A Framework for the RESTful Provisioning of Community-Contributed Web Services

by

Daniel Vijayakumar

A Thesis
Presented to
The University of Guelph

in partial fulfilment of requirements
for the degree of
Master of Science
in
Computer Science

Guelph, Ontario, Canada

© Daniel Vijayakumar, August, 2014
ABSTRACT

A Framework for the RESTful Provisioning of Community-Contributed Web Services

Daniel Vijayakumar  
University of Guelph, 2014

Advisor:  
Dr. Xining Li

Co-Advisor:  
Dr. Qusay H. Mahmoud  
University of Ontario Institute of Technology

Despite research spanning more than a decade in the area of Web services, the industry has not yet seen a consensus regarding an effective method for the provisioning of Web services. In recent years, a new paradigm for implementing Web services based on Representational State Transfer, the successful architectural foundation of the Web, has seen greater interest in the industry. This thesis proposes a framework for the implementation of platforms for provisioning community-contributed Web services using the Representational State Transfer (REST) paradigm. A prototype implementation of the framework has been developed for use as a reference. The results of evaluating the prototype demonstrate the feasibility and extensibility of the framework, particularly in the dynamic programming problem sub-domain.
Acknowledgments

First and foremost, I give my whole-hearted thanks to Dr. Qusay H. Mahmoud for taking a huge leap of faith by taking me on as his student. I am forever grateful for first meeting him in the Distributed Systems course that he taught me in the senior year of my undergraduate studies. Although it has taken me a little more than two years to truly enjoy the area of Web Services, it all started back in that course. Without a doubt, I find it remarkable that he has always trusted in my ability to produce good research, when I have always held doubts, many of them being irrational and rather detrimental to my overall well-being. But beyond all of the above, he has been extremely patient and gracious beyond all bounds with regards to allowing me to work at my very slow pace. I will always remember the moments of joy that I experienced during the many times I would receive an e-mail from him granting permission to postpone an upcoming milestone. I also thank him very much for the generous funding he provided for me, without which my continued studies would not have been an option.

Secondly, I sincerely thank Dr. Xining Li for eliminating any need for me to transfer away from Guelph by serving as my official resident advisor. His kind gesture played a key role in allowing my studies to continue smoothly.

There have been other professors who have had varying levels of influence on me during my time at Guelph. Dr. Charlie Obimbo has consistently been my friend and a welcome source of encouragement in many of my tenuous moments. Dr. William Gardner was the first of my professors to spark my interest in writing and maintaining complex software systems. Dr. Mark Wineberg taught me my favourite graduate course, Experimental
Design, in which his fine exposition of introductory philosophy of science along with his mastery of statistical methods wrought in me a love for scientific research. Dr. Joe Sawada introduced me to the painful but wonderful world of \LaTeX. To all of the above, as well as the many other scholars and professionals in the world that have left some mark on me, I give my warm salutations.

I certainly can not forget the entire staff at the School of Computer Science for taking care of all my administrative and financial requirements. I am particularly thankful for being assigned Teaching Assistantships that I have learned a lot from; even more than the students that I have taught.

Finally, I thank all of my wonderful friends and family for sticking with me throughout my education in various ways: Anna Maria and Ashley, for consistently being amazing role-models of hardwork, perseverance and professional excellence; Matthew, for his unrelenting emotional support; Job, for empathizing with me, being a graduate student himself; my sister, Caroline, and her husband, Augustine, for their practical aid at various crucial times; and my parents, Lita and Paul, for the unconditional love and affection they have always showered me with.
# Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Motivation ........................................... 2
1.2 Contributions ........................................ 4
1.3 Use Cases ............................................ 5
    1.3.1 Integrated Development Environments .............. 5
    1.3.2 Mobile Code-Offloading .......................... 6
1.4 Thesis Outline ...................................... 6

2 Background & Related Work 8

2.1 REST ................................................. 8
2.2 Service Registries .................................. 10
    2.2.1 Universal Business Registry .................. 10
    2.2.2 Programmable Web ............................ 12
    2.2.3 Modern Industry ............................... 12
2.3 Web Intents ......................................... 15
2.4 Shifting Paradigms ................................ 17
2.5 Summary ............................................ 17

3 Proposed Framework 19

3.1 Overview ............................................ 19
3.2 RESTful Foundation ................................. 21
    3.2.1 Service as a Resource ....................... 23
    3.2.2 Addressing Strategy ......................... 25
3.3 Framework Components .............................. 26
    3.3.1 Uniform Interface ............................. 26
    3.3.2 Service Cacheing ............................. 31
    3.3.3 Registration and Storage .................... 34
    3.3.4 Search and Discovery ......................... 36
    3.3.5 Service Composition ......................... 36
3.4 Summary ............................................ 37
List of Tables

3.1 Uniform interface definition for all interactions .......................... 27
A.1 Comparing average times of calculating fibonacci index of C code and implementation ...................................................... 89
A.2 Response times of the general performance experiment .................. 90
A.3 Throughput measures of the general performance experiment ............ 90
## List of Figures

3.1 Overview of Framework .................................................. 20  
4.1 System Architecture of Prototype Implementation .................. 40  
4.2 Amazon Web Services Configuration for the Prototype Implementation .. 43  
5.1 Native and prototype performance of computing Fibonacci sequences between 35 and 45 inclusive .................................................. 70  
5.2 Histogram of the aggregation of all five runs of the general performance experiment .................................................. 76
Chapter 1

Introduction

Service-Oriented Architecture (SOA) has been a very attractive paradigm for the design and development of distributed systems over the past decade. It dates back to the 1980s when distributed system architects sought to solve the problem of loosely coupled and language-agnostic components within complex applications. One of the original motivations behind this was the enabling of independent components (especially legacy/archaic code) within a modern system to be seamlessly integrated with the rest of the architecture as it continually evolved (Alonso et al. 2004). The middleware concepts that emerged such as remote procedure calls (RPC) and object request brokers (ORB) further pushed architects to investigate whether such distributed integration was possible not just within applications but between applications; and if between applications, then across the borders of enterprises. With the advent of the Web, this vision seemed very much possible. Research in this venture eventually led to what we now refer to as Web services.

A very general definition of a Web service can be given as “an application accessible to other applications over the Web” (Alonso et al. 2004). Such applications offer an interface (known in the context of the modern Web as a Web API) between some valuable functionality or data of the entity providing the service and parties (both external and internal) who may have an interest in consuming this data. There is no conceptual constraint
to the scope of publishing: if desired, a service can be made available to any interested party on the Web. Additionally, services can serve as building blocks for the creation of more complex services. These composite services, referred to as *Mashups* in the context of the modern social data-driven Web (Koschmider, Wilde, and Zirpins 2011), are themselves services; this detail is abstracted away from the service consumer.

As a consequence of the above features, application developers are able to enhance the value of their applications by integrating with such services to create features that would be impossible or infeasible to be developed in-house\(^1\). Not only can they take advantage of the valuable data offered by service providers, they are also able to extend the functionality of the provisioned services and feed their novel contributions back in to the ecosystem in the form of composite services.

### 1.1 Motivation

Web services standardization and research efforts have largely restricted themselves to the world of enterprise and Business-to-Business (B2B) integration. Furthermore, the specification for the Universal Description Discovery and Integration (UDDI) has failed to realize the open and optimistic vision of the Universal Business Registry (UBR) (Al-Masri and Mahmoud 2008; Pautasso, Zimmermann, and Leymann 2008; Pilioura and Tsalgatidou 2009). Arguably, the publishing and discovery of services is the most important aspect of the Web services model (Ferris and Farrell 2003) and the failure of UDDI has encouraged developers to seek other paradigms for service-oriented development.

---

\(^1\)For example, it has become very important in recent years for applications to be integrated with social media and this can be achieved through Web APIs provided by a myriad of entities such as Facebook, Twitter and Instagram to name a few.
Furthermore, the past decade has seen a shift towards more distributed decentralized development models, particularly with the rise of Free and Open Source Software a la Richard Stallman. The number of such projects developed in an open and collaborative fashion is no small number. For some perspective, the popular source code version-control service GitHub\(^2\) is single-handedly home to more than 4 million heterogeneous codebases (Doll 2012). More importantly, a decent number of these projects are valuable to the industry, thus demonstrating that the enterprise is not the only existing software-development community.

An interesting question is whether the Web services paradigm can be brought into these vastly distributed yet valuable projects. Perhaps such codebases could be exposed as services to be used by application developers via Web APIs. Hearkening back to the pattern of use and reuse, such an effort would greatly multiply the effectiveness and reach of these projects. The foundational problem is that of providing a platform for a non-restrictive eco-system of community-contributed services, which is currently inadequately addressed by Web services.

### 1.2 Contributions

More recently, some service providers have attempted to use an architectural model known as REpresentational State Transfer (REST) as an alternative paradigm to the traditional Web services model. REST detracts from the functional approach of standard SOA and instead approaches service development from the perspective of data, or resources."
Instead of focusing on exposing functionality to manipulate private valuable data, REST emphasizes the exposing of the data itself via a uniform interface that is constant in semantics. This is such a shift of paradigm from SOA that it has garnered its own name, Resource-Oriented Architecture (ROA) (Richardson and Ruby 2008). The credibility of ROA comes largely from the fact that REST is a considerably more simple and abstract architectural paradigm and has been the very reason for the success of the World Wide Web, serving as its underlying architecture.

Based on my exploration into the problem, I propose that the REST paradigm is a potential solution to the problem of provisioning an open and collaborative community of Web services that is not restricted to the enterprise. Such a diverse and open conglomerate of services offers as key benefits the openness of a collaborative community and the simplicity and robustness of the underlying REST architecture. This open, simplified and scalable access lowers the barrier of entry for developers and naturally facilitates increased innovation, the returns of which benefit all participants of the community.

The main contributions of this thesis are as follows:

- An abstract framework that demonstrates how the architectural principles of REST, specifically the modelling of Web services as ‘RESTful’ resources, can be applied to implement an open platform such that

  1. any developer can register self-sufficient pieces of code as services,

  2. any developer can discover these services,

  3. any party is able to invoke these services and consume the output,

  4. any developer can create composite services using existing services, and
5. the entire platform is exposed via a simple and unified interface.

- A reference implementation that I have created by applying the constraints of the proposed framework.

- An empirical evaluation of the implementation with an emphasis on the foundational aspect of runtime performance in order to draw inferences into the feasibility of the proposed framework as a whole.

1.3 Use Cases

To further solidify the motivation for this research, consider the following use cases that can be birthed from the implementation of the proposed framework.

1.3.1 Integrated Development Environments

By integrating the implementation with Integrated Development Environments (IDEs), developers would be able to incorporate into their applications complex features that are very difficult to develop in-house via a simple menu button which could interface with the REST API of the implementation. They could explore services that offer the exact/similar functionality desired. If no suitable service exists, the developer may choose to create a brand new service or a composite service by leveraging existing services; all from within the IDE. This new service would now be available to any other developer in the world.
1.3.2 Mobile Code-Offloading

Such an implementation may also be appealing to the mobile sector, where high power consumption, low computing resources and minimal improvements in battery efficiency (Cuervo et al. 2010; Fernando, Loke, and Rahayu 2013) have given rise to interesting energy optimization and micro-management problems. Mobile applications that wish to take advantage of the functionality offered by participating services would simply invoke a REST API that interfaces with the services by passing input and receiving output. The processing of the data is thus outsourced to these services that are external to the mobile device, thereby preserving both battery and computational resources.

1.4 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 lays the foundational background for the problem area and records my investigation into the existing literature that is relevant to the problem; Chapter 3 presents a framework for the RESTful provisioning of community-contributed Web services; Chapter 4 describes my proof-of-concept reference implementation of the proposed framework; Chapter 5 documents the design of experiments for empirically evaluating the reference implementation as well as the analysis of the results thereof; and I finally conclude this thesis in Chapter 6 by summarizing the research problem and the relevant results, followed by a brief discussion of the parts of this research that have been offered as avenues for future work.

---

3Note that this is in contrast with attempts to partition code to execute both remotely and locally. Instead, all computation would always be performed remotely. While there is a clear advantage in being able to focus all local resources on critical tasks, some peers may disagree in preference of a partitioning and optimization approach (Chun and Maniatis 2010; Kemp et al. 2012).
Chapter 2

Background & Related Work

In order to understand the problem I am attempting to solve, it is important to be grounded in the necessary information as well as discuss any relevant existing work. This chapter presents the same; each section contains my critique of the discussed work.

2.1 REST

Coined by Fielding in his Doctoral dissertation (Fielding 2000), the REpresentational State Transfer (REST) architectural style has proven to be the cornerstone of the Web’s success at scale.

A summary of the main principles that most of the literature agrees with is as follows (Davis 2012):

1. The central entity of a RESTful system is a resource. Every resource also has a unique identifier/address, standardized as the Uniform Resource Identifier (URI), within a namespace. It can be insightful to observe Fielding’s formal description of a resource:

   More precisely, a resource $R$ is a temporally varying membership function $M_R(t)$, which for time $t$ maps to a set of entities, or values, which are equivalent. The values in the set may be resource representations and/or resource identifiers. (Fielding 2000, ch. 5, pg. 88)
The equivalency of value set members is a crucial point.

2. All resources can be interacted with via a standard **uniform interface**, through which the state of a resource can be manipulated.

3. Resources are not accessed directly; rather, all resources contain at least one **representation**, which represents the state of the resource at the time of access. Multiple representations can also be manifested through the use of multiple media types.

4. Requests on resources and responses must be self-contained, i.e. **stateless**.

5. The flow of the system is primarily, if not only, through navigation of **hyperlinks** found in resource representations. Fielding referred to this as **Hypermedia As The Engine of Application State** (HATEOAS).

The idea of applying the principles of REST towards Web services has received great interest throughout the literature in recent years. It has brought about the rise of a new paradigm known as the Resource-Oriented Architecture (ROA) where resources are considered first-class citizens instead of services (SOA) (Richardson and Ruby 2008).

### 2.2 Service Registries

As mentioned before, the search and discovery step is key to the success of Web services. The concept of a registry or repository of service listings has been heavily associated with this so it is important to explore this topic.
2.2.1 Universal Business Registry

Collaboration efforts of enterprises such as Ariba, IBM and Microsoft who were interested in B2B (Business-to-Business) transactions in the late twentieth century resulted in a standard known as the Universal Description Discovery and Integration (UDDI) (Alonso et al. 2004). It served the role of service discovery within the well-known WS-* stack. It was created in order to address the then popular idea of a Universal Business Registry (UBR). As Alonso et al. further note, the initial goal was to support service directories all over the world such that any party could freely publish service descriptions as well as query such directories for interesting services. UDDI was much criticized even in its early days due to the small population of services in such directories (Sleeper 2002). But interestingly enough, Sleeper defended the criticism by pointing out that web services were not intended for public use, but within the restricted enterprise arena. This has unfortunately turned out to be quite the case, as recent literature seems to indicate (Lanthaler and Gutl 2010; Pilioura and Tsalgiatidou 2009). As Pilioura et al. observe, the UBR concept has seen more success in the form of private intranet registries and among few enterprise circles who possess a strong level of mutual concord.

A recent exploratory study of real-world WS-* Web services (Al-Masri and Mahmoud 2008) further confirms this trend of privatizing UBRs. Al-Masri et al. have used their Web Service Crawler Engine to scrape Web services from various sources throughout the Web two times over two years, one in October 2006 and the second in October 2007, leading to two rich datasets that were statistically analyzed. A striking finding from the study is
that while the total number of services in the datasets increased by 131%\textsuperscript{1}, the number of services crawled from UBRs like UDDI grew only by 12.6%\textsuperscript{2} yet the number of services crawled from Web service search engines increased by a whopping 286%\textsuperscript{3}. This interestingly shows a reduced interest in publishing services via registries in favor of search engines. The intuition for this is not hard to grasp; after all, the Web of documents relies completely on search engines in order to search and discover web pages. It could be argued that Web services may be heading this way. But Al-Masri et al. also state that search engines have a tendency to return incomplete or outdated results of Web services and only provide access to WSDL (Web Service Description Language) documents with no context of a business host, as opposed to the business-centric model of UDDI. Even more interesting is their conclusion that “the fact remains that discovering and selecting Web services through a standard, universal access point facilitated by a targeted Web services’ crawler is inherent and inevitable.”

\textbf{2.2.2 Programmable Web}

The ProgrammableWeb\textsuperscript{4} is currently the most popular modern centralized repository of listings of Web services. Some of the information it stores includes service descriptions, their interfaces (APIs) and their mashups. It functions as a portal site where users can register their APIs and mashups, manually search and retrieve API descriptions, and view analytics on APIs and mashups, such as popularity and usage patterns. The literature critiques its minimalistic keyword and category-based search functionality which further

\begin{flushright}
\textsuperscript{1}2199 services in 2006 to 5077 services in 2007. \\
\textsuperscript{2}1248 services in 2006 to 1405 services in 2007. \\
\textsuperscript{3}951 services in 2006 to 3672 services in 2007. \\
\textsuperscript{4}www.programmableweb.com
\end{flushright}
inhibit its ability to support the exploratory discovery of services. While it is a major departure from the intricate UDDI specification and is a modern service registry in that it predominantly houses registrations of RESTful services, it is structured very much like the telephone directory of UDDI. Furthermore, extra manual effort is required of end-users to sift through the vast amount of service listings\(^5\) in order to consume services (Verborgh et al. 2012; Lee and J.-H. Kim 2012; Gomadam et al. 2008; Lee and J.-S. Kim 2012), as opposed to UDDI which natively supports the possibility of automatic service invocation via its dynamic late binding feature (Alonso et al. 2004).

### 2.2.3 Modern Industry

The idea of a central repository or listing of APIs continues to be propagated in the commercial world today. One such solution, APIHub\(^6\), is a central hub for developers to discover and keep track of APIs while service providers can publish their APIs. Providers declare the type of their API (open, public, external), the protocol used to consume it (REST, SOAP, GData, HTTP) the format of the response (JSON, JSONP, XML, RSS) and any authentication/security requirements (OAuth, API Key, SSL support, std user/pass, none). Developers (consumers) can search for APIs by categories (APIHub boasts a rich ontology and taxonomy of describing APIs) (Sharma 2012). They can even interact with some of them and test the various API calls by providing inputs to an interface on the web page and receiving real-time responses.

However, developers will still have to set up a connection with the specific API provider

---


\(^6\)www.apihub.com
that they choose, obtain authentication tokens/API keys, learn the API and implement API calls by themselves in their codebase. In essence, APIHub only acts as a central location or hub for the exchange of APIs. It does not provide a unified interface to all APIs. For example, one can search for photography APIs and view a list of APIs from various providers of image manipulation APIs, however these APIs are separate entities that must be consumed separately. Thus, there might very well be some overlap among APIs.

A solution that is a bit closer to the idea of an entire community of services is the Apigee API platform\(^7\), which is a platform that essentially enables enterprises to create an application ecosystem out of their business data and expose it to developers. The platform provides various services such as analytical tools to gain insights on their data. A more interesting tool is their Gateway Services which the platform uses to create and manage APIs to the data or services of the provider, thereby offloading this time-consuming process. Additionally, their App Services tool provides external (and internal) developers the tools to create new applications using the APIs created by the Gateway Services in tandem with a myriad of the most common applications that Apigee has compiled and integrated with their platform.

A similar approach to Apigee’s App Services is Singly’s SDK/API\(^8\) that integrates and unifies various services such as Facebook, Twitter, and Instagram (they boast integration with over 35 services as of June 2013), which allows developers to smoothly consume these services without having to worry about individually subscribing to each of them. In other words, it connects applications with services. This allows developers to focus on features


for their applications using these services without the need to provide their own solutions. Singly is continually increasing its integration with services upon its customers’ requests.

However, the problem with both Apigee App Services and Singly is that they provide integration with only common or popular services. Additionally, their platforms are closed systems whose growth are controlled by themselves alone. Also, they cater for developers who know which specific service they would like to consume. For example, services such as Facebook and Instagram offer specific functions that cannot by themselves be abstracted by the platforms.

### 2.3 Web Intents

Knowing that many diverse remote services are used on a daily basis, it is quite ideal that web applications would be able to seamlessly integrate with each other. As documented above, this has proven to be quite a challenge largely due to the difficulty of unifying/aggregating new services. In order to address this issue, a framework for seamless client-side service discovery and inter-application communication known as Web Intents (Kinlan n.d.) was conceived. It is modelled after the similarly-named system in the Android platform (Kinlan 2012).

The system involves three parties: Client, User Agent and Service Provider. The Client is a user that is in need of a particular service for its own workflow. It registers this need with the Intents server via the User Agent. This registration is expressed in the form of an Intent, which is composed of an Action and a Type. In advance, Service Providers register with the Intents server and declare themselves capable of handling a specific Action and
Type. From these two parameters, the server creates matches and provides a list of Service Providers to the User Agent, which in turn presents this list to the Client. The Client chooses the desired Service Provider and the chosen Service Provider, then performs the requested action. Input and output between the Client/Service Provider is optional but in most cases will exist.

For example, consider a music website that includes a feature which allows users to share an audio file via some social medium. The web site (the Client) would register an Intent with the Web Intents server, which would accordingly return a list of Service Providers who offer such functionality (such as SoundCloud, ReverbNation, or even Facebook or E-mail). The Action of the Intent would be “SHARE”, and the Type of the Intent would be “audio/mp3” (a MIME type) and the input would be the desired audio file of mp3 extension stored as a binary blob. The User Agent (in this case, the browser) would present this list of Providers to the web site and the user would choose one. Assuming that the SoundCloud service was chosen, it accordingly carries out the sharing operation according to its registered API and perhaps even returns some feedback to the Client, once again via the User Agent.

Web Intents has unfortunately not enjoyed practical use and the w3 specification that was being actively developed has been halted and published as just an informative Working Group Note as of May 23rd 2013 (Billock, Hawkins, and Kinlan 2013).

A few papers that are somewhat unrelated to our research have used Web Intents in interesting ways and may contain some advantageous insights for our own work. An et al. have designed a light-weight network protocol for the Internet of Things/Web of Objects paradigm, where many heterogeneous loosely coupled nodes provide web services to each
other using sensors and actuators and communicate with each other using web APIs (An et al. 2013). They have extended the Web Intents project to suit their own system where each node acts as both a Client and a Provider to others. An Intent server exists in the network and is used by the nodes to perform actions and interact with each other. Since all of this would have to be performed without human intervention, the authors had to automate the Intent registration and Provider choosing steps using an action negotiation algorithm to suit their needs. Lyle et al. use Web Intents along with other elements of their stack for the purpose of discovering and choosing services (Lyle et al. 2013). Their stack aims to address the issue of web browsers accessing services and APIs offered by native devices and in personal networks while avoiding proprietary extensions and maintaining security and reliability.

2.4 Shifting Paradigms

In an effort to update the literature on the modern use of Web services/APIs, Maleshkova et al. have conducted a thorough investigation of the landscape of public Web APIs and have documented their results and analyses (Maleshkova, Pedrinaci, and Domingue 2010). The authors collected a subset of APIs from ProgrammableWeb in 2010 and reported on the description details of APIs, the invocation and authentication mechanisms, input and output parameters, number of operations, number of mashups, dates, the provision of complementary documentation and other general information. An important observation is that of the usage of RESTful APIs. Out of a representative sample of 222 services, RESTful APIs formed only about 32% of the sample. But in a separate 2012 study (Lee and J.-H. 2012).
Kim 2012) of the same directory, the number of representative RESTful services doubled to 72% while the total number of services also increased. This massive shift towards RESTful services from the standardized WS-* stack offers insights towards paradigm preferences among developers.

2.5 Summary

This chapter has provided us with significant insight into the failure of the UBR concept. Various explanations have been proposed throughout the literature yet the most fascinating observation is the consistent lack of coherence between the vision of a global registry of services and the actual research direction undertaken by the industry. We have seen that strong emphasis has consistently been placed solely on advancing specific EAI and B2B use cases, which inhibits the participation of valuable non-enterprise parties in the service ecosystem.

We have seen that the rigid design of UDDI and ProgrammableWeb strongly resembles that of a telephone-directory for businesses. An important observation is that the Web has never worked this way. To drive the point home, consider a thought experiment: imagine if users of the Internet had to sift through a white-pages book in order to locate specific websites. Perhaps an improvement would be a yellow-pages catalogue with deep classification and categorization of websites. Nonetheless, users would still have to manually type the address in order to point their browser to the selected website. They would have to repeat this cycle of tedious search, discovery and manual address entry for every single new website they wish to visit. This thought experiment undoubtedly sounds very
absurd (and it should). We must then remember that the term ‘Web’ in Web services is no coincidence.

The more recent initiative of Web Intents which aims to tackle the issue from a decentralized perspective seems promising but has not been explored much in the literature. Finally, we have seen that there seems to be an increased preference for the application of the RESTful paradigm to that of Web services.

The next chapter contains my proposal for a framework that uses REST in an attempt to address the lack of support for provisioning services beyond the limits of the enterprise.
Chapter 3

Proposed Framework

In this chapter, we will explore how an ecosystem of Web-based services can benefit directly from the underlying architecture of the Web. The proposed framework aims to respond to the failure of UBR by presenting an alternative that is much more natural to the Web. In particular, I argue that a universally open, collaborative library of services founded upon the architectural principles of REST is a strong candidate for the fulfillment of the original UBR vision. After presenting an illustrative overview of the various components of the framework in Section 3.1, the RESTful foundation of the framework is laid out in Section 3.2. Finally, each component of the framework is described in Section 3.3.

3.1 Overview

A big picture overview of the framework is presented in Figure 3.1. The various components of the framework are illustrated from a practical perspective. Each component is elaborated in Section 3.3. But first, the RESTful foundation of the framework will be presented in Section 3.2.
Figure 3.1: Overview of Framework. Arrows indicate requests to the server. Although not shown due to spatial constraints, it can be assumed that the server returns the corresponding responses.
3.2 RESTful Foundation

Since we are not adhering to the WS-* specifications, the first step involves applying certain constraints to the definition of a Web service in order to naturally mesh with REST. An important point to keep in mind is that careful emphasis must be placed to not draw away from the original vision of Web services. The term *Web Service* has been defined at varying levels of specificity throughout its time. Arguably the most specific definition\(^1\) is provided by Webopedia\(^2\) and is associated very tightly with the traditional WS-* stack (see Chapter 2). Recall from Chapter 1 that the least restrictive view of a Web service describes it as an application whose operations can be accessed by other applications over the Web. Our goal is to adhere, at minimum, to this general blanket. I now present my constrained definition of a service within the context of this framework:

**Definition 3.2.1** A *Web service* (referred to in short as a *service*) is defined as a modular piece of code which, like a function, accepts certain input arguments, performs its task based on the inputs, and accordingly produces the output.

In the WS-* stack, it is not an oddity for a business entity to provide a Web service that is a group of every single operation or function that it offers under a single WSDL document. I detract from this notion of an entire application consisting of various operations that either form a cohesive whole or are unrelated to each other. Instead, a service is enforced to focus on a **single operation**. For example, a service that performs a binary search of a given value on a given data would be violating the single operation requirement.

---

\(^1\)As per (Alonso et al. 2004).
if it performed its own sorting routine prior to performing the search. Instead, the service
must abstract away the sorting code to a separate service that is dedicated to the specific
sorting algorithm.

Such modularity encourages reuse and is a well-received software engineering prin-
ciple. This constraint is very similar to the informal description discussed in (Klompmaker
and Reimann 2008) as “every calculation or function that is to be executed on the device”.
This is in the context of a service-oriented framework hybridized with middleware prin-
ciples that aims to simplify the development of mobile-based multiplayer games by encour-
aging functionality reuse. Elsewhere (Bao et al. 2011), it is argued that standard functions
in programming languages are quite similar to services in that a function “receives some
predefined parameters, calculates them\(^3\) and returns an expected result”.

One may observe that the focus on individual operations/functions instead of entire ap-
plications hearkens back to the low-level tightly-coupled RPC middleware (see Chapter 2).
However, RPC mechanisms are founded on distributed shared state. This leads us to our
next constraint, which is the RESTful denunciation of maintaining application state among
queries. It is also loosely inspired by the functional programming paradigm of avoiding
state, mutable data and side-effects. The latter is a little more nuanced and will be touched
on in Section 3.3.2.1.

Now, one may argue that I have in fact creeped out of the blanket definition em-
phasized in the beginning of this section by reducing a Web service from an application
consisting of operations to an operation itself. However, note that we can still consider
a single-operation application to be an application. Furthermore, the modularity offered

\(^3\)Referring to the calculation of the result based on the input parameters.
by this constraint is a crucial enabler of service composition as specified by the proposed framework\textsuperscript{4}.

Thus, we have a constrained definition of a Web service in the context of the proposed framework that stays comfortably within the generalized blanket definition of a Web service. We can now use REST to describe the architectural elements of the proposed framework.

3.2.1 Service as a Resource

Recall from Chapter 2 that in the Resource-Oriented Architecture paradigm, the data that needs to be exposed via services is modelled as the resource, as per the data-driven architecture of REST. Since we wish to offer the service of exposing the entire library of Web services themselves, we refer to services as the resources in the framework. We can formally express a service as a resource using the temporally varying membership function mentioned in Chapter 2. First, note that the formal definition of a function can be paraphrased as a mapping between elements of a set (domain) to the elements of another set (co-domain) \textbf{such that no element of the domain can map to more than 1 element of the co-domain}, while an element of the co-domain may be mapped to by multiple elements of the domain. The notion of a function in most programming languages differs from this mathematical definition in that one-to-many correspondence may certainly exist due to the added parameter of the time of invocation, a good example being a function that returns the current system time. Thus:

\textsuperscript{4}Section 3.3.5 will complete this picture.
**Definition 3.2.2** A service, given its set $I$ of all possible $n$-tuple input parameters, where $n \in \mathbb{N}$, and its set $O$ of corresponding output values, is a resource $S$ such that $R \subseteq M_S(t)$, where $R$ is the function $I \mapsto O$.

This simply means that a service, being a RESTful resource, maps to specific representations based on its potential input parameters and the state of the service at specific instances of time. This temporal variation is very much like that of a function; the retrieved service representation may be different depending on when it was requested.

There are two kinds of representations of services. The first is the service description, which provides various information on the service, including its source code. This is analogous to retrieving a document representation of a resource and is thus very straightforward. Now, recall that a resource representation is simply a sequence of bytes. Thus, the output value of a computation performed by a service can be modelled as the result of invoking the service with a given tuple of input parameters. This service invocation result is the second kind of representation of a service.

Note that service descriptions are directly related to the state of the service. When the description is modified, such as by modifying the source code, the state of the service has consequently been modified. On the other hand, service invocation results never alter the state of a service. They simply reflect the state of the service in relation to its source code at the time of invocation. Lastly, note that the input parameters are obviously irrelevant to the value set of a service when retrieving service descriptions. Nonetheless, they must be included in the definition in order to account for service invocation results.

---

5Furthermore, content negotiation can be used to offer service descriptions in multiple formats.
The above model of service as a resource is key to enabling REST to become the architectural cornerstone of the proposed framework.

3.2.2 Addressing Strategy

Since every resource must be uniquely identifiable, Web services in the proposed framework are no different. The standardized URI scheme\(^6\) has demonstrated to be an excellent fit for uniquely identifying resources on the Web, thus it is reasonable to employ it for the context of the framework. For example, a service which computes the sum of a sequence of numbers may be identified as addnums. Assuming the namespace example.com and the HTTP URI scheme, the complete URI (to be precise, URL) for the service could be http://example.com/services/addnums.

As mentioned before, the invocation representations of a service can be seen as the various output values that are returned based on the provided input parameters\(^7\). Based on definitions 3.2.1 and 3.2.2, we are guaranteed a URI for each tuple of input parameters for any given service. Continuing the example of the addition service, let us assume that we have two cases:

1. Compute the sum of 2 and 3.

2. Compute the sum of 5, 6 and 7.

The corresponding URLs would be:


\(^7\)Long after devising this strategy, I coincidentally found a very similar idea in (Bao et al. 2011), albeit it is employed in the paper for a different purpose.

This example scenario will be used as an illustration medium for the rest of this chapter unless otherwise specified.

### 3.3 Framework Components

Building on the above RESTful architectural foundation, we can begin describing various components of the framework, beginning with the use of the uniform interface in this context.

#### 3.3.1 Uniform Interface

We are now prepared to define the uniform interface as it corresponds to the Web service resources in the proposed framework. A general overview is given in table 3.1.

The following sections 3.3.1.1 to 3.3.1.5 contain some commentary on the tabularized description.

#### 3.3.1.1 GET Method

When retrieving service descriptions, it is possible to support multiple description formats via HTTP content-negotiation. For example, a client requesting the description of a service would most likely prefer a human-readable HTML document if it is a browser, or a machine-processable JSON or XML response if it is an automated crawler or bot. This option is preferably specified via a query string (Richardson and Ruby 2008) but passing
<table>
<thead>
<tr>
<th>Method</th>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GET</strong></td>
<td>/&lt;service-id&gt; [?&lt;field1&gt;=&lt;value1&gt; ...&amp;&lt;fieldN&gt;=&lt;valueN&gt; ]</td>
<td>Retrieve the service description of the service-id, optionally receiving only specific fields as per the optional query string.</td>
</tr>
<tr>
<td></td>
<td>/&lt;service-id&gt;/invoke?&lt;param1&gt;=&lt;value1&gt;.. .&amp;&lt;paramN&gt;=&lt;valueN&gt;</td>
<td>Retrieve the corresponding output from invoking service-id with the given input parameters (param1,...,paramN).</td>
</tr>
<tr>
<td><strong>POST</strong></td>
<td>/&lt;service-id&gt; ?&lt;field1&gt;=&lt;value1&gt; ...&amp;&lt;fieldN&gt;=&lt;valueN&gt;</td>
<td>Update one or more specific field (including source code) of service-id, as specified by the query string, with the corresponding body data of the request.</td>
</tr>
<tr>
<td></td>
<td>/&lt;service-id&gt;/invoke</td>
<td>Retrieve the corresponding output from invoking service-id with the given input parameters passed through the body data of the request.</td>
</tr>
<tr>
<td><strong>PUT</strong></td>
<td>/&lt;service-id&gt;</td>
<td>Register service-id newly with the registry, or replace existing service.</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>/&lt;service-id&gt; [?&lt;field1&gt;=&lt;value1&gt; ...&amp;&lt;fieldN&gt;=&lt;valueN&gt; ]</td>
<td>Delete service-id from the registry or one or more specific fields if provided through the optional query string.</td>
</tr>
</tbody>
</table>

Table 3.1: Uniform interface definition for all interactions with Web services through the registry. Note that all URLs are assumed to be prefixed with http://example.com/services. This was not included due to spatial constraints.

this as metadata through the HTTP request header is also acceptable yet not guaranteed to be honoured due to the quirks of user agent-based content-negotiation.

Note that service invocation is performed via the read-only GET method not only due to the obvious conceptual analogy to reading output values but also because including the tuple of input parameters within the URL (via the query strings) effectively tags that
specific output value with an address. Consequently, output values can then be viewed as subordinate resources to the parent service resource with their own unique identifiers, which is again possible due to definitions 3.2.1 and 3.2.2. Apart from simplified access, the idempotency of GET enables cacheing and will be elaborated on in Section 3.3.2.

However, using the GET method along with query strings for service invocations requires a major trade-off to be made due to the potential for indefinitely large URLs. Interestingly enough, this is stated in the HTTP/1.1 specification as a rare condition. While the specification does not specify a limit, it categorizes this as an erroneous condition because nearly all user agents and servers enforce a limit for practical purposes. Additionally, the specification identifies a scenario where malicious clients can exploit a Web server (or middleware/proxy) that does not enforce such a limit by causing buffer overflows. A naive solution would be to use the POST method and pass the input parameters via the request body. In fact, the response may also be cacheable if the URL is concatenated with a strongly collision resistant hash of the input parameters. However, this contradicts the standardized write-only semantics of the POST method, turning it into a “data-handling process” that is analagous to an RPC operation (Richardson and Ruby 2008). A better solution would be to encourage dividing problems into sub-problems such that services can work on chunks of input data in a streaming fashion. While this initially incurs significant HTTP message round-trip time overhead, response cacheing would offset the overhead in the long-run.

Nonetheless, the proposed framework does not enforce implementations to restrict service

---

8 http://tools.ietf.org/html/rfc2616#section-10.4.15
10 Technically, the HTTP/1.1 standard explicitly includes this as part of the standardized functions of POST. However, this is still discouraged as per Richardson et al.
invocations to the GET method for another reason discussed in Section 3.3.1.2.

### 3.3.1.2 POST Method

As per the HTTP/1.1 specification\(^\text{11}\), some of the standardized functions for the POST method include “Annotation of existing resources”, “Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles” and “Extending a database through an append operation”. In other words, the creation of a subordinate resource (or child resource) is an appropriate operation under POST. While the literature does not explicitly encourage POST as an update/replacement method and generally pushes for using PUT, I argue\(^\text{12}\) that POST is very much appropriate for updating a **specific field** of a service description. Note that a specific field can be uniquely identified by a URL through the query string as a filter when requesting the service description via an HTTP GET (see table 3.1). Thus, the service description as well as its constituent fields can be considered as subordinate resources. Updating a subordinate resource in the context of the proposed framework fulfills the definition of idempotency by the HTTP/1.1 specification\(^\text{13}\). This is because since the subordinate resource contains a unique identifier, the resultant state of the service description will be the same every time the update is performed with the same input parameters. Thus, it is appropriate to use HTTP POST for updating service descriptions and reserve PUT solely for the entire service resource object in the interest of simplified

---

\(^{11}\)http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.5  
\(^{12}\)Loosely based on reasoning provided by (calcote2008putorpost) in Thomas Boutell (2008). *What is the maximum length of a URL.* URL: http://jcalcote.wordpress.com/2008/10/16/put-or-post-the-rest-of-the-story (visited on 04/16/2014)  
\(^{13}\)http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.2
Despite the general frowning on performing read-only operations using POST, this would be necessary in the proposed framework for services which expect input parameters that may be classified as sensitive data. Passing such data through the request header quite intuitively opens up many security holes. Instead, the data should be sent through the request body. Since the HTTP/1.1 specification does not exactly prohibit cacheing POST responses, appropriate invocation representations may indeed be cached as per the method mentioned in Section 3.3.1.1.

3.3.1.3 PUT Method

Consider the following statement by Richardson et al.:

The difference between PUT and POST is this: the client uses PUT when it’s in charge of deciding which URI the new resource should have. The client uses POST when the server is in charge of deciding which URI the new resource (which always turns out to be a subordinate resource) should have. (Richardson and Ruby 2008, ch. 4, pg. 99, comments in parentheses mine)

Hence, the PUT method is used when registering a new service (creating a new resource) or when replacing an entire service with a new one (completely replacing an existing resource).

Note that the source code of a service is still considered a property or field of the service and is thus a subordinate to the service. Hence, it must be replaced/updated only via HTTP POST.

\footnote{Note that the entire description can be updated idempotently as well so long as the query string of the URL contains every single field of the description.}

\footnote{Of course, this does nothing to protect against security holes on the transport level, such as malicious packet sniffing. This may be mitigated by implementing the proposed framework on HTTPS over TLS/SSL.}

\footnote{http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.5}
3.3.1.4 DELETE Method

DELETE is the idempotent antonym of HTTP PUT and POST (when used on subordinate resources) in terms of functionality. The proposed framework provides the feature of deleting specific fields of a service description for certain niche process flows where it may be useful.

3.3.1.5 Other Methods

Other more recently popularized methods such as PATCH are excluded from the interface definition for purposes of simplicity and overall familiarity. However, other standard methods such as HEAD can be safely supported if desired.

Note that content-negotiation can be employed in order to decide between service description formats.

3.3.2 Service Cacheing

One of the best explanations for the inherent scalability of the RESTful architecture is its consequent cacheing mechanism (Davis 2012). This has been applied successfully to the Web via the underlying HTTP protocol, especially the Conditional GET feature (Kurose and Ross 2010) coupled with the entity tag (ETag)\(^{17}\). Cacheing is an extremely important feature of the proposed framework as it drastically reduces the amount of time spent transferring response data or computing results of a service invocation.

\(^{17}\text{http://www.w3.org/Protocols/rfc2616/rfc2616-secl4.html\#sec14.19} \)
3.3.2.1 Service Code

The immediate use-case for caching in the proposed framework is for the purposes of avoiding the retrieval of source code to be executed as part of a service invocation request. This works no different from standard caching of Web resources. As long as the service code has not been modified, the cached code can be considered valid. Hash-generation algorithms employed by version-control systems are very appropriate for use as ETags. As soon as the code is modified, a new ETag must be created and a cache-revalidation directive must be specified in the next HTTP response header.

3.3.2.2 Invocation Results

The most interesting application of caching in the context of the proposed framework is for storing responses to service invocations. However, not all responses can be cached. This leads us to a discussion on referential transparency. According to this principle (Keller and Sleep 1986), an expression will always evaluate to the same value as long as it does not have any side-effects. A function composed of referentially transparent expressions/statements is commonly referred to as a pure function, whereas the contrary case is known as an impure function. In other words, given a specific tuple of input parameters, a pure function will always return the same corresponding value every time it is invoked. For example, a function that performs a mergesort on a given collection of times is deterministically guaranteed to produce the exact same result every time for the corresponding exact same tuple of input parameters\(^\text{18}\). Examples of impure functions include obtaining

\(^{18}\)That is, given the input tuple \((9, 4, 7, 2), \text{ 'ascending' }\), the function \texttt{mergesort(data, order)} will always return the output \(\{2, 4, 7, 9\}\).
the current time and database append operations.

We can extend this concept to the context of the proposed framework. To use the words of Fielding, ‘pure’ service invocation representations are static while ‘impure’ invocation representations vary temporally. Recall that an HTTP server can only issue a status code of 304 (i.e., the cached response is still valid) once it has verified with either the ETag of the response or the last date of modification. The former is out of question in this case because the ETag of the service invocation response can only be computed after the service invocation is complete. At that point, there is no point in issuing the ‘cached’ response since the server has already spent CPU time to compute the result of the invocation. However, observe that as long as the source code has not changed, we can be completely sure that a successful pure service invocation will always produce the exact same response from the corresponding input parameters. The change in source code modifies the state of the service. Thus, we can cache the response by checking if the source code has not been modified; as soon as this does not hold true, a cache-revalidation directive must be specified. This can be easily done by taking advantage of the ETag computed as described in Section 3.3.2.1.

An interesting boundary case that must be considered is that not all functions are exclusively pure or impure. Certain functions may possess a subset of input parameter tuples that result in deterministic output while the remainder may result in varying outputs each time. A good example of this is a function that generates a pseudo-random integer based on a predetermined seed value. A service provider might implement such a service such that if no input parameters are provided, the service uses the value of the current time as the seed; else, the specified seed is used. Since the current time will always be
different, the response for that specific input parameter tuple should not be cached, while the responses for the remaining tuples should be cached. Implementations of the proposed framework must support this feature in order to ensure that service invocation is not just scalable but also results in the correct output every time.

3.3.3 Registration and Storage

In the traditional WS-* stack, there are two approaches to publishing Web services. The centralized approach involves the tModel architecture of UDDI, which was designed such that only references to externally hosted WSDL descriptions are stored in service registries. The decentralized approach, known as Web Services Inspection Language (WSIL) (L.-J. Zhang, Cai, and J. Zhang 2007), takes a document-oriented approach to service representation such that the WSIL descriptions of one service may link to the WSIL description of another service, creating a distributed chain of linked service descriptions and eliminating the need for a central registry. With respect to the centralized approach of UDDI, one of the main findings from the literature review was that both UBRs and search engines often produced stale data. In some cases, the WSDL document would be so outdated that the service was no longer accessible at the specified URL. One explanation for this is that providers of registries do not have control over the actual registered services since they are located externally, i.e. outside of the jurisdiction of the registry. Thus, extra correspondence between service providers and service registries is required in order to ensure that they both are up-to-date with each other. The decentralized WSIL approach does not suffer from this problem due to the absence of a central broker, yielding a peer-to-peer model. However, this very absence causes its own set of problems related to validity and trust.
The proposed framework seeks a balance between the two approaches by storing the entire service data, including source code, within the platform. That is, instead of providing listings of references to services, the platform provides the entire services themselves, thus avoiding instances of broken links and stale information while still serving as the single authoritative registry of services. The registry is ideally implemented as a decentralized storage solution using various data replication and synchronization techniques in order to avoid being performance bottleneck or a single point-of-failure. This is left as an implementation detail. To clarify further, the centralization is not from the perspective of data storage, but rather from the perspective of authority. In other words, there is only a single authoritative ‘registry’ for a specific library of services. How the integrity of service data is specifically maintained across distributed storage nodes is not relevant to the framework from a big picture perspective.

3.3.4 Search and Discovery

A key feature of REST is the HATEOAS principle. Navigation in the standard document Web is driven by this mechanism. To elaborate, whenever the representation of a resource is received, it more often than not contains links to other resources. Upon selecting a link, the representation for the selected resource is retrieved, which in turn may contain links to other resources. This gradual transitioning between application states as opposed to providing access to all application states at any given time is a consequence of the stateless constraint of REST and is also another crucial factor (apart from caching) for the scalability of the Web.

From the literature review, it was found that the effective implementation of HA-
TEOAS for search and discovery inevitably requires venturing into semantic annotations and linked data (Lanthaler and Gutl 2010). For purposes of simplicity, this was left out of the scope of this research and would be suitable for future work.

3.3.5 Service Composition

Service-to-Service (S2S) consumption is the application of the traditional concept of service composition to the constrained definition of Web services. It is a feature that allows services to function as clients to each other and perform invocations between each other. This also encourages modular design. For example, a service that implements the binary search algorithm may invoke another service that performs sorting right before it performs the binary search. Another benefit of this construct is the interesting ability to form recursive invocation patterns. For example, a service that computes the factorial of a given number may do so recursively by consuming itself. For example, the factorial of 5 may be retrieved via the URL /services/factorial?num=5, which would multiply 5 by the factorial of 4 via the URL /services/factorial?num=4, which would multiply 4 by the factorial of 3 via the URL /services/factorial?num=3, and so on and so forth. Lastly, since services act as HTTP clients when retrieving the results of invoking other services, the cacheing mechanism of Section 3.3.2.2 is made available for use via the standardized HTTP cacheing interface. The architecture of the implemented platform should include a distributed cacheing solution that takes advantage of this feature. This server-side native cacheing significantly improves overall performance for invocations of pure composite services for all potential service consumers, as opposed to the typical client-side cacheing that only benefits the particular client.
3.4 Summary

This chapter presented my proposed framework for the provisioning of open community-contributed Web services via application of the architectural principles of REST which have proven successful in the implementation of the Web. Particularly, we saw the constrained the notion of a Web service to that of a self-contained piece of code that is very much like a standard function. We also saw the crucial modelling of Web services as self-contained hypermedia resources that contain two kinds of representations - service descriptions and invocation results. The latter is made possible via a URI scheme involving the input parameters of a service invocation. Being resources, all operations on services can be performed via the intuitive uniform interface, specifically the four standard HTTP verbs. Being modular pieces of code, services are provided with a rich composition pattern by means of retrieving the results of HTTP GET invocations on other services within the system. The ability to cache certain service invocations was particularly highlighted as a strong enabler of performance and scalability. It was also emphasized that the framework currently offers no particular opinion on search and discovery apart from the HATEOAS constraint.

The next chapter describes the implementation of a reference prototype based on the constraints of the proposed framework.
Chapter 4

Prototype Implementation

The second contribution of this thesis is a prototypical proof-of-concept reference implementation of the framework that I have proposed. This chapter presents the prototype and discusses various facets in detail. It also documents various design challenges that were required to overcome as well as any assumptions and tradeoffs that were made. First, the architecture of the implementation is described in Section 4.1. This is followed by a discussion on the rationale behind the implementation language in Section 4.2. A note on the implementation of cacheing is provided in Section 4.3. Finally, the implementation of services in the prototype is described in Section 4.4.

4.1 Architecture

A graphical illustration of the reference implementation architecture is given in Figure 4.1. The following sections discuss the architectural aspects of the implementation and highlight some points of interest.
4.1.1 Overview

All requests are first processed by a single load balancer which distributes traffic across multiple worker nodes/servers. These nodes are identically configured servers that are able to independently process any kind of request, be it registering a service or retrieving the result of a service invocation. A single central service database contains all service data, specifically service descriptions and service source code. All worker nodes must
access this database alone in order to obtain the aforementioned data or contribute new data. All workers also have access to a dedicated central service invocation cache that stores the results of previously completed service invocations. This provides the benefit of significantly reduced response times for composite services due to potentially avoiding many time-consuming service invocations. Each worker server also possesses a distinct in-memory cache that is used for fast access to previously executed source code, allowing for the elimination of repeated database queries.

The overall caching mechanism works as follows. For all service invocation requests, the worker server that has been assigned the request first checks the service invocation cache to see if the result for the request exists, assuming the result is cacheable. If it exists, the result is returned to the client and the request is complete; otherwise, the service must actually be invoked, so the source code is retrieved from the service database. Before obtaining the code, the worker checks its own cache to see if it already contains the service’s source code. If it exists, the cached code is used; otherwise, the code must be downloaded remotely from the service database, after which the code is stored in the worker’s cache for future requests. In either case, the result of the invocation is then stored in the central service invocation cache for future requests and is then returned to the client, successfully completing the request.

It is important to have a dedicated commonly accessible cache for service invocations as opposed to a distinct local cache per node. Once a computation has been performed by any node, all other worker nodes, including future nodes that may be added when scaling horizontally, must be able to benefit from the cached result. One limitation with this approach is that increasing cache storage capacity by adding multiple cache nodes
introduces the problem of lazy/asynchronous data replication. This problem was left out of the scope of this research for purposes of simplicity. On the other hand, allowing each worker server to locally cache source code neatly fits the usage scenario and conveniently allows the service invocation cache to be exclusively used for caching invocation results. Finally, the single central database approach avoids the aforementioned synchronization problem in the case of the service data. Nevertheless, an actual implementation of the framework must certainly consider for its storage configuration multiple service databases and service invocation caches.

### 4.1.2 Hosting Platform

Initially, all the components of the architecture were hosted on a single machine via localhost and multiple ports. While this was enough for certain purposes, a more realistic implementation would leverage multiple machines ideally distributed across the Internet. Instead of constructing a LAN-based data center, the decision was made to take the virtualization route using the Infrastructure-as-a-Service (IaaS) solutions offered by Amazon via their Amazon Web Services (AWS)\(^1\) catalog. AWS offers great freedom in configuring a virtualized distributed architecture using its large array of IaaS services; consequently, the onus is on the user to ensure that the designed architecture is scalable, despite the IaaS services themselves being robust. However, automatic scaling is indeed offered to a certain extent and I found it to be sufficient for the purposes of this research. Another advantage is that the prototype instantly benefits from the security and reliability of the AWS infrastructure. Finally, the just-in-time provisioning of components, intuitive Web-based dashboard

\(^1\)http://aws.amazon.com/
interface and affordable variable cost-model (hourly pay-as-you-go rates) opened up many more options for experimentation. Figure 4.2 illustrates the design of the prototype implementation on AWS. The following sections briefly explain all of the components.

### 4.1.3 EC2

Elastic Cloud Compute (EC2) is the main virtualization solution offered via AWS. An EC2 instance is a virtual machine that runs a specially configured operating system that is pre-packaged as an Amazon Machine Image (AMI). It must be attached to some form
of storage, most commonly an EBS volume (see Section 4.1.5). For the prototype implementation, I strictly used multiple EC2 instances running Ubuntu 14.04 Server Edition to host the various components of the architecture instead of leveraging the corresponding AWS services\(^2\) since the entire prototype was already implemented prior to being hosted on AWS. While the prototype certainly wasn’t able to fully realize the potential of AWS, this allowed me to make the most of my time constraints and did not significantly affect the evaluation process. All EC2 instances were powered by hardware class \texttt{m3.medium}, which offers a single core of a 2007 Intel server CPU clocked at approximately 3.0-3.2 GHz and approximately 3.5 GB RAM.

4.1.4 NodeJS Worker

Each worker node hosts a single NodeJS (version \texttt{0.10.21}) process (see Section 4.2.2) that is capable of independently handling any and all requests. A single Redis (version \texttt{2.8.9}) process was used as the local cache for storing service source code.

4.1.5 EBS Volume

An Elastic Block Storage (EBS) volume is a persistent block-level standard rotating disk storage offered by AWS. Each EC2 instance was attached to an EBS volume of size 8 GB. In particular, the EBS volume for a worker node was pre-installed with NodeJS and Redis. By packaging all of this information into an EC2 AMI, new independent nodes can be easily launched and added to the worker fleet whenever extra compute capacity is required.

\(^2\)For example, instead of using Amazon’s DynamoDB service for the central database storage, an EC2 instance hosting CouchDB (see Section 4.1.8).
4.1.6 Elastic Load Balancer

A single Elastic Load Balancer (ELB) instance controls the distribution of traffic to the fleet of worker nodes. As part of the distribution algorithm, the ELB performs health-checks in order to route traffic to healthy instances. This is performed through a customizable ping, which in the case of the implementation was configured as an HTTP GET to path `/ping` every 5 seconds for each worker node. If the ping request receives a response within 10 seconds, the node is considered healthy and is able to process incoming traffic. If the ping request does not receive a response from the node within the threshold, the timeout is recorded and another ping is issued. This continues a total of 10 times until a response is received. If all 10 pings result in timeouts, then the node is considered unhealthy and is automatically removed from the worker fleet. Apart from the task of adding worker nodes to the load balancing group and the initial configuration of the ping, all load balancing is performed automatically by the ELB instance under the hood. The benefit of AWS was apparent in the fact that not a single modification of implementation code was required to enable load balancing. Finally, note that the traffic distribution algorithm is not of particular interest to this research.

4.1.7 Redis Cache

A single Redis (version 2.8.9) process hosted on a separate EC2 instance serves as the dedicated central service invocation cache, and is used to store the results of service invocations for all worker nodes. Redis is an in-memory key-value distributed cache store

---

3This is also partly due to the centralization of storage and the independence of each worker node in the architecture.

4http://redis.io/documentation
that supports a wide range of data structures such as strings, hashes and lists. It supports data replication, although it is irrelevant to the implementation. The striking feature of Redis is that despite storing data in-memory, persistence can be realized by periodically dumping to the disk. This proved to be extremely useful for the purposes of evaluating the implementation since it allowed the instance to be launched and terminated at will without losing the data cached until then. Finally, an interesting thing to note is that new worker nodes that may be added in the future automatically have access to this cache since the static IP of the instance is hard-coded into the worker node source code that is packaged in the AMI.

4.1.8 Couch DB

Like the Redis Cache, a dedicated EC2 instance hosting a single CouchDB\(^5\) (version 1.5.0) process serves as the central service disk-based permanent storage database. It is a NoSQL data store with eventual consistency, which means that data is synchronized among nodes lazily, sacrificing immediate consistency for vast gains in performance (CouchDB n.d.). Since participating services are not expected to have a well-defined structure to their description, a non-relational data store is a reasonable approach. CouchDB uses a document-model where each database consists of structured data known as documents that are stored as JSON (JavaScript Object Notation) data. Document creation, modification, deletion, retrieval and many other operations can all be performed via a REST API which returns responses in JSON format. All the documents of a database can also be aggregated and specially queried to obtain meaningful information from them using JavaScript view

\(^5\)http://couchdb.apache.org/
functions, which are equivalent to standard map-reduce functionality. As for the eventual consistency feature, it is irrelevant as only a single node is used in the architecture. Finally, just like the Redis Cache, the static IP of the CouchDB instance is hard-coded into the worker node source code, allowing new nodes to automatically have access to the service database.

4.2 Implementation Language

While the ultimate goal is a language-interoperable implementation, this adds a significant amount of complexity to the problem since we would need to support generic interpreters for executing service code written in different languages. For purposes of simplicity, the reference implementation was restricted to support a single programming language, JavaScript. It was also chosen as the sole language to implement the entire application logic. Despite having its own set of quirks and oddities, it is nonetheless a very popular dynamic language used for developing Web applications and possesses a number of strengths that were particularly important in the context of the reference implementation.

4.2.1 Asynchrony

As we wish to embrace the Web as a platform instead of merely a transport medium, every single interaction within the reference implementation system, with a few exceptions, is an HTTP request-response pattern. Additionally, no extra verbs or status codes were added to the HTTP protocol for purposes of simplicity and compatibility. Since HTTP

\footnote{On a light-hearted note, the reasons for why it is still a popular language despite its phenomenal quirks continue to remain a mystery to me.}
interactions form the bulk of operations throughout, we can clearly see that the majority of operations are blocking in nature. For example, consider the common case of retrieving a piece of service code from the database. This is conventionally a sequential operation that would block the server process, rendering it idle until the code is received from the database. In light of such use cases, an asynchronous pattern of program flow based on delayed execution of code via callback procedures is beneficial.

JavaScript is a strong candidate for satisfying the heavy asynchronous architecture of the prototype as it was one of the earliest languages to implement such design patterns that are extremely crucial to the performance and user experience of Web browser interfaces. Graphical User Interfaces (GUI) are well-known for their use of event-driven programming patterns. Generally, such an application would consist of a single thread of execution that continually loops and listens for and handles events that are emitted by UI components. This pattern was extended earlier on to the browser and JavaScript continued to evolve with these patterns in mind\(^7\). Furthermore, readers may be very familiar with the AJAX pattern, which stands for Asynchronous JavaScript and XML\(^8\). AJAX allows JavaScript code that makes requests to external HTTP servers asynchronously and updates a page with the returned response via a callback function at a later unknown time. Without the AJAX pattern, the browser would simply enter a blocking state until the response is received, which is certainly detrimental to the user experience. This is very much like the event-loop pattern of GUIs mentioned above.

\(^7\) Another example of this would be the binding of a listener that reacts to a `onclick` event that may emit from the user clicking on a particular button.

4.2.2 External Libraries

While the choice of language is important, the library support for the language is arguably of more practical relevance. JavaScript is a particularly attractive option in this dimension as well due to a low-level Web framework known as NodeJS\(^9\). It employs a single-threaded event-driven model for handling concurrent requests, very much like the GUI and browser patterns discussed above, as opposed to the more common multi-threaded approach like Apache HTTP\(^{10}\). Expanding on the brief description of the event-driven pattern above, applications in NodeJS are built around handling events that are emitted due to various triggers. These events are handled via callback functions which are executed only when the event is actually triggered and then caught. Until this takes place, the application effectively sleeps. For example, a simple HTTP web server can be designed to respond to incoming HTTP requests from clients. When a client connects to the server, an event is triggered and if it is designed to be handled, the respective callback function is executed. Certainly, this is not a novel concept in the domain of Web servers. In fact, NodeJS was inspired by the already existing Nginx Web server\(^{11}\) which employs this pattern. But what is striking about NodeJS is that it leverages JavaScript’s native asynchronous features such that the simple event-driven pattern actually exists throughout the entire platform. It is not simply a framework for HTTP servers; rather, it functions as a networking extension to the language itself.

---

\(^9\)http://nodejs.org/about/
\(^{10}\)http://httpd.apache.org/
\(^{11}\)http://nginx.org/
4.2.3 Full-Stack Development

The above two advantages of JavaScript, native asynchronous patterns and library support, provide the basis for its third strength: it can be used for full-stack development of Web applications. While languages such as Python may be supplemented by external libraries that allow asynchronous patterns (which are in any case inferior to native language patterns), they do not offer this convenience to the best of my findings. Being able to develop the entire implementation using a single language certainly proved to be useful.

4.2.4 Performance

JavaScript is currently a considerably performant dynamic language as a by-product of the aggressive competition amongst the major browser vendors to increase the rendering efficiency of their respective product. In particular, NodeJS utilizes the V8 virtual machine, which is written in efficient C++ code by Google, as the runtime for executing JavaScript code.

4.2.5 Limitations

Despite its perks, JavaScript does exhibit limitations that are better addressed by other languages. For example, there is surprisingly weak support for multi-threading at the time of writing. Most other languages offer mature libraries for this feature, often being based on the POSIX model. Effective concurrency is an absolute necessity in order to scale well due to the usage scope of the implementation, so the lack of mature multi-threading libraries was initially quite detrimental to the performance of the prototype. This can be
clearly seen in the case of CPU-intensive service invocation requests. Since the NodeJS event-loop operates on a single thread, not much else can be done until the computation is completed, resulting in a long queue of requests to be processed.

One workaround is to fork a child process for each service that is invoked that lasts until the execution of the source code is completed, after which the child process is terminated and deallocated. Input parameters and invocation results can be transferred between the two via inter-process communication channels, such as UNIX sockets. As per (NodeJS n.d.), it costs around 10 MB of memory to spawn a new NodeJS process. This is unfortunately a very high cost compared to spawning a new thread and performing a context switch, reducing the total number of service invocation requests that can be served concurrently by a single worker node. On the brighter side, the high cost is offset by the reliability gains yielded by using Linux processes instead of threads. Currently, if a service invocation fails while executing the corresponding source code in a child process due to the process crashing, the worker node continues normally. In the case of threads, an improperly handled exception could cause the entire worker node’s process to terminate unexpectedly. Furthermore, sharing memory (via threads) can potentially leave room for vulnerabilities that may be exploited by malicious service code.

Nevertheless, the choice of implementation language is not a crucial factor to the development of an implementation of the proposed framework from a big picture perspective. Implementing the prototype in other languages such as Python and Java would certainly offer valuable insights, but this is left for future work.
4.3 HTTP Cacheing

Minimal effort was required to implement the cacheing mechanism described in Section 4.1.1. The built-in HTTP library of NodeJS natively handles ETag generation from responses and the verification thereof. In the case of service composition via invoking other services, extra work was needed to model worker nodes as HTTP clients, however the cacheing framework described by the HTTP protocol was still used with no modifications whatsoever.

4.4 Service Implementation

A total of forty-nine diverse services, including ten special services known as modules, have been created so far. This section briefly discusses how services are implemented in the prototype.

All services are implemented as modular pieces of JavaScript code suitable for execution on a NodeJS V8 runtime. As mentioned in Section 4.2.5, whenever a service needs to be invoked, a separate child process is created, the raw source code is retrieved from the database/local cache and the retrieved code is executed in a sandboxed environment via the vm library\(^\text{12}\). The sandbox is crucial for security purposes because any service source code can potentially contain malicious statements. This is especially true due to the openness of the service eco-system. Only the bare minimum libraries as well as a few helper utilities are made available to the sandboxed execution environment. Apart from preventing malicious code from damaging the system, the sandbox is configured in such a way that external

\(^{12}\)http://nodejs.org/api/vm.html#vm_sandboxes
JavaScript code can be migrated to the prototype as a service with minimal adjustment.

4.4.1 Modules

In order to further ease the migration of external JavaScript code, a new kind of service was created, known as a *module*. It is a service that cannot be consumed (invoked) but offers its source code to other services to build on top of\(^\text{13}\). Module services work very much like the NodeJS module model. Any service that wishes to use a module service must import it using the standard NodeJS `require` function. Normally, this function searches the local filesystem for the specified file which contains the JavaScript code to be imported; if found, the code is loaded into memory\(^\text{14}\) and is appended to the `Global` object of the requesting code’s context. In the case of the prototype, however, modules are stored remotely in the service database. To address this, a mechanism for remotely importing JavaScript modules was created. Specifically, each service invocation sandbox environment is provided access to a function called `require_db`\(^\text{15}\) that abstracts away the details of remotely retrieving the module source code from the database and importing it into the sandbox context. The function accepts as input the globally unique ID of the module service and a callback function which receives as input the imported module object.

This intentionally models the design of local software libraries in order to encourage modular development of services. For example, the prototype currently contains a module that wraps an entire library that provides high-performance functions for matrix and vector operations, known as `glMatrix`\(^\text{16}\). Using the module mechanism, the entire JavaScript code

---

\(^{13}\)Note that modules can indeed build on top of other modules as well.

\(^{14}\)In JavaScript, even source code can be represented as objects in-memory.

\(^{15}\)Source code is listed in appendix B.1.

\(^{16}\)http://glmatrix.net/
was registered as a module service called `glmatrix` without any extra modification to the source code. A standard service for performing chain multiplication of any number of 3-dimensional square matrices called `multiplymatrix3` was registered as a dependent of `glmatrix`. Its source code is shown in Listing 4.1. Note the use of the `require_db` function in line 3 to import the module service code. Finally, note that modules can also import other modules, which may also be dependent on other modules, and so on and so forth.

```javascript
if (!sandbox.args || sandbox.args.length < 2) sandbox.res("No/Insufficient input provided.").error(400);
else {
  require_db('glmatrix', function(err, required_module) {
    if (err) return sandbox.res(err).error();

    var lib = required_module.mat3;

    // First, add the first two matrices...
    var result = lib.mul([], sandbox.args[0], sandbox.args[1]);

    // ...then add any remaining ones
    var n = sandbox.args.length; // total number of operands
    for (var i = 2; i < n; i++) {
      result = lib.mul([], result, sandbox.args[i]);
    }

    sandbox.res(result).json().send();
  });
}
```

Listing 4.1: Implementation of the matrix chain multiplication service

An interesting problem to investigate for future work is whether the module mechanism can be abstracted away into the framework as a component that can work regardless of the language of the source code.
4.4.2 Service-to-Service Consumption

Recall that the framework facilitates the creation of composite services via the standard orchestration technique of services invoking other services. In the prototype, this was implemented as a helper utility that is accessible to all services via a simple Javascript function known as `consume_service`\(^{17}\). It provides an intuitive abstraction for retrieving the result of invoking any service registered with the system that is not a module. This takes place in the form of a new HTTP GET request, the invokee acting as a client to the invoked service. The result of the invocation can then be used by the client service to perform its tasks. Furthermore, since the function simply performs a separate HTTP GET request, there is nothing stopping a service from invoking itself. This allows for the recursive pattern described in Section 3.3.5.

All of this further enhances the modularity and reuse of services. Consider as an example the recursive implementation of a service called `levenshtein` that computes the Levenshtein distance between two strings. The source code for the service is provided in Listing 4.2. Note the series of recursive service invocations in lines 24 to 41. Also, observe that another service called `minNums` is invoked in order to compute the minimum of a list of numbers. Certainly, such a function could be re-used in so many other situations apart from computing Levenshtein distances. Hence, it makes sense to implement it as a separate service that can be invoked by any other service at will.

\(^{17}\)Source code is listed in appendix B.2.
```javascript
/**
 * Code adapted from recursive pseudocode given at http://en.wikipedia.org/wiki/
 * Levenshtein_distance#Recursive
 * Accessed on May 1, 2014
 */

if (!sandbox.args || sandbox.args < 2)
  sandbox.res("No/Insufficient input provided.").error();

var s = sandbox.data.s || sandbox.args[0];
var length_s = s.length;
var t = sandbox.data.t || sandbox.args[1];
var length_t = t.length;

// Base case: empty strings
if (length_s == 0)
  sandbox.res(length_t).integer().send();
if (length_t == 0)
  sandbox.res(length_s).integer().send();

// Test if the last characters of the strings match
var cost = s[length_s-1] == t[length_t-1] ? 0 : 1;

// Compute the minimum of deleted character from s, deleted character from t and
// deleted character from both
consume_service('levenshtein', {s: s.substring(0, length_s-1), t: t}, function (err, status, from_s) {
  if (err) sandbox.res(err).error();
  from_s = parseInt(from_s, 10);
  consume_service('levenshtein', {s: s, t: t.substring(0, length_t-1)}, function (err, status, from_t) {
    if (err) sandbox.res(err).error();
    from_t = parseInt(from_t, 10);
    consume_service('levenshtein', {s: s.substring(0, length_s-1), t: t.substring(0, length_t-1)}, function (err, status, from_both) {
      if (err) sandbox.res(err).error();
      from_both = parseInt(from_both, 10);
      consume_service('minNums', [from_s + 1, from_t + 1, from_both + cost], function (err, status, min) {
        if (err) sandbox.res(err).error();
        min = parseInt(min, 10);
        sandbox.res(min).integer().send();
      });
    });
  });
});
```

Listing 4.2: Implementation of the Levenshtein string distance calculation service
4.5 Summary

This chapter has presented a detailed look into the implementation of the prototype. Although the prototype does not yet implement all components of the proposed framework, the main features were captured with the aim of exploring the runtime feasibility of such a system. Thus, more emphasis was placed on service invocation performance/scalability and programming abstractions for implementing new services, while lesser emphasis was placed on search and discovery.

The following chapter continues this thesis by documenting the evaluation of the prototype.
Chapter 5

Evaluation and Results

The final contribution of this thesis is the empirical evaluation of the reference implementation of the proposed service provisioning framework. We explore the behaviour of such a system in order to infer the feasibility of the framework itself. In this chapter, a discussion on feasibility and its translation to evaluation metrics is presented in Section 5.1, followed by a detailed documentation of the design and rationale behind all experiments and methodologies in Sections 5.2.1 and 5.3.1. The results of evaluation are then presented in Sections 5.2.2 and 5.3.2 and the chapter concludes in Sections 5.2.3 and 5.3.3 with my analysis and interpretation of the data.

5.1 Feasibility

For all practical purposes, a model or framework is ultimately ‘viable’ or ‘feasible’ only if it is able to solve existing problems with statistically significant improvements and is also attractive to adoption by the industry. While the former is fairly straightforward in that it involves the inference of quality via analysis of quantifiable data, the latter is a greater challenge because it is arguably very difficult to predict the success of technology, even when it is promising. As we have seen in previous chapters of this thesis, this was
precisely the case for the WS-* stack. While it is indeed a valid implementation of SOA and is still effective in complex private enterprise contexts, it has ultimately failed to gain the worldwide traction that was initially optimistically envisioned. While there are indeed various explanations and models for the prediction of adoption, they currently remain out of the scope of this research. In any case, the former quantitative analysis is of prime and foremost importance and that is the focus of this thesis. We will first discuss the qualitative requirements germane to service-oriented systems that exist in the literature.

5.1.1 Quality-of-Service

The concept of Quality-of-Service (QoS) is very prevalent in the task of discovering relevant services. It is based on the well-known networking concept of providing previously agreed upon contracts that guarantee a certain level of a parameter. Services may evaluate themselves on such different parameters and may then offer guarantees on meeting these requirements (Alonso et al. 2004; Singh and Huhns 2006; Kurose and Ross 2010). These quantitative measures can be used as advanced filters in the automatic search and discovery process. As far as I am aware, the first fleshing out of these measures with respect to Web services was given by Mani and Nagarajan in their IBM whitepaper (Mani and Nagarajan 2002). Their work was later included in the textbook by Singh and Huhns.

Instead of using QoS as a parameter of service search and discovery, we can use it in the evaluation of the reference implementation itself. Of the nine described parameters, we may be initially interested in availability, accessibility, integrity, performance, reliability and security.
Availability

The measure of the ability of the service to meet its own responsibility of being ready, i.e. available to serve clients during its offered times of service.

Accessibility

The likelihood that the service can actually be reached despite being available due to the impact of external factors such as network congestion and high demand; in other words, the scalability of the service.

Integrity

The correctness of transactions with the service, which can be viewed as analogous to the database concept of atomic transactions. We can also consider the computational correctness of the service, i.e. whether it provides the correct result.

Performance

This measure is particularly interesting due to its multi-dimensional nature. There are two main metrics: throughput and latency/response time; the former being the number of requests the service can handle in a specific time period; the latter being the round-trip time (RTT) of a single service invocation operation.

Reliability

Being closely related to availability and accessibility, it measures how often a service succeeds in providing its offered level of service.
Security

This measure is very broadly scoped, encompassing features such as confidentiality of information and nonrepudiation/authentication of clients.

5.1.2 Implementation Evaluation

Despite having a fair amount of overlap among each other, the above six measures of quality offer distinct perspectives on service evaluation. In particular, integrity and security carry their own unique set of challenges that warrant separate research efforts and are thus left for future work. This thesis is primarily concerned with the foundational performant feasibility of the proposed framework. As such, the measures on which I have based the implementation evaluation are performance, availability and reliability, with an extended investigation of performance. The literature identifies these measures as the runtime features of QoS (Ran 2003). Note that scalability is also a runtime feature yet it is discarded from this particular evaluation due to the scope of the prototype.

Recall that although in the context of Web services these measures are used to evaluate individual services, we are on the other hand interested in evaluating the entirety of the implemented eco-system of exposed services as a whole. But since these measures are actually abstract concepts of distributed systems QoS in general, they are valid measures for evaluating the reference implementation. Furthermore, since the entire eco-system of community-contributed services is exposed via a single RESTful API, the use of the above QoS metrics is further validated.
5.1.3 Operational Metrics

In order to model the four selected measures, we need to represent them as specific operational metrics/parameters. The notion of availability has been defined by Bellido et al. in the context of a RESTful system based on HTTP status codes. In accordance with Mani et al., performance is defined in terms of throughput and response time. Finally, scalability is measured in terms of the preceding metrics, similar to the approach by Ran.

5.1.3.1 Availability/Reliability

In their paper, Bellido et al. have analyzed the application of well-established control-flow patterns of traditional service composition in the context of RESTful services (Bellido, Alarcón, and Pautasso 2013). They specifically argue that the centralized nature of traditional choreography and orchestration based on a single component is far removed from the stateless and decentralized context of modern services based on the REST architecture. In the evaluation of their implementation, they agree with the standard metric of availability as the total number of successful service invocations in proportion to the total number of requests. But they interestingly define this in the context of REST services using HTTP status codes. Specifically, any 2xx (success) or 3xx (redirection) response is classified as a successful invocation, while 4xx (error in client’s request) and 5xx (error from server) responses are considered as failed requests.

This definition of availability meshes well with the reference implementation since it is based on the HTTP application protocol. However, certain distinctions must be made. Firstly, failures resulting from the client-side are technically irrelevant to the actual mea-
sure, so we can discard 4xx responses from the evaluation. Secondly, on the occasion of service-responsible 5xx errors, clients will not be offered the opportunity to retry their request. This constraint is applied for purposes of control and simplicity.

With this approach of using standard HTTP status codes, we can obtain a metric that combines both availability and reliability. Specifically,

\[ A = \frac{N_{\text{Success}}}{N_{\text{Total}}} \]

where \(N_{\text{Success}}\) is the number of service invocation requests that yield a 2xx or 3xx code and \(N_{\text{Total}}\) is the total number of service invocation requests. In the remainder of this chapter, I will simply refer to this metric as “availability”.

### 5.1.3.2 Throughput and Response Time

It is very straightforward to measure the throughput (TP) as the capacity of the service to successfully serve client requests within a specific amount of time. We can be more specific by drawing from the availability metric defined in Section 5.1.3.1. Specifically, the total number of requests that result in a 2xx or 3xx response code over a specific period of time.

As per Ran, the response time (RT) is the average, maximum or minimum time taken to complete a request as guaranteed by the service provider (Ran 2003). However, I measure it as the difference between the time at which the request was initiated by the client and the time at which a successful response of HTTP status code 2xx or 304 (Not Modified) was received and acknowledged by the client. Out of the 3xx redirection codes, only 304 is considered a ‘success’ because it signifies that the cached response is still valid. Most
of the other 3xx codes are directives for explicit redirections. Thus, the response time will include all time spent during the transit of data including all redirects that need to be followed, until one of the aforementioned success codes is encountered. The rationale for this is very intuitive from the point of view of the client/user who, all things held constant, is not concerned about the intricate complexities of the flow of data as long as the response is received in a reasonable amount of time.

5.1.4 Reference Benchmark

In order to make a reasonable judgement on the feasibility of the framework, it is important to have an external reference benchmark to compare with the evaluation results of the implementation. Unfortunately, there is no particular prior work that is a good enough fit for such a role. Firstly, the vast majority of research involving performance evaluations has been performed on work related to the WS-* standards as opposed to RESTful Web services. Secondly, although the entire platform is itself a Web service, it is very different from a typical Web service so benchmarks compared against QoS measures of known Web services is not a valid method. A better method would be to compare the performance of the implementation with the performance of UBRs such as UDDI, however this is also not valid since the UDDI API is only limited to publishing, searching, and other functions that are not related to service invocation. A particular exception to this would be that of dynamic late binding, however this would have to also include the search and discovery step, which is not a part of the service invocation process in the proposed framework.

Due to the lack of a suitable reference benchmark, the performed experiments will be explorative in nature as opposed to comparative. Nonetheless, this should be sufficient to
gain insight into the feasibility of the proposed framework.

5.2 Cacheing

A small introductory experiment was performed in order to gain some perspective on the crucial role of cacheing in the service invocation process.

5.2.1 Experimental Design

The Fibonacci service, located at /services/fib, was created in order to perform the cacheing experiment. It calculates the value of the given input index within the Fibonacci sequence of integers. It expects a non-negative integer as input and produces a positive non-zero integer as output. The following is typical pseudo-code for the recursive implementation of the Fibonacci sequence:

```python
# Assume n >= 0 always.
def fib(n):
    if n < 2
        return 1
    else
        return fib(n - 1) + fib(n - 2)
```

While this is normal for a regular local implementation, such an implementation on the prototype would be a problem as it is a strongly CPU-bound problem, particularly due to the large number of recursive function calls. However, observe the following pseudo-code in the context of the prototype:

```python
# Assume n >= 0 always.
def fib(n):
    if n < 2
        sendResponse(1)
```

63
else
    consumeService('fib', n - 1, def (output1)
    consumeService('fib', n - 2, def (output2)
    sendResponse(output1 + output2)

The `consumeService` function is an implementation of the Service-to-Service consumption concept mentioned in Section 3.3.5. It expects three input parameters: the unique ID of the service, any optional input parameters, and lastly a callback function to be executed upon completion of the consumption. The result of the S2S consumption is passed as a parameter to the callback function and can then be used for whatever purpose. This S2S function can also be nested, i.e. called within the callback of another S2S function. This takes advantage of the event-driven style of programming that is a hallmark of frameworks like NodeJS. The `sendResponse` function simply relays the final output to the original client.

Note that `consumeService` is simply syntactic sugar for a normal HTTP GET request. In other words, `consumeService('fib', n - 1, ...)` is simply `GET /services/fib/result?index=<n-1>`, where `<n-1>` varies according to the corresponding recursive iteration. Consequently, a sizeable number of iterations will be avoided as previously cached responses will be employed. Coupled with the stellar ability of NodeJS to efficiently handle large numbers of concurrent HTTP requests due to its single-threaded event-loop model, we can expect long-term reasonable performance from the prototype compared to a static native solution.

The full implementation of the Fibonacci service is listed in Appendix C.
5.2.1.1 Methodology

The prototype implementation was compared with a native ANSI C implementation of the Fibonacci sequence function, based on the standard pseudo-code in Section 5.2.1.

Two sets of experiments were performed: native implementation and prototype implementation. For each case, the values for the indices between 35 and 45 inclusive were calculated. This corresponds to 11 runs (representing the 11 values) for each case. For each run, 10 computations were performed. The average of the 10 repetitions forms the observation for the run. In both cases, the metric to be measured is the time spent from the execution/invocation of the function/service until the receipt of the resultant final value. For the native implementation, this is known as the execution time and for the prototype it is known as the response time. Finally, the service invocation cache was reset for every single repetition. This was done in order to offer a fair comparison with the native implementation.

5.2.1.2 Test Environment

The native C implementation was compiled on a Linux Desktop installation running Ubuntu 12.04 based on kernel version 3.2.0-59-generic-pae. The processor was an AMD Deneb-family clocked at 3.0 GHz. The standard GCC compiler at version 4.6.3 was used with no explicit compiler optimizations. The function was called from a test file which performed all the necessary runs and dumped the output to a file. Execution time was calculated by computing the difference of timestamps at the start and end of execution via the clock() function from time.h.
The prototype implementation was hosted at localhost on a MacBook Pro with a Intel Core i7 mobile processor clocked at 2.9 GHz, running OSX Mavericks, NodeJS version 0.10.21. The CouchDB database and Redis store were also hosted on the same machine. The decision to host the server on a machine inferior to the Desktop was made in order to press any CPU-bottlenecks of the prototype even harder such that the benefits of cacheing would be more easily visible.

5.2.2 Results

The results of the cacheing experiment are summarized in Table A.1 in Appendix A.1. The graphs in figure 5.1 provide a more illustrative description of the data.

5.2.3 Analysis

We can observe some very interesting patterns in the data. Firstly, it is plainly obvious that the native implementation performs better overall when simply considering the average over all runs of about 4.5 seconds. However, this is very misleading. Upon closer inspection, we can see that the average is greatly skewed by the significant majority of the runs being quite low. As the runs increase, the execution increases at an alarming rate. In fact, the blue graph in figure 5.1 roughly looks like an exponential function. This is no surprise since that approximately models the time complexity of the recursive Fibonacci sequence. But comparing this with the prototype implementation, we see a completely different picture. While the response time is quite large right from the beginning (the minimum is around 12 seconds compared to the minimum of 0.2 seconds for the native implementation), it consistently grows at a very miniscule rate of only about 3% on average compared
Figure 5.1: Native C and prototype performance of computing Fibonacci sequences between 35 and 45 inclusive. The pink graph represents the prototype and the blue graph represents the native implementation. **Lower times are better.**

to the whopping 63% of the native implementation. Since the prototype implementation grows at a much slower rate than the native implementation, they both converge quite dramatically just after the run at index 44. This is clearly illustrated in figure 5.1. Assuming that this slow rate of growth continues well beyond the sampled runs\(^1\), the prototype implementation is clearly a much better candidate for this particular set of experiments.

\(^1\)The maximum variance over the sets of 10 runs for all 11 indices for both the native C implementation and the prototype was observed as 0.1002 s\(^2\), the minimum approaching even closer to 0. This minimal deviation from the average for each index leads me to believe that the prototype will continue to grow slowly even beyond the sample indices of the experiment assuming, of course, that it maintains its theoretically free caching mechanism.
If both the implementations are algorithmically the same, why does the prototype implementation offer much better time complexity and hence scalability? The obvious explanation is the crucial leveraging of HTTP cacheing for service invocations. Since the recursive algorithm for Fibonacci must frequently traverse previously computed nodes and recompute them, the ability to cache previous computations culls away many computations. This cacheing mechanism is quite analogous to the concept of memoization in dynamic programming. While the native C implementation can definitely be modified to support memoization, and will consequently always perform better than the prototype, its Achilles’ Heel would be the trading of space for CPU time. It would not scale well due to the relatively restricted constraints on space in local devices (particularly for mobile/ubiquitous devices). On the other hand, the proposed framework is particularly designed to scale well when exchanging space for time due to leveraging a sophisticated distributed data storage strategy. Furthermore, in the context of this particular prototype implementation, event-loop Web servers like NodeJS are excellent at handling large volumes of I/O requests and crumble when performing time-consuming CPU-intensive tasks. Hence, it is even more advantageous for them to trade away CPU time as much as possible.

5.3 General Performance

The main experiment evaluates the performance of the entire system on grounds of throughput, response time and availability.
5.3.1 Experimental Design

This experiment was performed in order to obtain overall statistics of the implementation. A total of six diverse services were used as follows in order to obtain a realistic usage scenario:

1. Fibonacci. CPU-intensive; recursive; I/O-intensive due to recursive requests.

2. Levenshtein Distance. CPU-intensive; recursive; I/O-intensive due to recursive requests; I/O-intensive due to invocation of another service for computing the minimum of a list of numbers.

3. Multiply $3 \times 3$ Matrices. CPU-intensive; single invocation.

4. Google Static Maps. I/O-intensive due to invocation of an external service; transfer of binary image data

5. AES Encryption. CPU-intensive; single invocation.

6. AES Decryption. CPU-intensive; single invocation; data used corresponds to data used in AES Encryption.

These services are described in detail along with their source code in Appendix C.

5.3.1.1 Methodology

The ideal method of testing overall performance is to simulate some sense of realistic traffic. In order to do this, a script was developed (presented in Appendix D.1 for generating such simulations.)
First, different test cases were created by randomly generating input parameters for each of the six services mentioned in Section 5.3.1. A total of 500 requests, each with randomized parameters, were created for each service yielding a total of 3000 requests.

Second, the entire set of 3000 requests were shuffled randomly into a queue such that each request had equal likelihood to be issued. However, once a request was completed, it would not be returned back to the queue.

Third, all requests were issued to the implementation server in an asynchronous fashion, i.e. the entire queue was sent to the server as opposed to waiting for the response of each request before issuing the next request. This allowed a more realistic simulation of server load, giving us some insight into scalability in terms of ability to handle many concurrent requests. Finally, the response of each request was written into a CSV file storing the invoked service, response code, response time, corresponding response message (including error messages) and the request URL (which documents the input parameters). Note that the response was not stored because the services were verified in advance that they return the correct response whenever they issue a status code of 200.

The entire process was repeated a total of five times in order to calculate average statistics.

5.3.1.2 Test Environment

The implementation was hosted on Amazon’s Amazon Web Services (AWS) via the Elastic Compute Cloud (EC2) service, which is a virtualization platform. A total of six instances were created: four worker servers, one cache server and one DB server. Each instance hosted a Linux Server installation running Ubuntu 14.04 based on kernel ver-
version 3.13.0-24-generic. All instances were of class m3.medium, which sports a single core 2007 server CPU rated at about 3.0-3.2 GHz, about 3.5 GB of RAM and 4 GB persistent storage on a single SSD drive. It would have been more favorable to choose an instance class that sported a multi-core CPU however the implementation was not optimized to take advantage of multiple cores, hence the benefits would be negligible. The four worker servers were running NodeJS version 0.10.25 and were exposed to the Internet via a single load-balancer using Amazon’s Elastic Load Balancer (ELB) service. All worker servers accessed the same cache and DB storage, located separately on the respective instances. The cache server was running Redis version 2.8.9 and the DB server was running CouchDB version 1.5.0. All instances were fresh installations and no other user processes were running in the background. The cache was emptied at the very beginning of the experiment, but not at the beginning of each run. In other words, the cache from the previous run carried forward onto the next run.

The test script was executed on the same Desktop mentioned in Section 5.2.1.2.

5.3.2 Results

The response times of the five runs including the average across the five runs are presented in Table A.2 in Appendix A.2. The throughput of each run was calculated as follows:

\[ TP_{run} = \frac{3000}{T_{run}} \]

where \( TP_{run} \) is the throughput measured for the particular run and \( T_{run} \) is the time taken to complete the entire run\(^2\). The measured data is presented in Table A.3 in the same

\(^2\)Note that a total number of 3,000 requests were issued for each run, as per Section 5.3.1.1.
Figure 5.2: Histogram of the aggregation of all five runs of the general performance experiment.

Appendix.

Although not documented in this thesis, all responses had a status code of 200 OK, indicating that every single invocation, across all 5 runs, was successful. Thus, the prototype exhibited availability $A = 1.0$ in this particular experiment.

The histogram of the aggregation of the five runs (a total of 15,000 requests) plotted in figure 5.2 provides an illustrative view of the response time distributions.
5.3.3 Analysis

The first observation is that the minimum response time of the first run was much higher than that of all the other runs. This is in large part due to the cache being initially empty. Services that extensively depend on previously found solutions like Fibonacci and Levenshtein Distance will be affected by this initially. Nonetheless, runs 2 through 5 depict the usual situation where the cache contains some pre-existing values prior to the invocation of the service.

The prototype performed well in terms of availability. All responses across all runs were successfully completed with a 2xx status code. However, this does not give us much insight into the general case since multiple loads were not compared. Due to time constraints, this has been left for future work, thus no inference can be made currently on the availability of implementations of the proposed framework.

The average total time for performing all 3,000 service invocations was approximately 15 minutes yet the average time for a single invocation was clocked at approximately 8 minutes, i.e. a little over half of the total time. Note the very high average standard deviation of approximately 4.5 minutes. This is evident from the vast average difference of 16 minutes between the average shortest response time and the average longest response time. This difference is in fact approximately equal to the average total time, indicating a very large spread of values. This can also be gleaned from inspecting the histogram of response times across all 15,000 requests in Figure 5.2, which illustrates this spread. This implies that there is a good chance that a service invocation could take anywhere between 1 second and 16 minutes, regardless of which service was being invoked. With such an
unreliable range of response times for any service invocation request, we can conclude that the prototype performed quite poorly.

Due to the nature of the experiment, the worker servers were definitely under heavy load. What we can glean from this is that the implementation did not perform well under a constant load of requests. One possible explanation is the fact that the implementation, based on NodeJS, was not able to take advantage of effective multi-tasking via threads. Firstly, the spawning of a new NodeJS process adds a guaranteed 30-50 millisecond overhead to the response time. Additionally, performing context switches between threads of a single process is faster compared to between separate processes, i.e. switching between threads of different processes. An important future work would be porting the prototype implementation to a multi-threaded architecture and comparing the performance.

5.4 Summary

This chapter presented a detailed description of the experiments performed to evaluate the reference implementation. The results of the experiments were also documented and corresponding analyses were performed. Some important observations were made.

- The caching of invocation responses clearly demonstrated a significant improvement even in comparison to a native implementation. Being able to avoid performing potentially long calculations on previously computed requests saves a lot of time and allows the implementation to scale better. The main reason being the problem of exchanging space for time is negated since space is abstracted away from the problem due to the use of the distributed caching system. Thus, the implementation
would excel in domains that involve many dynamic programming problems, such as Evolutionary Computing and Genetic Algorithms.

- The average response time of approximately 8 minutes along with the very widely spread distribution of the response times across the five runs of the general performance experiment show that the prototype implementation did not perform well under load. This was hypothesized to be due to the lack of multi-threading in the implementation architecture. Further experiments that compare this performance with multi-threaded implementations of the framework were recommended.

The next chapter draws this thesis to a close by summarizing all findings and adding further thoughts as appropriate.
Chapter 6

Conclusion and Future Work

The remarkable idea of components of a system distributed all over the Web yet seamlessly communicating with each other in order to perform tasks showed promising signs of fruition with the advent of Web services. We have seen, however, that despite significant research efforts the industry has not yet reached any sort of consensus regarding the provisioning of services. The various issues regarding the complex specifications known as the WS-*, particularly the concept of UBR and the UDDI registry, were briefly discussed in Chapter 2. We have also seen that service providers and consumers have been increasingly interested in a new paradigm for Web services known as Resource-Oriented Architecture which is based on Fielding’s Representational State Transfer architectural proposition for the HTTP protocol, on which the Web was built. Lastly, we have seen that quality software no longer solely emerges from the enterprise sector. The collaborative development style that is prominent within the open source community has yielded its sizeable share of valuable software. These observations led to the interesting question of whether the concept of Web services could be aided by the collaborative software development.

This thesis presents the results of my exploration into applying REST to the problem of provisioning Web services contributed by an open and collaborative community of

\[1\] In fact, almost all research pertaining to this thesis was conducted with the aid of third-party open source tools.
providers and consumers. The main contributions are as follows:

- A framework for the provisioning of community-contributed Web services based on the REST paradigm.

- A prototype implementation of the framework that may be used as a reference for other implementations.

- A Web interface to the prototype that demonstrates practical usage of the framework.

- Evaluation of the performance of the prototype that offers insights into the feasibility of the framework.

While exploring REST, the cacheing feature of the HTTP protocol was identified as one of the main reasons for the natural scalability of the Web. It was highlighted as a particularly attractive application to the framework with respect reducing the number of times services need to be actually invoked to provide results to previously requested input parameters. The main experiments performed for the prototype evaluation did indeed demonstrate the significant benefits of cacheing results of ‘pure’ services. Specifically, it was seen that while the actual response times were relatively high compared to the execution time of native local implementations of the tested services, the rate of increase in response time as the computations increased in intensity were relatively much lower, such that eventually the response time of the tested service implemented in the prototype would be lower than the execution time of the native implementation. However, this was only true for those services which performed repeated computations of previously computed results, i.e. overlapping subproblems. Thus, it was inferred that the framework would be particularly effective in
terms of performance when services are of the dynamic programming problem domain. This is because the cacheing mechanism of the framework is very much like the memo- 
ization optimization technique of dynamic programming problems which trades space for 
time. The main benefit of the framework is that the space theoretically comes for free, as 
it is abstracted away from the execution environment into the implementation architecture 
itself.

Further experiments were performed to gain insight into the general performance of 
the prototype. The most striking result was the very deviation from the mean as well 
as the wide spread of response times for a service invocation request. This showed that 
the chance that the observed response time could be anywhere between the lowest and 
highest response time was quite high despite the specific service that was being invoked. 
The best explanation that was proposed was the lack of multi-threading support in the 
implementation. However, more experimentation is required to confirm this, particularly 
by comparing with a multi-threaded implementation.

6.1 Future Work

Many interesting problems that were out of the scope of this thesis remain open to 
research. Some of them are listed below:

- Examining the difference in performance of implementing the service invocation 
  component of the framework via a multi-threaded approach compared to the single-
  threaded event-loop approach of the prototype.

- Performing evaluation of the facets of feasibility other than performance.
• Exploring the Semantic Web and leveraging it in the proposed framework in order to enhance the search and discovery process.

• Enabling cross-language interoperability. Specifically, allowing developers to contribute services written in any language of their choice.

• Investigating the application of the HATEOAS principle to automated service composition and invocation (Parastatidis et al. 2010).

• Creating strategies for addressing potential conflicts with source code licensing.

• Implementing processes for evaluating both functional and non-functional QoS metrics. Particularly, determining if a particular service has been implemented correctly is a very interesting problem.
Bibliography


Alonso, Gustavo et al. (2004). Web services: concepts, architectures and applications. Springer.


Kinlan, Paul (2012). *Web Intents*. Online collaborative source code repository using the Git version control system. Version b12e8a798c2eaac06e274882412a507b104d8a90. URL: https://github.com/PaulKinlan/WebIntents#web-intents (visited on 06/25/2013).


Appendix A

Experimental Data

This appendix contains a tabulated presentation of the results of the experiments mentioned in chapter 5.
A.1 Cacheing

<table>
<thead>
<tr>
<th>Run</th>
<th>Average Execution Times</th>
<th>Percentage Increase Between Runs</th>
<th>Run</th>
<th>Average Response Times</th>
<th>Percentage Increase Between Runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>0.1430</td>
<td>0.0000</td>
<td>35</td>
<td>11.3878</td>
<td>0.0000</td>
</tr>
<tr>
<td>36</td>
<td>0.2310</td>
<td>0.6154</td>
<td>36</td>
<td>11.6974</td>
<td>0.0272</td>
</tr>
<tr>
<td>37</td>
<td>0.3730</td>
<td>0.6147</td>
<td>37</td>
<td>12.1412</td>
<td>0.0379</td>
</tr>
<tr>
<td>38</td>
<td>0.6030</td>
<td>0.6166</td>
<td>38</td>
<td>12.4288</td>
<td>0.0237</td>
</tr>
<tr>
<td>39</td>
<td>0.9780</td>
<td>0.6219</td>
<td>39</td>
<td>12.6934</td>
<td>0.0213</td>
</tr>
<tr>
<td>40</td>
<td>1.5820</td>
<td>0.6176</td>
<td>40</td>
<td>13.0660</td>
<td>0.0294</td>
</tr>
<tr>
<td>41</td>
<td>2.7430</td>
<td>0.7339</td>
<td>41</td>
<td>13.4724</td>
<td>0.0311</td>
</tr>
<tr>
<td>42</td>
<td>4.4820</td>
<td>0.6340</td>
<td>42</td>
<td>13.7584</td>
<td>0.0212</td>
</tr>
<tr>
<td>43</td>
<td>7.2680</td>
<td>0.6216</td>
<td>43</td>
<td>14.0030</td>
<td>0.0178</td>
</tr>
<tr>
<td>44</td>
<td>11.7620</td>
<td>0.6183</td>
<td>44</td>
<td>14.3056</td>
<td>0.0216</td>
</tr>
<tr>
<td>45</td>
<td>18.9250</td>
<td>0.6090</td>
<td>45</td>
<td>14.6401</td>
<td>0.0234</td>
</tr>
<tr>
<td>Min</td>
<td>0.1430</td>
<td>0.6090</td>
<td>Min</td>
<td>11.3878</td>
<td>0.0178</td>
</tr>
<tr>
<td>Max</td>
<td>18.9250</td>
<td>0.7339</td>
<td>Max</td>
<td>14.6401</td>
<td>0.0379</td>
</tr>
<tr>
<td>Average</td>
<td>4.4627</td>
<td>0.6303</td>
<td>Average</td>
<td>13.0540</td>
<td>0.0255</td>
</tr>
<tr>
<td>Std Dev.</td>
<td>6.0158</td>
<td>0.0370</td>
<td>Std Dev.</td>
<td>1.0782</td>
<td>0.0060</td>
</tr>
<tr>
<td>Variance</td>
<td>36.1897</td>
<td>0.0014</td>
<td>Variance</td>
<td>1.1625</td>
<td>3.5815 × 10⁻⁵</td>
</tr>
</tbody>
</table>

Table A.1: Results from experimental evaluation comparing average times of 10 runs per index (maximum variance recorded was 0.1001858) as well as their growth. **Lower times and growth are better.** Also note that 0.0 growth is discarded when calculating min, max and average.
A.2 General Performance

<table>
<thead>
<tr>
<th>Run</th>
<th>Mean (s)</th>
<th>Standard Deviation (s)</th>
<th>Min (s)</th>
<th>Max (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>433.704621</td>
<td>241.336186804302</td>
<td>1.204</td>
<td>854.481</td>
</tr>
<tr>
<td>2</td>
<td>444.137735666667</td>
<td>252.027095690197</td>
<td>0.825</td>
<td>879.165</td>
</tr>
<tr>
<td>3</td>
<td>478.936931666667</td>
<td>282.295822883816</td>
<td>0.920</td>
<td>935.404</td>
</tr>
<tr>
<td>4</td>
<td>464.748838666667</td>
<td>283.273033634957</td>
<td>0.936</td>
<td>980.555</td>
</tr>
<tr>
<td>5</td>
<td>472.672313666667</td>
<td>269.928082345072</td>
<td>0.900</td>
<td>1023.968</td>
</tr>
<tr>
<td>Average</td>
<td>458.840088133333</td>
<td>265.772044271669</td>
<td>0.957</td>
<td>934.7146</td>
</tr>
</tbody>
</table>

Table A.2: Response times of requests across the five runs from the general performance experiment. **Lower times are better.**

<table>
<thead>
<tr>
<th>Run</th>
<th>Throughput (req/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.51090310960688</td>
</tr>
<tr>
<td>2</td>
<td>3.41232874375117</td>
</tr>
<tr>
<td>3</td>
<td>3.20717037771915</td>
</tr>
<tr>
<td>4</td>
<td>3.05949181840896</td>
</tr>
<tr>
<td>5</td>
<td>2.92977905559549</td>
</tr>
<tr>
<td>Min</td>
<td>2.92977905559549</td>
</tr>
<tr>
<td>Max</td>
<td>3.51090310960688</td>
</tr>
<tr>
<td>Average</td>
<td>3.22393462101633</td>
</tr>
</tbody>
</table>

Table A.3: Throughput measures (req/s) of the five runs from the general performance experiment. **Higher measures are better.**
Appendix B

Selected Source Code

Selected snippets of the prototype implementation source code that were discussed in the thesis are presented here.

B.1 Import Module From Database

```javascript
var path = require('path');
var fs = require('fs');
var tmp = require('tmp');
var utilities = require('../libs/utilities');

function require_blob(blob, callback) {
  if (!callback) callback = () => { console.log("Successfully required blob."); }
  if (!blob) callback(new Error("Module with no contents."));

  // Create a temp file (it is automatically unlinked on process exit)
  tmp.file((err, filepath, fd) => {
    if (err) return callback(err);

    // Write the blob to the file
    fs.write(fd, blob, 0, blob.length, null, function (err, written, buffer) {
      if (err) {err.message = "[RBLOB] " + err.message; return callback(err);}
      if (written != buffer.length) return callback(new Error("[RBLOB] File was not written properly.")));}

    // Require the module and pass it to the callback
    callback(null, require(filepath));
  });
}

function import_code(code, callback) {
  var binary_blob = new Buffer(code, 'base64'); // Must decode from base64

  // Import the module from the binary blob
  require_blob(binary_blob, function (err, required_module) {
    if (err) return callback(err);
  });
```
callback(null, required_module);
});

// The closure here allows the sandbox object to be used by any service
module.exports = function setup (sandbox) {
    if (!sandbox) throw new Error("No sandbox global context provided.");

    // Remember references to the global context
    require_db = sandbox.require_db;

    return function _require_db_multiple (docnames, callback) {
        if (!docnames || docnames.length == 0) callback(new Error("No service id(s) provided."));
        if (!callback) callback = function () {console.log("Successfully included module(s).")};

        // Retrieve all modules from the database
        if (!utilities.isArray(docnames)) docnames = [docnames]; // convert the argument to an array, if needed

        var required_modules = [];
        (function _populate (i) { // I realize that this recursive approach was unnecessary...I could have just used Array.forEach(). Either way, this is a good learning experience.
            // Base case
            if (i == docnames.length) { // We have finished requiring all modules
                if (required_modules.length != docnames.length) {
                    callback(new Error("[REQDB] Not all modules were successfully imported."));
                }
                else {
                    if (required_modules.length == 1) required_modules = required_modules[0];

                    // Do something with the required modules
                    try {
                        callback(null, required_modules);
                    } catch (e) {
                        console.log("[REQDB] Runtime error: ", e);
                        e.message = "[REQDB] Runtime error: " + e.message;
                        callback(e);
                    }
                }
            } else {
                // For each service...
                var docname = docnames[i];
                // console.log(docname);

                // HTTP caching
                var opts = {
                    hostname: 'localhost',
                    port: 5984,
                }
            }
        })(0);
    }
}
    path: '/services/' + docname,
    method: 'HEAD',
    headers: {"User-Agent": "ServiceRegistry/0.0.1 (importing module " +
docname + ")"}
};

    var ServiceCache = require('../libs/cache-service')('localhost', '6379',
null, function (err) {callback(err)});
    ServiceCache.read_cache({
    hashkey: 'service:' + docname,
    key: 'etag'
}, {
    hashkey: 'service:' + docname,
    key: 'code'
}, function (err, cachedETag, cachedResponse) {
        if (err)
            return
    callback(err);

        // If the etag exists AND we also have a cached response, then attach
        the etag to the request header
        if (cachedETag && cachedResponse) {
            opts.headers["If-None-Match"] = cachedETag;
        }

        // Perform a separate HEAD request (only because nano apparently doesn
't permit custom headers in the db APIs) and look for the ETag
        var http = require('http');
        var req = http.request(opts, function (res) {
            var headers = res.headers;

            // If the response is 304, use the cached code
            if (res.statusCode == 304) {
                import_code(cachedResponse, function (err, required_module) {
                    if (err)
                        return
                callback(err);

                    // Push to the modules array
                    console.log("Cached Source Code");
                    required_modules.push(required_module);
                    _populate(i+1);
                });
            }

            // otherwise, obtain the fresh source code
            else {
                http.get('http://localhost:5984/services/' + docname, function (res) {
                    var body = "";
                    res.on('data', function (chunk) {
                        body += chunk;
                    });
                    res.on('end', function () {
                        body = JSON.parse(body);
                        if (!body.source_code)
                            return
                    callback(new Error("[REQDB] No source exists for this service.")));

                    // Cache the base64-encoded source code binary blob
                    ServiceCache.write_cache({
Listing B.1: Utility for remotely importing module source code

B.2 Service-To-Service Consumption

```javascript
function build_service_url (docname, args) {
    // var serviceURL = "http://localhost:3000/api/invoke/" + docname;
    var serviceURL = "http://localhost:3000/services/" + docname + "/result";

    if (args && typeof(args) !== 'function') { // Need to check for function type
        // because in some cases args may still contain the callback function
        // console.log("These are the args = ", args);
        var utilities = require('../libs/utilities');
        if (utilities.isArray(args) || typeof(args) !== 'object') {
            args = {data: args};
        }
        var qs = require('qs');
        serviceURL += "?" + qs.stringify(args);
    }

    // console.log("\t\t\tServiceURL is: ", serviceURL);
    return serviceURL;
}
```
function relay_response (status, response, callback) {
    if (status === undefined || response === undefined)
        return callback(new Error("Invalid response."));
    callback(null, status, response);
}

function preprocess_by_type (type, output) {
    var match = /(text|image|audio|video|application|message|multipart|extension-token)\/(\[\s\]x20|x00-\x1f|x7f{0,8},;:\"\"/\[\s\]x20|x00-\x1f|x7f{0,8}\]|\[\s\]x1f|\x7f{2,7})/.exec(type);
    if (!match) // Assume that the data is bad if the mime type is not valid
        return null;
    var top = match[1], sub = match[2];
    console.log(top, sub);
    switch (top) {
        case 'text':
            return output.toString('utf8');
        case 'image':
            return output;
        case 'video':
            return output;
        case 'audio':
            return output;
        case 'application':
            switch (sub) {
                case 'json':
                    return JSON.parse(output);
                default:
                    return output;
            }
        default:
            return output
    }
}

/**
 * This helper function enables services to consume other services in the
 * registry, either by name + arguments (standard function invocation style)
 * or via a direct URL, either absolute or relative. The result of the
 * invocation, including any errors that may have occurred, is simply relayed
 * to the callee via a callback.
 * Note: the direct URL approach theoretically enables services to be located
 * anywhere on the Internet provided they follow the required response scheme,
 * i.e. {status: <status>, output: <output>}. If the scheme is not followed,
 * an error is raised.
 * @param {String} docname unique name/id of the service, or a valid URL
 * @param {*} args input parameters to the service
 * @param {Function} callback to be executed upon receipt of response
 */
// var validServiceURL = /(http://localhost:3000|\:\\)(/api\:\\invoke\:\\)\.(+))/;
var validServiceURL = /(http://localhost:3000|ˆ)(/services/[0-9a-zA-Z\-\_]*)\((.*)/;

module.exports = function _consume_service (docname, args, callback) {
  if (arguments.length < 2) {
    return sandbox.res.sendError("[CSERV] Nothing to be done.");
  }
  if (arguments.length == 2 && typeof(args) === 'function') {
    callback = args;
  }
  if (!callback) callback = function () {console.log("Successfully consumed service.");};

  try {
    // If URL was not provided, process the query and convert it to a consumable URL
    var match = validServiceURL.exec(docname);
    if (!match) {
      docname = build_service_url(docname, args);
    } else if (!match[1]) { // forces the host to be the correct one if missing
      docname = "http://localhost:3000" + docname;
    }

    // Parse the URL into its constituents as per the URL spec
    // (This is done to 1) enable HTTP caching and 2) attach additional headers if needed)
    var url = require('url');
    var options = url.parse(docname);
    options.query = options.query || "";
    options.headers = {"User-Agent": "ServiceProvision/0.0.1 (consuming service at " + options.pathname + ")"};

    // Check the cache to see if the URL has been queried before
    var ServiceCache = require('../libs/cache-service')('localhost', '6379',
      null, function (err) {callback(err)});
    ServiceCache.read_cache({
      hashkey: 'service:' + options.pathname,
      key: 'etag'
    }, {
      hashkey: 'service:' + options.pathname + ':query',
      key: options.query
    }, function (err, cachedETag, cachedResponse) {
      if (err) return callback(err);

      // If the etag exists AND we also have a cached response, then attach the etag to the request header
      if (cachedETag && cachedResponse) {
        options.headers["If-None-Match"] = cachedETag;
      }
    // Consume the service and execute the provided callback
    // console.log(options);
    var http = require('http');
http.get(options, function (res) {
  // console.log(res.headers, res.statusCode);
  // If the response is 304, forward the cached response
  if (res.statusCode == 304) {
    relay_response(res.statusCode, JSON.parse(cachedResponse), callback);
  }
  else {
    // process the rest of the response
    var body = new Buffer(0);
    res.on('data', function (chunk) {
      body = Buffer.concat([body, chunk]);
    });
    res.on('end', function () {
      var type = res.headers['content-type'];
      var response = preprocess_by_type(type, body);
      relay_response(res.statusCode, response, callback);
    });
    // If caching is permitted (it won’t be allowed for impure functions), store the etag + response in the remote cache
    if (res.statusCode == 200) {
      var cacheControlPattern = /no-cache|no-store|must-revalidate|max-stale=0/;
      if (res.headers.etag && !cacheControlPattern.test(res.headers['cache-control'])) {
        ServiceCache.write_cache({
          hashkey: 'service:' + options.pathname,
          key: 'etag',
          value: res.headers.etag
        }, {
          hashkey: 'service:' + options.pathname + ':query',
          key: options.query,
          value: JSON.stringify(response)
        });
      }
    }
  }
}).on('error', function (err) {
  console.log('[CSERV] Could not consume the service: "', err);
  err.message = '[CSERV] Could not consume the service: "' + err.message;
  callback(err);
}); // on http.get error
}

} catch (e) {
  console.log('[CSERV] Runtime error: "', e);
  e.message = '[CSERV] Runtime error: "' + e.message;
  callback(e);
}

Listing B.2: Utility for consuming another service registered within the system

93
Appendix C

Sample Services

Out of the many services that were implemented in the prototype, a select few have been featured in this thesis. This appendix presents a closer look into each of them.

C.1 Fibonacci

This service implements the naive recursive algorithm of computing the members of the Fibonacci sequence. Although there are more efficient algorithms, such as the iterative algorithm and the Golden Ratio approximation, the recursive implementation offers us insight into the built-in memoization capability of the prototype implementation by means of its service invocation results caching mechanism.

```javascript
var n = parseInt(sandbox.data, 10) || 0;

if (n < 0) sandbox.res(n).integer().send();
else if (n == 0 || n == 1) sandbox.res(n).integer().send();
else {
    consume_service('fib', n-1,
    function (err, status, output1) {
        if (err) return sandbox.res(err).error(503);
        output1 = parseInt(output1, 10);
        consume_service('fib', n-2,
        function (err, status, output2) {
            if (err) return sandbox.res(err).error(503);
            output2 = parseInt(output2, 10);
            sandbox.res(output1 + output2).integer().send();
        });
    });
}
```

Listing C.1: Sample service: Calculation of the Fibonacci sequence
C.2 Levenshtein Distance

The Levenshtein distance is one measure for the ‘edit’ distance between two strings, the distance being defined as the minimum number of single-character insertions, deletions or substitutions needed to change one word into the other. The implemented service, yet again, follows a naive recursive algorithm for the same reasons as the Fibonacci service of gaining insight into the service invocation cacheing mechanism.

```javascript
if (!sandbox.args || sandbox.args < 2)
    sandbox.res("No/Insufficient input provided.").error();

var s = sandbox.data.s.toString() || sandbox.args[0].toString();
var length_s = s.length;
var t = sandbox.data.t.toString() || sandbox.args[1].toString();
var length_t = t.length;

// Base case: empty strings
if (length_s == 0)
    sandbox.res(length_t).integer().send();
if (length_t == 0)
    sandbox.res(length_s).integer().send();

// Test if the last characters of the strings match
var cost = s[length_s-1] == t[length_t-1] ? 0 : 1;

// Compute the minimum of deleted character from s, deleted character from t and deleted character from both
consume_service('levenshtein', {s: s.substring(0, length_s-1), t: t}, function (err, status, from_s) {
    if (err) sandbox.res(err).error();
    from_s = parseInt(from_s, 10);
    consume_service('levenshtein', {s: s, t: t.substring(0, length_t-1)}, function (err, status, from_t) {
        if (err) sandbox.res(err).error();
        from_t = parseInt(from_t, 10);
        consume_service('levenshtein', {s: s.substring(0, length_s-1), t: t.substring(0, length_t-1)}, function (err, status, from_both) {
            if (err) sandbox.res(err).error();
            from_both = parseInt(from_both, 10);
            consume_service('minNums', [from_s + 1, from_t + 1, from_both + cost], function (err, status, min) {
                if (err) sandbox.res(err).error();
                min = parseInt(min, 10);
```
C.3 Matrix Multiplication

Using the glmatrix module service as a foundation, this service computes the multiplication of an ordered list, of any feasible size, of $3 \times 3$ matrices. Along with all of the other services that are based on glmatrix, this service demonstrates the value of module services in easing the migration of existing JavaScript libraries to the prototype as services. Recall that the glmatrix module contains verbatim the source code of the actual open source glMatrix library.

```javascript
if (!sandbox.args || sandbox.args.length < 2) sandbox.res("No/Insufficient input provided.").error(400);
else {
  require_db('glmatrix', function(err, required_module) {
    if (err) return sandbox.res(err).error();

    // Store a reference to the function that we need from the imported module
    var lib = required_module.mat3;

    // First, add the first two matrices...
    var result = lib.mul([], sandbox.args[0], sandbox.args[1]);

    // ...then add any remaining ones
    var n = sandbox.args.length; // total number of operands
    for (var i = 2; i < n; i++) {
      result = lib.mul([], result, sandbox.args[i]);
    }

    sandbox.res(result).json().send();
  });
}
```

Listing C.3: Sample service: Matrix chain multiplication
C.4 Google Static Maps

This service is simply a minimalistic client to the Google Static Maps API V2. The original service takes as input various cartographic parameters via the query string of a specially-constructed URL accessed through a standard HTTP GET request and if successful, responds with a raw PNG image of the generated map. Being exposed through a simple HTTP interface, it was very easy to create a service for the prototype that serves as a proxy client to the original service; all it does is forwards all requests to the original service. This service demonstrates the relative ease of integrating certain valuable external Web services. As long as the external service is accessible over some form of socket connection, a client for it should theoretically be possible. This interestingly implies that it should be possible to implement proxy services even for external WS-* services. This is an interesting avenue for future exploration.

It must be pointed out that many external service providers enforce usage limits on their APIs for various service consumers. This is usually done via some sort of client identification via API keys or OAuth tokens. In the future, the proposed framework should take this into consideration. For now, a personal key is being used with the prototype.

```javascript
if (sandbox.data && typeof (sandbox.data) !== 'object')
    return sandbox.res("Invalid input provided. You must provide a JSON object containing the parameters.").error();

require_db('googlemaps', function (err, required_module) {
    if (err) return sandbox.res(err).error();
    var request = required_module('staticmap');
    var params = sandbox.data || {};

    // Some defaults/pre-processing
    params.center = params.center || "43.5303377,-80.2260301"; // UoG approx lat/lon
    params.zoom = params.zoom || 10;
    params.size = params.size || '300x200';

    request(params, function (err, response, body) {
```

1https://developers.google.com/maps/documentation/staticmaps/
C.4 Sample service: Client service to the Google Static Maps API

```javascript
if (err) return sandbox.res(err).error();

sandbox.res(response).image('png').impure().send();
```

Listing C.4: Sample service: Client service to the Google Static Maps API

C.5 AES Encrypt/Decrypt

Two separate services were implemented for performing encryption/decryption of given textual data using the AES algorithm. Note that instead of writing a stand-alone implementation of the specification, the functions from the standard cryptographic library packaged with NodeJS were conveniently used. Recall that certain modules are made available to the sandbox environment in which service code is executed; this module is one of them.

```javascript
var crypto = require('crypto');

var input = sandbox.data || "Good morning!"; // If no input given, use all default settings

var plaintext = input['data'] || input['plaintext'] || input;
var algorithm = input['algorithm'] || 'aes192';
var password = input['password'] || 'helloworld';
var input_encoding = input['input_encoding'] || 'utf8';
var output_encoding = input['output_encoding'] || 'hex';

var cipher = crypto.createCipher(algorithm, password);
ciphertext = cipher.update(plaintext, input_encoding, output_encoding);
ciphertext += cipher.final(output_encoding);

sandbox.res(ciphertext).string().send();
```

Listing C.5: Sample service: AES Encryption

```javascript
var crypto = require('crypto');

var input = sandbox.data || "b917f604040e28c7af1851011b7a848d!"; // If no input given, use all default settings

var ciphertext = input['data'] || input['ciphertext'] || input;
var algorithm = input['algorithm'] || 'aes192';
var password = input['password'] || 'helloworld';
var input_encoding = input['input_encoding'] || 'hex';
```

Listing C.5: Sample service: AES Encryption
var output_encoding = input['output_encoding'] || 'utf8';

var decipher = crypto.createDecipher(algorithm, password);
plaintext = decipher.update(ciphertext, input_encoding, output_encoding);
plaintext += decipher.final(output_encoding);
sandbox.res(plaintext).string().send();

Listing C.6: Sample service: AES Decryption
Appendix D

Evaluation Scripts

The following listing presents the JavaScript code that was used to randomly generate various test cases for evaluating the general performance of the prototype implementation, which was performed in Section 5.3. Each test case is simply represented as a service invocation URL that can be queried against the prototype server. The script is capable of executing all test cases and creating CSV reports by itself, or dumping all the test case URLs into a file that can be used by other applications for more sophisticated tests, if desired.

```javascript
var http = require('http');
var request = require('request);
var qs = require('querystring');
var fs = require('fs');
var utilities = require('../../webintents/libs/utilities');

function log (counter) {
    return function _log (filename, delim, fields) {
        var field_names = Object.keys(fields);

        // Write the header row on first run
        if (counter == 1) {
            var output = "";
            for (var i = 0; i < field_names.length; i++) {
                output += field_names[i];
                if (i != field_names.length - 1)
                    output += delim;
            }
            fs.writeFileSync(filename, output);
        }

        // Dump the results in a new row
        var output = "\n";
        for (var i = 0; i < field_names.length; i++) {
            output += fields[field_names[i]];
        }
        fs.writeFileSync(filename, output);
    }
}
```
if (i != field_names.length - 1)
    output += delim;
}
fs.appendFileSync(filename, output);
}
}

var dump = log(1);

function randomInt (min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min
}

function randomFloat (min, max) {
    return Math.random() * (max - min) + min
}

function randomString (length) {
    var buf = new Buffer(length);
    for (var i = 0; i < length; i++) {
        do {
            buf[i] = randomInt(48,122)
        } while (buf[i] >= 58 && buf[i] <= 64 || buf[i] >= 91 && buf[i] <= 96);
    }
    return buf.toString();
}

function generate_fib (min, max, runs) {
    var data = [];
    for (var r = 0; r < runs; r++) {
        data.push({num: randomInt(min, max)});
    }
    return data;
}

function generate_levenshtein (min_length, max_length, runs_per_length) {
    var data = [];
    for (var i = min_length; i <= max_length; i++) {
        for (var r = 0; r < runs_per_length; r++) {
            data.push({s: randomString(i), t: randomString(i)});
        }
    }
    return data;
}

function generate_matrix (min, max, size) {
    // Always assumes a square matrix
    var matrix = new Array(size);
    for (var i = 0; i < size*size; i++) {
        matrix[i] = randomFloat(min, max);
    }
    return matrix;
}

function generate_matrices (min, max, size, matrices, runs) {
    var data = [];
    for (var r = 0; r < runs; r++) {
        var datum = {};
        for (var m = 1; m <= matrices; m++) {
            datum['mat' + m] = generate_matrix(min, max, size);
        }
        data.push(datum);
    }
function generate_staticmap () {
    return {
        center: randomFloat(-90, 90) + ',' + randomFloat(-180, 180),
        zoom: randomInt(0, 20),
        size: '640x640'
    };
}

function generate_staticmaps (runs) {
    var data = [];
    for (var r = 0; r < runs; r++) {
        data.push(generate_staticmap());
    }
    return data;
}

function generate_aesencrypt (length) {
    return {plaintext: randomString(length), password: randomString(length)};
}

function generate_aesdecrypt (aesencryptdata) {
    return aesencryptdata.map(function (element) {
        var cipher = require('crypto').createCipher('aes192', element.password);
        var ciphertext = cipher.update(element.plaintext, 'utf8', 'hex');
        return {ciphertext: ciphertext + cipher.final('hex'), password: element.password};
    });
}

function generate_aesdata (min_length, max_length, runs) {
    var data = [];
    for (var r = 0; r < runs; r++) {
        data.push(generate_aesencrypt(randomInt(min_length, max_length)));
    }
    return data;
}

function generate_randomtest (test_cases) {
    var data = [];
    var tests = Object.keys(test_cases);
    while (tests.length) {
        // Pick a test at random
        var test_index = randomInt(0, tests.length - 1);
        var test = test_cases[tests[test_index]];
        if (!test.length) {
            // If test has no cases, delete test
            tests.splice(test_index, 1);
            continue;
        }
    }
// Pick a case at random
var case_index = randomInt(0, test.length - 1);
var test_case = test[case_index];

// Generate URL
var pathname = '/services/' + tests[test_index] + '/result?' + qs.stringify(test_case);

// Push URL to data array
data.push(pathname);

// Delete case
test.splice(case_index, 1);
}

return data;

/////

*/
* Activate/Deactivate test cases by uncommenting/commenting them respectively
* Currently, only the levenshtein and aes encryption/decryption test cases have been selected
*
*/
var test_cases = {
  // 'fib' : generate_fib(0, 125, 100),
  'levenshtein' : generate_levenshtein(4, 4, 25),
  // 'factorial' : generate_fib(0, 50, 5, 1),
  // 'addmatrix2': generate_matrices(1000000, 10000000, 2, 10, 10),
  // 'multiplymatrix3': generate_matrices(1000000, 10000000, 3, 10, 100),
  // 'addmatrix4': generate_matrices(1000000, 10000000, 4, 10, 10),
  // 'googlestaticmaps': generate_staticmaps(100),
};

var aesencryptdata = generate_aesdata(100, 1000, 10);
test_cases['aes-encrypt'] = aesencryptdata;
test_cases['aes-decrypt'] = generate_aesdecrypt(aesencryptdata);

/**** BEGIN TEST ****/

var outfile = process.argv[2] || './results/test.csv';
var delim = process.argv[3] || "\",";

var test_cases = generate_randomtest(test_cases);
var url = {
  protocol: 'http:',
  hostname: 'dummy.url.com', // Enter the appropriate hostname
}

/*** Uncomment this to dump the URLs to a file for use in curl, httpperf, siege, etc. ***/
// fs.openSync(outfile, "w");
// for (var i = 0; i < test_cases.length; i++) {
//   fs.appendFileSync(outfile, test_cases[i] + "\n");
// }
for (var i = 0; i < test_cases.length; i++) {
    url.pathname = test_cases[i];
    (function (query) {
        var stamp = Date.now();
        // request.get({url: url}, function (err, res, body) {
        request.get(url.protocol + "//" + url.hostname + url.pathname
        , function (err, res, body) {
            var end = Date.now();
            var RT = end - stamp;
            var code = res ? res.statusCode || -1 : -1;
            var mssg = err ? err.toString() : "OK";
            console.log("URL: %s\n", query);
            console.log("\tStatus: %d %s Response Time: %d ms Start Time: %d End Time: %d\n", code, mssg, RT, stamp, end);
            dump(outfile, delim, {
                response_code: code,
                response_time: RT,
                response_mssg: mssg,
                service_id: (/\services\/[a-zA-Z0-9-]+\result/.exec(query)) [1],
                query: query
                });
        });
    })(url.pathname);
}

Listing D.1: Script used in the evaluation of general prototype performance