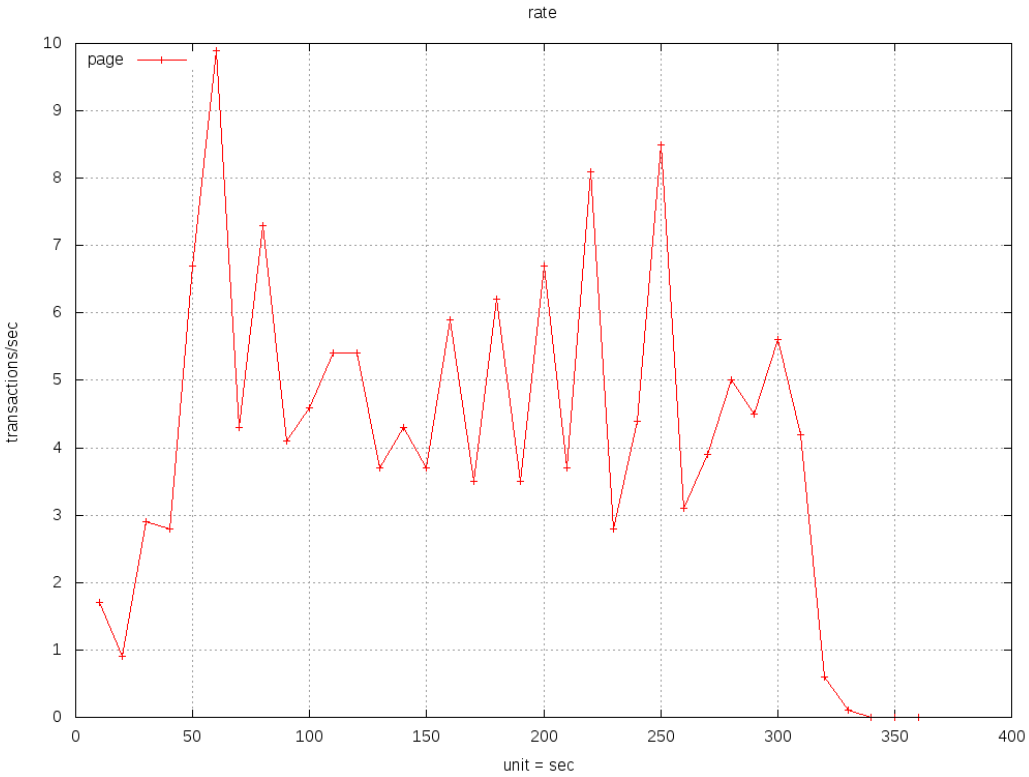

Figure B.1: Tsung Experiment 1: 5qps for 5 minutes.

Name	Highest rate	Total
error_abort_max_conn_retries	27.5 / sec	3,058
error_connect_etimedout	6 / sec	597
error_connect_system_limit	126.1 / sec	18,918

Table B.6: Errors

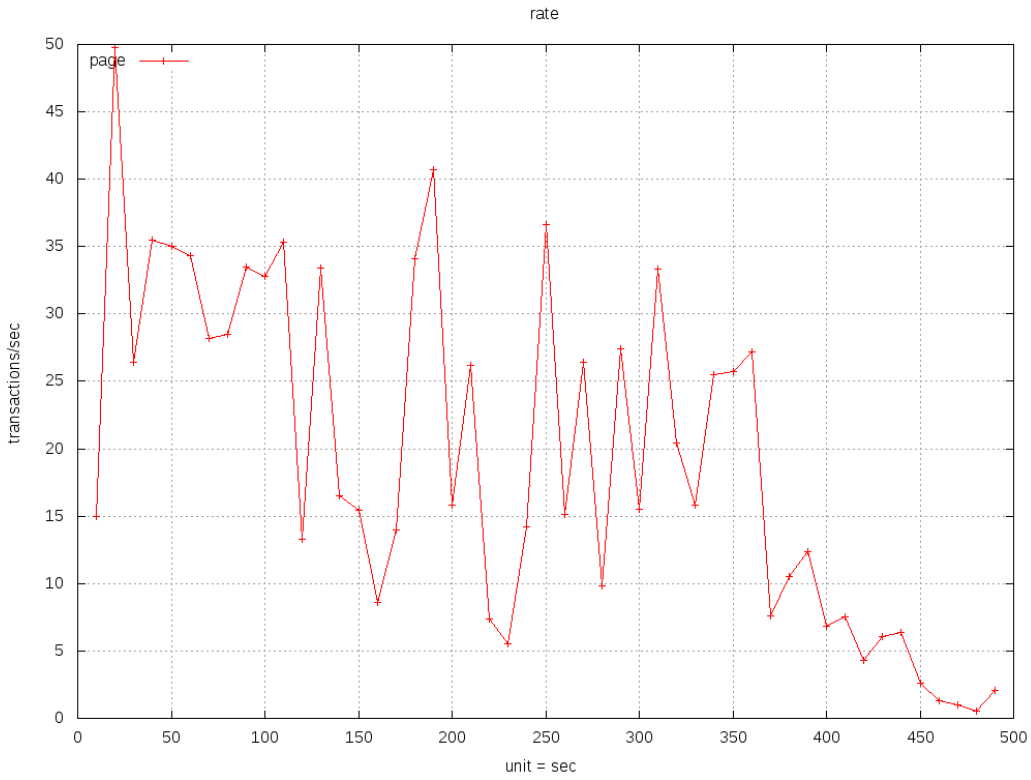

Figure B.2: Tsung Experiment 2: 50qps for 5 minutes.

B.2 Cloud GAE Production Server Test Results

B.2.1 Load Test Data for 5 qps over 5 minutes

Name	Highest 10sec mean	Lowest 10sec mean	Highest rate	Mean	Count
Connect	18.67 msec	13.99 msec	5.8 / sec	14.45 msec	1,410
Page	0.26 sec	0.14 sec	5.7 / sec	0.23 sec	1,410
Request	0.13 sec	68.21 msec	11.3 / sec	0.12 sec	2,820
Session	2.26 sec	1.18 sec	5.7 / sec	1.68 sec	1,410

Table B.7: Main Stats

Name	Highest rate	Total
size_rcv	893.01 Kbits/sec	27.03 MB
size_sent	14.29 Kbits/sec	437.87 KB

Table B.8: Network Throughput Stats

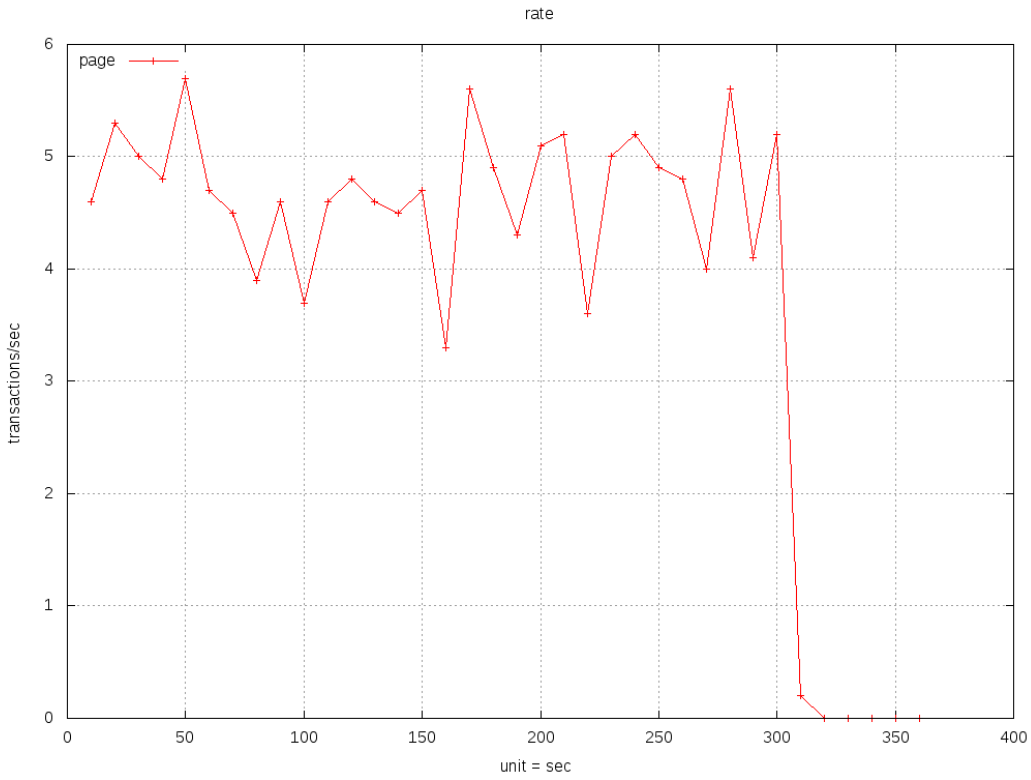

Figure B.3: Tsung Experiment 1: 5qps for 5 minutes.

B.2.2 Load Test Data for 50 qps over 5 minutes

Name	Highest 10sec mean	Lowest 10sec mean	Highest rate	Mean	Count
Connect	17.59 msec	14.04 msec	47.9 / sec	14.54 msec	13,219
Page	0.60 sec	0.23 sec	48.7 / sec	0.31 sec	13,219
Request	0.30 sec	0.12 sec	97.1 / sec	0.16 sec	26,438
Session	3.24 sec	1.59 sec	48.8 / sec	1.73 sec	13,219

Table B.9: Main Stats

Name	Highest rate	Total
size_rcv	7.46 Mbits/sec	253.31 MB
size_sent	119.60 Kbits/sec	4.01 MB

Table B.10: Network Throughput Stats

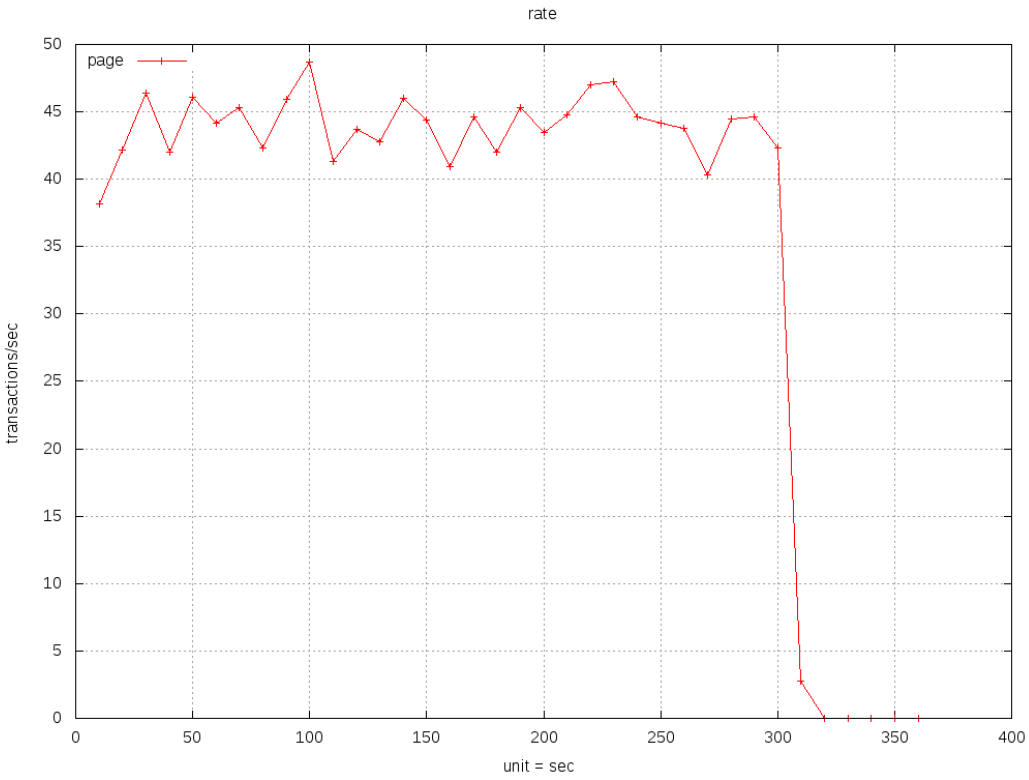

Figure B.4: Tsung Experiment 2: 50qps for 5 minutes.

B.2.3 Load Test Data for 100 qps over 5 minutes

Name	Highest 10sec mean	Lowest 10sec mean	Highest rate	Mean	Count
Connect	16.20 msec	13.99 msec	91.9 / sec	14.41 msec	25,444
Page	0.93 sec	0.29 sec	93 / sec	0.40 sec	25,444
Request	0.46 sec	0.14 sec	185.8 / sec	0.20 sec	50,888
Session	2.93 sec	1.65 sec	92.1 / sec	1.81 sec	25,444

Table B.11: Main Stats

Name	Highest rate	Total
size_rcv	14.26 Mbits/sec	487.68 MB
size_sent	229.04 Kbits/sec	7.72 MB

Table B.12: Network Throughput Stats

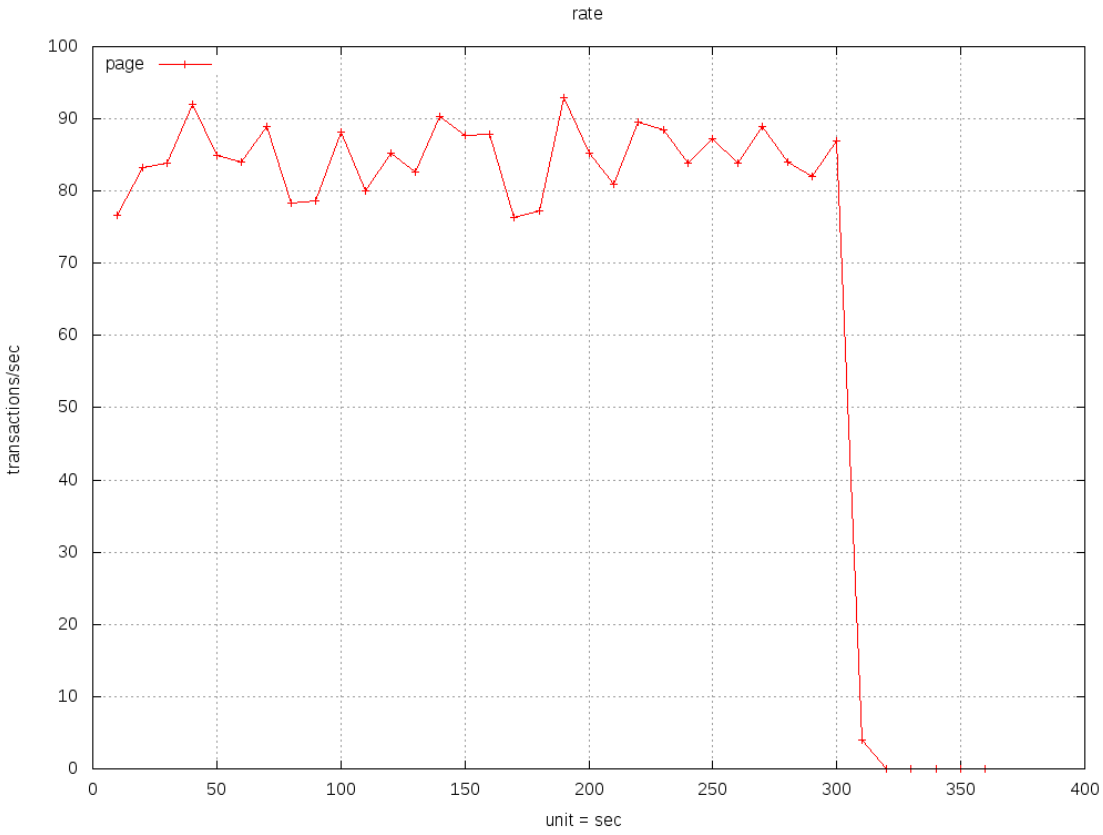

Figure B.5: Tsung Experiment 3: 100qps for 5 minutes.