

Dependable IPTV Hosting

A Thesis Submitted to the College of
Graduate Studies and Research
In Partial Fulfillment of the Requirements
For the Degree of Computer Science
In the Department of Computer Science
University of Saskatchewan
Saskatoon

By

DANIEL MERINO

© Copyright Daniel Merino, March, 2012. All rights reserved.

Permission to Use

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

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

Head of the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan S7N 5C9

ABSTRACT

This research focuses on the challenges of hosting 3rd party RESTful applications that have to meet specific dependability standards. To provide a proof of concept I have implemented an architecture and framework for the use case of internet protocol television. Delivering TV services via internet protocols over high-speed connections is commonly referred to as IPTV (internet protocol television). Similar to the app-stores of smartphones, IPTV platforms enable the emergence of IPTV services in which 3rd party developers provide services to consumer that add value to the IPTV experience. A key issue in the IPTV ecosystem is that currently telecommunications IPTV providers do not have a system that allows 3rd party developers to create applications that meet their standards. The main challenges are that the 3rd party applications must be dependable, scalable and adhere to service level agreements. This research provides an architecture and framework to overcome these challenges.

ACKNOWLEDGEMENTS

I would like to thank Jacqueline Arroyo, my mother, for supporting me throughout my scholastic career. I also would like to thank Ralph Deters for providing me with the opportunity to do research under his guidance; support and giving me the liberty to explore various research topics. Finally, I would like to thank Telecommunications Research Labs, specially the Head of Converged Digital Media R&D Andrew Kostiuk, for the privilege of working with the topics mentioned in this research, support and funding.

TABLE OF CONTENTS

page

<u>ABSTRACT.....</u>	<u>ii</u>
<u>ACKNOWLEDGEMENTS.....</u>	<u>iii</u>
<u>LIST OF TABLES.....</u>	<u>ix</u>
<u>LIST OF FIGURES.....</u>	<u>x</u>
<u>LIST OF ABBREVIATIONS.....</u>	<u>xiii</u>
<u>INTRODUCTION.....</u>	<u>1</u>
1.1. Internet Protocol Television.....	3
<u>PROBLEM DEFINITION.....</u>	<u>5</u>
2.1. Dependable execution of 3 rd party services.....	7
2.2. Scalability of 3 rd party services.....	7
2.3. Service Level Agreements 3 rd party services.....	8
<u>LITERATURE REVIEW.....</u>	<u>9</u>
3.1. Web Services SOA - WS* and REST.....	10
3.1.1. Web Services using SOA - WS*.....	10
3.1.2. Service Registries.....	11
3.1.3. Service Repositories.....	11
3.1.4. Service Definitions.....	11
3.1.5. Service Frameworks.....	11
3.1.6. SOA Architectural Workflow.....	11
3.1.6.1. Boundaries are explicit.....	12
3.1.6.2. Services are autonomous.....	13
3.1.6.3. Services share schema and contract, not class.....	13
3.1.6.4. Services compatibility is determined based on policy.....	14

3.1.7.	WSDL	14
3.1.8.	Summary: SOA-WS*	16
3.1.9.	Web Services using RESTful Architectures	16
3.1.9.1.	Request Line	17
3.1.9.2.	Richardson Maturity Model on REST	20
3.1.9.3.	Level 0	21
3.1.9.4.	Level 1	21
3.1.9.5.	Level 2	21
3.1.9.6.	Level 3	22
3.1.10.	Summary: Web Services SOA-WS* VS REST	23
3.2.	Dependable Web Services	23
3.2.1	Summary: Dependable Web Services	25
3.3.	A new approach of defining Dependability in Web Services	26
3.3.1.	Attributes of Dependability	28
3.3.1.1.	Confidentiality	28
3.3.1.2.	Reliable	28
3.3.1.3.	Availability	29
3.3.1.4.	Safe	29
3.3.1.5.	Integrity	30
3.3.1.6.	Maintainability	30
3.3.2.	Summary: Adding Dependability to Web Services	30
3.4.	Fault Injection	31
3.5.	CAP Theorem	32
3.5.1.	Summary: CAP Theorem	34
3.6.	Databases	34

3.6.1.	Summary: Databases.....	37
3.7.	Cloud Computing.....	37
3.7.1.	Infrastructure as a Service.....	38
3.7.2.	Platform as a Service	38
3.7.3.	Software as a Service (SaaS)	38
3.7.4.	Summary: Cloud Computing	40
3.8.	Sandboxing	41
3.9.	Service Level Agreements	42
3.10.	Summary of Literature Review.....	44
3.11.	Issues Tackled in this Research	46
<u>IMPLEMENTATION.....</u>		47
4.1.	Overview.....	47
4.1.1.	Detailed explanation of how a request is handled by the architecture.....	48
4.2.	Core Architectural Components	52
4.3.	Worker API.....	56
4.4.	Workers - Resources	60
4.5.	Simplify IPTV Development	68
4.6.	Service Level Agreements	68
<u>EXPERIMENTS</u>		69
5.1.	Setup	70
5.2.	Load Generation.....	71
5.3.	EX1 - Overhead Test	71
5.4.	EX2 - Dependability vs. Performance Test	73
5.5.	EX3 - Scalability Test.....	76
5.6.	EX4 - Service Level Agreements Test.....	76

<u>Results</u>	78
6.1. EX1	78
6.1.1. Services without the architecture	78
6.1.2. Services with the architecture	82
6.2. EX2	86
<u>Summary and Contribution</u>	91
<u>Future Work</u>	93
8.1. Future Work	93
8.1.1. Web Sockets.....	93
8.1.2. CQRS	94
8.1.3. Pre-Processed Static Resources Hosted on the Cloud	95
8.1.4. Publish Subscribe Client Cache Push	95
<u>LIST OF REFERENCES</u>	96
<u>LIST OF WEBSITES</u>	101

LIST OF TABLES

<u>Table</u>	<u>page</u>
Table 3-1. HTTP Verb List.....	17
Table 3-2. Common HTTP Response List.....	19
Table 3-3. Cap Theorem Breakdown.....	32
Table 6-4. Overhead 10 RPS Summary.....	79
Table 6-5. Overhead 20 RPS Summary.....	80
Table 6-6. Overhead 30 RPS Summary.....	81
Table 6-7. Overhead Architecture10 RPS Summary.....	83
Table 6-8. Overhead Architecture 20 RPS Summary.....	84
Table 6-9. Overhead Architecture 30 RPS Summary.....	85
Table 6-10. Dependability VS Performance Summary.....	86
Table 6-11. Scalability Instances Summary.....	88
Table 6-12. Scalability Summary.....	89

LIST OF FIGURES

<u>Figure</u>	<u>page</u>
Figure 1-1. MRML Code for Store Front	2
Figure 1-2. Rendering of MRML Store Front	3
Figure 2-2. Basic IPTV Setup.....	5
Figure 3-1. SOA Architecture.....	12
Figure 3-3. Sample HTTP requests with their Status Response	17
Figure 3-4. Sample HTTP Response	17
Figure 3-5. Richardson Maturity Model [14]	20
Figure 3-6. Dependability Tree [22]	27
Figure 3-7. Cap Theorem Balance	32
Figure 4-1. Architecture.....	47
Figure 4-2. HTTPS Request generated by user. If found in cache, return request.	48
Figure 4-3. Create an operation to be sent to the 3 rd party services.....	49
Figure 4-4. Send the operation to the 3 rd party service	50
Figure 4-5. Save the operation to the cache.....	51
Figure 4-6. Distributing the load for scalability and dependability	52
Figure 4-7. Login	57
Figure 4-8. Create Projects	57
Figure 4-9. List of all the projects created by the user.....	57
Figure 4-10. Upload a worker resource that plugs into the architecture.....	58
Figure 4-11. List of all workers resources uploaded by user	58

Figure 4-12. Give the worker permissions.....	59
Figure 4-13. Create instances of the worker projects so they can accept requests	59
Figure 4-14. Find a free port for the resource	60
Figure 4-15. Hard coded example of how to permissions are given	61
Figure 4-16. Interface that allows to adhere to the RESTful MVC	61
Figure 4-17. Hard coded example of a worker binding	62
Figure 4-18. Dependency injection at runtime using a central HTTP based repository ...	62
Figure 4-19. Round-Robin in memory lookup of available resources.....	63
Figure 4-20. Personalized Application Hub.....	64
Figure 4-21. Applications the user has purchased	65
Figure 4-22. Twitter Application	66
Figure 4-23. Facebook Application	66
Figure 4-24. Alerts sent from a mobile device	67
Figure 5-1. Cloud Instances	70
Figure 5-2. Services Layout.....	73
Figure 5-3. Faulty Services Layout.....	75
Figure 5-4. Unavailable Services Layout.....	75
Figure 6-1. Overhead 10 RPS – 10 Minutes	79
Figure 6-2. Overhead 20 RPS – 10 Minutes	80
Figure 6-3. Overhead 30 RPS – 10 Minutes	81
Figure 6-4. Overhead Architecture 10 RPS – 10 Minutes	83
Figure 6-5. Overhead Architecture 20 RPS – 10 Minutes	84
Figure 6-6. Overhead Architecture 30 RPS – 10 Minutes	85
Figure 6-7. Dependability VS Performance – 10 Minutes	87
Figure 6-8. Scalability – 10 Minutes.....	89

LIST OF ABBREVIATIONS

API Application Programming Interface

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

JSON JavaScript Object Notation

PaaS Platform as a Service

REST Representational State Transfer

RSS RDF Site Summary

SaaS Software as a Service

SLA Service Level Agreement

SOAP Simple Object Access Protocol

TCP Transmission Control Protocol

Telco Telecommunications Provider

URI Uniform Resource Identifier

WSDL Web Service Definition Language

XML Extensible Markup Language

CHAPTER 1 INTRODUCTION

IPTV is delivering TV services through internet protocols over a high-speed connection. IPTV platforms differ from internet-based multimedia platforms (e.g. Crunchyroll [W1], Netflix [W2], YouTube [W3], iTunes [W4], Amazon Video on Demand [W5], Roku [W6], Google TV [W7], etc.) in terms of content, delivery and costs.

Subscription based IPTV offers its subscriber's TV content in addition to the stored content of multi-media platforms. The regular internet-based multimedia video-on-demand streaming and/or downloading services are therefore a superset of the services provided by the internet-based multimedia platforms.

To ensure that content-providers grant access to premium content, subscription based IPTV platforms offer very dependable (secure, safe, reliable and available) service delivery. Using bandwidth provisioning and secure protocols, it becomes possible to allow subscribers instant access to highly sought after digital assets (e.g. new tv-shows, new movie releases in HD) without compromising the DRM constraints of the content owners.

However all this comes at nearly twice the costs based on regular video on demand internet-based multimedia. In Canada (Saskatchewan) the video on demand Netflix service can cost a user \$38 a month (30\$ internet + 8\$ Netflix), a medium live IPTV service package from SaskTel is however, \$72 (internet included).

To combat the migration of customers from IPTV to basic internet-based multimedia services, IPTV subscription providers have begun the move towards interactive IPTV platforms that allow for apps on the TV.

Interactive IPTV platforms allow providers to blur the lines between classical TV and computers. Platforms like Microsoft Mediaroom allow IPTV providers to embed applications

into the video-stream and thus increase the interactivity of TV (see figure 1-1 and figure 1-2). Microsoft Mediaroom [W8] is one of the leading platforms in this market segment. Especially telecommunications companies (telcos) across the globe (e.g. AT&T, Bell, Deutsche Telekom, BTVision, etc.) favor Mediaroom since it offers backend support that fits their specific needs. The Microsoft Mediaroom platform is a server centric IPTV solution that is based on XML documents delivered over HTTP. This server-centric design is partially due to resource limitations and the need for a secure solution.

The application below is an example that provides an overlay over-the-top of the video and the user is able to interact with the “Shopping Channel” to purchase items.

```
<Text id="TVTextProductInformation"
  top="160" left="172"
  height="59" width="358">
  If you like to purchase click BUY NOW
</Text>
<Button id="TVButtonBuyNow" left="200"
  top="242" width="220">
  BUY NOW
  <Actions>
    <Event type="onclick"
      action="DialogAction" />
  </Actions>
</Button>
```

Figure 1-1. MRML Code for Store Front

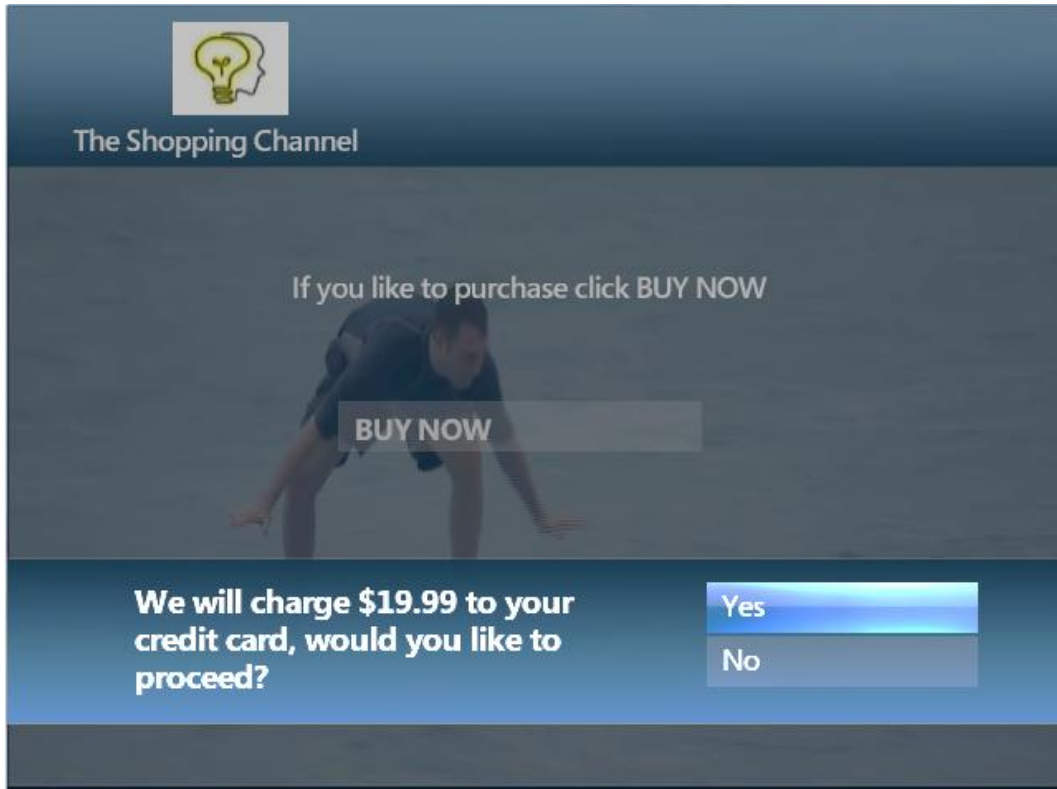


Figure 1-2. Rendering of MRML Store Front

1.1. Internet Protocol Television

As of 2010, it was announced on the broadband forum year-end report that there has been a total increase in IPTV telecommunications subscribers by 34.6% [W9]. On the side of internet-based media, Internet related streaming applications are becoming more popular because home entertainment devices are becoming media clients that are augmenting the video experience. Some of these devices are the Xbox 360 [W10], PlayStation 3 [W11] and Nintendo Wii [W12]. In the next few years, it is expected that consumers want to use interactive television experiences by default such as the ones that come from Samsung Smart TVs which connect directly to a network connection to augment user experience with widgets and internet content.

As the interactive experience from users increases, the number of simultaneous requests increases. The requests can even come from multiple devices [1][2][3], which means that

multiple active connections and sessions must be kept for each client. This causes the load on the servers to increase. To increase diversification, interactivity and to appeal to group audiences [4][5] in the IPTV applications ecosystem, 3rd party developers must be allowed to deploy IPTV applications. There has been one particular API created to give a personalized IPTV experience [6] but one common problem is that subscription companies must be able to control 3rd party applications. Also a system must be created to help 3rd party applications handle large amounts of load. Therefore, making requests directly to 3rd party applications is not feasible. Consequently, special measures must be taken to create a dependable and robust system. Finally, many of the 3rd party developers must be able to count on specific service contracts to provision their applications and to check the health of the applications.

The remainder of this thesis is structured as follows: Chapter 2 defines the problem, Chapter 3 has a compilation of work previously done that I use to support our work, Chapter 4 goes over our implementation, Chapter 5 has an evaluation, and Chapter 6 has the results from the evaluation, Chapter 7 is the conclusion of the research and Chapter 8 contains future work.

CHAPTER 2 PROBLEM DEFINITION

In this research the goal is to create an ecosystem for 3rd party developers to create their own IPTV applications to develop a highly diverse IPTV applications. Such 3rd party service providers will develop their own applications to be deployed on my IPTV architecture. Allowing them to fully develop their applications with their own creativity will generate TV content to be relevant, personalized, and different. This 3rd party development will allow for a great diversity of applications. Hence, 3rd party services can be used to increase the experience of users who subscribe to an IPTV service.

To be able to have a 3rd party ecosystem telcos must be able to host 3rd party IPTV services; a large architectural support is needed for a large number of diverse applications with many concurrent users. To be able to understand how to allow 3rd party hosting into the IPTV space, an overview of how the telco companies setup their IPTV infrastructure is required. Figure 2-2 is a diagram of a basic IPTV setup for telco companies.

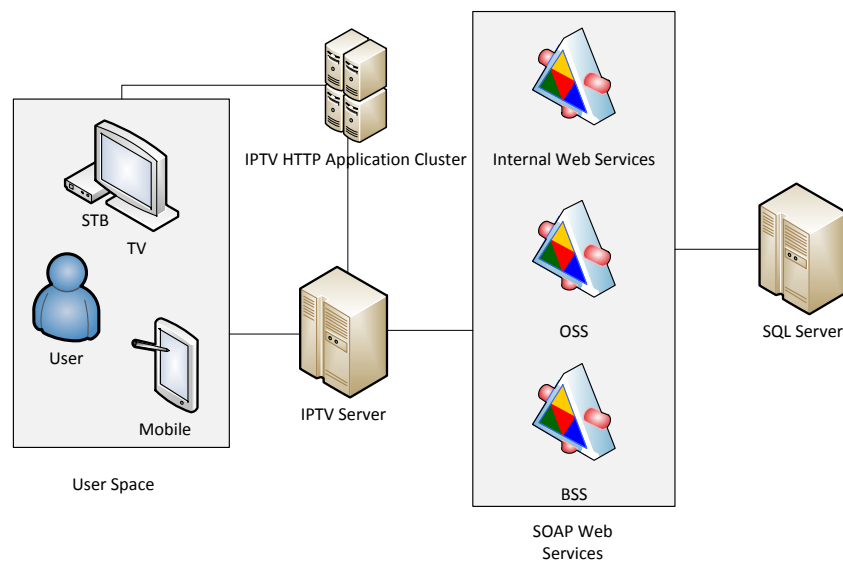


Figure 2-2. Basic IPTV Setup

The left hand side of the diagram represents the user space. The user space involves one or multiple set-top boxes and devices that can make requests to web services, such as mobile devices. On the other side, the IPTV deployments involve an IPTV server. The IPTV server can give IPTV related information, which includes several types of web services, and a basic HTTP server. The IPTV applications reside on a very large dedicated server or cluster as depicted by the IPTV HTTP Application Server on the diagram above. There are several web services in the IPTV server that allow developers to build a distributed architecture and allow for extension. They are mainly broken down into internal web services and API (subdivided into Operational Support Systems and Business Support Systems) SOAP web services. The internal web services are not meant to be for developers' use but only for internal IPTV components that have to interact with each other. The OSS API web services allow for control over multiple functional aspects of the environment, such as video on demand, electronic program guide, emergency alert system, and many others. The BSS API web services allow control over user billing-related and package management aspects. The functionality of these services must be exposed to third party developers so that they can work with them. Also, as stated on the server documentation, if the web services are executed concurrently without being throttled, general performance will suffer greatly due to the concurrent process model. Many of these services require a connection to the database and will hang until the work has been completed. Because we do not have access to the code to these web services, it is only possible to speculate that the reason for performance issues is due to heavy CPU bound operations and non-asynchronous operations with a database.

In this research the goal is to integrate middleware that will incorporate to this architecture to provide a dependable, scalable, and open architecture. Having a dependable, scalable, and open architecture to 3rd party developers IPTV architecture has many challenges and therefore, more

components (and as a result, overhead) need to be added to overcome the challenges. The primary challenges that I plan to include are:

- IPTV Architecture that allows for 3rd party code
 - Dependable execution of 3rd party services
 - Scalability of 3rd party services
 - Service Level Agreements 3rd party services

2.1. Dependable execution of 3rd party services

3rd party applications must interact with each other, and yet be able to perform without behavior alteration from other applications. Behavior alteration, intentional or unintentional, is a concern since ideally one machine would run multiple IPTV applications as tenants. For example, one application should not be able to delete other files, and application errors should stay within the application. For this goal there are two properties that must be included: secure and safe execution of code. From a security standpoint, applications should not affect other applications' behavior. From a safety standpoint, applications that are designed to be unaffected by other applications are isolated from danger. With either option, it is needed to prevent alteration of behavior across applications.

2.2. Scalability of 3rd party services

The stress due to requests/load on the applications varies extensively and is linked to the television programs being watched by users. For example, if there is an application that enhances the viewing experience of the Super Bowl by showing real time statistics of players, it could potentially bring the viewer numbers up to 48.66 million viewers in the United States of America at once, as once before, during the Super Bowl XLII (New York Giants vs. New England Patriots) in 2008. Because buying hardware to be able to keep up with infrastructure needs would be extremely expensive and many resources would be wasted, especially when they are not necessary, a cloud computing component must be used. Since there is interdependence

between components, a publish-subscribe system will be used to report when resources are ready and are working at their full capacity. It is important to note that since resources need to be deployed very efficiently, the environment must be able to respond quickly.

2.3. Service Level Agreements 3rd party services

To ensure quality of service, we require service level agreements [6], which would allow us to plan to handle requests in specified amounts, while providing the most accurate data. To be able to allow an open environment, the development of a custom platform for the 3rd party code is necessary while trying to reduce the overhead. All of the applications should be managed in such way that pushing updates to resources and their dependencies is automated, requiring minimal or no downtime.

In order to handle large loads of simultaneous users, the applications must be distributed and have several workers. This makes things difficult, as calculating quality of service for clients involves many variables.

The main goal of a service level agreement is to provide some promise based on best estimates of how the applications will behave under different types of loads. This way developers can do their capacity planning to make sure the provisioning of resources is efficient.

CHAPTER 3 LITERATURE REVIEW

This chapter discusses related research in the following fields: Web Services, Dependable Web Services, Fault Injection, Cloud Computing, CAP Theorem, Database Storage, Service Level Agreements and Sandboxing.

The importance of Web Services is that they are a well understood pattern of distributed systems. Since the architecture has many components and many of these components are not under our control, it is required to add dependability to web services. Sandboxing is investigated to provide the means for the dependable system with 3rd party applications. Fault injection is researched as a means to test the dependability vs. performance considerations of the architecture and the framework. Cloud computing is also significant because it simplifies provisioning to the growing architecture needs due to the large amount of users while trying to keep costs to the minimum. The CAP theorem is presented because it puts constraints on any distributed architecture. An overview of databases is done to showcase the alternatives available and how they are affected by the CAP theorem. Service Level Agreements are researched to be able to provide contracts with 3rd party applications.

3.1. Web Services SOA - WS* and REST

Web Services are a well understood pattern actively being used in distributed systems [W13]. Web services are a standardized mechanism in which distributed applications can communicate with each other. In simple terms each application becomes exposed to the Internet by using web services [8]. One of the main points of web services is that they are designed to be modular and extensible which gives them great extensibility. We will use web services since our architecture has many components that need to communicate with each other. For this task I have chosen to explore SOA and REST. Based on the findings I will explain where each component makes sense to be used as they are very different and therefore have many diverse strengths/weaknesses.

3.1.1. Web Services using SOA - WS*

WS-* is a compilation of specifications that have been passed by W3C [W14], OASIS [W15], and WS-I [W16]. The main points of WS-* are using Simple Object Access Protocol (SOAP) [W17] for data transfer, Web Services Description Language (WSDL) [W18], WS-Security (WSS) [W19], and many others.

All of the design principles of WS-* are what is commonly called Service Oriented Architectures (SOA) [W20]. On an ideal SOA architecture, there are four main types of SOA support mechanisms that allow it to be a well-known distributed architecture [8]:

- Service Registries
- Services Repositories
- Service Definition
- Service Frameworks

3.1.2. Service Registries

Service Registries are public services where web services can expose their location and capabilities. This is the primary source where consumers go to find web services. Extensive research has been performed to make sure the service registries can give relevant information to find the best-suited web services [10].

3.1.3. Service Repositories

There are many different types of service repositories but their main function is to work as metadata sources for the web services, primarily, to hold service descriptions and policies [11]. Some of the information includes service level agreements (SLA) and security requirements. Service repositories are extremely important as they are used for design, implementation, and deployment of web services. Finally, service repositories help version control the repositories with their contracts.

3.1.4. Service Definitions

Contracts and behaviors are defined in a XML file using the web service definition language (WSDL).

3.1.5. Service Frameworks

The platforms allow the developers to create an abstraction layer to support the SOA architecture by providing design time support and automatic implementation of the requirements based on the contracts. In some cases they also provide runtime support on some frameworks.

3.1.6. SOA Architectural Workflow

Figure 3-1 shows the SOA architecture workflow. The Universal Description Discovery and Integration (UDDI) [8] service is a repository, where web services can register and provide their operational information, WSDL. From the UDDI, they can be discovered by clients, who can then invoke services, which are usually behind a service bus [7].

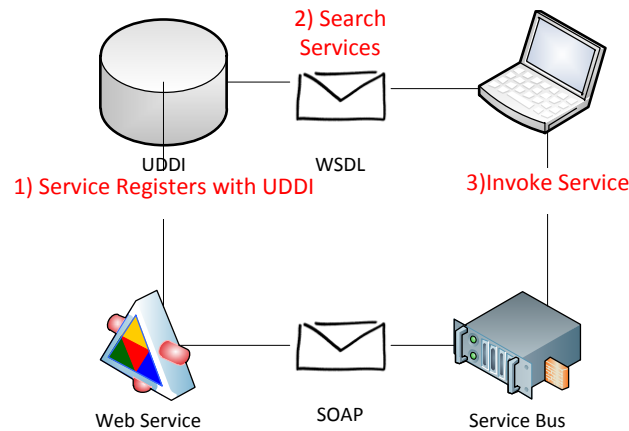


Figure 3-1. SOA Architecture

For the creation of web services, there are several platforms and the most common ones are .NET and Java.

Service-orientation has evolved over the years but many still abide by Don Box’s design guidelines. Don Box specifies that there are four main principles, which he calls tenants [W21]:

- Boundaries are explicit
- Services are autonomous
- Services share schema and contract, not class
- Service compatibility is based on policy

3.1.6.1. Boundaries are explicit

This is a very important notion that separates RPC style communication from SOA. This is an architectural style, which acknowledges that in a distributed architecture, messaging is usually the most expensive operation and the most susceptible to network failure. The basic architectural pattern of SOA makes it clear that there are boundaries and that there are specific costs when working with the boundaries.

3.1.6.2. Services are autonomous

SOA principles define an architecture, where there is not a single component but multiple distributed components. Because there is no single entity, all of the components are strictly decoupled and allow for individual deployment, updates, upgrades and extension. This also applies to failures of the consuming applications. SOA applies principles that allow for failures that do not have to be propagated to the whole system for it to be maintaining functionality. Finally, under this SOA principle, all of the messages are to be proofed because they may come from malicious senders, may be malformed, or may even not have the proper privileges.

3.1.6.3. Services share schema and contract, not class

In SOA, the abstraction levels of structures called schemas, and behaviours are all called contracts. In contrast with classic object orientated programming, the pattern is to combine structure and behavior under the same abstraction; this is to avoid marshaling objects across a network and simplify the consumption of services. Services publish their structures and behaviors to allow many different types of heterogeneous clients to have their own specifics. This way, only the transport and communication level is under the contract facilitating the distributed architecture by providing a verification and validation required. This further has another benefit, as evolving each service is far easier than the whole distributed architecture, and backward compatibility does not depend on objects but on contracts. The backward compatibility can be added by adding SOAP headers, which distinguishes messages for different contracts.

3.1.6.4. Services compatibility is determined based on policy

Policy allows for every service to publish its capabilities and requirements that allow consumers to implement policy. This policy is in place to ensure the ongoing operation of the services. The policies allow for assertions, which are based on unique names that are globally consistent through the time and space, based on any service.

3.1.7. WSDL

The WSDL is a contract which the clients use to communicate with the web service. The contract allows for structured communication without having to share the same implementation, which specifies to Don Box's main principles behind service orientation. Figure 3-2 is a basic description snippet of a WSDL.

```
<MESSAGE NAME="GETSHOWREQUEST">
  <PART NAME="TIME" TYPE="XS:STRING"/>
</MESSAGE>

<MESSAGE NAME="GETSHOWRESPONSE">
  <PART NAME="SHOW" TYPE="XS:STRING"/>
</MESSAGE>

<PORTTYPE NAME="SHOWBYTIME">
  <OPERATION NAME="GETSHOW">
    <INPUT MESSAGE="GETSHOWREQUEST"/>
    <OUTPUT MESSAGE="GETSHOWRESPONSE"/>
  </OPERATION>
</PORTTYPE>
```

Figure 3-2. Snippet from WSDL file

WSDL documents have predefined elements that allow the contract to be defined. The predefined elements are:

- **Types**– a container for data type definitions using some type system (such as XSD).
- **Message**– an abstract; typed definition of the data being communicated.
- **Operation**– an abstract description of an action supported by the service.
- **Port Type**–an abstract set of operations supported by one or more endpoints.
- **Binding**– a concrete protocol and data format specification for a particular port type.

- **Port**– a single endpoint defined as a combination of a binding and a network address.
- **Service**– a collection of related endpoints.”

Most of the major business vendors like Microsoft WCF [W22], IBM SOA [W23], Oracle (previously BEA) [W24], and many others, are supporting SOA. Therefore, SOA currently runs most business operations.

One of the problems with SOA is that it relies usually on complex constructs and sometimes described as “big” [12]. The tooling provided by vendors for SOA creates a big abstraction layer. The abstraction layer generates increased overhead and can cause side effects if the user does not know how to master a specific framework. This is because most of the frameworks rely on configurations that are located in XML files, like web.config in WCF.

SOA is taxing on clients due to XML parsing. Clients have to create SOAP document envelopes [W25] and transfer all of the data through POST commands. On thin clients, this type of operation can take a significant toll on performance, as thin clients usually prefer to have few light threads working in the background, besides the user interface thread [W26].

SOA can also cause significant problems on caching operations. Because files are transferred back and forth between the client and servers, the documents have to be parsed to decipher the data that needs to be sent from the cache. Also, deciphering what is cacheable and what is non-cacheable can be a challenge. It is fairly easy to design an SOA architecture that does not have a semantic way of invalidating cache.

SOA forces the client to keep state information. The client must know all of the operations that it must perform. Usually the clients must be updated every time operations change on the server. This can be very costly, as this means that the client application must know a lot of the

details of the server and how they work. Security in SOA is very easy to implement, but it is also taxing on clients because it requires parsing the documents.

SOA forces the client to keep state information. The client must know all of the operations that it must perform. Usually the clients must be updated every time operations change on the server. On the other side, because SOA has great tools available, it allows developers to get started in minutes and update their services with a few clicks.

3.1.8. Summary: SOA-WS*

SOA is a great architectural pattern for implementing web services quickly due to the large number of tools available on the platforms. Sadly, the main problem in SOA comes from the overhead in tooling, transfer, security protocols, and a caching mechanisms make it less optimal to scale.

3.1.9. Web Services using RESTful Architectures

To be able to understand RESTful architectures one must first understand the HTTP protocol. The protocol is a request response based protocol in which a client/consumer sends requests to a server, Figure 3-3 and the server sends a response, Figure 3-4.

A HTTP request is composed of:

- Request Line
- Headers (optional)
- Empty line (as a separator)
- Body (optional)

Result	Protocol	Host	URL
200	HTTP	www.google.ca	/
200	HTTP	ssl.gstatic.com	/gb/images/b_8d5afc09.png
200	HTTP	www.google.ca	/images/nav_logo83.png
200	HTTP	www.google.ca	/images/srpr/logo3w.png
200	HTTP	www.google.ca	/extern_js/f/CgJlbhICY2ErMEU4ACwrMFo4ACwr
204	HTTP	www.google.ca	/csi?v=3&s=webhp&action=&srt=238&e=1725

Figure 3-3. Sample HTTP requests with their Status Response

```
GET http://www.google.ca/ HTTP/1.1
Host: www.google.ca
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: NID=50=VNQXwXJ9jknXQzkaW0Xw8d1_PejkEW2ndmh1zI_khRZEajK0o-
```

Figure 3-4. Sample HTTP Response

3.1.9.1. Request Line

The request line is composed of nine verbs that tell the server how the request should be processed.

Table 3-1. HTTP Verb List

HEAD	Requests for metadata information for the resource. The metadata is composed of all of the headers involved in a usual GET response but without the body of the response. It is an idempotent operation.
GET	Is a retrieval operation for a resource that should not have any side effects. It is an

	idempotent operation.
POST	Sends data as form of the body. The goal of this operation is to annotate existing resources, posting an HTML form, data-handling, or appending to a database.
PUT	Sends request data that creates/updates a current resource on the server. It is an idempotent operation.
DELETE	Removes a resource from the server. It is an idempotent operation.
TRACE	Echoes the request back to the consumer/client. The goal is to check for any changes, additions, or removals of intermediate resources. It is an idempotent operation.
OPTIONS	Is mainly a response that serves all of the possible HTTP methods supported by the resource. It is usually to check functionality. It is an idempotent operation.
CONNECT	Changes the request and falls back to a TCP/IP tunnel, which can be used to encrypt communications on an unencrypted connection.

PATCH	Provides partial modifications at any point of the resource.
-------	--

A well-formed HTTP response should contain a response status and a body, depending on the resource on the server.

Some of the most common response status codes are:

Table 3-2. Common HTTP Response List

100	Continue
200	Ok
201	Resource Created
301	Moved Permanently
307	Temporary Redirect
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
405	Method Not Allowed
408	Request Timeout
505	Internal Server Error

On the other hand, RESTful architectures are very different. They were proposed by Roy Fielding. He describes REST as “a hybrid style derived from several of the network-based

architectural styles and combined with additional constraints that define a uniform connector interface.” [13]

REST, as in SOA, is a distributed system that allows the service side and the consumer side to be decoupled. The main difference between REST and SOA is hypermedia and the uniform connector interface, which is explained below using the Richardson Maturity Model [14].

3.1.9.2. Richardson Maturity Model on REST

Leonard Richardson [14] explains the main points of REST as levels of maturity in the protocol

Figure 3-5.

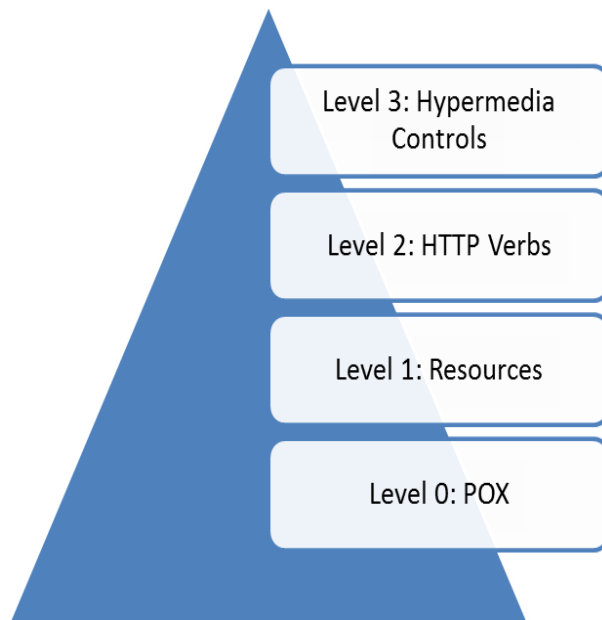


Figure 3-5. Richardson Maturity Model [14]

3.1.9.3. Level 0

Level 0 is a just a plain transport level for data for remote interactions using the HTTP protocol. This is how the remote procedure invocation works by posting messages from one machine to another, usually using XML documents.

3.1.9.4. Level 1

Level 1 is composed of resources. Resources are important because they allow the system to be decoupled and to abide by the single responsibility principle to a specific resource. This allows for a resource to have decoupling from the interface level with the data that the resource contains. This decoupling allows for a better implementation of the resource, as updates to a single resource can be easily differentiated from other resources by URI.

3.1.9.5. Level 2

Level 2 has distinct verb interaction with the resources. The distinct use of verbs allows for the use of idempotent verbs, which permits retrieval operations to happen at any point without any consequence. It might not seem like an important feature but due to this feature, it is possible to build a scalable and an available system using features of HTTP. Using only the verb GET for retrieval implies that a system that depends on sub systems is capable of keeping a resource cached (as long as there is enough memory on the machine) and not having to alter the state of the sub systems. This allows for the resource to perform extremely fast and even be replicated among multiple machines, performing as that resource. The only time a resource cache has to be invalidated is when a CREATE, UPDATE, or a DELETE operation has been performed, that can be easily distinguished by the method used when using verbs for interaction.

3.1.9.6. Level 3

Level 3 is being able to keep state on the server. The crucial difference between all of the other levels and this level is that this level is hypermedia. Hypermedia is a runtime retrieval system that allows users or programs to retrieve related information particular to a specific subject. It uses a decentralized manner to retrieve resources. As opposed to SOA, REST is much more flexible and therefore, more scalable. As previously discussed, SOA has many platforms that leverage the nature of contracts and behavior in service orientation. The main disadvantage with this approach is that the consumer is hard bound to the service by the type of data that it must receive and by the specific methods that it has to call. In a properly developed REST service by just having to know the specific uniform connector interface the entire interface contract can be removed from the client-service interaction because the client is just following the servers behaviors.

The flexibility of REST can be seen by the use of the “content-type” and “accept” headers in REST requests. The important aspect of hypermedia is that state is kept by the uniform connector interface. The biggest impact of REST is due to hypermedia and the URI. The server keeps the state of the client and therefore, fewer resources are needed by the client.

RESTful services resources usually have clear specifications of a small set of verbs to perform operations over those resources. The RESTful services that use GET and POST as their primary verbs, are called lo-REST [12].

The services that use the full set of verbs are called hi-REST [12]:

- GET
- PUT
- POST
- DELETE

3.1.10. Summary: Web Services SOA-WS* VS REST

Based on background research it is clear to see that RESTful architectures can provide greater scalability. RESTful architectures require less overhead because they are usually custom built, are easily cacheable due to the semantics of their verb operations. Finally they are great for thin clients as the operational state is kept on the client.

3.2. Dependable Web Services

Dependability in web services has been tackled in many areas in the WS* with SOA. The following is a review of the work performed in that area but there is almost no information about dependable RESTful web services. From here I will look into the definition dependability and strategies to enhance it in web services. In many of the cases dependability has been described as the trustworthiness of a service [15] under the web service environment. For a service to be fully dependable it is necessary to look into several areas where services can fail. A list of potential problems include:

- “1) Crash of services*
- 2) Crash of server*
- 3) Hang of service*
- 4) Corruption of data*
- 5) Duplicate messages*
- 6) Omission of messages*
- 7) Delays” [16]*

Based on this list it is necessary to have control over the hardware, software and communication of the services to be able to identify when there might be a failure like the one mentioned above. Some of the techniques [16] to detect failures are designed based on

middleware detecting/rejecting corrupt data, duplicated messages, omitted messages, timeouts and giving proper error messages.

To be able to prove that it is a dependable architecture it is necessary to use known methods of determining dependability. The two methods available [15] are modeling or measurement techniques. Modeling requires full access to the 3rd party service to generate a model of the service behavior and measurement techniques involve basically blackbox testing. Measurement techniques are necessary on this system because there are many components we will not have full access to and which might not reside under our system.

Under the work performed “Web Services Dependability and Performance Monitoring” it was established that it was possible to develop a tool that could measure

1. availability
2. functionality
3. performance
4. faults/exceptions.

From here it is possible to dive deeper into each one of the properties.

Looking at “Enhancing Web Services Availability” [17] it is possible to also think in terms of the system as that is composed of infrastructure, middleware and application availability. High availability [17] can be described as $r = (1-p)^{l \cdot m \cdot n}$ where l is the number of activities per day, n is the number of tiers in a web service architecture and m is the number of web services the activity utilizes, p is the probability the service will fail leaving r as the probability of having the system available. With the introduction of a highly available system it is possible to look into the functionality of the system. For the functionality of the system it is appropriate to look into reliability. The reliability of the system is highly dependent of the availability of the system

using web service replication. There are three main ways of using replication to provide reliable web services. The first technique is N-version programming [19] which involves having different implementations for web services. If there is a faulty web service then other implementations can be tried since they might be able to avoid the error state. It is necessary to avoid using this versioning method as it would put a high level of overhead on 3rd party developers by making multiple versions, it would be difficult to deploy and not be cost effective. Active Replication [20] [21] is the second technique, active replication is basically sending messages in a FIFO manner in a multicast to multiple services which would perform the same task. This way multiple services can respond. The one that responds the fastest and with a correct answer is the one that is taken. The final third technique is using passive replication [19] which involves having a replication manager, fault detection and fault notifications, recovery and logging.

Enforcing dependability in web services is a tricky task because web services are not meant to be transactional by nature as described by “Dependability in the Web Services Architecture” [21]. There are two main problems: management of transactions and locking of resources. In the management of transactions a common interface is required that some web services may not have and it violates the autonomy and isolation of web services. Finally locking of resources is not appropriate for web services because of their autonomy and for scalability reasons. On the good side, because all of the subscribers are known to the applications since they must sign up with the telco company for the paid service, it is not required to worry about how many resources are needed to be provisioned, since the exact number of users are known.

3.2.1 Summary: Dependable Web Services

There has been a lot of work performed under the WS* SOA stack for dependable web services. Much of the research in WS* SOA involves increasing availability and reliability through availability. For reliability there were two main methods that are viable, active and passive

replication. It is also clear the functionality and performance are also key aspects of dependability. From here I will aim to specify a standardized way to define dependability, directed towards web services using the REST architectural pattern.

3.3. A new approach of defining Dependability in Web Services

Allowing 3rd party developers to integrate their services to an open IPTV architecture causes discrepancies when attributing failures or blame. Most of the time, the telecommunication companies will be attributed with the blame that their IPTV service is not working correctly, when in reality, it is a failure from a 3rd party application. Service call complaints, on average, cost \$8; therefore, it is vital to have dependability over the 3rd party applications.

For our purpose, we have broken down dependability into the following components based on the fundamental concepts of dependability [22]:

- Confidentiality
- Reliability
- Availability
- Safety
- Integrity
- Maintainability

These fundamental concepts have led to a view of dependability regarding the main affliction factors:

- Faults
- Errors
- Failures

Using those concepts we plan to increase dependability using the attributes of dependability by providing:

- Fault prevention
- Fault tolerance
- Fault removal
- Fault forecasting

We can summarize this information with Figure 3-6, as the dependability tree from the “Fundamental Concepts of Dependability” [22]:

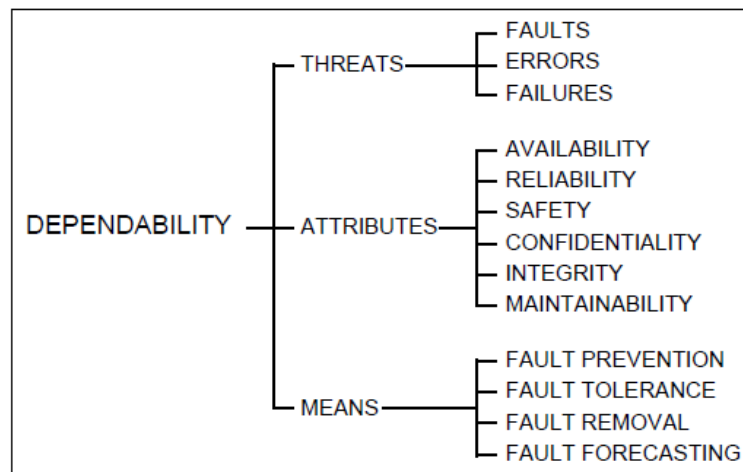


Figure 3-6. Dependability Tree [22]

Errors

We are treating errors as a state that causes a failure and may propagate to cause other failures.

Fault

We define faults as the reason why errors appear in the system.

Failure

We define failures as a breakdown or a malfunction in the system.

3.3.1. Attributes of Dependability

To ensure that our system is dependable, we need to add the following attributes to our architecture. Since our system is based on web services, we have to apply the following attributes to the web services components in the architecture.

3.3.1.1. Confidentiality

“Ensuring that information is accessible only to those authorized to have access” [23]. To add value to the user experience, 3rd party applications must have access to user information. In some cases, this user information can be anything from username to personal address. Therefore, it is important to make sure that no information leaks by just using architecture. For most usual purposes, the standard technology, which provides secure transfer of information, is Secure Sockets Layer using HTTPS [W27].

3.3.1.2. Reliable

“The probability that the software will give the correct result...” [24]. A user must be able to open up IPTV programs from their set top box and interact with them. More importantly, the program will be able to perform its function to the best possible effort. To be able to ensure that we are able to provide the correct service, we will work the benefits of the HTTP protocol status codes. Using status codes, we are able to ensure that the right result is being delivered (in the 200 level) in a specified amount of time to keep quality of service. One of the techniques available is fault removal and reliability of service is strongly coupled with our ability to be highly available. We can **remove the faults** to make sure that they never reach the users and we can use other services to correct service using high availability.

3.3.1.3. Availability

“The ability of an item to be in a state to perform a required function at a given instant of time or at any instant of time within a given time interval, assuming that the external resources, if required, are provide” [W28]. One of the main issues with web services is availability. This is an important issue, especially when a large amount of users are not able to perform actions, due to unavailability problems with the web services. This lack of assurance can cause extreme dissatisfaction, lower expectations, and eventually, cause users to leave a provider for another. Using high availability tied with high reliability we can increase **fault tolerance**. If we look at the definition of reliability in the sense of “probability of failure-free operation of a computer program for a specified time in a specified environment” [25], the services provided by 3rd party applications must have enough availability to increase the probability of a failure free operation and be available for the users to have the expected result.

We expect to have a high level of redundancy, which allows for a highly availability environment. Also, we will have services that easily attach to the architecture and therefore, increase capabilities. One of the advantages is that we are able to select the web services that perform the best and skip the ones that are in an error state, which cause errors to propagate. Finally, in some cases, even multiple services that perform the same function can be invoked at the same time to increase chances of a correct result that will return in the smallest amount of time possible.

3.3.1.4. Safe

“Absence of catastrophic consequences on the users and the environment” [26]. The architecture must be safe and shield failure of one service from another. Failure in a 3rd party system must be isolated in order to avoid the creation of cascading failures that affect other parties. We are concerned with the following reliability problems [W29]:

- “A failure [that] could cause the client system to crash while performing an operation, and;
- An outage [that] could disrupt connections from the client system to other services.”

3.3.1.5. Integrity

“Absence of improper system state alterations” [22]; for this, we deal with unauthorized changes to the states of the web services. Therefore, we are shielding and sandboxing each web service to make sure that only the web service itself can change its state, fault prevention.

3.3.1.6. Maintainability

“Ability to undergo repairs and modifications” [22]; in a highly redundant and available environment, a lot of the modifications and repairs to the system must occur in a fast manner and be automated to be feasible. Also, it is important that all of the changes done to the architecture are under version control and as modular as possible. Modularity is an important aspect, as only the smallest unit change will require the least effort to change in a distributed environment and to make rollback easier.

3.3.2. Summary: Adding Dependability to Web Services

To be able to have dependable web services the following properties will be added to our architecture:

- Availability
- Reliability
- Safety
- Confidentiality
- Integrity
- Maintainability

3.4. Fault Injection

Fault injection is a well proven technique to assess the dependability of a system [27]. This is a technique specially designed to see how a system behaves when an error occurs and how errors are handled. In our particular case we primarily want to make sure that any errors found in the system will not propagate to other areas and are contained. The containment of the errors, and when possible, fixing the errors will be the primary way to check the correctness of our system.

To prove that the architecture has the attributes of dependability that we mentioned above, we are going to introduce fault injection as a way to validate dependability [28]. Our goal is to:

- “Identify dependability bottlenecks”
- “Be aware of the behavior changes due to the presence of faults”
- “Error detection and recovery”

Using these goals, we will be able to successfully quantify the dependability of the system. To quantify the dependability of the system it is necessary to add meaningful and specific faults to the system. It was decided not to use random corruption of bytes because to give better coverage we want to change individual values that will cause specific failures. With the specific failures it is possible to have specific test coverage in a black box system like the one created. Many systems that provide dependability rely on a backward recovery system which usually means that it will be able to roll back to erase the error state [29]. In our case we do not want to have to roll back errors as they block the progress of other requests so the technique we use is forward which can be simplified as exception handling [29]. The main focus of the fault injection is researching how the system can move forward without having to go backward causing locking of resources.

3.5. CAP Theorem

The possibility of a distributed architecture that has consistent, available, and partition-tolerant properties is a balance issue, Figure 3-7, with any distributed web service centric architecture [29].

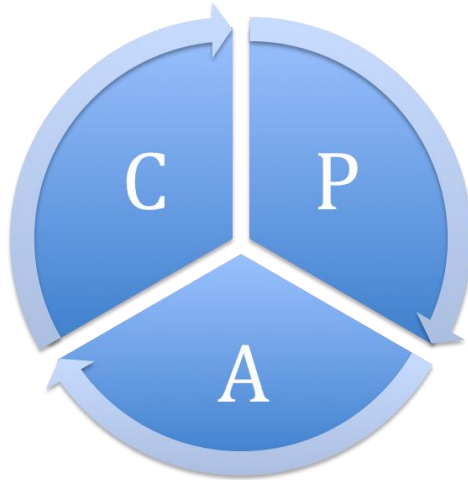


Figure 3-7. Cap Theorem Balance

Table 3-3. Cap Theorem Breakdown

Consistency	The systems with all of the nodes have access to the exact same data.
Availability	The system is capable of responding to requests, even if the responses are out of date or reporting that no correct response could be found.
Partition Tolerance	The system is able to still operate even when some messages cannot be delivered.

Consistency is broken down into hard state and soft state. Hard state involves having all nodes stop working until all of them share the same data. Soft state is having all nodes working even if the data is a bit stale. Of course the complexity increases significantly as the system grows. When there are more nodes in the hard state, the time when a lock is in place will increase until the data is propagated to all the nodes, therefore making the system temporarily unavailable. In the soft state, as the number of nodes increases, the inconsistency of every node increases as a function of latency and will be able to integrate the new data.

Availability can be measured by the number of nodes that have the same data. As the number of nodes with the same data increases, consistency decreases in a soft state system. Also, for data to be fully available all of the nodes must share all of the data and therefore bandwidth will be an issue. To lessen the problem fast, large pipe networks transports must be used. The main problem arises when even a single message is lost because then the specific data is not available.

Partition tolerance means that messages may be lost but no problems arise since the system has countermeasures. These countermeasures can consist of data being partitioned by specific data value keys between nodes. Also for further checks under replication such as two face commits can be added but that increases latency, which can become a very bad problem in larger distributed systems since components have to interact with many other components increasing time spent with each transaction.

Consistency and Availability

By providing consistency and availability means that the system has the **accurate data available on all of the nodes**. The main problem with this approach is that there is no partition tolerance; therefore if even one single message is lost the system becomes unavailable.

Partition Tolerance and Availability

By providing partition tolerance and availability it means that the system has **higher resilience because even when messages are lost, the system can still function and the system will always reply with some response**, even if the response is old or an error message.

Consistency and Partition Tolerance

Providing consistency and partition tolerance means that the system has all of the **data synchronized and it is able to deal with lost messages**. The main problem with this approach is that the system will have to use locks to make sure the data is up to date, making the system unavailable.

Because we want to create a fully available system we have decided to forego consistency for a highly available and partition tolerant system that has as few locks as possible.

3.5.1. Summary: CAP Theorem

Consistency, availability, and partition-tolerance are some key elements necessary to have a perfect distributed system. Unfortunately, the main point of the CAP theorem is that it is impossible to deliver all three properties; only two can be served at any time. The main problems increase in complexity as the number of nodes in the system increases. We will sacrifice consistency to acquire the highest level of availability and partition tolerance.

3.6. Databases

MySQL, Microsoft SQL, and PostgreSQL [W30] all follow the ACID principle properties [W31]:

- Atomicity
- Consistency
- Isolation

- Durability

Because of the properties of these databases we will have the ACID properties on our architecture when performing transactions. These high performance databases still have a drawback especially since they work by locking. Locking prevents operations to be performed concurrently and therefore, decreases performance. Also when tables grow large, performance decreases significantly. When tables grow significantly large that they do not fit in RAM and tables are not indexed in a careful fashion, performance will suffer significantly. As the databases grow larger the harder it is to join data, especially if the data tablets with data have not been optimized to be indexed. Even if they have been indexed, indexing causes overhead when writing the data. These problems occur specially on large data sets. To be able to deal with large datasets companies are now offering databases on the cloud. Databases on the cloud allow developers to store large quantities of data and not worry about the storage problems. Some examples are:

- Microsoft SQL Azure [W31]
- Xeround [W32]

The main problem is that these offerings do not particularly provide performance guarantees but mostly storage capacities guarantees.

Since these ACID based databases promote data normalization, it takes longer for the databases to aggregate the appropriate data requested. Secondly, one of the problems arising from performance in databases is that most developers use querying frameworks or object relational mappers to create the queries on a higher abstraction level. Many of these frameworks are very popular:

- LINQ to Entities [W33]

- Hibernate [W34]
- Active Record [W35]

The main problem is that a great degree of expertise on the specific framework is required to make sure the query is written in an optimal fashion. The most common problem is N+1 queries, when fetching lists with specific information; each row must be checked as its own query [W36]. This is because the primary query did not fetch enough information and has to perform multiple queries. This can be prevented by eager loading techniques.

Recently, there has been a movement for many developers to drop normalized storage for denormalized storage. This is what is commonly known as No-SQL. No-SQL storage is (mostly) unstructured storage that is combined with sharding [W37] and caching in most cases. There are many different available solutions for No-SQL databases, such as:

- MongoDB [W38]
- Cassandra [W39]
- CouchDB [W40]
- SimpleDB [W41]
- Google Big Table [W42]
- Azure Table Storage [W43]

Usually, each row will contain different types of data and all of the data that a single request should need. This allows for single fast queries.

No-SQL excels at throughput and scalability. The main reason why No-SQL databases are extremely fast is because they have constraints that favor doing queries with identity keys for each item. Also, they tend to facilitate scalability as they are designed for distributed storage. This means that a database might only have to handle some queries based on specifications of the

identity keys. This is done especially for partition tolerance and availability, which increases scalability. Since partition tolerance and availability are primarily taken with No-SQL, consistency is neglected.

They work under the BASE approach [W44]:

- Best Available
- Soft state
- Eventual consistency

This approach allows for multiple data replicas to increase availability and scalability that might not be in sync with each other (consistent) but give the best performance. We would like to use the BASE approach for caching our information since it will give the best availability and performance for non-critical information.

3.6.1. Summary: Databases

With the use of SQL and No-SQL databases it is possible to have the best availability and keep the ACID properties in transactional systems.

3.7. Cloud Computing

It soon becomes clear that for large telecommunication companies providing IPTV services, there are many important issues, such as scalability. As of 2008, Sasktel had 70,463 Max subscribers [W45]. This means that every single one of those users is consuming high levels of bandwidth to interact with IPTV content. As the number of Max subscribers increases and the subscribers buy companion devices, the number of thin clients grows exponentially. Most web applications are not optimized to scale to such a large audience. There are three types of cloud computing:

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)

- Software as a Service (SaaS)

3.7.1. Infrastructure as a Service

It provides developers with dedicated virtual machines that they must fully manage. This also means that the developers have full control over the whole server and what operating system, features, and software it contains. This is what Amazon currently provides on their EC2 [W46] cloud. This type of approach has the advantages that the computers can have any type of operating system, license, and security features, which are fully controllable. Due to this full control, IaaS requires much more maintenance. If 3rd party developers create applications under an IaaS it would mean that they would have to manage everything themselves.

3.7.2. Platform as a Service

It is a cloud architecture that was built to conform to specific sets of standards and uses. The developers have very little to no control over how the virtual machines behave. This is what Heroku [W47], AppHarbor [W48], and Windows Azure [W49] provide on their platforms. The advantage of using these platforms is that developers just have to know how the platform stack works and they are able to have freedom on how to develop functionality, as long as they stay within the limits of the platform. Deviating from the main goals of the platform will usually cause things to be difficult to maintain or develop.

3.7.3. Software as a Service (SaaS)

SaaS allows clients to use software functions exposed through an API, usually over web services. Some examples include Facebook Graph API, Google, Twitter, and Ebay. The main advantage of this approach is that the level of abstraction is high and developers mainly have to worry about their business product, instead of the how a platform works or of the infrastructure.

Cloud architectures are very appealing because of [W50]:

1. “Cost Efficiency

- a. Dealing with burst loads, only pay what is used
- 2. Storage Capabilities
 - a. Unlimited storage capacity, only pay what is used
- 3. Redundancy
 - a. The data is securely stored; most cloud providers give up to 3 levels of redundancy”

As described by Amazon, there are many benefits for many small companies, such as [W51]:

- “Scale Capacity on demand
- Turns fixed costs into variable costs
- Always available
- Rock-solid reliability
- Simple APIs and conceptual models
- Cost-effective
- Reduces time to market
- Focuses on Product & core competencies”

In a cloud context [31], there are several benefits for thin clients and REST:

- *“Rest is stateless*
 - *minimizing the impact of network volatility*
- *REST is URL based*
 - *therefore easy to invoke*
- *REST responses are usually HTTP based*
 - *therefore discrete*
 - *also minimizes the impact of network volatility*
- *REST delivery can be made very succinct*
 - *lends itself to constrained memory environments*
 - *no superfluous protocol elements”*

One of the main and most popular techniques to increase availability is to use intermediaries. It helps by making services available from a number of devices and it can utilize a fast content delivery network (CDN). The CDN network also reduces latency, which is an extremely important variable, when scaling web applications, especially if clients are all over the world, because resources are tied up until the network delivers content. The CDN network uses its own resources and does not use any resources from a business application, when transferring data. There are two types of intermediaries: functional and optimizing [32]. With either method, one of the biggest helpers of scalability is to avoid optimization but to have a high level of redundancy [33].

One of the main benefits of RESTful architectures in the cloud is that they provide multiple layers, where content can be cached. Also, it provides the HEAD [W52] verb request, which allows a developer to check if a resource has been changed since it was last accessed by providing a hash called ETag [W53]. This is especially important in layered systems, where resources might have to be aggregated; only the resources that have changed have to be fetched again.

Using HTTP verbs, it is possible to optimize systems because the GET operation is idempotent and it is only meant for retrieval of information. PUT, POST and DELETE can change the state of a resource and therefore can be used to invalidate the cache for the GET operations.

3.7.4. Summary: Cloud Computing

Cloud computing can help provision enough resources to support the large needs for hosting 3rd party IPTV services. In this way 3rd party developers can have SaaS cloud applications that can scale. With the use of REST having multiple resources that cache content is very effective as the cache can be easily invalidated when needed.

3.8. Sandboxing

“Sandboxing is a technique for creating confined execution environments to protect sensitive resources from illegal access. A sandbox, as a container, limits or reduces the level of access its applications have.” [34]. The main idea of sandboxing is creating confined environments that are fully under control of a supervisor. There are three main techniques for this tasks which include access control lists (ACLs) [34] and special purpose sandboxes specific for the application/service being run [34] and application sandboxes based on system call contexts [35].

The following are the primary techniques available for sandboxing:

- Virtualization
- Rule-based Execution

We have looked extensively into virtualization but sadly the costs of giving each 3rd party service is currently at least 12 cents per resource on amazon EC2 . If we want to have high availability and reliability the services must be replicated to multiple dedicated instances which multiply the costs based on the availability and reliability desired making it very expensive and wasteful since some resources might not be heavily used and it could waste CPU cycles. One option would be to buy a large machine and virtualize several environments but that carries several performance drawbacks due to the overhead of each of the virtualization environments [34]. The main advantage to the virtualization approach is that it has full fault isolation as it is on its own environment.

We have chosen a rule-based execution to avoid the performance drawbacks of virtualization and utilize resources in the best way possible. In particular, we have chosen application domain under the .NET framework which has several advantages [W54] [W55] that we will explain below.

One of the main advantages to .NET AppDomains is that it is possible to have specific configurations for each service. This allows us to have expressive policies for each 3rd party application. Each policy is built from several rules, rule based configuration for each application under the application domain. Each application can have a smaller memory footprint; multiple applications can run under the same process, this can even increase performance as these applications can share the .NET runtime libraries. The way that the AppDomains are set up also provides full fault isolation for each application that runs in the master process which means that multiple applications can run in the same process without affecting each other. Faults from one application domain will not crash the entire host application or affect other applications in different domains.

3.9. Service Level Agreements

“Service Level Agreements (SLA)s are signed between two parties for satisfying clients, managing expectations, regulating resources and controlling costs” [36]. Most of the time, these guaranties involve parameters such as response time, throughput, and a condition stating that when there is a deviation and/or failure to meet the agreement, the consumer must be informed.

In a complex environment, metrics must be captured and aggregated to have some relevant data for the SLA parameters. For this, a supervisor must be injected to be able to capture and monitor all of the data to make sure the SLA are not violated. Currently, there is the WSLA language [6], which is capable of describing these types of service level agreements for web services.

The descriptions include:

- Parties, Roles and Actions
- SLA parameters, measures, aggregation and appointment of a supervisor

- Service Level Objectives and Action guarantees

We want our system to be able to have the following functionality:

- SLA parameters, measurements, aggregation and appointment
- Service Level Objectives and Action guarantees

Using this information we can generate table and give different users different types of services based on the categories as described in “A concept for QoS integration in Web services” [37] of processing times and services.

Table 3-4. Types of SLA based services

Class of Service	Platinum	Gold	Bronze
Max Processing Time	.10ms	.30ms	.70ms
Throughput	5000 requests/s	1000 requests/s	500 requests/s
Price per service usage in hours	\$0.24	\$0.14	\$0.05

SLA based web service quality monitoring allows to differentiate between the types of services that are provided to 3rd party applications. This can be best performed by a system that is broken into three different areas [38]:

1. Measurement
2. Monitor
3. Analyzer

The way the SLAs are set is a bit different because we are working with services backed by the cloud. This means that we can drive provision based on the SLAs. The work performed by “Autonomic SLA-driven Provisioning for Cloud Applications” explores specific approaches on WS* based web services towards [39]:

- Adaptive adjustment
- Cost-effective resources allocation
- Detection and removal/replacement of stale resources
- Component replication and migration depending on load variations

For the focus of this research I am mainly interested in being able to provide a way of providing the events/alerts to be able to later allow for the approaches mentioned above. That would be the focus for the future work of this thesis since we are primarily working with RESTful web services.

3.10. Summary of Literature Review

Using the review from the CAP Theorem, Web Services, Dependable Web Services, Fault Injection, Cloud Computing, Service Level Agreements and Sandboxing I have identified the following patterns. From the CAP theorem we have decided that we will focus on availability and partition tolerance. We have chosen RESTful web services as they are easy to cache and have inherited properties that we can use to provide the greatest dependability. I will use availability, reliability, safety, confidentiality, integrity and maintainability to have a dependable system. Fault injection is used to check the dependability of the system. Cloud computing is used to provision resources and provide scalability in conjunction with REST. Service level agreements are used to make sure the system is able to keep its promises to 3rd party developers in terms of performance and sandboxing using rule-based execution prevents any harm to the system because it does not require full operating system virtualization.

Table 3-5. Summary of Literary Review

<p>Web Services SOA –WS* and REST</p>	<ul style="list-style-type: none"> • Adoption of web services [W13] • Distributed applications with web services [8] • Finding suitable Web Services [10] • Service descriptions and policies [11] [23] • Rest over SOA [9]
<p>Dependable Web Services</p>	<ul style="list-style-type: none"> • Definition [15] [16] [17] [26] • Fundamental Concepts of Dependability [22] • High Availability [18] [32][33] • Dependability/Reliability [16] • Assessing dependability [15] • Dependability Techniques [19][20][21]
<p>Fault Injection</p>	<ul style="list-style-type: none"> • Testing dependability using fault injection [27] • Fault injection techniques [29]
<p>Cloud Computing</p>	<ul style="list-style-type: none"> • Cloud architecture benefits [W50] [W51] • REST with cloud computing [31] • Cloud computing middleware [32]

CAP Theorem	<ul style="list-style-type: none"> • CAP theorem [30]
Database Storage	<ul style="list-style-type: none"> • ACID properties [W31] • BASE [W44]
Services Level Agreements	<ul style="list-style-type: none"> • Definition [36] • Service level agreement properties [36] • Service level agreement services [37] • Service level agreements core components [38] • Techniques of service level agreements on a cloud environment [39]
Sandboxing	<ul style="list-style-type: none"> • Sandboxing techniques [34] [35] • .NET framework sandboxing [W54][W55]

3.11. Issues Tackled in this Research

The issues not covered by the literary review that are covered in this thesis are the following:

- Dependability using Restful web services instead of WS-*
- Offering a new definition for web service dependability based on availability, reliability, safety, confidentiality, integrity and maintainability
- Hosting 3rd party web services in a sandboxed environment
- Scalable interactive cloud IPTV architectures
- RESTful tracking of SLA agreements

CHAPTER 4 IMPLEMENTATION

This chapter explains the IPTV multi-tenant implementation, based on the requirements outlined in Chapter 2. The IPTV multi-tenant architecture and framework intended to deal with the research goals of **dependability**, **scalability** and **service level agreements**. Figure 4-1 is an overview of the multi-tenant architecture.

4.1. Overview

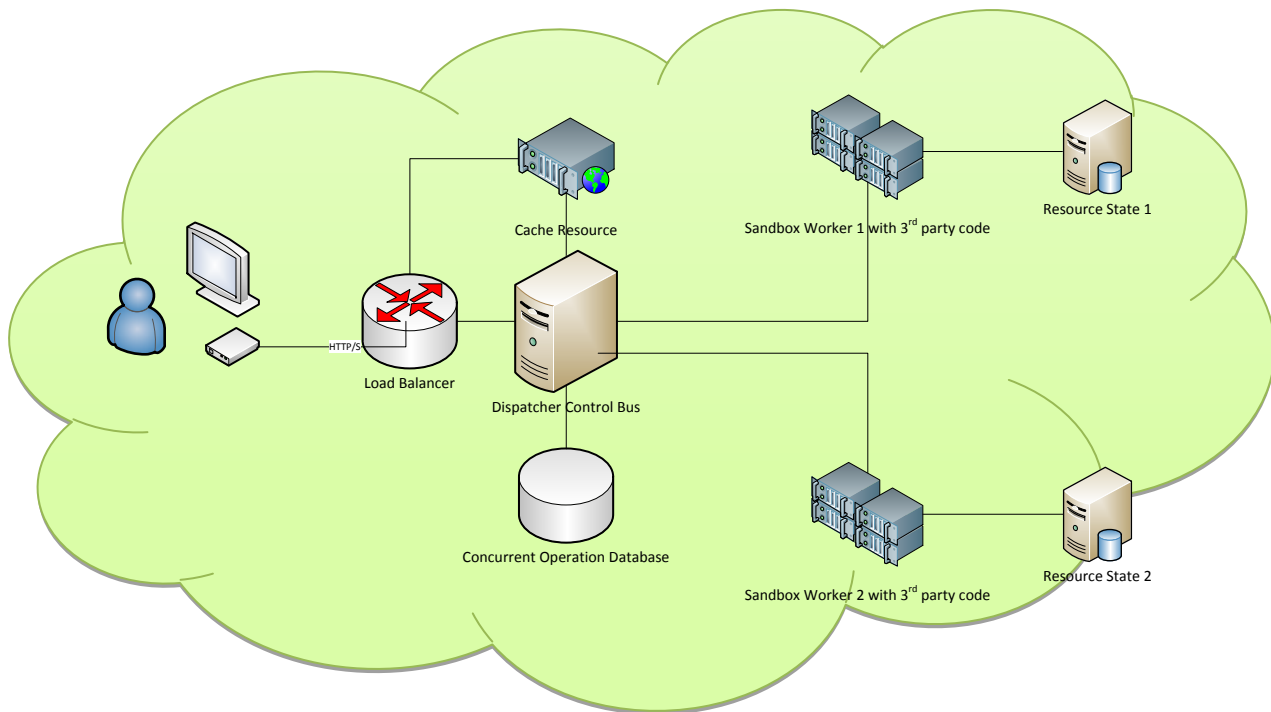


Figure 4-1. Architecture

4.1.1. Detailed explanation of how a request is handled by the architecture

First a user request is generated from a user set top box. The set top box creates a request that gets routed to one of the load balancers in the architecture. The load balancers will then check if the resource has been found on memory. If the request has been found on memory it will respond to that request as per Figure 4-2.

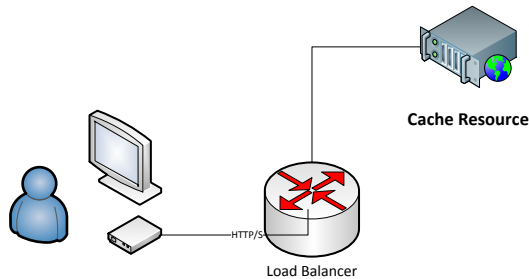


Figure 4-2. HTTPS Request generated by user. If found in cache, return request.

If not cached, the load balancer will decrypt HTTPS at the load balancer level. From there the load balancer will create an operation to be sent to the dispatcher control bus to be processed and wait asynchronously for a result, as in Figure 4-3. The dispatcher control bus is where all of the requests are logged and managed to be able to provide most of the dependability properties on the architecture. The processes and dependability properties at this level of the architecture are explained in detail later in this chapter.

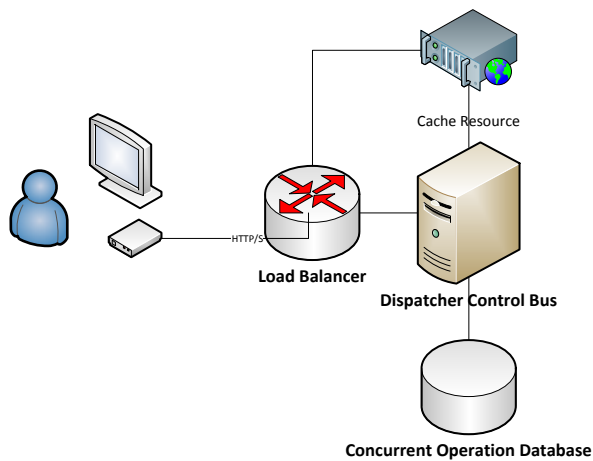


Figure 4-3. Create an operation to be sent to the 3rd party services.

The dispatcher control bus will create an operation to be sent to the 3rd party services to be processed and wait asynchronously for a result, as in Figure 4-4. The 3rd party services are stored in sandboxes environments where from there they can communicate to other sources to create/read/update/delete their state.

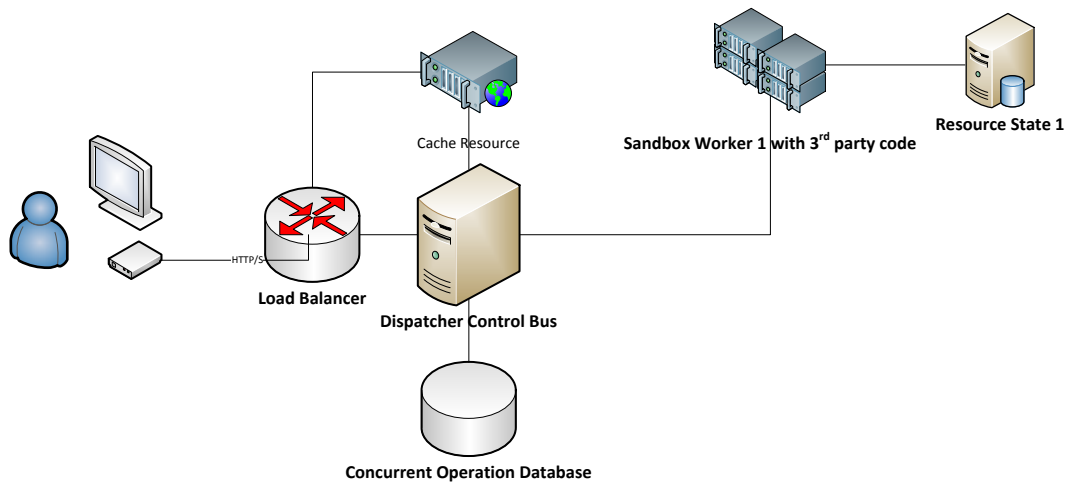


Figure 4-4. Send the operation to the 3rd party service

The dispatcher control bus will then receive the result where then the result will then can be saved. The result can then be saved at the load balancer level or the dispatcher control level. The main difference is that the invalidation of cache can have the greatest control at the dispatcher control bus level since it can perform quick head requests to check if a resource has changed. Saving the result at the load balancer level will have the greatest performance but least cache control, as shown in Figure 4-5.

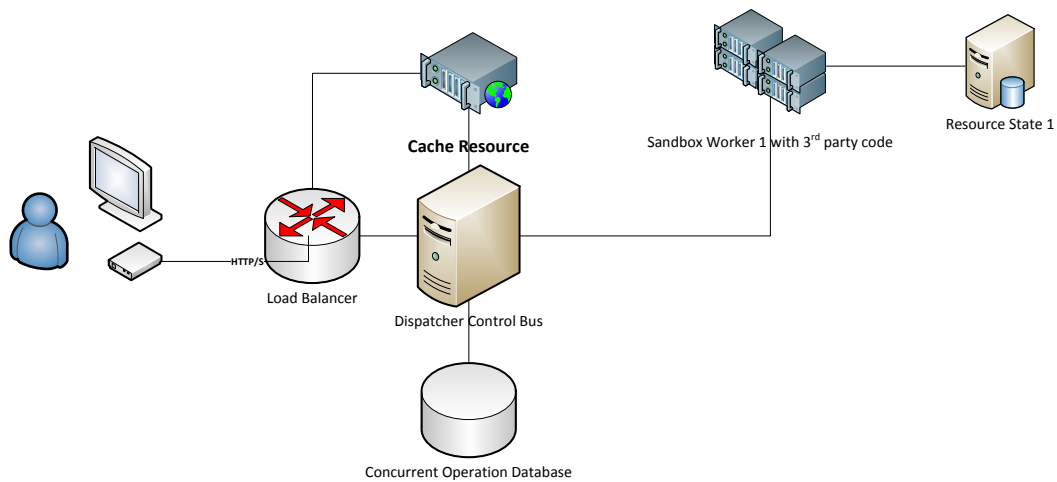


Figure 4-5. Save the operation to the cache

To distribute the workload, the next request that is not found in the cache will then be sent to a different 3rd party resource, as in Figure 4-6. In the case a request fails another resource can be tried or even multicasting a request to multiple 3rd party resources depending on the priority of the request.

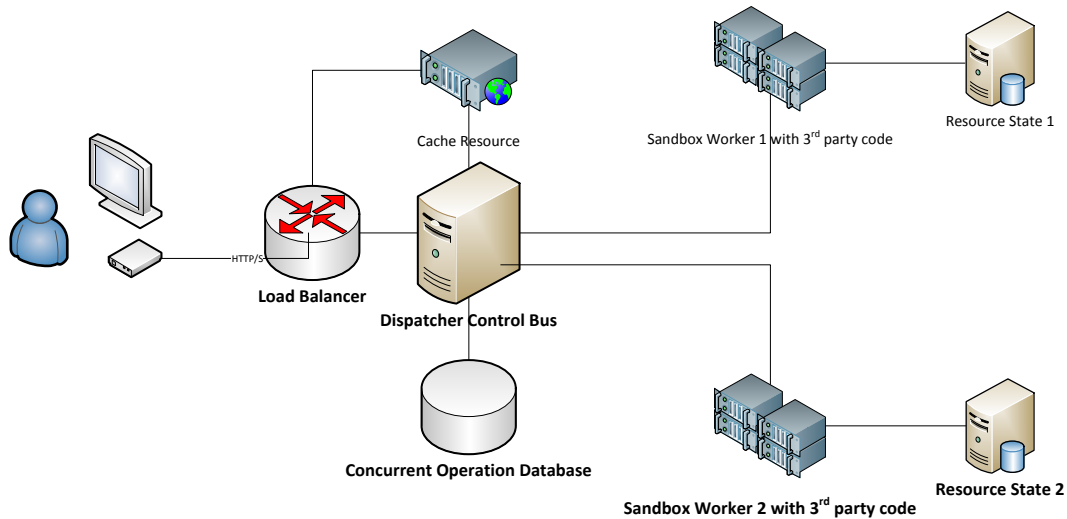


Figure 4-6. Distributing the load for scalability and dependability

4.2. Core Architectural Components

Reverse proxy – The reverse proxy acts as a load balancer, which is an entity that is specialized to receive a larger number of requests and decrypt incoming requests. The main goal of the load balancer is accept all of the connections of users and offload all of the SSL decryption. The load balancer is also responsible for caching the content. It is based on event-driven architecture, which makes it very fast and highly scalable. The event-driven architecture is based on epoll/kqueue. Firstly, by setting up the architecture in this way, we are able to use the spoon-feeding technique. The spoon-feeding technique basically allows for the resources of the load balancer to be consumed when dealing with slow clients.

Restful Service Dispatcher Bus – The service bus is in charge of distributed workers. It uses a round robin algorithm to distribute the requests. Because the service bus has control over the requests from beginning to end, it is able to prioritize, cancel, and augment all of the requests. The dispatcher bus is also able to cancel requests that can cause harm to the architecture, which is considered **fault prevention**. Due to its properties the restful service dispatcher bus can perform **fault removal**. We have added a special feature to the dispatcher bus, which is for workers to automatically bind to the dispatcher bus. This allows for multiple workers to be started asynchronously and increase the power of the architecture. The main goal of the Restful Service Dispatchers is to offload the load from the service workers that actually return data. The service workers should have the minimum amount of resources to keep a service level agreement; therefore, it is vital that their resources are not wasted. For this, we will use Little's Law [W56]:

$$N_Q / N = XR_Q / XR = R_Q / R$$

Using Little's law, we can gather that the queuing time delay is the same as the percentage of the total cycle time (R) when the queued items (N_Q) is the same to the total number of items currently being processed (N). One of the main ways to speed up the queuing time delay is to minimize the number of items actively being processed by the main system. This can be done by not wasting resources from allowing slow clients to directly connect to the workers, as they increase the queuing time delay.

The implementation of the Dispatcher Control Bus is based on the .NET framework. The main advantage is that it is a reliably fast runtime environment. We decided not to use a standard server to host the service bus because while there are many are great frameworks, they have a few drawbacks for dynamic web applications that do not conform to default guidelines:

- Do not work well with prolonged sustained connections
- When a server project are loaded for the first time, can take up to several seconds for the first request to load if the default configuration is used
- If the server workers become idle, application performance is diminished on the first hit (this can be mitigated by not recycling application pools and warming up applications but this could create potential problems, when trying to meet service level objectives)
- Use a threading based model

For this, we have developed our own server from scratch. It is an event driven asynchronous server under the .NET framework. The server performs fairly well; the average throughput of the server is 5,000 requests per second, which is much higher of the basic, non-cached, typical website running under IIS or Apache.

One of the main advantages of using the Dispatcher Control Bus is that it has the role of a supervisor allowing us to inject dependability to the architecture. The Dispatcher Control Bus helps to provide confidentiality, reliability, availability, maintainability, integrity and monitor the service level agreements:

Confidentiality is added to this architecture because the Dispatcher Control Bus sends the information only to the specified resources. This way, we are limiting the data that is available through the network to specific machines running the 3rd party code. As a result, we are able to minimize the unauthorized disclosure of information.

Reliability is performed by the Dispatcher Control Bus by checking the return status of every request that is performed by the resources. If the return status of the resource result is not what is expected, then the Dispatcher Control Bus can send the request again to a different redundant

resource to be able to correct the error state of the first resource. This way we are able to correct the service.

For **availability** we are using k-safety. Basically it is safety in numbers as the distributed system is capable of responding requests as long as at least $K+1$ services are operational, where K is the number of tolerable failures. The Dispatcher Control Bus architecture allows for several resources to perform the same task. It does not matter which way the resources perform the task. It is very easy to create multiple resources that bind to the dispatcher bus and when a fault occurs, they are ready to correct any problems with other resources with the services provided.

The systems **maintainability** is partly performed by the Dispatcher Bus by being able to route requests to different locations. This is very important when hardware fails or when partial failures happen because the requests can be routed and throttled to the functional servers.

The Dispatcher Control Bus is capable of accepting and blocking requests for **integrity**. This way it is capable of staying within its technical parameters and prescribed limits. Also it is capable of blocking requests that might be harmful to the resources. For example, some of the resources might only allow specific verbs to be called within the system but not externally, the dispatcher bus is capable of filtering those requests.

Service Level Agreements are governed because the Dispatcher Control Bus has control over the incoming and outgoing traffic of requests/responses to the clients and we are able to collect all the relevant data required to fulfill the service level agreements. If the agreements may not be maintained then the Dispatcher Control Bus contacts a resource that alerts the developers that there might be a possible problem. This is part of the action guarantees of a service level agreement which helps prevent future problems and report them which is part of **fault**

forecasting. Alerts are email based and developers can either see if the problem is an error on their code or provision more resources in our architecture.

4.3. Worker API

The worker API is a SOA based interface that allows developers to quickly scale their resources. The main reason for using SOA is that it allows developers to quickly get started with the use of SOA tooling that generates the code for them. This is because many developers will want to automate deployment and by giving them access to the SOA API it will be a lot easier for them to get started. We have also developed a website that allows developers to scale their resources and communicates to the same SOA backend.

The website allows us to create projects Figure 4-8. Each account may hold many projects Figure 4-9. Then it is possible to add third party plugin workers Figure 4-10. Each account can have several workers Figure 4-11. All of the plugins must be given permissions to perform specific tasks such as access the file system. Figure 4-12 shows how to give each plugin worker permissions within the architecture. From there it is easy to deploy multiple instances of those workers Figure 4-13. On this site we are also capable of viewing all the statistics and health of all the workers. For management, we have decided to add a secure website that allows developers to easily deploy their projects. The projects are a container for workers that run 3rd party code. The management site allows several developers to work on different parts of the 3rd party code, deploy independent parts, and to extend the sizes of projects, without having to change the whole project. This is due to the fact that projects work based on hypermedia and the fact that the structure of the projects is separated based on worker resources.

Account Information

User name

Password

Remember me?

Log On

Figure 4-7. Login

Create

Project

Project Name

Description

Create

Figure 4-8. Create Projects

Index

[Create New](#)

	Name	Description
Edit Details Delete	Store	Buy IPTV Applications

Figure 4-9. List of all the projects created by the user

Create

Plugin

Plugin Name

Version

Description

PluginExamples.dll ;

Location

Figure 4-10. Upload a worker resource that plugs into the architecture

Index

[Create New](#)

	Name	Version	Description	UploadLocation
Edit Details Delete	Store	1	Store v1	C:\Worker

Figure 4-11. List of all workers resources uploaded by user

Create

Permission

Plugin
Store ▼

PermissionType
Choose... ▼

Create


Figure 4-12. Give the worker permissions

Create

ProjectPlugin

Project
Store ▼

Plugin
Store ▼

Instances
2 

Start

Create

Figure 4-13. Create instances of the worker projects so they can accept requests

4.4. Workers - Resources

The workers pre-initialize all resources at startup. To increase performance, the dependencies of the workers are optimized by NGEN. NGEN stands for Native Image Generator and it creates native code for a specific system so that the just-in-time compiler does not have to perform work when the workers are started. Also, one of the benefits is that the native images are able to share the dependencies, when multiple instances of the workers that share the dependencies are started. This is extremely important as multiple workers are able to consume less memory because they are able to share the dependencies. The startup time is minimized because the JIT compiler is not needed, as the image is now native. Because the services are able to startup very fast, servers can scale much faster. Each worker uses its own port within a machine. Ports are found with the following code on Figure 4-14 and it checks up to port 10000.

```
for (; curport < 10000; curport++)
{
    if (!busyPorts.Contains(curport))
    {
        return curport.ToString();
    }
}

throw new ApplicationException("There are no free ports!");
```

Figure 4-14. Find a free port for the resource

Each worker has their own sandbox. The workers house all 3rd party code in a dependable fashion.

Safety

The workers are sandboxed and dependable because they use AppDomains. The AppDomains allow for the 3rd party code to have specific permissions, which may not harm other workers, if

the code is not trusted. Figure 4-15 is an example of how to set permissions, we have hardcoded the permissions this example so it is easier to understand.

```
PermissionSet permSet = new PermissionSet(PermissionState.None);
permSet.AddPermission(new SecurityPermission(SecurityPermissionFlag.Execution));

plug = new Plugin(workersdir + dllName, nameSpaceDll);
```

Figure 4-15. Hard coded example of how to permissions are given

There is overhead from using AppDomains but the benefits allow for a multi-tenant architecture. Each worker loads the 3rd party code at runtime. The code adheres to an interface, Figure 4-16, designed to use the HTTP verbs as the main functions. To enforce HTTP guidelines, the GET verb only take parameters that can be passed as query strings. The POST verb is able to take an object, which is the body of the request.

```
public class Plugin : Infrastructure.IWebPlugin
{
    IWebPlugin wp;

    public Plugin(string location, string instance)
    {
        Assembly asm = Assembly.LoadFrom(location);
        wp = (IWebPlugin) asm.CreateInstance(instance);

        if (wp == null)
            throw new ArgumentException("Please check the namespace of your assembly!");
    }

    public WebResult OnGet(string[] parameters)
    {
        return wp.OnGet(parameters);
    }

    public WebResult OnPost(object o)
    {
        return wp.OnPost(o);
    }
}
```

Figure 4-16. Interface that allows to adhere to the RESTful MVC

To simplify the platform as a service for 3rd party developers, the code allows for templates, such as MVC pattern. The templates from the views can be downloaded from digital link libraries. Using the HTTP verb TRACE, the workers automatically bind to the service bus. Figure 4-17 is an example of information the service passes in the headers of the binding request to the service bus with hard coded values to simplify understanding.

```
WebRequest re = WebRequest.Create("http://127.0.0.1:8080");
re.Method = "TRACE";
re.Headers.Add("port", port);
re.Headers.Add("address", "127.0.0.1");
```

Figure 4-17. Hard coded example of a worker binding

To make it easier for 3rd party developers, all of the dependencies can be downloaded from a repository.

Then all of the plugins generated by the third party developers are stored in a central repository. The central repository allows for newly created instances to grab code at runtime for updating dependencies on the fly. Figure 4-18 show the code responsible for grabbing dependencies from the code repository.

```
Directory.CreateDirectory(workersdir);
if (File.Exists(workersdir + filename))
    return;

WebClient we = new WebClient();
we.DownloadFile("http://plugins.codingadventure.com/" + filename, workersdir + filename);
```

Figure 4-18. Dependency injection at runtime using a central HTTP based repository

From the management site, developers can upload their DLLs so that they conform to the specific interface in the HTTP verbs that it can handle.

Mapping is done at runtime when the worker binds to the resource manager. This mapping was designed so that there is no overhead for the resource manager to check a worker database. The workers' addresses are all in memory in a dictionary. All requests then come out from a queue to a round robbing distribution, Figure 4-19.

```
RequestInfo re;

if (cq.TryDequeue(out re))
{
    result = (HttpWebResponse)await WebRequest.Create(new Uri(resourceList[Current])).GetResponseAsync();

    if (Current == (resourceList.Count() - 1))
    { Current = 0; }
    else
        Current++;
}
```

Figure 4-19. Round-Robin in memory lookup of available resources

To show the effectiveness of the architecture, we have developed applications that demonstrate different common scenarios faced by 3rd party developers:

- Personalized user transactions Figure 4-20 and Figure 4-21
- Calls to 3rd party services Twitter (Figure 4-22) and Facebook (Figure 4-23)
- Long pooling real-time notifications Figure 4-24

Personalized user transactions come from an application hub that we have developed to mimic an application store for mobile devices. The application store is the main component of the system, as it has to be able to allow users to purchase applications and redirect them to other applications. The application hub is completely personalized to every user. For this to happen, the application hub allows each user to log in securely using its portal.



Figure 4-20. Personalized Application Hub

The portal is built on asynchronous controls that belong to Mediaroom 2.0 and therefore allow for great user experience. The user experience is enhanced because the main page is loaded first and then it performs the asynchronous requests for the menus. The menus load gradually and allow the users to interact with the menus while they load. We have opted for this compartmentalization, as it allows for great use of cacheability. It enhances the use of cache because the main page is a compromise of requests that are small and easily cacheable, instead of large pages customized for every user. Because the asynchronous requests return specially formatted and regular XML, the services can be developed under many different environments. In our case, we have developed in C# using the new dynamic features of version 4.0. This enables different types of objects to be passed to the services, which can then be formatted to this special XML, as long as some basic properties are on the object.

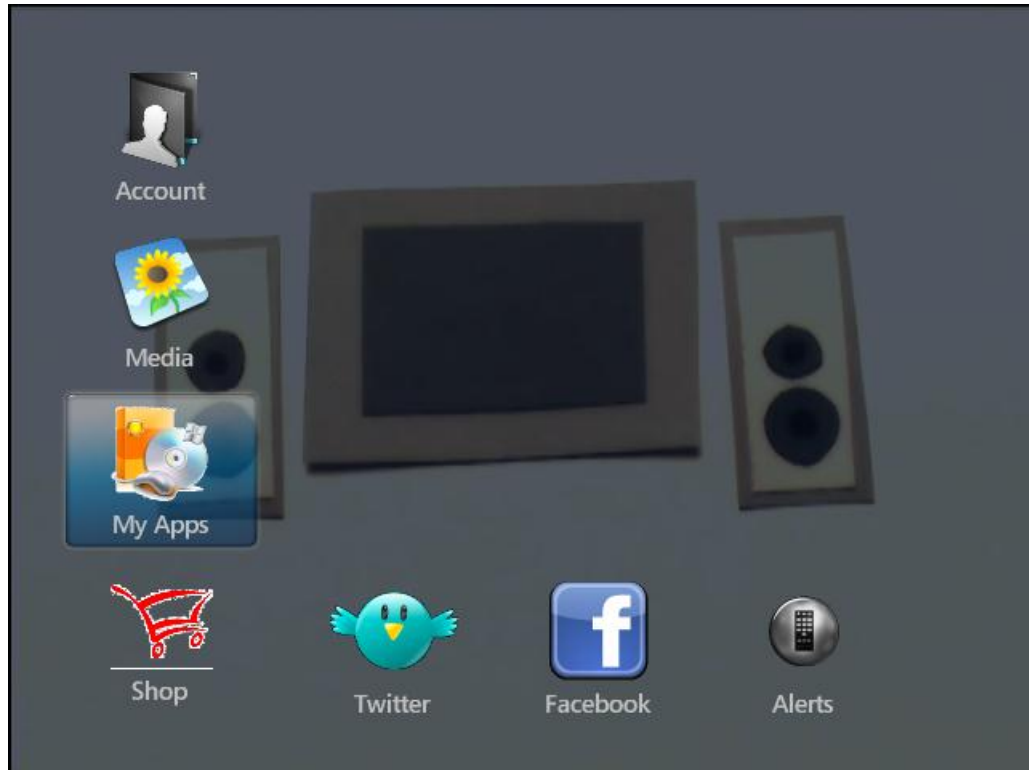


Figure 4-21. Applications the user has purchased

For the Twitter and Facebook applications, the service requests run through a web service that converts all of the data at runtime. Because of the nature of the process, everything is run through the asynchronous controls. This allows for complete control over any errors that might happen when performing the requests, as their competition is out of our scope. If the requests fail within a specific amount of time, the default content is returned to the user to explain that the 3rd party service has failed. This also allows for all of these requests to be cached so that if they failed, they can be retrieved from the cache.



Figure 4-22. Twitter Application



Figure 4-23. Facebook Application

The long pooling real-time notifications example Figure 4-24 involves having requests performed with lower frequency but the server does not respond to the requests immediately, therefore the servers keep an open connection and it is based on a Hypermedia as the Engine of Application State (HATEOAS) [W57] to retrieve new application code (XML) and state. Also, to make sure the messages are returned to the users as fast as possible, they are saved on a server that stores the messages in memory. The mechanisms are making constant requests to databases, such as VoltDB, or using a publish-subscribe mechanism, such as the one from Redis. Both of those approaches would be the ones followed, if there are thousands of users for critical notifications.

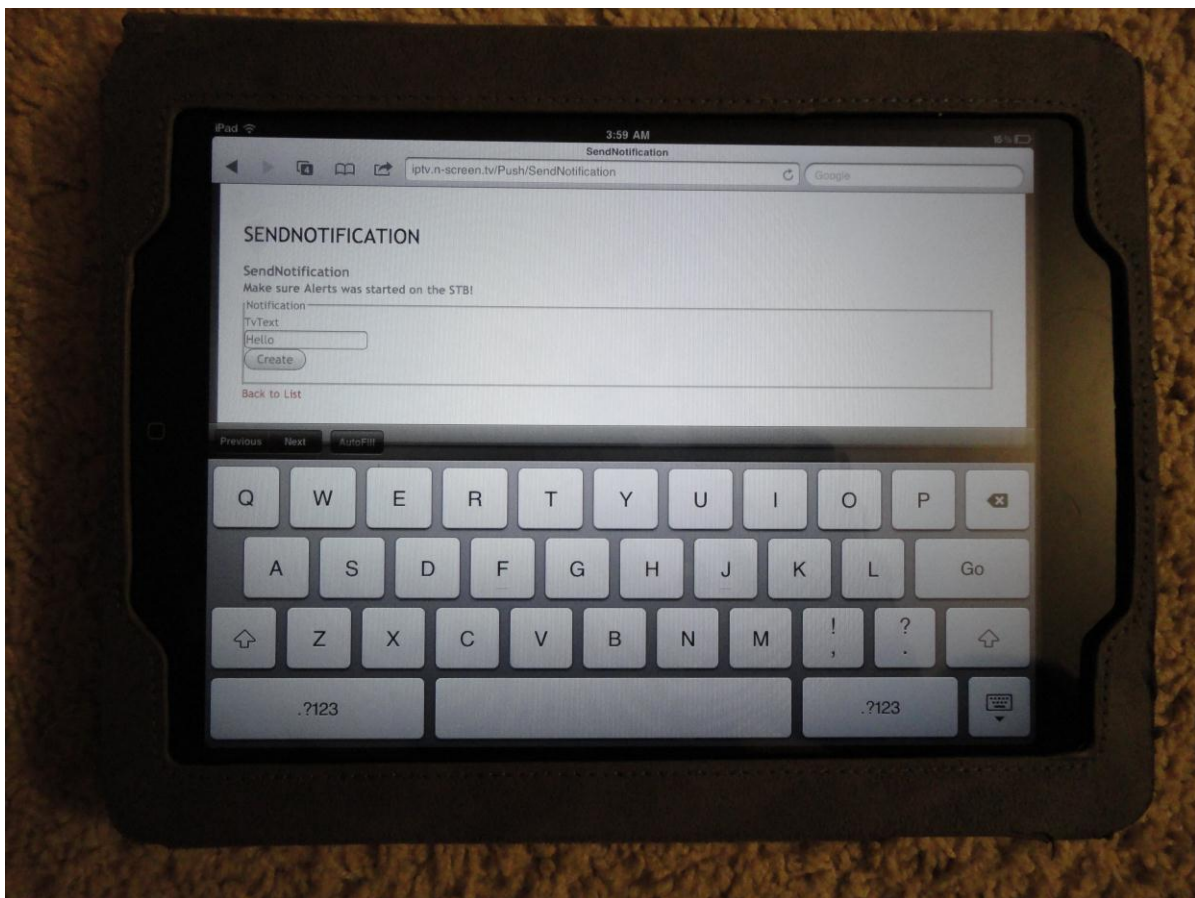


Figure 4-24. Alerts sent from a mobile device

4.5. Simplify IPTV Development

Mediaroom IPTV development has significant barriers of entry for 3rd party developers. Development of the applications is done on a tightly protected specification developed by Microsoft. To enable ease in usability for all users, all 3rd party applications must follow similar design guidelines for the user interface that we must set up.

Many third party developers need to integrate their own web services into the IPTV architecture. This way the 3rd parties developers will not have to develop their code twice and it will enable them to use their existing infrastructure to consistently show the same content using the Mediaroom platform.

Development and deployment must be created in such way so that it is very simple and intuitive for first time developers. This will help with the steep learning curve that comes with the learning process for Mediaroom IPTV development due to the lack of resources available.

4.6. Service Level Agreements

The service level agreements are implemented by using the Dispatcher Control Bus. The Dispatcher Control Bus holds all of the service level agreements. It keeps track of all of the outstanding requests and keeps a log of all of the responses of the requests. Using this information it is capable to ensure the following service level objectives of the 3rd party services:

- Requests/Second
- Internal response time

All of this information is aggregated and can be queried by a RESTful api.

CHAPTER 5 EXPERIMENTS

The following list of experiments ensures that we have covered all of the problems relating to building a dependable IPTV architecture that is capable of running 3rd party code, as stated on chapter 2 and explained in chapter 3.

Table 5-1. Test Summaries

Goal	Properties	Experiment
Ability to execute 3 rd party code in a dependable environment	<p>Being able to run execute binaries on the architecture with the following properties:</p> <ul style="list-style-type: none"> • Confidentiality • Reliability • Availability • Safety • Integrity • Maintainability 	<ul style="list-style-type: none"> • EX1 <ul style="list-style-type: none"> ○ Overhead of the properties • EX2 <ul style="list-style-type: none"> ○ Tests dependability using faulty and unavailable services while measuring performance
Scalability	Being to handle growing amounts of stress	<ul style="list-style-type: none"> • EX3 <ul style="list-style-type: none"> ○ Handling increasing loads until maximum throughput is reached then

		<p>increasing resources</p>
<p>Service Level Agreements</p>	<p>Being able to stay within a specified service level agreement and when those objectives are not met, to notify the 3rd party developers</p>	<ul style="list-style-type: none"> • Ex4 <ul style="list-style-type: none"> ○ Violation of specified service agreements and prompt notification

Table 5-1 shows the designed experiments, each test is designed to test dependability, scalability, and the service level agreements. We have designed each experiment to test the properties that we have introduced in the architecture.

5.1. Setup

The set-up of the experiments will be on the Amazon EC2 cloud. The machines will use the specs of the default instances [W58]:

Small Instance – default*

- 1.7 GB memory
- 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)
- 160 GB instance storage
- 32-bit platform
- I/O Performance: Moderate
- API name: m1.small

Figure 5-1. Cloud Instances

We will have in total four machines. Not all of the machines will be used at the same time but their roles are detailed within each experiment.

5.2. Load Generation

To create a realistic load, we will use our applications and capture the data generated. We will use the applications we developed:

- Application Hub (for personalized transactions)
- Facebook and Twitter (calling external services)
- Notifications application

To capture the data, we will use a packet sniffer. The packet sniffer captures all of the requests coming in and out of the set-top box. We will also make sure to capture the appropriate think time from the packet sniffer. Think time is the amount of time a user takes absorbing the content from one page until they perform an action, such as clicking the remote, to view another page with different content. Because not all users behave the same, the think time will be randomly changed for specific experiments mentioned.

5.3. EX1 - Overhead Test

First, we will test how much overhead is added by the architecture. For the test, we will use two machines. One will use the service dispatcher bus and another will have a service running a simple application. Load testing will be performed on the service itself. The service is stripped of all of the functionality, except for serving HTTP requests. The tests will involve firing sequential GET requests to the service directly, following the pattern outlined below. The load we have outlined below is a minimal load to make sure we are only measuring the overhead. We have chosen a minimal load to make sure our measurements are not affected by bottlenecks in any part of the system.

Table 5-2. EX1 Tests Overview

Requests/Second	Duration (minutes)
10	10
20	10
30	10

Afterwards, we will introduce the dependability aspects to the service and perform the same load. Finally, we will introduce the dispatch service bus and run the same load. From there, we will generate graphs showing the amount of overhead from the framework and architecture.

The first part of the test involves using a normal version of the custom-built http server consuming data from middleware and 3rd party data providers (Facebook). Since we are subjecting the server to a low load of requests per second then we do not expect to have any significant variation with the latency. This is done to be able to chart only the latency of the server without having to worry that any component is going through a starvation of resources. Our hypothesis is that the latency of the service will be constant since the amount of work performed by the server is constant. The only changes that we might see are if there are any network problems between the clients to the server or between the server and 3rd party state resources (Facebook).

The second part of the test involves the whole architecture, which has caching components. The caching components have the advantage of not having to perform calculations or contact background resources and therefore are able to return the requests right away. Due to this

advantage we expect that the architecture might increase the performance of the system by lowering latency. Since we are only going to keep the cache available for 1 second for the requests we expect to have a high variance in the latency since the work performed will vary depending on the arrival rate and time of the requests.

5.4. EX2 - Dependability vs. Performance Test

For this test, we will use three different computers. The first computer will have the service dispatcher coordinator running as a single process. The other two computers will have a total of two services; this means the whole architecture has four services running as in Figure 5-2.

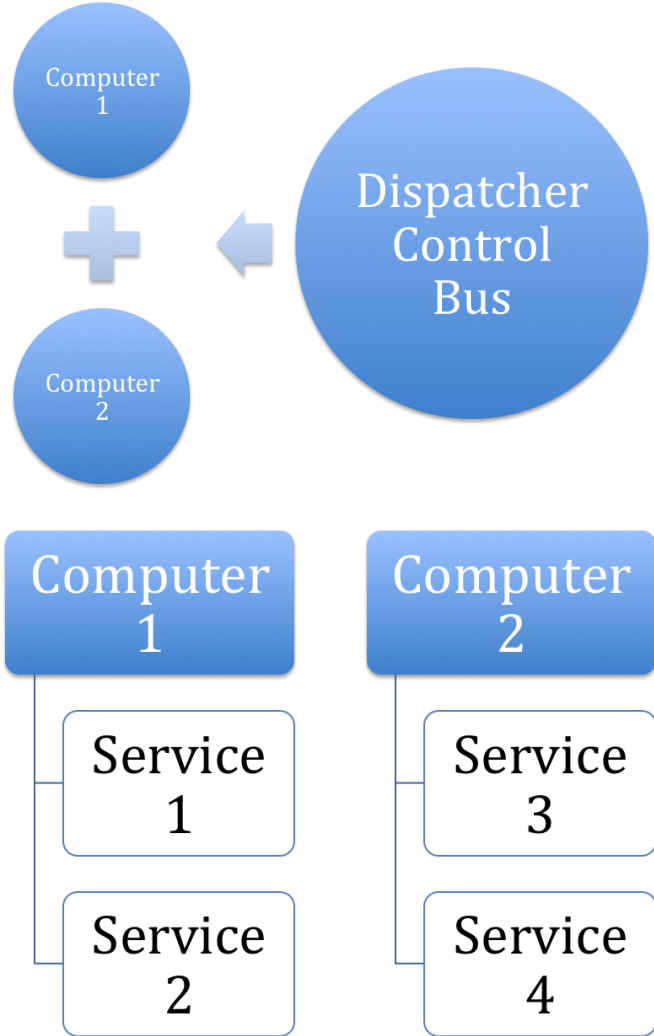


Figure 5-2. Services Layout

All of the services will be returning almost the same state, including an identification number for the service performing the work. At the beginning, all of the services will be running. We will perform the following load. The following load is based on half of the subscribers from SaskTel to simulate the traffic from a city, such as Saskatoon. We are interested in measuring the performance of the system for an application that a user might check once or twice a day, such as a Daily Deal application for Saskatoon. We will assume that, based on the number of SaskTel subscribers quoted in chapter 3, there are 70,463 that are actively using an IPTV applications from 10 pm to 11 pm, which includes Primetime television. Also, each subscriber checks the application four times during this period, which is a one-hour period, which is 3600 seconds.

$$\frac{(Users \times Requests)}{Time} = \frac{(70,463 \times 4)}{3,600}$$

Table 5-3. Dependability VS Performance Test

Requests/Second	Duration (minutes)
78.3	10

After the first run, within machine service one, we will make one of the services perform faulty Figure 5-3 and throw exceptions, and another to be fully operational. We will, then, run the tests again.

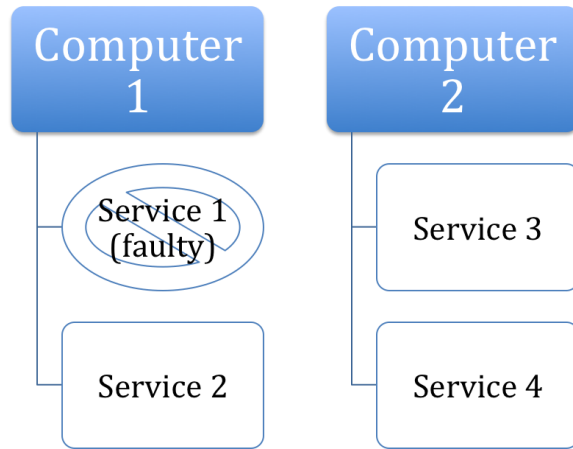


Figure 5-3. Faulty Services Layout

This will test the performance of the architecture when there are faulty services under the same load. For the last part of this test, we will bring back the services on machine one to be fully operational and on machine two, one of the services will be unavailable Figure 5-4, and upgrade the service to a new code base under the same load.

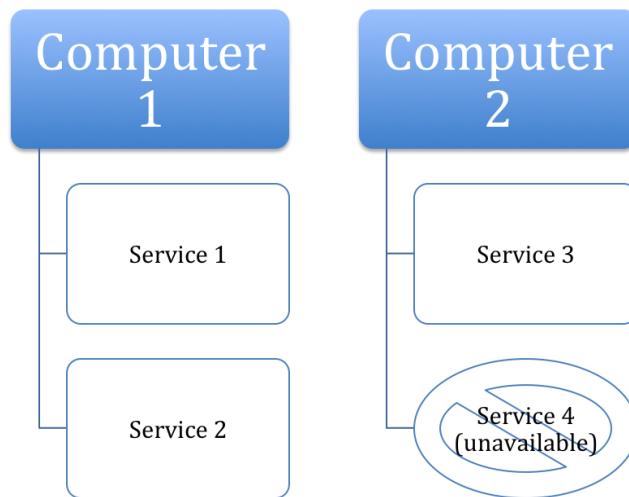


Figure 5-4. Unavailable Services Layout

Using those results, we will create a graph to illustrate how the performance of the system is affected by providing the dependability properties.

5.5. EX3 - Scalability Test

For this test, we will generate the maximum amount of load the application can handle. At the beginning, we will only use the dispatcher bus on one machine and one service on another machine. We will find out the throughput of the architecture using the data captured from the load generation from real generated traffic one standard deviation from the curve. Next, we will increase the number of services until we find the precise saturation amount when throughput plateaus, no matter how many services are added. Finally, we will add a load balancer and introduce two dispatcher busses, each with a single service. From there, we will increase the number of services until the throughput stabilizes.

Finally, we will introduce a third dispatcher bus and saturate all of the busses with resources until the throughput becomes stable. This will give us a good idea of the scalability capabilities of the architecture.

5.6. EX4 - Service Level Agreements Test

To test the service level agreements, we want to test the SLA parameters, measurements, and aggregation. This will be done through the specification of service level objectives and when the objectives are not met, then it will trigger the action guarantees

For this, we will enter the following objectives for a 3rd party service:

- Requests/Second
- Internal response time

When the service level objectives are not being met we should receive email notifications with the current load and the internal response time. This mechanism allows 3rd party administrators to start a new instance of their services in our architecture to meet higher demands.

Due to the large capacity of this system, we will develop services that are underperforming by wasting CPU cycles. This will allow us to easily breach the service level agreements.

For the experiment, we will have the following parameters:

- 100 requests/second
- 10 milliseconds internal response time

We will perform the following load to test the service level agreements:

Table 5-4. Description of SLA based Tests

Minutes	Requests/Second
0	50
1	60
2	70
3	80
4	90
5	100
6	110
7	120

CHAPTER 6 RESULTS

6.1. EX1

6.1.1. Services without the architecture

The experiments are the overhead experiments which measure the amount of operational costs of using the architecture. The data shows that as expected the latency had a constant tendency. The latency average was over 500 milliseconds because the raw data is being consumed and transformed from middleware and 3rd party data services (Facebook). This extra layer of abstraction is great for developing 3rd party applications as it is easier to develop applications using the APIs but it also causes for http servers to have higher latency. As with a normal system, the larger the amount of requests being handled by the architecture, there is an increase in the total latency time. The minimum values gathered show the fastest intervals of time when the results were received. The maximum values show the slowest times intervals when the results were received. The minimum and maximum spikes are very close to each other indicating a homogenous trend. The major spikes of the max values in latency are due to problems with latency from the 3rd party services, in this case, Facebook. For every request done to the architecture that requires 3rd party data, the architecture will contact directly those services. This means that for every request to the architecture for the Facebook application requires another request that goes to an API that connects to Facebook. Also 3rd party providers of data start to throttle requests to make sure they are not flooded with requests that could cause a denial of service attack. When those requests from Facebook are not received in time, the services show high latency as shown in the graph. The spikes can also be due to problems with I/O usage in the EC2 architecture. The following are the cumulative values we have gathered with the architecture:

Table 6-4. Overhead 10 RPS Summary

Summary	Milliseconds
Minimum	394
Maximum	31330
Average	630.3

Values of 394 milliseconds are great but values of 31 seconds would timeout, create horrible user experience and basically make the system unusable.

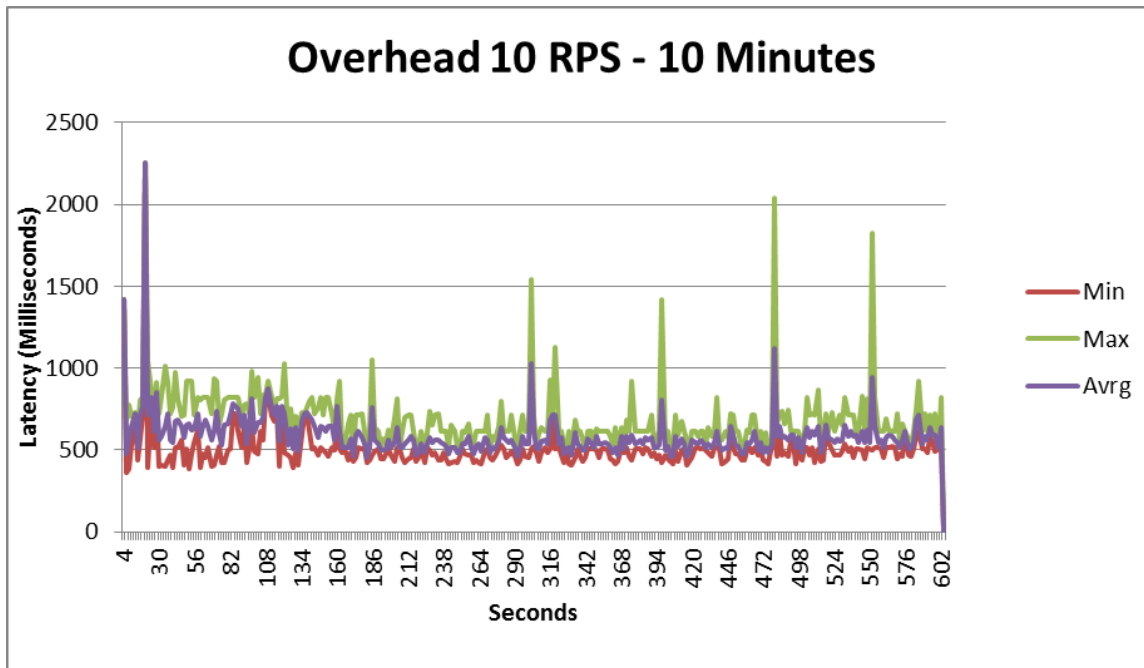


Figure 6-1. Overhead 10 RPS – 10 Minutes

In Figure 6-1 the spikes in this graph show the latency of the system under a load of 10 requests per second for a period of 10 minutes. Several spikes that can be due to throttling from Facebook, latency from shared I/O from EC2 or latency issues with Facebook.

Table 6-5. Overhead 20 RPS Summary

Summary	Milliseconds
Minimum	361
Maximum	2252
Average	576.7

Values of 361 milliseconds are great and values of up to 2 seconds are acceptable but would decrease user experience.

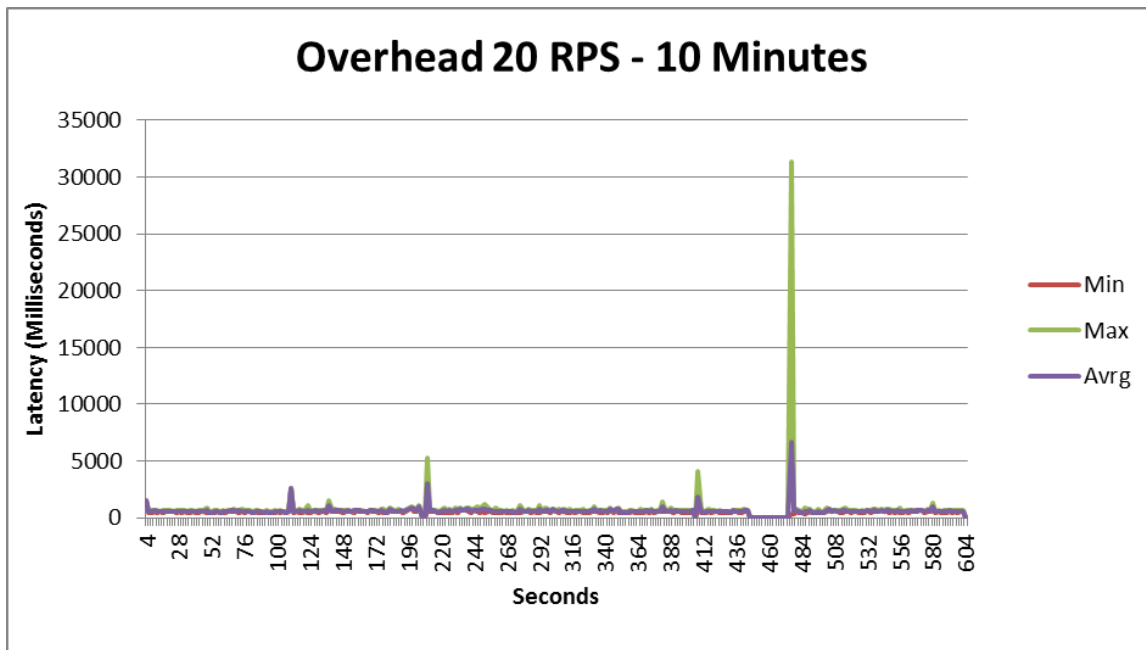


Figure 6-2. Overhead 20 RPS – 10 Minutes

In Figure 6-2 the spikes in this graph show the latency of the system under a load of 20 requests per second for a period of 10 minutes. The average value for spikes is under 576.7 milliseconds which is good. There is a huge spike in latency due to throttling from Facebook.

Table 6-6. Overhead 30 RPS Summary

Summary	Milliseconds
Minimum	422
Maximum	4410
Average	595.6

Values of 422 milliseconds are great and values of 4.4 seconds decrease user experience significantly but it is still usable.

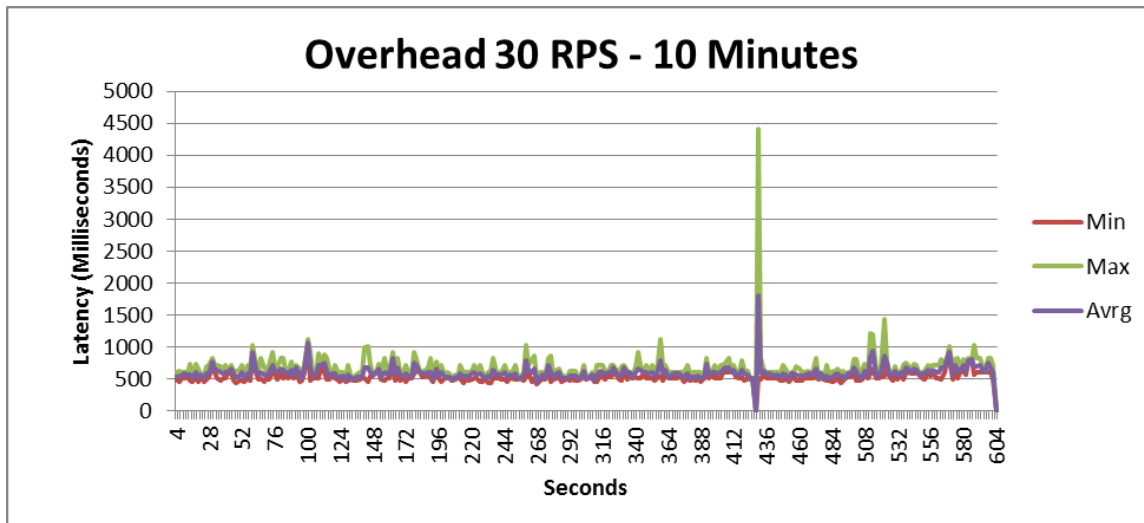


Figure 6-3. Overhead 30 RPS – 10 Minutes

In Figure 6-3 the spikes in this graph show the latency of the system under a load of 30 requests per second for a period of 10 minutes. The average value for spikes is under 576.7 milliseconds. There is a huge spike in latency due to throttling from Facebook, latency from shared I/O from EC2 or latency issues with Facebook. At this point we see that consistently there is a spike around 400-430 seconds into the test. It is due to throttling of Facebook.

6.1.2. Services with the architecture

The architecture performs much better as the number of requests increase. The cache is available for one second and it only takes one request with the proper result to be able to respond to any of the other requests the front side of the architecture. The amount of requests being handled becomes a function of the amount of requests the front side of the architecture can handle.

Therefore one of the important things we have learned is that the front instances with the dispatcher bus should be most powerful to be able to handle loads as fast as possible. For this experiment we are only using one backend resource service but if we introduce multiple ones the load will be balanced across multiple services which would decrease the load of each individual service. Then the bottleneck of the architecture is the front end dispatcher receiving the requests. At that point when it reaches saturation a larger front instance should be introduced or multiple dispatchers can be added which would be accessible through the same DNS address. This method of having powerful front side instances is vertical scalability and having several less expensive instances to communicate with backend services to perform state fetches or changes is horizontal scalability.

Table 6-7. Overhead Architecture 10 RPS Summary

Summary	Milliseconds
Minimum	0
Maximum	1261
Average	49.8

Values of 0 milliseconds are amazing, it means that the load generator would need smaller measuring magnitudes to capture the latency and values of 1.2 seconds are not bad.

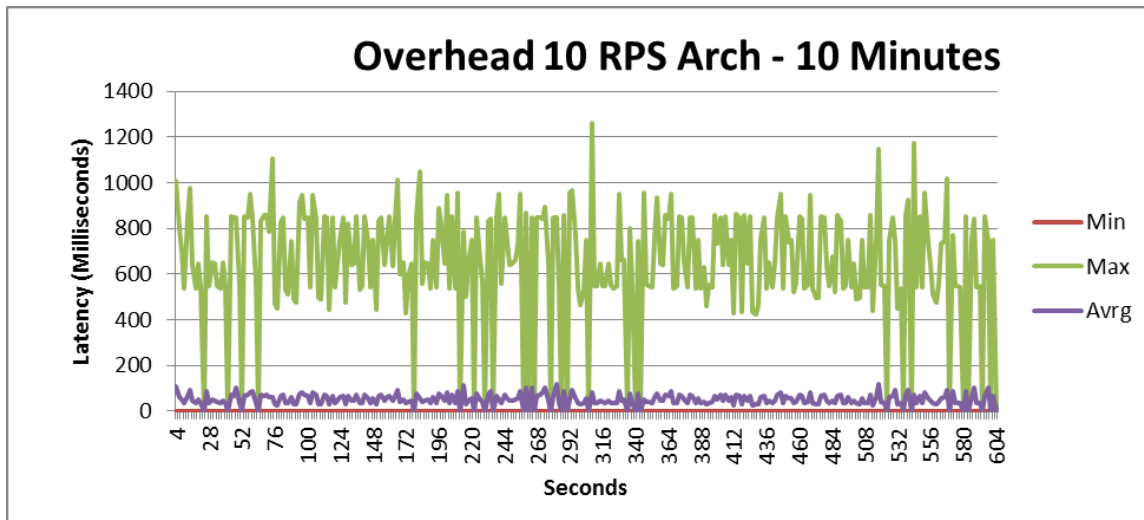


Figure 6-4. Overhead Architecture 10 RPS – 10 Minutes

In Figure 6-4 the spikes in this graph show the latency of the system under a load of 10 requests per second for a period of 10 minutes. The average value for spikes is under 49.8 milliseconds. Latency is significantly diminished due to caching. The max latency values are still present but a huge spike in latency due to throttling from Facebook.

Table 6-8. Overhead Architecture 20 RPS Summary

Summary	Milliseconds
Minimum	0
Maximum	3114
Average	35.9

Values of 0 milliseconds are amazing, it means that the load generator would need smaller measuring magnitudes to capture the latency and values of 3 seconds make the system usable but with significantly deminished user experience.

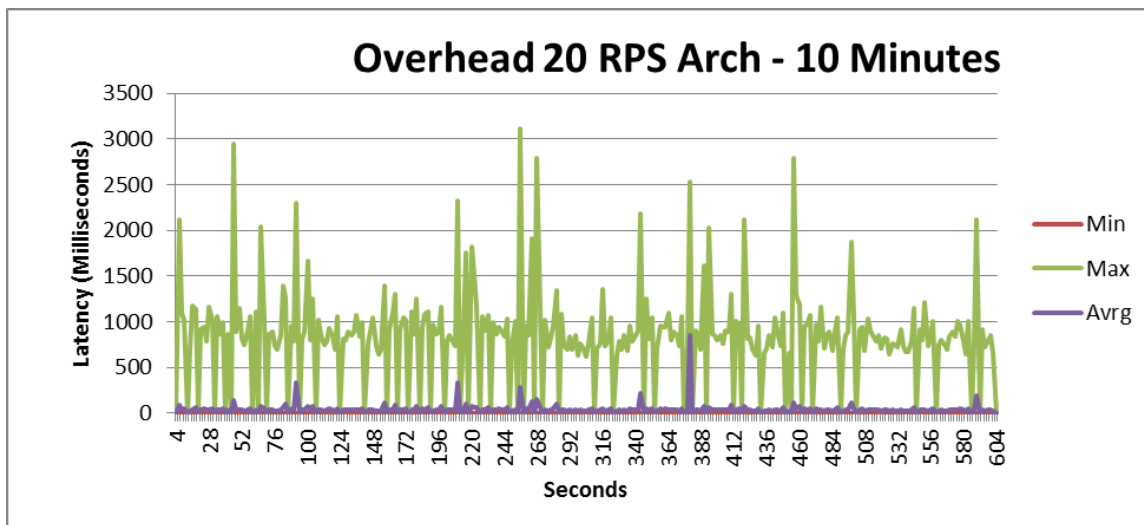


Figure 6-5. Overhead Architecture 20 RPS – 10 Minutes

In Figure 6-5 the spikes in this graph show the latency of the system under a load of 10 requests per second for a period of 10 minutes. The average value for spikes is under 49.8 milliseconds. Latency keeps going down due to the use of resource caching which prevents problems due to throttling or 3rd party latency.

Table 6-9. Overhead Architecture 30 RPS Summary

Summary	Milliseconds
Minimum	0
Maximum	1424
Average	13

Values of 0 milliseconds are amazing, it means that the load generator would need smaller measuring magnitudes to capture the latency and values of 1.4 seconds are not bad.

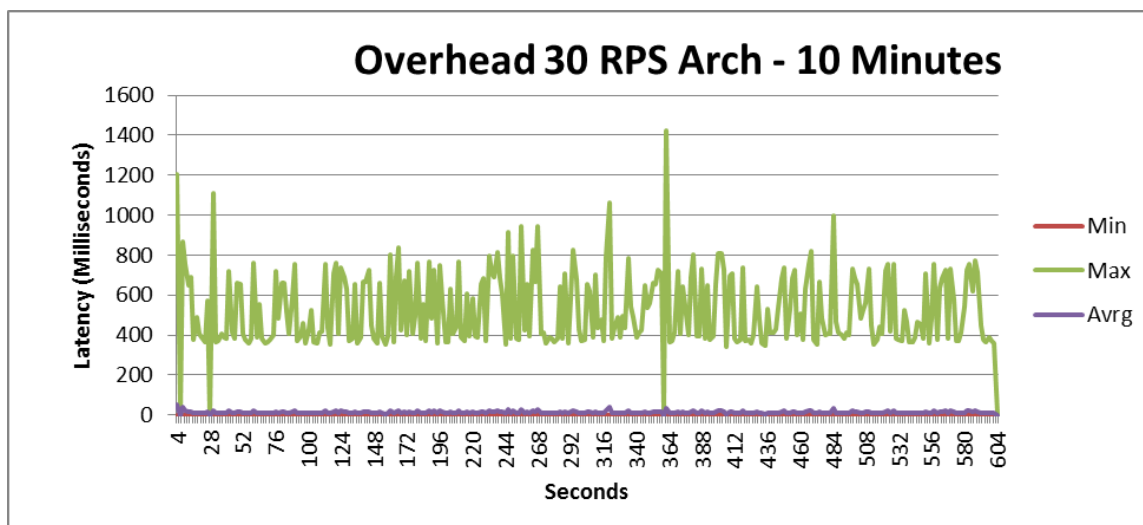


Figure 6-6. Overhead Architecture 30 RPS – 10 Minutes

In Figure 6-6 the spikes in this graph show the latency of the system under a load of 30 requests per second for a period of 10 minutes. The average value for spikes is under 13 milliseconds. Latency keeps going down. Comparably from having the architecture and not having the architecture there is an improvement of 576.7 milliseconds from to 13 milliseconds at 30 requests per second.

6.2. EX2

The test for dependability tests if the architecture is capable of handling expected load which being able to respond accordingly in a dependable way. In this case we are checking that no errors get returned on the tests and that the latency times are also minimal with the architecture. To make sure that the latency times are minimal we are using four backend resource servers. At four minutes into our experiment we are going to simulate maintenance of a service. The dependability of the service is affected primarily by the amount of requests the service that is temporarily offline. Also this offline service is causing extra overhead because the dispatch control bus takes some time to realize that the service is unavailable instead of continuously waiting to achieve the connection. In Figure 6-7 the spikes in this graph show the latency of the system under a load of 78.3 requests per second for a period of 10 minutes. The average value for spikes is under 9.5 milliseconds. Latency was minimal in the system.

Table 6-10. Dependability VS Performance Summary

Summary	Milliseconds
Minimum	0
Maximum	31516
Average	9.5

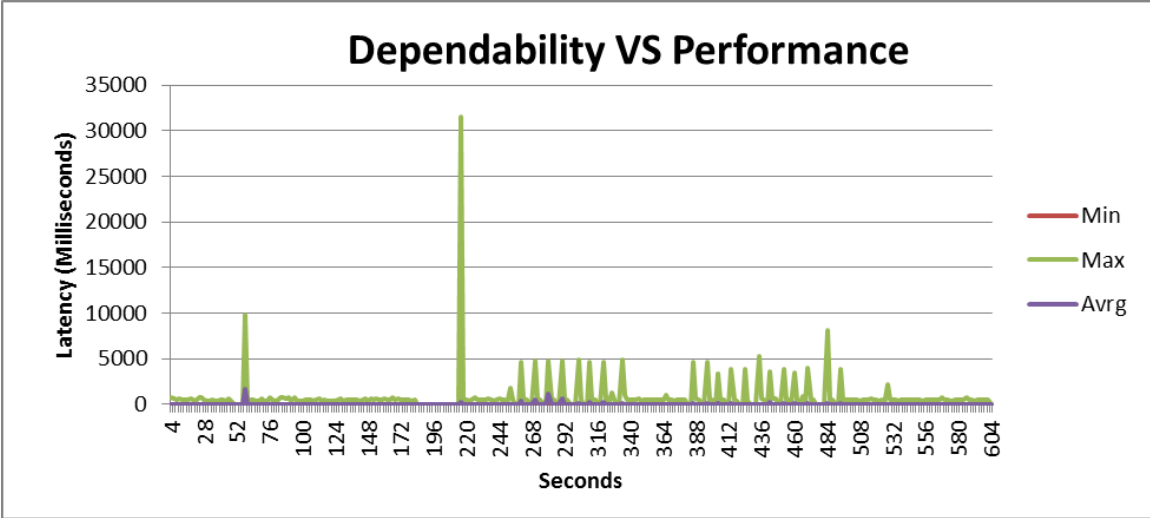


Figure 6-7. Dependability VS Performance – 10 Minutes

The spikes in this system show how performance fluctuates based on the damages in the system based on unavailability or errors. It also shows that even if the system undergoes unavailability or errors the system can recover by using replication and it is able to perform with low latency due to caching as can be seen from the average.

EX3

In this scalability experiment we incrementally increased the load using a step function. When there was a big latency spike in the system we introduced another instance of a service worker resource. We ended up with four service instances entered sequentially at specific times. The constant latency proves that as more instances are introduced into the system we are able to stabilise the system.

Table 6-11. Scalability Instances Summary

Time	Number of Services
0	1
84	2
284	3
436	4

Table 6-12. Scalability Summary

Summary	Milliseconds
Minimum	0
Maximum	8627
Average	1.9

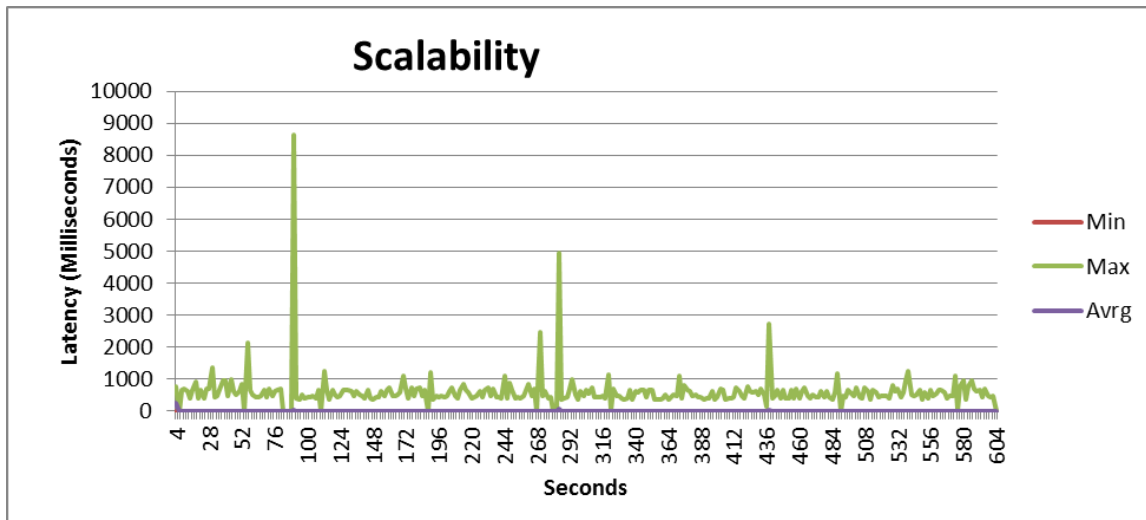


Figure 6-8. Scalability – 10 Minutes

In this case we have introduced four services since it was the number of services that allowed us to have a stable system, Figure 6-8, at the level of latency we experienced. It is also important to note that the latency average was only of 1.9 milliseconds making it a very high performance system that scales horizontally.

EX4

To test services level agreements we made sure to have restful reporting of the data. The data then is used for anotation service. The real time statistics can be accessed and succesfully show the breach of the agreements during the times marked on the following tablet in red.

Minutes	Requests/Second
0	50
1	60
2	70
3	80
4	90
5	100
6	110
7	120

The results show that the data from the system can be aggregated and used to provide comprehensive information to 3rd party developers and maintaners of the architecture. Using this information it is possible to ensure that 3rd party agreements are not breached and if they are breached then due compensations can be done. The advantage of having this information over a RESTful resource is that many other applications can consume this data to provide interesting reports whenever, for example, more than 100 request per second are not being responded by the 3rd party services in the architecture.

CHAPTER 7 SUMMARY AND CONTRIBUTION

As more services are being ported to the cloud it is important that we are able to have the same or higher dependability for those services. The current trend for developing high end services is to have 3rd party developers create applications. These applications become part of an ecosystem. To have dependable and scalable systems that are governed by service level agreements allows for the new type of ecosystems. It is possible to have all the benefits of 3rd party applications without the need to settle for less confidence in the overall user experience in the system. The main contribution of the research is that we have developed a fully working IPTV architecture that provides dependability, scalability and SLA agreements for TRILabs (www.trilabs.ca). The dispatcher control bus handles dependability properties through the REST protocol. Availability is achieved through redundancy of services. Reliability is achieved by using the HTTP protocol status codes. Integrity is achieved by making sure that state changes can only occur through the web service itself and isolating errors from propagating inside the system to other components through using sandboxing. Safety is achieved by isolating errors from propagating to the user and sandboxing of the resource services using appdomains. The architecture can scale horizontally by adding more services which balances the use of resources, higher performance is achieved by the use of caching. Finally the statistics of how these services are performing is captured by the dispatcher service bus and exposed through as a RESTful resource.

The system allows for multiple web services inside the same machine to conserve resources while shielding them from each other. Also it protects the user experience by preventing errors from propagating to the end users. With this architecture it is possible of having an architecture that outperforms regular consumption of 3rd party services. The performance boost is due to the

heavy use of caching and balancing the high request load to several web services. Using high redundancy it is possible to handle failure of multiple services as long as there are enough working replicas. High redundancy along with round-robin allows having high scalability since new 3rd party services are able to start up instantly and bind to the architecture at runtime. Finally the service level agreements allow us to check that the architecture is able to keep its promises to the 3rd party services.

CHAPTER 8 FUTURE WORK

8.1. Future Work

In the process of finding a working prototype for this research we have found many different aspects that could augment this thesis or other viable solutions.

8.1.1. Web Sockets

Sockets are the new standard for full duplex communication available in HTML5 enabled browsers. Sockets allow for real-time even driven communication between the client and server. The current standard for web transports basically are retrieval based which means the client has full control over when the information is sent to the client. The client has to initiate the integration and therefore the server is not capable of sending information to the client whenever it relevant to the client. This is especially important for displaying information such as notifications, alerts, medical and time sensitive information.

In the architecture for this research we used streaming techniques to circumvent the limitations of not having sockets available in the current IPTV frameworks. Some of the techniques are pooling, long pooling and streaming. Pooling involves the client continuously requesting data to the server with very short intervals to check if information has changed, this method has a high level of overhead as each client produces high levels of requests. Long pooling involves having requests performed with lower frequency but the server does not respond to the requests immediately, therefore the servers keep an open connection for a prolonged period of time until a little bit before the client connection times out. Finally, streaming is having an open connection due to a non-complete response; therefore multiple results can come from the same open connection. The main problem with streaming is that it

does not work well with proxy's that sometimes buffer requests and firewalls. All of these methods rely on the HTTP protocol and send/receive headers, which in some cases include latency. For our research we rely on the HTTP protocol to be able to provide caching and dependability due to the semantics of the headers. For future work, it would be interesting to implement this architecture using a mix of the REST HTTP protocol and the use of web sockets for real time event driven communication. The challenge would be when it would be proper to have RPC style communication of web sockets and when it would be proper to use REST with caching. The answer to this problem could be CQRS.

8.1.2. CQRS

Command Query Responsibility Segregation is a pattern designed for scalability and separation of concerns on large systems. CQRS is primarily used in distributed architectures for asynchronous propagation of updates to separate components. Like in our architecture it trades consistency over availability and partition tolerance as described previously with the CAP theorem. CQRS separates the reads and writes in an architecture and it uses events to propagate changes to the architecture. In this case it possible to have very complex logic involving the creation and update process of an object including the use an ACID database to hand all of the writes in the architecture and use a NoSQL database to handle all of the reads. Therefore it would be a good addition to the architecture since we can guarantee that all operations in the system are eventually consistent due to the ACID properties. All of the reads come from a very different de-normalized model of the data which require a preferably a single object from the database to display all the data needed for the request. Also by segregating the read and write operations it is possible to use this semantic pattern of CQRS in the use of sockets. Since CQRS allows for complex create and update logic of objects specific procedures can be handled so some objects are specifically flagged for the use of sockets. Finally, using a publish subscribe

system the 3rd party systems could automatically update our data with all of the information so that no fetching needs to be done from the 3rd party. Therefore all of the information would come directly from our NoSQL store and then the information would be sent from web sockets.

8.1.3. Pre-Processed Static Resources Hosted on the Cloud

Another technique available could be to preprocess all of the state required in the form of hypermedia files. This way the 3rd party developers only need to upload template files that then are processed for every single user that will be using the architecture. Whenever a new user signs up, an asynchronous process will generate the static resources in cloud storage. Then all of the files will be stored in CDN where the files can be served world wide in a very efficient and fast manner. Recently most CDN networks now support encrypted transport protocols (https), which would provide some security. The main problem with this technique might be being able to enforce security in complex scenarios. The main basic approach would be generate a few methods that would help 3rd party developers to create random resources addresses, and they should be random enough so that others will not be able to compromise others privacy.

8.1.4. Publish Subscribe Client Cache Push

For clients with higher capacities, it would be possible to have embedded RESTful databases inside the client. In this case the static resources could be pre-processed on the server and then they would be pushed directly to the client. Then these clients could directly talk with other clients lowering latency in LAN environments just like a P2P environment for dependability. This way 3rd party developers would still be able to create applications for these devices but we would have to be careful. Since the code would be hosted from inside the device there are problems with sandboxing the content and making sure privacy is still maintained to provide dependability.

LIST OF REFERENCES

- [1] S. Robertson, C. Wharton, C. Ashworth, M. Franzke, Dual device user interface design: PDAs and interactive television, Proceedings of the SIGCHI conference on Human factors in computing systems: common ground, p.79-86, April 13-18, 1996, Vancouver, British Columbia, Canada.
- [2] P. Cesar, D. Bulterman, and A. J. Jansen. Usages of the Secondary Screen in an Interactive Television Environment: Control, Enrich, Share, and Transfer Television Content. In Proceedings of the 6th European conference on Changing Television Environments (EUROITV '08), Manfred Tscheligi, Marianna Obrist, and Artur Lugmayr (Eds.). Springer-Verlag, Berlin, Heidelberg, 168-177.
- [3] A. Al-Hezmi, Y. Rebahi, T. Magedanz, S. Arbanowski, "Towards an Interactive IPTV for Mobile Subscribers," icdt, pp.45, International Conference on Digital Telecommunications (ICDT'06), 2006
- [4] P. Vorderer, 2000. Interactive entertainment and beyond. In Media Entertainment: The Psychology of Its Appeal, D. Zillmann and P. Vorderer, eds., Lawrence Erlbaum Associates, 21-36.
- [5] J. Masthoff, Group Modeling: Selecting a Sequence of Television Items to Suit a Group of Viewers. In Proceedings of User Model. User-Adapt. Interact.. 2004, 37-85.
- [6] L. Jihye, P. Shinjee, L. Jeongyeon, K. Munchurl, L. Sunhwan, K. Sangki, "Design of Open APIs for Personalized IPTV Service," Advanced Communication Technology, The 9th International Conference on , vol.1, no., pp.305-310, 12-14 Feb. 2007 doi: 10.1109/ICACT.2007.358361
- [7] A. Keller, H. Ludwig, The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services, Journal of Network and Systems Management, 2003-03-01, Springer New York, <http://dx.doi.org/10.1023/A>:
- [8] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana, "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI," Internet Computing, IEEE , vol.6, no.2, pp.86-93, Mar/Apr 2002 doi: 10.1109/4236.991449 <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=991449&isnumber=21386>
- [9] S. Vinoski, "REST Eye for the SOA Guy," 2007. IEEE Internet Computing, pp. 82-84, January/February
- [10] E. Al-Masri and Q. H. Mahmoud. 2007. Discovering the best web service. In Proceedings of the 16th international conference on World Wide Web (WWW '07). ACM, New York, NY, USA, 1257-1258. DOI=10.1145/1242572.1242795 <http://doi.acm.org/10.1145/1242572.1242795>

- [11] M. Sabou, J. Pan, Towards semantically enhanced Web service repositories, *Web Semantics: Science, Services and Agents on the World Wide Web*.
- [12] C. Pautasso, O. Zimmermann, F. Leymann. “*Restful web services vs. "big" web services: making the right architectural decision*”. 2008. In *Proceeding of the 17th international conference on World Wide Web (WWW '08)*. ACM, New York, NY, USA, 805-814. DOI=10.1145/1367497.1367606 <http://doi.acm.org/10.1145/1367497.1367606>
- [13] Roy Fielding. “*Representational State Transfer (REST)*” http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- [14] M. Fowler. “*Richardson Maturity Model: steps toward the glory of REST*” <http://martinfowler.com/articles/richardsonMaturityModel.html>
- [15] N. Looker , M. Munro. X. Jie, "WS-FIT: a tool for dependability analysis of Web services", *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International , vol.2, no., pp. 120- 123 vol.2, 28-30 Sept. 2004* doi: 10.1109/COMPSAC.2004.1342690
- [16] N. Looker, X. Jie, "Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection", *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. The Ninth IEEE International Workshop on , vol., no., pp. 163, 01-03 Oct. 2003*
- [17] S. Abraham, M. Thomas, J. Thomas, "Enhancing Web services availability" *e-Business Engineering, 2005. ICEBE 2005. IEEE International Conference on , vol., no., pp.352-355, 12-18 Oct. 2005* doi: 10.1109/ICEBE.2005.62
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1552915&isnumber=33062>
- [18] L.E Moser, P.M. Melliar-Smith, W. Zhao , "Making Web services dependable," *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on , vol., no., pp. 9 pp., 20-22 April 2006* doi: 10.1109/ARES.2006.79
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1625341&isnumber=34117>
- [19] E. Nourani, "A new architecture for Dependable Web Services using N-version programming" *Computer Research and Development (ICCRD), 2011 3rd International Conference on , vol.2, no., pp.333-336, 11-13 March 2011* doi: 10.1109/ICCRD.2011.5764144
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5764144&isnumber=5764069>
- [20] Y. Xinfeng, "Providing Reliable Web Services through Active Replication" *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on , vol., no., pp.1111-1116, 11-13 July 2007*

doi: 10.1109/ICIS.2007.151

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4276532&isnumber=4276339>

[21] X. Ye, Y. Shen, "A middleware for replicated Web services" Web Services, 2005. ICWS 2005. Proceedings. 2005. IEEE International Conference on , vol., no., pp. 2 vol. (xxxiii+856), 11-15 July 2005 doi: 10.1109/ICWS.2005.8

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1530855&isnumber=32665>

[22] A. Avizienis , J. C. Laprie, B. Randell, B. Randell. "Fundamental Concepts of Dependability". 2001. Technical Report Series university Of Newcastle Upon Tyne Computing Science, 1145(010028), 7-12. Citeseer. Retrieved from

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.120&rep=rep1&type=pdf>

[23] J.H. Hammer, G. Schneider, "On the Definition and Policies of Confidentiality" Information Assurance and Security, 2007. IAS 2007. Third International Symposium on , vol., no., pp.337-342, 29-31 Aug. 2007 doi: 10.1109/IAS.2007.20

<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4299796&isnumber=4299732>

[24] Poore, J.H.; Mills, H.D.; Mutchler, D.; , "Planning and certifying software system reliability," Software, IEEE , vol.10, no.1, pp.88-99, Jan 1993 doi: 10.1109/52.207234

<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=207234&isnumber=5302>

10.1109/WORDS.2003.1267504 URL:

<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1410959&isnumber=30574>

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1342690&isnumber=29573>

[25] JD Musa, A Iannino, Software reliability: measurement, prediction, application, McGrawHill, New York, 1987

2007, Pages 142-150, ISSN 1570-8268, 10.1016/j.websem.2006.11.004.

<http://www.sciencedirect.com/science/article/pii/S1570826807000066>)

[26] Zamojski, W.; Caban, D.; , "Introduction to the Dependability Modeling of Computer Systems," *Dependability of Computer Systems, 2006. DepCos-RELCOMEX '06. International Conference on* , vol., no., pp.100-109, 25-27 May 2006

doi: 10.1109/DEPCOS-RELCOMEX.2006.35

[27] Looker, N.; Jie Xu; , "Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection," *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. The Ninth IEEE International Workshop on* , vol., no., pp. 163, 01-03 Oct. 2003 doi:

10.1109/WORDS.2003.1267504

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1410959&isnumber=30574>

[28] H. Mei-Chen, T.K. Tsai, R.K. Iyer, "Fault injection techniques and tools" Computer , vol.30, no.4, pp.75-82, Apr 1997

doi: 10.1109/2.585157

URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=585157&isnumber=12687>

- [29] N. Sasikaladevi, L. Arockiam, "Extended WS-FIT model to enhance the fault tolerance in the dynamic composite web service" *Electronics Computer Technology (ICECT)*, 2011 3rd International Conference on , vol.5, no., pp.21-25, 8-10 April 2011
doi: 10.1109/ICECTECH.2011.5941949
- [30] G. Seth, N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". 2002. *SIGACT News* 33, 2 (June 2002), 51-59.
DOI=10.1145/564585.564601 <http://doi.acm.org/10.1145/564585.564601>
- [31] J. H. Christensen. "Using RESTful web-services and cloud computing to create next generation mobile applications". 2009. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA '09)*. ACM, New York, NY, USA, 627-634. DOI=10.1145/1639950.1639958
<http://doi.acm.org/10.1145/1639950.1639958>
- [32] K. Birman, R. V. Renesse, W. Vogels, "Adding High Availability and Autonomic Behavior to Web Services" 2004. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 17-26.
- [33] J. Salas, F. Perez-Sorrosal, M. Patio-Martinez, R. Jimenez-Peris. 2006. "WS-replication: a framework for highly available web services". In *Proceedings of the 15th international conference on World Wide Web (WWW '06)*. ACM, New York, NY, USA, 357-366.
DOI=10.1145/1135777.1135831 <http://doi.acm.org/10.1145/1135777.1135831>
- [34] L. Zhen, T. Jun-Feng, W. Feng-Xian, "Sandbox System Based on Role and Virtualization" *Information Engineering and Electronic Commerce*, 2009. IEEC '09. International Symposium on , vol., no., pp.342-346, 16-17 May 2009
doi: 10.1109/IEEC.2009.77
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5175134&isnumber=5175055>
- [35] Zhen Li; Hongyun Cai; Junfeng Tian; Wu Chen; , "Application Sandbox Model Based on System Call Context," *Communications and Mobile Computing (CMC), 2010 International Conference on* , vol.1, no., pp.102-106, 12-14 April 2010
doi: 10.1109/CMC.2010.77
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5471505&isnumber=5471384>
- [36] A. Sahai, V. Machiraju, M. Sayal A. V. Moorsel, F. Casati, "Automated SLA Monitoring for Web Services", *Management Technologies for E-Commerce and E-Business Applications*, Book Series, 2002, Springer Berlin / Heidelberg, Isbn: 978-3-540-00080-8 Url:
http://dx.doi.org/10.1007/3-540-36110-3_6 Doi: 10.1007/3-540-36110-3_6
- [37] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, J.S. Freie, "A concept for QoS integration in Web services", *Web Information Systems Engineering Workshops*, 2003. *Proceedings. Fourth International Conference on* , vol., no., pp. 149- 155, 13 Dec. 2003
doi: 10.1109/WISEW.2003.1286797
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1286797&isnumber=28684>

[38] L. Liang, S. Meina; Z. Xiaoqi, "*An SLA based Web Service quality monitor system*" Pervasive Computing (JCPC), 2009 Joint Conferences on , vol., no., pp.661-664, 3-5 Dec. 2009 doi: 10.1109/JCPC.2009.5420099
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5420099&isnumber=5420061>

[39] N. Bonvin, T.G. Papaioannou, K. Aberer, "*Autonomic SLA-Driven Provisioning for Cloud Applications*" Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on , vol., no., pp.434-443, 23-26 May 2011 doi: 10.1109/CCGrid.2011.24
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5948634&isnumber=5948590>

LIST OF WEBSITES

- [W1] Crunchyroll. <http://crunchyroll.com> , Last Accessed: January 20, 2012
- [W2] Netflix. <http://netflix.ca>, Last Accessed: January 20, 2012
- [W3] Youtube. <http://youtube.ca>, Last Accessed: January 20, 2012
- [W4] iTunes. <http://www.apple.com/ca/itunes/>, Last Accessed: January 20, 2012
- [W5] Amazon Video on Demand <http://www.amazon.com/gp/video/ontv/start> , Last Accessed: January 20, 2012
- [W6] Roku Stream Player. <http://www.roku.com/>, Last Accessed: January 20, 2012
- [W7] Google TV. <http://www.google.com/tv/>, Last Accessed: January 20, 2012
- [W8] Microsoft Mediaroom <http://www.microsoft.com/mediaroom/>, Last Accessed: January 20, 2012
- [W9] Year-End 2010 Broadband And Iptv Report Highlights: Global Broadband Penetration Is The Critical Driver Behind Broadband Forum's Ipv6 Solutions http://point-topic.com/content/press/YE2010_BB_IPTV_IPv6%20release%20FINAL%2023%20Mar%202011.doc
- [W10] Xbox 360 Live. <http://www.xbox.com/en-US/live?xr=shellnav> , Last Accessed: January 20, 2012
- [W11] PlayStation Video Store. <http://www.playstation.ca/ps/videostore.aspx>, Last Accessed: January 20, 2012
- [W12] Netflix at Nintendo Wii. <http://www.nintendo.com/wii/netflix>, Last Accessed: January 20, 2012
- [W13] SOA User Survey: Adoption Trends and Characteristics. 2008. Gartner, Last Accessed: January 20, 2012
- [W14] World Wide Web Consortium (W3C) <http://www.w3.org/>, Last Accessed: January 20, 2012
- [W15] OASIS: Advancing open standards for the global information society <http://www.oasis-open.org/home/index.php>, Last Accessed: January 20, 2012
- [W16] Web Services Interoperability <http://www.ws-i.org/>, Last Accessed: January 20, 2012
- [W17] SOAP Specifications. <http://www.w3.org/TR/soap/>, Last Accessed: January 20, 2012

[W18] Web Services Description Language. <http://www.w3.org/TR/wsdl>, Last Accessed: January 20, 2012

[W19] OSSIS Web Services Security (WSS) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss, Last Accessed: January 20, 2012

[W20] Understanding Service-Oriented Architecture. <http://msdn.microsoft.com/en-us/library/aa480021.aspx>, Last Accessed: January 20, 2012

[W21] A Guide to Developing and Running Connected Systems with Indigo. <http://msdn.microsoft.com/en-us/magazine/cc164026.aspx>, Last Accessed: January 20, 2012

[W22] Windows Communication Foundation. <http://msdn.microsoft.com/en-us/netframework/aa663324>, Last Accessed: January 20, 2012

[W23] IBM Service Oriented Architecture — SOA <http://www-01.ibm.com/software/solutions/soa/>, Last Accessed: January 20, 2012

[W24] Oracle Service-Oriented Architecture. <http://www.oracle.com/us/technologies/soa/index.html>, Last Accessed: January 20, 2012

[W25] SOAP Envelope Element. http://www.w3schools.com/soap/soap_envelope.asp, Last Accessed: January 20, 2012

[W26] Performance Considerations in Applications for Windows Phone. http://msdn.microsoft.com/en-us/library/ff967560%28v=vs.92%29.aspx#BKMK_Threads, Last Accessed: January 20, 2012

[W27] SSL and Online Trust Information Center. <http://www.verisign.com/ssl/ssl-information-center/index.html> , Last Accessed: January 20, 2012

[W28] ITU-T Recommendation E.800 <http://wapiti.telecom-lille1.eu/commun/ens/peda/options/ST/RIO/pub/exposes/exposesrio2008-ttnfa2009/Belhachemi-Arab/files/IUT-T%20E800.pdf>, Last Accessed: January 20, 2012

[W29] Web Service Scalability and Performance with Optimizing Intermediaries. <http://www.w3.org/2001/04/wsws-proceedings/mnot/wsws-nottingham.pdf>, Last Accessed: January 20, 2012

[W30] Brief Summary of Popular Database Systems. <http://www.paragoncorporation.com/ArticleDetail.aspx?ArticleID=22>, Last Accessed: January 20, 2012

[W31] Microsoft SQL Azure <http://www.windowsazure.com/en-us/home/tour/storage/>, Last Accessed: January 20, 2012

- [W31] ACID. <http://en.wikipedia.org/wiki/ACID>, Last Accessed: January 20, 2012
- [W32] Xeround the cloud database <http://xeround.com/>, Last Accessed: January 20, 2012
- [W33] Linq to Entities <http://msdn.microsoft.com/en-us/library/bb386964.aspx>, Last Accessed: January 20, 2012
- [W34] Hibernate. <http://www.hibernate.org/>, Last Accessed: January 20, 2012
- [W35] Class: ActiveRecord::Base. <http://api.rubyonrails.org/classes/ActiveRecord/Base.html>, Last Accessed: January 20, 2012
- [W36] Combating the Select N + 1 Problem In NHibernate. <http://ayende.com/Blog/archive/2006/05/02/CombatingTheSelectN1ProblemInNHibernate.aspx>, Last Accessed: January 20, 2012
- [W37] Shard (database architecture). http://en.wikipedia.org/wiki/Shard_%28database_architecture%29, Last Accessed: January 20, 2012
- [W38] MongoDB. <http://www.mongodb.org/>, Last Accessed: January 20, 2012
- [W39] Cassandra. <http://cassandra.apache.org/>, Last Accessed: January 20, 2012
- [W40] CouchDB. <http://couchdb.apache.org/>, Last Accessed: January 20, 2012
- [W41] Amazon SimpleDB. <http://aws.amazon.com/simpledb/>, Last Accessed: January 20, 2012
- [W42] Bigtable: A Distributed Storage System for Structured Data. <http://labs.google.com/papers/bigtable.html>, Last Accessed: January 20, 2012
- [W43] Table Service API. <http://msdn.microsoft.com/en-us/library/dd179423.aspx>, Last Accessed: January 20, 2012
- [W44] Eventual Consistency. http://en.wikipedia.org/wiki/Eventual_consistency, Last Accessed: January 20, 2012
- [W45] Sasktel Annual Report 2008. <http://www.sasktel.com/about-us/company-information/financial-reports/attachments/08-annual-report.pdf>, Last Accessed: January 20, 2012
- [W46] Amazon Elastic Compute Cloud (Amazon EC2). <http://aws.amazon.com/ec2/>, Last Accessed: January 20, 2012
- [W47] Heroku. <http://heroku.com/>, Last Accessed: January 20, 2012

- [W48] AppHarbor. <http://appharbor.com/>, Last Accessed: January 20, 2012
- [W49] Windows Azure. <http://www.microsoft.com/windowsazure/windowsazure/>, Last Accessed: January 20, 2012
- [W50] Six Benefits of Cloud Computing. <http://web2.sys-con.com/node/640237>, Last Accessed: January 20, 2012
- [W51] Building Highly Scalable Web Applications
<http://www.slideshare.net/iwmw/buildinghighly-scalable-web-applications>, Last Accessed: January 20, 2012
- [W52] Method Definitions. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>, Last Accessed: January 20, 2012
- [W53] Caching in HTTP. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html>, Last Accessed: January 20, 2012
- [W54] What is an application domain – an explanation for .NET beginners
<http://codebetter.com/raymondlewallen/2005/04/04/what-is-an-application-domain-an-explanation-for-net-beginners/>, Last Accessed: January 20, 2012
- [W55] AppDomain Class <http://msdn.microsoft.com/en-us/library/system.appdomain.aspx>, Last Accessed: January 20, 2012
- [W56] Notes on Little's Law <http://www.columbia.edu/~ks20/stochastic-I/stochastic-I-LL.pdf>, Last Accessed: January 20, 2012
- [W57] HATEOAS <http://www.infoq.com/news/2009/06/hateoas-dsl-protocol-description>, Last Accessed: January 20, 2012
- [W58] Amazon EC2 Instance Types, Last Accessed: January 20, 2012