

RESTFUL SERVICE COMPOSITION

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

By

Dong Liu

©Dong Liu, October 2013. 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

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C9

ABSTRACT

The Service-Oriented Architecture (SOA) has become one of the most popular approaches to building large-scale network applications. The web service technologies are de facto the default implementation for SOA. Simple Object Access Protocol (SOAP) is the key and fundamental technology of web services. Service composition is a way to deliver complex services based on existing partner services. Service orchestration with the support of Web Services Business Process Execution Language (WSBP EL) is the dominant approach of web service composition. WSBP EL-based service orchestration inherited the issue of interoperability from SOAP, and it was furthermore challenged for performance, scalability, reliability and modifiability.

I present an architectural approach for service composition in this thesis to address these challenges. An architectural solution is so generic that it can be applied to a large spectrum of problems. I name the architectural style RESTful Service Composition (RSC), because many of its elements and constraints are derived from Representational State Transfer (REST). REST is an architectural style developed to describe the architectural style of the Web. The Web has demonstrated outstanding interoperability, performance, scalability, reliability and modifiability.

RSC is designed for service composition on the Internet. The RSC style is composed on specific element types, including RESTful service composition client, RESTful partner proxy, composite resource, resource client, functional computation and relaying service. A service composition is partitioned into stages; each stage is represented as a computation that has a uniform identifier and a set of uniform access methods; and the transitions between stages are driven by computational batons. RSC is supplemented by a programming model that emphasizes on-demand function, map-reduce and continuation passing. An RSC-style composition does not depend on either a central conductor service or a common choreography specification, which makes it different from service orchestration or service choreography.

Four scenarios are used to evaluate the performance, scalability, reliability and modifiability improvement of the RSC approach compared to orchestration. An RSC-style solution and an orchestration solution are compared side by side in every scenario. The first sce-

nario evaluates the performance improvement of the X-Ray Diffraction (XRD) application in ScienceStudio; the second scenario evaluates the scalability improvement of the Process Variable (PV) snapshot application; the third scenario evaluates the reliability improvement of a notification application by simulation; and the fourth scenario evaluates the modifiability improvement of the XRD application in order to fulfil emerging requirements. The results show that the RSC approach outperforms the orchestration approach in every aspect.

ACKNOWLEDGEMENTS

It is one of the most brilliant decisions that I have made to start my PhD program at the Department of Computer Science, University of Saskatchewan in 2004. The faculty, staff, and students supported and helped me both professionally and personally. I chose my research topic and went through the long journey with the mentoring, inspirations and encouragements of Ralph Deters, my supervisor. My committee members, Chris, Nate, Eric, and Rick have been patient over years and helped me on every step towards this thesis. Nate contributed many valuable suggestions based on his careful reviews of my work. Patrick, My external reviewer, provided feedbacks to improve this document. Jan and Gwen took really good care of all the program-related procedures over years. It was my pleasure to stay in such a friendly environment made by the department faculty and staff.

I would like to thank Steward, Mike, Marina, Jinhui, Nathenial, Yuhong, Chris, Dylan, Diony, and Elder, who collaborated in the Science Studio project from the University of West Ontario, Concordia University, IBM, and Canadian Light Source. I also thank my colleagues at Canadian Light Source for their support for not only my duty R&D work but also my PhD research work.

The work described in this thesis was supported by NSERC, NRC, CIHR, CANARIE, Canadian Light Source, and the University of Saskatchewan. I would like to thank for their generous support without which I will not have the chances to work in the projects that connect perfectly to my research.

I would like to thank all the persons and organizations who contribute to the open-source software and free services that I have used during the development of this work.

Finally, I want to thank my family, including my parents and parent-in-laws, for their continuous supports and encouragements.

To Na, Albert, and Emily,
for their love, understanding, and patience.
And also to the Web,
for all the connections to this work.

CONTENTS

Permission to Use	i
Abstract	ii
Acknowledgements	iv
Contents	vi
List of Tables	ix
List of Figures	x
List of Listings	xi
List of Equations	xii
List of Abbreviations	xiii
1 Introduction	1
1.1 Distributed computing	1
1.2 Web services and composition	3
1.3 Challenges of web service composition	4
1.4 An architectural approach	6
1.4.1 Representational State Transfer (REST)	6
1.4.2 RESTful service composition	7
1.5 Contributions	8
1.6 Structure of the thesis	8
2 Web Services and REST	9
2.1 RPC	10
2.2 Object-oriented RPC	11
2.3 XML	13
2.4 XML-RPC	14
2.5 SOAP and WSDL	14
2.6 Stack of web service technologies	15
2.7 Web architecture	16
2.7.1 Web architectural elements	18
2.7.2 Web architectural constraints and rationale	19
2.7.3 Web standards and specifications	25
2.8 REST and RESTful services	26
2.9 Interesting properties of web services	29
2.9.1 Performance	29

2.9.2	Scalability	30
2.10	Lessons of distributed computing	32
2.11	Summary	34
3	Web Service Composition	35
3.1	A case study	35
3.2	Web service orchestration	37
3.2.1	WSBPEL	40
3.2.2	Essentials of WSBPEL	44
3.2.3	Technical drawbacks of WSBPEL	45
3.2.4	Other service orchestration approaches	46
3.3	Web service choreography	47
3.3.1	WS-CDL	49
3.3.2	Essentials of WS-CDL	52
3.3.3	Technical drawbacks of WS-CDL	52
3.4	Interesting properties of web service compositions	53
3.4.1	Performance	53
3.4.2	Scalability	55
3.4.3	Reliability	58
3.4.4	Modifiability	60
3.5	Summary	61
4	RESTful Service Composition	62
4.1	Describing software architectural style	63
4.2	Defining RESTful Service Composition	64
4.2.1	The problem domain	65
4.2.2	Element types and roles	65
4.2.3	Constraints	69
4.2.4	Consequences	71
4.2.5	An example	74
4.2.6	The connections to RESTful Service Composition	77
4.3	Programming RESTful Service Composition	82
4.3.1	Functional programming elements for RESTful service composition	82
4.3.2	Infrastructure for programming RESTful service composition	84
4.3.3	Related programming paradigms	85
4.4	Summary	87
5	Evaluation	89
5.1	XRD file transfer service	91
5.1.1	Experiment setup	91
5.1.2	Other implementation details	93
5.1.3	Measurement result and analysis	94
5.2	PV snapshot	96
5.2.1	PVs as RESTful services	96
5.2.2	Experiment setup	97

5.2.3	Measurement result and analysis	100
5.3	Notification in the presence of network partitions	103
5.3.1	Simulation parameters	104
5.3.2	Simulation results and analysis	106
5.4	Modification of the XRD file transfer service	109
5.4.1	Emerging requirements and corresponding modifications	109
5.4.2	Modifiability analysis	111
5.5	Summary	113
6	Conclusions	114
A	Scalability	117
A.1	Performance, capacity and scalability	117
A.2	Definition of scalability	118
A.3	Scalability models	119
A.3.1	Amdahl's law	119
A.3.2	Gunther's scalability model	119
	References	121

LIST OF TABLES

2.1	Comparison of RESTful web services and SOAP-based web services.	28
3.1	WSBPEL Activities and Corresponding BNF notations	41
3.2	BPMN notations	48
3.3	Common flow structures of WSBPEL and WS-CDL.	54
4.1	The semantics of HTTP methods for computation and computation instances in RSC.	70
4.2	File and composite file resource interfaces.	74
5.1	The application scenarios and the corresponding evaluation goals.	90
5.2	Testing environments and workload for the XRD file transfer service implemen- tations.	92
5.3	The semantics of common PV access methods.	97
5.4	The experiment setup to evaluate different implementations of the PV snapshot application.	98
5.5	The response time in milliseconds for disconnected PVs and corresponding scale factor.	102
5.6	The parameters and metrics for notification simulation.	107
5.7	Parameter values specified for the simulations.	107
5.8	The orchestration and RSC interfaces for Requirements 1 to 5	112

LIST OF FIGURES

1.1	The trend of distributed computing.	2
1.2	The spoke-hub pattern of service orchestrations' structure.	5
2.1	Common interaction sequence of RPC. 1/6 and 3/4 are local calls. 2/5 is a remote call. Marshalling happens in 1/2 and 4/5, and unmarshalling in 2/3 and 5/6.	11
2.2	The web service specifications and their dependencies.	17
2.3	A process view of a web architecture.	19
2.4	The stack of web standards.	25
2.5	The derivation of REST [38].	26
2.6	The throughput and residence time of a web service versus the number of concurrent active threads.	31
3.1	Two roles of conductor service in a service orchestration.	37
3.2	Two architectural approaches for session sharing.	57
4.1	The flow structure of RSC compared with typical hub-spoke structure of service orchestration.	71
4.2	Asynchronous conversations composed of synchronous messaging.	73
5.1	The message sequences of the orchestration and RSC-style implementations for XRD file transfer.	93
5.2	The image transfer time from CLS to UWO.	95
5.3	The response time of various implementations of the PV snapshot application for three different snapshot tasks.	102
5.4	Two implementations for notification.	105
5.5	The state transitions of a notification task.	106
5.6	The failure ratio and latency of orchestration and RSC implementations obtained from simulation.	108

LIST OF LISTINGS

3.1	A BNF notation for describing service endpoint interface.	38
3.2	The interface of scan service.	38
3.3	The interface of CLS data service.	39
3.4	The interface of UWO data service.	39
3.5	The interface of UWO processing service.	39
3.6	A service orchestration implementation of the XRD scenario in WSBPEL. . .	42
3.7	BNF notation for WS-CDL.	50
3.8	A choreography for XRD image processing.	51
4.1	Part of an orchestration of RESTful services.	74
4.2	The relaying service of uwo_data_service.	76
5.1	A single-threading implementation of the snapshot based on the PV class. . .	98
5.2	A single-threading implementation of the snapshot based on the <code>ca</code> class. . .	98
5.3	A multi-threading implementation of the snapshot based on the PV class. . .	99
5.4	A multi-threading implementation of the snapshot based on the <code>ca</code> class. . .	99
5.5	A node.js implementation of the snapshot based on the EPICS <code>caget</code> command line tool.	100

LIST OF EQUATIONS

2.1	Web resource	17
2.2	Software architecture	18
2.3	Little's law	29
3.1	Availability	58
A.4	Scalability	118
A.5	Discrete scalability	118
A.12	Scale factor	118
A.13	Linear solution of scale factor measured by throughput	119
A.14	Linear solution of scale factor measured by residence time	119
A.16	Speedup	119
A.18	Amdahl's law	119
A.19	Gunther's scalability model	120

LIST OF ABBREVIATIONS

AIO	Asynchronous Input/Output.
AJAX	Asynchronous JavaScript And XML.
ALS	Advanced Light Source.
API	Application Programming Interface.
ASP	Active Server Pages.
B2B	Business To Business.
B2C	Business To Consumer.
BNF	Backus-Naur Form.
BPMI	Business Process Management Initiative.
BPMN	Business Process Model and Notation.
BT	BitTorrent.
CA	Channel Access.
CCD	Charge-Coupled Device.
CIFS	Common Internet File System.
CLS	Canadian Light Source.
COD	Code-On-Demand.
CORBA	Common Object Request Broker Architecture.
CPS	Continuation-Passing Style.
CPU	Central Processing Unit.
CREST	Computational REST.
DB	Database.
DCOM	Distributed Component Object Model.
DNS	Domain Name System.

DSM	Distributed Shared Memory.
DTD	Data Type Definition.
EAI	Enterprise Application Integration.
EPICS	Experimental Physics and Industrial Control System.
FTP	File Transfer Protocol.
GUI	Graphical User Interface.
HATEOAS	Hypermedia As The Engine Of Application State.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
HTTPS	Hypertext Transfer Protocol Secure.
I/O	Input/Output.
IDL	Interface Definition Language.
IETF	Internet Engineering Task Force.
IOC	Input/Output Controller.
JSON	JavaScript Object Notation.
JSP	JavaServer Pages.
JVM	Java Virtual Machine.
LAN	Local Area Network.
MEP	Message Exchange Pattern.
MIME	Multipurpose Internet Mail Extensions.

MTBF	Mean Time Between Failures.
MTTF	Mean Time To Failure.
MTTR	Mean Time To Repair.
MVC	Model-View-Controller.
OMG	Object Management Group.
ONC	Open Network Computing.
OOP	Object-Oriented Programming.
P2P	Peer To Peer.
PV	Process Variable.
REST	Representational State Transfer.
RFC	Request For Comments.
RMI	Remote Method Invocation.
RPC	Remote Procedure Call.
RSC	RESTful Service Composition.
SEDA	Staged Event-Driven Architecture.
SGML	Standard Generalized Markup Language.
SOA	Service-Oriented Architecture.
SOAP	Simple Object Access Protocol.
TCP	Transmission Control Protocol.
UDDI	Universal Description Discovery and Integration.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
UWO	University of Western Ontario.

VHDL	VHSIC (Very-High-Speed Integrated Circuits) Hardware Description Language.
VM	Virtual Machine.
W3C	World Wide Web Consortium.
WAN	Wide Area Network.
WF	Windows Workflow Foundation.
WS	Web Services.
WS-CDL	Web Services Choreography Description Language.
WS-I	Web Services Interoperability.
WSBPEL	Web Services Business Process Execution Language.
WSDL	Web Service Description Language.
WWW	World Wide Web.
XAML	Extensible Application Markup Language.
XHR	XMLHttpRequest.
XML	Extensible Markup Language.
XRD	X-Ray Diffraction.
YAML	Yet Another Markup Language.

CHAPTER 1

INTRODUCTION

The Web is more a social creation than a technical one.

—Tim Berners-Lee, *Weaving the Web*

1.1 Distributed computing

Distributed computing has been largely shaped by popular programming languages and paradigms. A “Déjà Vu” was observed [100]: Remote Procedure Call (RPC) appeared in the 1980’s with procedural languages, and object-oriented RPC prospered in the 1990’s after Object-Oriented Programming (OOP) languages became mainstream. The development of information representation methods also impacted distributed computing technologies. For example, Simple Object Access Protocol (SOAP) is based on Extensible Markup Language (XML). Figure 1.1 shows significant events related to programming languages, standards and techniques in distributed computing over the time. While more and more distributed computing paradigms, standards and implementations emerged, language-, platform-, and vendor-independence have always been a theme of the developments.

Among all the events, the emergence of the Web, or World Wide Web (www), has had the greatest influence. The Web might be the first real success among all attempts to develop large scale language-, platform-, and vendor-independent distributed systems. The Web has influenced the development of distributed computing since the 1990’s. The Web showed more advantages as a distributed computing platform than others. It was rediscovered by the name of “Representational State Transfer (REST)” [39] with the tide of Web 2.0. Besides the Web, cloud computing [8] and mobile computing have also brought new problems and

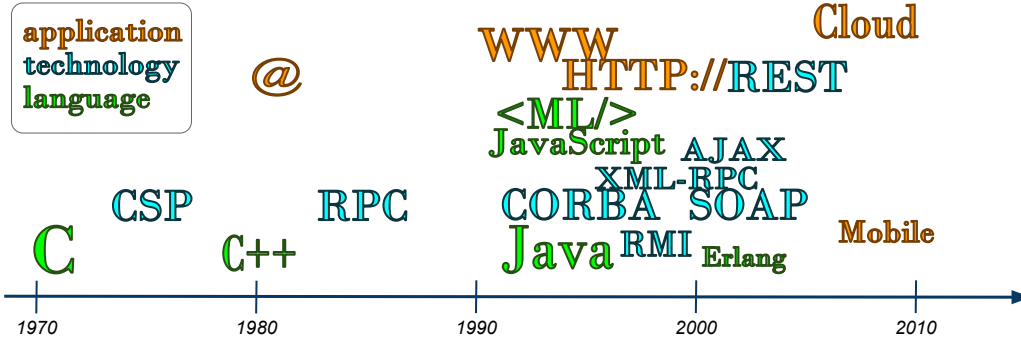


Figure 1.1: The trend of distributed computing.

technologies to distributed computing. Distributed applications got more and more diverse from Enterprise Application Integration (EAI) to Business To Business (B2B), Business To Consumer (B2C), and Peer To Peer (P2P). Computing devices extended from dedicated servers to personal computers and mobile devices. More and more distributed applications run on the Internet beyond Local Area Networks (LANs) and Wide Area Networks (WANs).

As its concept suggested, distributed computing always involves communication between computers. However, I found that it was incomplete to view distributed computing merely from a computer-computer communication perspective¹, when reviewing the successes and failures of distributed computing technologies. That is, distributed computing is more than making the communication between computers *correct* and *efficient*. The other perspectives that should be taken into the view are the *correctness* and *efficiency* of developer-computer communication and developer-developer communication.

Developer-computer communication is the activity where a developer, or even an experienced user, uses available interfaces to define and modify an application. Developer-computer communication also refers to the activity where the machine informs a developer of the state of an application and aids the developer in finishing the application. For developers, it could mean programming, testing, debugging, modifying and maintaining an application. For users, it could mean participating and finishing an application.

Developer-developer communication is the activity that two or more developers or teams collaborate in order to define and modify an application. For developers, it could mean

¹I did not use the term “interaction” here in order to avoid the confusion with terms like human-computer interaction.

documenting, sharing, reviewing, and modifying programming artifacts. For users, it could mean sharing application states with other users, and even learning to program based on documentation.

1.2 Web services and composition

The Service-Oriented Architecture (SOA) is a paradigm for designing, implementing and utilizing services in different ownership domains in a loosely-coupled and interoperable way [35]. A service-oriented system is a group of applications that interact with other(s) by providing and/or consuming services. A service refers to a concrete autonomous computation capability for retrieving and processing information. Service interfaces are specified by document(s) rather than specific programming or binary code [20]. There are already numerous “services” running in enterprise and personal computing environments, like “create an order”, “fulfil an order”, and “check email”. However, most of those services were not ready for easy access due to their heterogeneous platforms, languages, and ownership. The industry has developed many approaches to enabling easy access to the services.

Generally speaking, Web Services (WS) are a collection of technologies on the basis of SOAP, covering the areas of messaging, security, process, and management. SOAP is a standard for exchanging XML-represented information between two distributed applications, or two web services [79]. SOAP originally referred to “Simple Object Access Protocol”, which indicated that the structured information was indeed objects specified by object-oriented programming languages. The term “web” in “web service” came from the fact that most web services leverage Hypertext Transfer Protocol (HTTP) to *transport* SOAP messages² although other transportation protocols can also be used.

High-order services, taking advantage of service composability, provide their functionalities by consuming partner services. Such high-order services, or service compositions, can be implemented basically in two ways, service orchestration and service choreography. Service orchestration depends on a conductor-like central service that acts as service consumers of partner services. Service choreography, on the other hand, does not have a central service

²Although HTTP is often used for just transportation, it is an *application* layer network protocol.

and assumes that all partner services are collaborative enough to be able to achieve their common goal. Because the partner services of a service orchestration do not need to know any details about the orchestration, service orchestration is much easier to implement than choreography. Service orchestration is the dominant approach to composing web services currently.

1.3 Challenges of web service composition

SOAP aimed to address the interoperability problem in object-oriented RPC by standardization. Before SOAP, it is almost impossible to program the client and server sides of a RPC program in two different languages. WS has achieved a big progress beyond its predecessors like RPC and XML-RPC. However, the interoperability issue remained for web services developed in different languages and platforms. Even if the SOAP-based WS standards are followed by a web service vendor, the interoperability is still doubtful. Therefore, the Web Services Interoperability (WS-I) organization³ has published another set of standards to guide the development and testing of web services towards true interoperability, for example, the interoperability between a web service developed in Java and the other in .Net. The first guiding principle of the WS-I basic profile is “No guarantee of interoperability” [106].

Besides interoperability, some other interesting properties⁴ — performance, scalability, reliability, and modifiability — are also challenges for SOAP-based web services. RPC-based technologies aimed to achieve convenience in programming by allowing developers to program a RPC the same as a local one. SOAP, derived from RPC, aimed to provide the same convenience to object-oriented programmers while taking advantage of XML as a standard message format and HTTP for transportation through firewalls. However, the convenience of SOAP misled developers as did RPC. The illusion that a remote method call is the same as a local one introduces many pitfalls for the correctness of developer-computer communication. Furthermore, SOAP has not addressed several other important aspects in computer-computer communication and developer-developer communication. This will be further discussed in

³See <http://www.ws-i.org/>

⁴The details of these properties are discussed in Chapter 2.

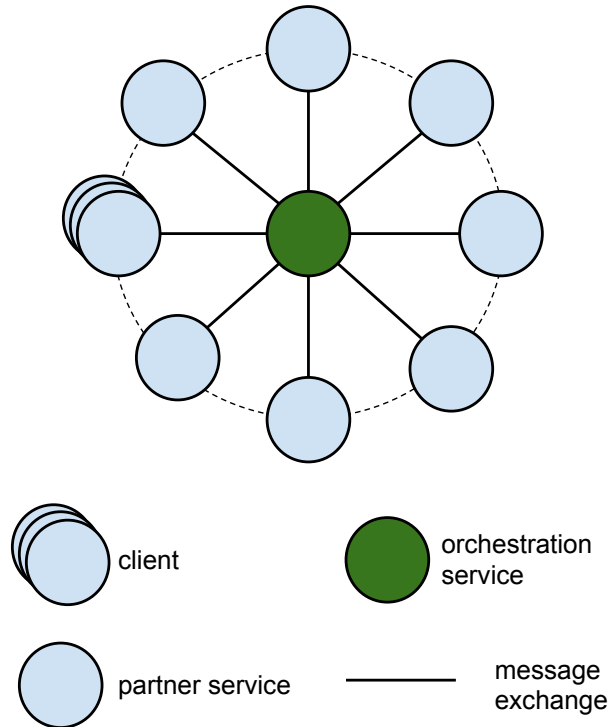


Figure 1.2: The spoke-hub pattern of service orchestrations' structure.

Chapter 2.

Web service orchestration inherits the shortcomings from web services — deficiencies of interoperability, performance, scalability, reliability, and modifiability. Some of these shortcomings become even more critical for server compositions. A service orchestration is normally constructed in a spoke-hub pattern shown in Figure 1.2. The central conductor service resides at the hub, and clients and partner services are at the spoke ends.

1. The central conductor service needs to support many concurrent orchestration instances and handle more Input/Output (I/O) operations than normal services, which can bring more performance and scalability challenges.
2. The central conductor service is a single point of failure and, therefore, is critical to the reliability of a service composition.

1.4 An architectural approach

There could be many approaches to addressing the challenges for service compositions, for example, developing and using web service languages or frameworks with better interoperability and scalability. A more generic approach, I think, is to address these challenges at the architectural level. This is because:

A software system’s non-functional properties are shaped by its architecture.

Performance, scalability, reliability, and modifiability are non-functional properties of software systems. Many design decisions have to be made at the architectural level in order to fulfil those non-functional requirements [45, 68]. Software architecture is a set of elements that are selected and organized according to particular constraints and rationale [86].

An architectural approach is generic. Although a system’s non-functional properties are influenced by specific infrastructure and the language or framework used to program it, an infrastructure or language or framework solution for scalability is often specific to a certain context and will be quite limited in a heterogeneous environment. On the contrary, an architectural approach can benefit many systems no matter what infrastructures or languages/frameworks they are based on.

1.4.1 Representational State Transfer (REST)

Representational State Transfer (REST) was coined in 2000 by Roy T. Fielding, who was the contributor to several web standards, including HTTP [14, 36] and Uniform Resource Identifier (URI) [15], one of the core contributors of Apache httpd, one of the first modern web servers⁵, and also one of the founders of Apache Software Foundation. In his PhD thesis [38], Fielding wrote, “The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax ...” , and “encompasses the

⁵See http://httpd.apache.org/ABOUT_APACHE.html

fundamental constraints upon components, connectors, and data that define the basis of the web architecture, and thus the essence of its behavior as a network-based application”.

REST aimed to improve the performance, scalability, simplicity and visibility of network-based applications. The web architecture [17] can be considered as a typical RESTful application. Furthermore, REST naturally encourages correct and efficient developer-computer communication and developer-developer communication. Chapter 2 discusses more details of REST’s advantages.

There have been many debates about SOAP/WS versus REST from industry [88] to academics [85] since the early 2000s. About ten years later, big web media and application vendors like Google, Yahoo, Microsoft, Facebook and Twitter have all started to use REST to label their products. However, the software developer community still has no agreement on whether or how an application is *critically* RESTful, and some hope a maturity model⁶ can help. In fact, some mature technologies of the current Web, for example, the cookies, did not follow the REST style [38]. In my opinion, it is more important to *apply* and *extend* the architectural style to real problems than to assert dogmatically whether a solution is truly RESTful.

1.4.2 RESTful service composition

When comparing different architectural approaches to improving the performance, scalability and modifiability of service orchestration, I found that many valuable aspects of those approaches were included in or can be derived from REST. Therefore, I use the term RESTful Service Composition (RSC) to refer to the architectural style that I present in this thesis in order to address the challenges for service composition. What makes a RESTful service composition? How to program such a RESTful service composition? Can this RESTful service composition approach bring better scalability, reliability and modifiability than normal web service composition approaches? These are the questions discussed by this thesis.

My research focuses on not only the architectural style for service compositions but also the programming model to realize it. System architecture can be designed to be independent

⁶Richardson Maturity Model: steps toward the glory of REST. See <http://martinfowler.com/articles/richardsonMaturityModel.html>

of the programming languages or frameworks used to implement it. However, a *programming model* can ease the implementations of an architectural design. A programming model abstracts a group of programming problems and provides the solution in the form of Application Programming Interface (API) design and usage.

1.5 Contributions

The contributions of this thesis are the following:

- I present an architectural style of RESTful Service Composition (RSC) that is thoroughly different from traditional service orchestration approaches. A RSC is partitioned into computational stages, and the transition between stages are driven by baton passing. The definition of RSC is the key contribution of my thesis.
- I present a programming model for RSC that is designed for the specific development requirements of RSC by leveraging functional programming technologies. The programming model is a supplement to RSC.
- The architectural style is evaluated in terms of performance, scalability, reliability and modifiability. The results quantitatively show that RSC outperforms the service orchestration approach in all the real-world scenarios used in the evaluation.

1.6 Structure of the thesis

Chapter 2 reviews the basic technologies of web services, compares pros and cons of RPC, Web Services, and REST, and discusses performance and scalability of web services. Chapter 3 reviews the technologies of web service composition, and discusses reliability and modifiability. Chapter 4 derives the architectural style of RSC, and presents a corresponding programming model. Chapter 5 evaluates RSC by comparing the performance, scalability, reliability and modifiability of composition applications in four scenarios. Chapter 6 is the conclusions. Some contents of this thesis have appeared in my previous publications [72, 73, 74, 75].

CHAPTER 2

WEB SERVICES AND REST

It's called Accessibility, and it's the most important thing in the computing world.

—Steve Yegge, *Stevey's Google Platforms Rant*

The Simple Object Access Protocol (SOAP) protocol was standardized by the World Wide Web Consortium (W3C) in 2000¹, and later became the foundation of web service technologies. The activities of SOAP development can be traced back to 1998, when Don Box aimed to “replace DCOM with XML” within Microsoft [88]. SOAP leveraged Extensible Markup Language (XML) and Hypertext Transfer Protocol (HTTP) to tackle the problems of object-oriented Remote Procedure Call (RPC) — vendor-, environment-, and system- dependency, and most significantly, inability to work for the Internet applications [19]. In order to solve the firewall problem for the Distributed Component Object Model (DCOM), SOAP used HTTP for message transportation, while HTTP offers much more capabilities than transportation. During an interview in 2009², Box mentioned that the SOAP team was not familiar with web technologies back in 1998, which somehow explained why HTTP, an application protocol, was used as a transportation mechanism by SOAP from the beginning. The Web got more and more attention with the Internet boom in late 1990s and the tide of Web 2.0 in 2000s. The extensive development and usage of web applications brought better understanding of web architecture to the distributed computing community.

¹See <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>

²See <http://www.infoq.com/interviews/box-soap-xml-rest-m>

2.1 RPC

A distributed application is much more complicated than a non-distributed one of the same purpose, for example, a network file system versus a local file system. The major difference between them is the complexity of communication. Distributed components can be programmed in various languages, and be deployed on diverse operating systems. Remote Procedure Call (RPC) is a programming model that allows the programmers to develop a distributed application without worrying about network communication details. In RPC, the call of a procedure residing in a different address space can be programmed almost the same as the call of a local procedure from a programmer's perspective. In this way, the complexity of remote communication is handled by the RPC library and tool-generated stubs. The idea of RPC can be traced back to Internet Engineering Task Force (IETF) Request For Comments (RFC) 707 [103] published in 1976 [104]. RPC improved the productivity of distributed application developers by its convenience, and also generated a mirage that a distributed application can be programmed the same as a local one.

The first implementation of RPC on Unix was SUN RPC, which was developed for the NFS, SUN's network file system. Other RPC implementations had a quite similar design to SUN RPC. It was later standardized as Open Network Computing (ONC) RPC [95]. A developer normally uses a tool named `rpcgen` to generate server and client stubs from an interface description of the procedure. A stub is a piece of code to marshal/unmarshal the parameters and return value. The developer needs to write the client code that implements client logic and calls the client stub, and also the server code that implements procedure logic and is called by the server stub. Both the client code and server code have no significant difference from the caller and callee of a local procedure call. The sequence of interactions between the client code and the server code in an RPC is shown in Figure 2.1.

The programming model of RPC involves two types of communication in distributed computing — developer-computer and computer-computer. As shown in Figure 2.1, the programming interface is for developer-computer communication, and the address boundary is for computer-computer communication. RPC makes the programming interface extremely easy, and technically hides the complexity of address boundary from the developers.

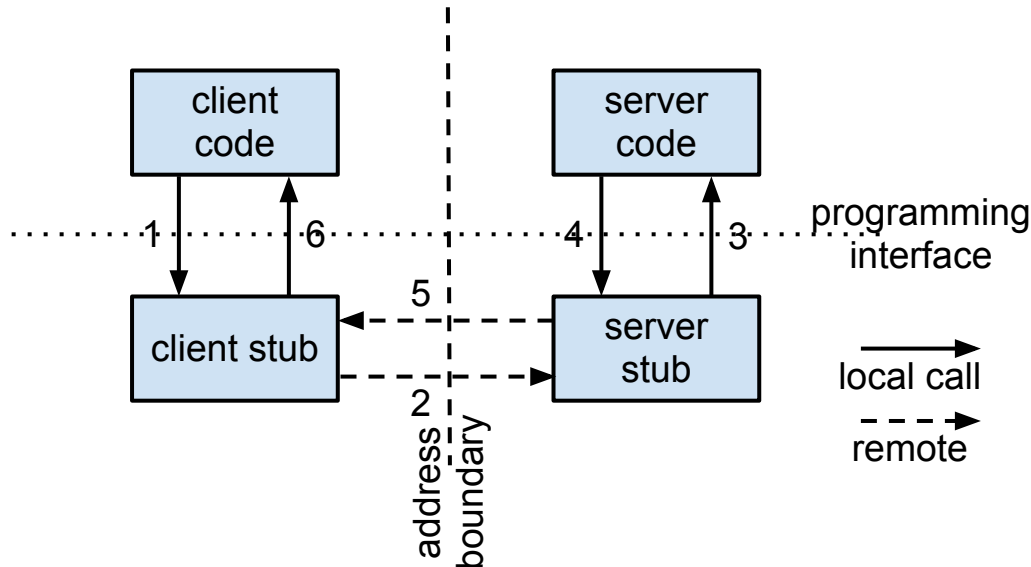


Figure 2.1: Common interaction sequence of RPC. 1/6 and 3/4 are local calls. 2/5 is a remote call. Marshalling happens in 1/2 and 4/5, and unmarshalling in 2/3 and 5/6.

RPC's *convenience* has its cost. Two of the most significant drawbacks of RPC are

Opaque messages Most RPC developers have no idea about how the procedure parameters and returned values are marshalled/unmarshalled. It is too expensive to develop an intermediary application that can interpret the message and process it between the client stub and server stub even for the developers with deep knowledge of RPC. The messages are not human-readable.

Tight coupling between client and server The client and server stubs are always released as a pair. It is almost impossible to program the client and server sides in different languages, to develop a compatible client or server side without the right stub, or to update one side's stub without breaking the compatibility.

2.2 Object-oriented RPC

When Object-Oriented Programming (OOP) became the mainstream programming paradigm, RPC also had its object-oriented version, for example, Java Remote Method Invocation (RMI). The object-oriented RPC approaches were very similar to the original RPC. A language-

specific tool helps the developer to generate the client stub and server stub³, and the object marshalling/unmarshalling⁴ are carried out by the library. The problems for RPC remained in object RPC.

In order to develop distributed objects compatible with various object-oriented languages, the Common Object Request Broker Architecture (CORBA) was standardized⁵ by the Object Management Group (OMG). The inter-language compatibility was achieved by standardizing the mapping from an Interface Definition Language (IDL) to specific languages like C++ and Java.

CORBA rose in the second half of the 1990s. Its growth was inhibited by Microsoft's DCOM, and later was overwhelmed by the rise of web technologies. CORBA provided a technical solution to the language- and platform- dependency problem of RPC. However, it is much more complicated than RPC. For example, the CORBA 2.0 specification is 634 pages long, and only covered the mapping from IDL to C, C++ and Smalltalk. In order to implement CORBA, every programming language needs a specific mapping specification and corresponding implementation. With CORBA, it is still impossible to update one side without breaking the client-server compatibility, which yields the incompatibility between different versions of the same application. Besides these issues, CORBA was also criticized by its standardization approach [52].

At the end of the 1990s, Business To Business (B2B) and Business To Consumer (B2C) applications started to attract more attention in the area of distributed computing, where EAI used to be the focus. These applications brought two new requirements:

1. The application must work over the Internet across organizational, geographical, and network boundaries. The most common hurdles are the firewalls.
2. Security must be considered to protect organizations and consumers in an application. No application can go to production without security measures.

Unfortunately, RPC, object-oriented RPC and distributed objects technologies failed to fulfil these two requirements. In order to apply RPC technologies to B2B and B2C, web tech-

³It is also called a skeleton for Java RMI.

⁴Serialization/deserialization are often the terms used for objects.

⁵See <http://www.omg.org/spec/CORBA/>

nologies like XML and HTTP were used for object marshalling/unmarshalling and transportation. However, it does not mean that there is no successful usage of RPC-based technologies. Google, Facebook, and twitter have all invented their own *new* RPC frameworks: protocol buffer⁶, thrift⁷, and finagle⁸.

2.3 XML

The rise of the Web brought a bunch of new technologies to the distributed computing community. Among these was the Hypertext Markup Language (HTML), which introduced a simple yet powerful way to represent information. The success of HTML attracted developers to work on a new markup language in W3C based on the same root of HTML — Standard Generalized Markup Language (SGML) — since the mid-1990s⁹ in order to represent information not merely for web browsers¹⁰. The result was the Extensible Markup Language (XML), and it was once even considered the “silver bullet” for information exchange [77].

XML’s structure is more restricted than HTML, while its vocabulary is more extensible. The former factor benefited the development of XML parsing and query works. The latter factor backed its expression capability with the support of the Data Type Definition (DTD) and XML schema. Basically, XML can be used in any case when some information needs to be presented in a machine-readable structured way. Therefore, it has been used to represent digital media, data or objects to be persistent or serialized, interface descriptions, and also deployment descriptions.

Ten design goals were proposed for XML [23, 22]. During its development and applications over more than a decade, most of the goals have been achieved. However, the success of one goal — “XML documents should be human-legible and reasonably clear” — is still questionable. Nowadays, XML has been used to represent very complicated structured information, among which many were generated by machines. Such XML documents are not just “wordy” for humans, but exceed the reading capability of developers. Using only-machine-readable

⁶See <http://code.google.com/p/protobuf/>

⁷See <http://thrift.apache.org/>

⁸See <http://twitter.github.com/finagle/>

⁹See <http://www.w3.org/XML/hist2002>

¹⁰See <http://www.w3.org/TR/WD-xml-961114.html>

XML documents for developers to document and communicate had negative impacts on some XML-related applications.

The Yet Another Markup Language (YAML)¹¹ and the JavaScript Object Notation (JSON)¹² are other approaches to representing structured data. Both YAML and JSON are designed for good human readability that is considered a shortcoming of XML [13].

2.4 XML-RPC

The problems of object-oriented RPC mentioned in Section 2.2 were addressed by XML-RPC in 1998¹³. The idea was straightforward — to use HTTP for transportation in order to let messages go through firewalls, and to use XML to encode the method calls, returns, and faults. Note that the HTTP POST method is used for all the request/response in the XML-RPC specification.

XML-RPC has been implemented by many languages, including some relatively new languages like Erlang and Clojure. Like RPC, many implementations provide a code generation tool for convenience. Because the specification does not include an IDL, the implementations have different approaches to defining the interfaces. The XML-RPC specification has never been formally standardized by an organization. This makes it impossible for XML-RPC to become a language- and platform-independent approach.

XML-RPC also has siblings like JSON-RPC and YAML-RPC. As their names suggest, they use JSON or YAML to encode messages instead of XML.

2.5 SOAP and WSDL

The Simple Object Access Protocol (SOAP) was inspired by XML-RPC, and RPC was one of its design goals [21]. SOAP defined a more complicated way to exchange structured information for distributed applications than XML-RPC. SOAP extended XML-RPC's message structure by adding a header part for meta information in front of the message body.

¹¹See <http://yaml.org>

¹²See <http://json.org>.

¹³XML-RPC Specification, see <http://www.xmlrpc.com/spec>

SOAP 1.1 was loosely specified in various aspects, which hindered the implementations' compatibility. For example, the specification did not clearly define how to locate the remote object to access. There were three mechanisms to locate a remote object: the HTTP request URI, the non-standard `SOAPAction` HTTP header, or an XML namespace in the SOAP message. There was no agreement on which of the three mechanisms should be enforced.

A big difference between SOAP and XML-RPC is the Web Service Description Language (WSDL), which is a must-have for all SOAP-based web services. WSDL is an XML-based language to describe exposed service endpoint operations, message formats to invoke the operations, and the binding to a network protocol [30]. Most SOAP implementations provided two tools, one to generate code from WSDL and the other to generate WSDL from code, namely `wsdl2code` and `code2wsdl`. These tools enable two development models of web services: code-first or WSDL-first. In practice, most web services were developed in a code-first way, and the WSDLs were generated. Writing a correct WSDL document is more difficult than writing a correct program. The machine-generated WSDLs documents were too complicated to be interpreted by programmers, and therefore, they were rarely read by humans.

2.6 Stack of web service technologies

A set of service-oriented principles was summarized by Don Box, one of the original designers of SOAP, and were known as four tenets [20]:

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

These four tenets were considered the guideline for web service development activities that Microsoft took part in. SOAP and WSDL were designed based on them, and the design of other web service specifications were also influenced.

Service registry and discovery was considered as an essential part of service-oriented computing. This had its root in agent-based computing [1]. Universal Description Discovery and Integration (UDDI) is an XML-based specification for registering and discovering SOAP- and WSDL-based web services. UDDI also refers to the service registries implementing the specification. Customers can get answers to “who, what, where, and how” questions about services from a UDDI. The answer to the question “who” is about the enterprise supplying services; the answer to “what” is about the services provided; that to “where” is the service Uniform Resource Locators (URLs) or email addresses; and that to “how” is about interfaces to interact with the services. In 2006, major vendors of UDDI closed their public UDDI services¹⁴. This did not imply that there was no need for sharing service descriptions, but it did indicate that UDDI was not accepted by the community. The reason could be that UDDI was too complicated or the specification did not, in fact, reflect the needs of the community.

Based on SOAP and WSDL, a large stack of specifications have been developed. The specifications covered almost all aspects of EAI and B2B applications: messaging, resources, transactions, security, management, workflow, and interoperability. Figure 2.2 shows the web service standards stack. The standards at the top depend on those beneath them. A more detailed illustration of web service standards can be found on the innoQ website¹⁵. The number of the standards is still growing, and new versions of some existing standards have been published. The heavy stack showed the wide-ranging applications of web services. On the other hand, it also shows the overwhelming complexity of web service technologies. The industry will eventually examine whether these specifications address the real requirements like the case of UDDI.

2.7 Web architecture

The first web server, browser, and web page was developed by Tim Berners-Lee, and started to be online by Christmas 1990 [16]. Three years later, the load on the first web page

¹⁴See <http://uddi.microsoft.com/about/FAQshutdown.htm> .

¹⁵See <http://www.innoq.com/soa/ws-standards/poster/> .

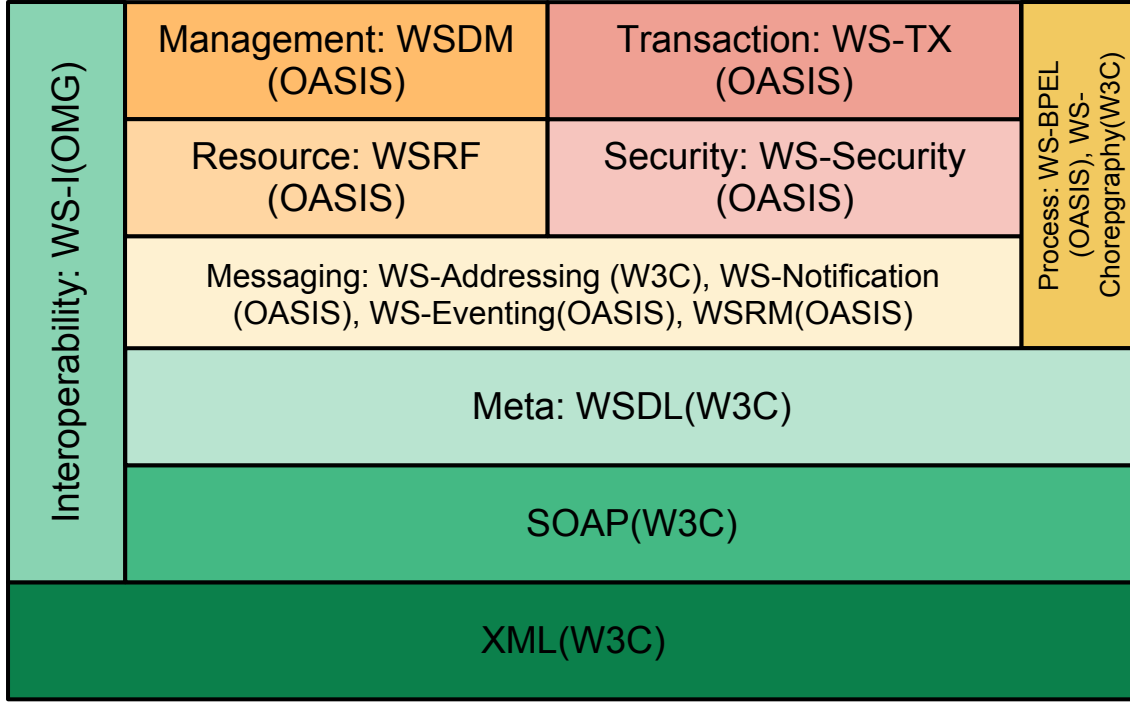


Figure 2.2: The web service specifications and their dependencies.

increased 1000 times¹⁶. By 2008, there were already 1 trillion unique URLs¹⁷. More than 2 billion people are web users to date¹⁸. The Web is so far the largest distributed system.

The Web is “a network of information resources” [89]. A resource is the key abstraction of the Web [38]. A resource can be any piece of information that can be named. Fielding defined a resource as “a temporally varying membership function” [38].

$$r : \mathbf{T} \longrightarrow P(\mathbf{RE}), \quad (2.1)$$

where \mathbf{T} is the time, \mathbf{RE} is the set of *representations*, and $P(\mathbf{RE})$ is the power set of \mathbf{RE} . A resource is a mapping to a set of *representations*, or equivalently, a set of *identifiers*. At a certain time, a source is identified or represented by a *member* of a subset of \mathbf{RE} . The membership might not change for a static resource, and varies for a dynamic one. A resource identifier, in the form of a Uniform Resource Locator (URL), has hierarchies each level of

¹⁶A Little History of the World Wide Web, see <http://www.w3.org/History.html>.

¹⁷See <http://googleblog.blogspot.ca/2008/07/we-knew-web-was-big.html>

¹⁸See <http://www.internetworldstats.com/emarketing.htm>

which can be resolved by a corresponding naming authority. For example, the domain name can be resolved by a Domain Name System (DNS), and the path of a resource can be resolved by the server hosting the resource. A resource representation is the data to describe the state of the resource with metadata. The type of data is called a media type, and specified by the Multipurpose Internet Mail Extensions (MIME) standard [41].

I think a process view of resources is more accurate. The actual entity of a resource can be thought as a “feeling” process in Whitehead’s language [105]. Such a process results in a representation of the original “data” that is the piece of information of interest. The representation is not only determined by the original data, but also by the way that a specific subject feels it. Specifically, a web resource’s representation is decided by its URL, the access method used, the headers of the request, and content negotiations (if any).

According to Perry and Wolf [86], software architecture can be defined as a tuple of elements, constraints¹⁹, and rationale.

$$architecture = \{elements, constraints, rationale\} \quad (2.2)$$

The elements are abstractions of the components that contain data, process data, or connect components with each other. The constraints are the rules based on which the elements are configured and organized. The rationale is the reasons or motivations for the decisions of elements and constraints. There are always pros and cons for a design decision, and the rationale is a comprehensive evaluation of every aspect. The following subsections discuss the architecture of the Web.

2.7.1 Web architectural elements

Software architecture can be described by views [7]. Web architecture is shown in Figure 2.3 from a process perspective. This viewpoint describes the common elements and their organization. The elements shown in the figure include user agent, cache, DNS, HTTP connection, proxy, reverse proxy, and original server. It shows two scenarios. In the first one, the

¹⁹Perry and Wolf originally used the term **form** in their paper.

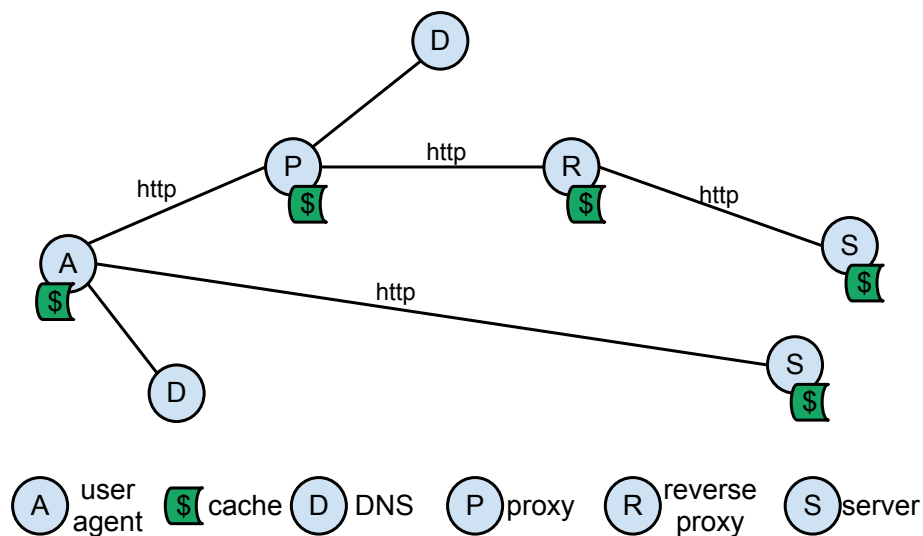


Figure 2.3: A process view of a web architecture.

user agent sends a request via the proxy, the proxy forwards the request to a reverse proxy resolved by a DNS, and the reverse proxy then passes the request to the origin server hosting the resource. The proxy and reverse proxy are called *intermediaries*. During the process, if any intermediary has a valid cache for the request and the request indicates acceptance of cached response, a cached response will be sent back. In the second scenario, the user agent sends a request directly to the origin server resolved by a DNS.

2.7.2 Web architectural constraints and rationale

Client-server

Client-server is the most fundamental relationship between web elements. A web server is always in an active listening state, and ready to receive requests from web clients. The web clients can be distributed across networks, in a state of connected, or thinking, or offline. Not all the web elements are either client or server. The intermediaries need to play both roles. The client-server separation of concerns can still help abstract their design and programming into client parts and server parts. Web programmers often need to distinguish client-side code from server-side code when they are dealing with some data elements like Active Server Pages (ASP) and JavaServer Pages (JSP), which have code for both sides woven in one file.

With client-server, the client side and server side can evolve independently. Furthermore, it is more straightforward for the server to update a resource when needed, and then all clients can get the update. A drawback of client-server is that the server will be a performance and scalability bottleneck when numerous clients are consuming it at the same time.

Stateless communication

The basic Message Exchange Pattern (MEP) of the Web is the request-response pattern. A client might need to perform a sequence of request-response with or without interruptions in order to achieve a goal. Such a sequence is abstracted as a *session*. The requests at the latter part of a session often depend on the information contained in leading request-responses in order to continue the session. Such information can be maintained solely on the client side, or on the server side, or on both sides. The stateless communication constraint suggests that any request should carry *sufficient*²⁰ information for the server to understand the request and continue the session. Therefore, the communication is stateless such that there is *no necessary* state dependency between a request and others. This releases the server from keeping the state of sessions belonging to different clients. However, this does not conflict with the fact that a server should maintain shared state of a resource. Stateless communication increases the scalability of a web server, and also eases the recovery or change of a session.

In practice, cookies have been used for session states since the early days of the Netscape browser. A cookie is a piece of state information generated on an origin server and saved on a user agent. When a session state is required, the origin server sends a response with a **Set-Cookie** header directive. If the user agent enables cookies from the origin server, the cookie will be saved on the agent. The cookie will be attached on all further requests sent to the URLs that satisfied domain and path selections until its max age is reached[67].

There are three intrinsic problems with the usage of cookies. Firstly, the implicitness of cookies prevents a user from being aware of the setting of a cookie and the state information contained in a cookie. Therefore, it is impossible for a user to manage the cookies saved on her/his agents efficiently. Secondly, an origin server can set cookies on a user agent even

²⁰I use *sufficient* here instead of *necessary* used by Fielding in the sense that the information contained in a request assures the continuation of the session.

if the requested resource has nothing to do with a session. A user's browsing history can be silently revealed by an origin server that hosts resources, like images or ads, linked by resources on other origin servers. Thirdly, an origin server can arbitrarily set a cookie's max age to be much longer than a real session's lifetime. In this way, the origin server can get all the history of a user agent visiting it. Cookies are supported by most browsers, and have been used by many servers for various purposes, including tracking users²¹.

Cache

A cache stores a limited amount of replicated data temporarily in order to make future retrieval of these data faster. The HTTP protocol specified the capability, expiration and validation of cache. The cache-control-related meta information is specified by HTTP headers. A web cache can be deployed on user agents, origin servers, or intermediaries. Web caches can reduce the response latency by partially or completely avoiding the interactions between a user agent and an origin server. Therefore, user-perceived performance can be improved. Since the load on an origin server can be reduced by web caches, the scalability is also improved. When partial failure happens, a user agent can still get resource representation from a web cache, which benefits the system's reliability.

The usage of caches can easily introduce inconsistency between cached resource representations and those on an origin server. Although HTTP defines headers and mechanisms to expire or re-validate cached representations, inconsistency can always happen. It might even be impossible to keep consistent at any time if we want to maintain high availability of a resource on the Web [43]. In such cases, the strategy of *eventual consistency*²² [99] can be applied.

Layered system

The Web was designed to be a layered system. In order to cooperate with intermediaries, HTTP messages are designed to be transparent. Intermediaries can redirect, cache, check, and

²¹See <http://collusion.toolness.org/> for a demo.

²²Pat Helland might have been the first to coin this term. See <http://blogs.msdn.com/b/pathelland/archive/2007/05/15/memories-guesses-and-apologies.aspx>.

transform HTTP messages. A load balancing intermediary improves scalability. A firewall intermediary implements security. A cache intermediary improves scalability and reliability.

On the other side, intermediaries also brought issues. For example, intermediaries introduce extra latency to end-to-end message exchanges. Intermediaries also introduce more failure points, and make it more difficult to identify the source of partial failures. In order to overcome these drawbacks, intermediaries are normally light-weight, highly efficient and extra reliable.

HTTP and HTML

The HTTP protocol was specifically designed for the Web. It follows the constraints of client-server, stateless communication, cache, and layered system by nature. RPC works well with HTTP request-response Message Exchange Pattern (MEP), which makes some developers think that HTTP was developed as an RPC mechanism. The most significant difference between HTTP and RPC is that the former introduced a generic interface. All resources can be accessed via the same set of methods: `Get`, `Post`, `Put`, `Delete`, and `Patch` [33] for resource manipulation; `Options` and `Head` for metadata retrieval; and `Trace` for testing and diagnosis. A resource only needs to support a subset of the methods. The uniform interface makes it possible for web elements to evolve independently without breaking the interface compatibility, which is almost impossible for clients and servers in RPC [107].

Although request-response is the basic MEP, HTTP was designed to encourage agent-intermediary-server negotiations. A negotiation is a conversation in which two parties try to reach an agreement by a sequence of request-response exchanges. HTTP defined message headers, status codes, and mechanisms for content negotiation. Content negotiation aims to provide the best resource presentation to a user agent from either an origin server or an intermediary. A content negotiation can be driven by a user agent, an origin server, or an intermediary. Content negotiations improve the compatibility between clients, servers, and intermediaries, and therefore, bring better user experience.

Negotiation can happen during the transportation of an HTTP request message. When a request contains a relatively big body, it will be inefficient for the origin server to reject the request after parsing the body. The performance can be improved if a user agent sends the

headers first and waits till receiving a continuation signal from the server. The status code 100 was designed for such situations.

An HTTP message is composed of a start line, headers and a message body if necessary. The start line is a request line for request messages, or a status line for response messages. The start line and all message headers are transferred in plain text. The headers indicate meta-information about a message, including general headers, request or response headers, and entity headers. By interpreting the start line and headers, an intermediary should be able to get sufficient information for processing the message. In this sense, the message is *self-descriptive* for the intermediaries.

With no prior knowledge about a resource, the conversation between a user via an agent (client) and a resource normally starts from an unconditional **GET** of the resource. **GET** is both *safe* and *idempotent*. By safe, it means the request will not cause any significant change to the target resource. By idempotent, it means more than one identical request has the same side-effects as a single request. The safeness and idempotence of **GET** make a URL bookmark always a good place for a client to start an application. The server hosting the resource or an intermediary then sends back a representation of the resource. The representation contains the current state of the resource, and also controls such as links and forms for the client to trigger the state transfer of the application. Via those controls, the client can choose to retrieve representations of other resources or to manipulate resources. Note that a resource is always hidden behind the interfaces. After an application is initiated, every representation sent back from the server contains the current application state. Therefore, the client has all the application state. And at the same time, the server should maintain the shared state. A shared state is the information that both the server and the client must keep in order to make a state transfer.

The start line and header lines in an HTTP message are all plain texts, which is not efficient from a transportation point of view. It was designed in order to improve a message's transparency to all connectors and to gain performance by incremental processing. The HTML is also designed in a similar way for incremental rendering by a user agent [89], which results in better user-perceived performance.

Although the HTTP standard specified message formats unambiguously, it still recom-

mended tolerance during message parsing. For example, accept more than one whitespace or tab character at the places where only a single space is required; and use a linefeed as the line terminator though the carriage return linefeed was for that in the specification. Similarly, the HTML specification suggested that a user agent renders only the recognized valid HTML elements and attributes. The design rationale is to achieve the maximum compatibility among applications with the existence of developer mistakes and other faults. It is still a good practice for developers to validate the HTML pages against the validators in order to avoid faults that are not warned by browsers during testing.

Code-on-demand

The representation of a resource is often a combination of text, hyperlinks, images, graphics, and media streams. The Web takes advantage of a more powerful representation element — mobile code. A piece of mobile code is a program that a user agent retrieves from an origin server as a part of resource representation. The user agent then loads the code and executes it on demand on the client side. The most popular form of such executable code is JavaScript, whose kin includes Java Applet, Flash, and Silverlight. A user agent needs to have built-in support or extended plug-ins installed to execute mobile code.

The Code-On-Demand (COD) enables rich user interfaces beyond traditional hypermedia. A representation can be modified instantly according to user local context and real-time inputs, which can be used to mimic many characteristics of desktop application software. Furthermore, by using the XMLHttpRequest (XHR) API and asynchronous programming style, an agent can retrieve representations of interesting resources either on demand or proactively without blocking the interaction between a user and the current rendered representation. Such an approach based on JavaScript is called Asynchronous JavaScript And XML (AJAX). COD augmented the Web to a higher order of dynamics, and can transform hypermedia representations into *computational* representations.

The power of COD raised some security issues for the Web. One of the most well-known problems is cross-site scripting: a user agent executes malicious mobile code in a resource representation that contains sensitive information such as user identity. A piece of mobile code might be so computation-intensive that a user agent can be slowed down when executing

HTTPS(IETF)	HTTP authentication (IETF)	JavaScript(ECMA)	
HTTP(IETF)		DOM(W3C)	CSS(W3C)
URL(IETF)	MIME(IETF)	HTML(W3C)	

Figure 2.4: The stack of web standards.

it. Embedding a lot of such mobile code in a representation can result in a non-responsive, or even crashing, user agent.

Java Applet was used as a COD technology as early as JavaScript was. However, JavaScript has been much widely used on the Web than Java Applets. Besides the issues of Java's runtime reliability and version incompatibility, a significant difference between those two is that Java Applet is byte-code and JavaScript is plain text. A Java Applet is not transparent to user agents and intermediaries, and therefore, it is difficult to retrieve information from it automatically. There is no way to index or inspect a Java Applet, while it is not a problem for JavaScript and hypertext. The lack of transparency could also contribute to the failing of Flash²³.

2.7.3 Web standards and specifications

The standard stack for the Web is much smaller than that of web services. Basically, there are two independent groups of specifications: application and representation. As shown in Figure 2.4, the left side is for applications, and the right side is for representation. The Multipurpose Internet Mail Extensions (MIME) is a standard that was originally developed for email application. The Hypertext Transfer Protocol Secure (HTTPS) and HTTP authentication are two widely implemented specifications for web security.

²³Flash will not be supported by HTML5 [54].

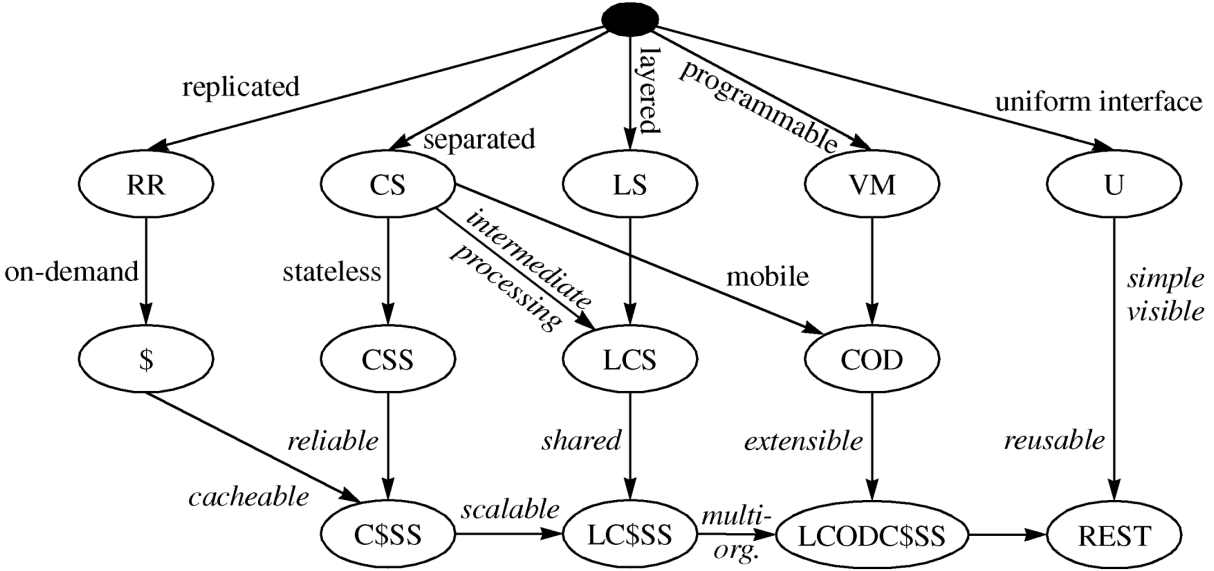


Figure 2.5: The derivation of REST [38].

2.8 REST and RESTful services

Representational State Transfer (REST) was first described in Fielding’s PhD thesis [38]. It was derived from a set of network-based architecture styles. A software architectural style is a collection of constraints that is abstracted from individual architectures sharing the same characteristics, or derived from other architectural styles. An architectural style can be used to guide the design of a specific architecture. The relationship between an architectural style and an architecture instance is analogous to that of a design pattern [42] and a design. The derivation of REST from related architectural styles is illustrated in Figure 2.5.

The derivation started from the null at the very top. The first style added was **RR (replicated repository)**. The idea of **\$ (cache)** was directly derived from **RR** with on-demand capability added. The separation of concerns principle resulted in **CS (client-server)**, which then became client-stateless-server with the stateless communication constraint. The combination of **CS** and **LS (layered system)** yielded **LCS (layered client-server)** with intermediaries. The programmable **Virtual Machine (vm)** enabled the execution of mobile code transferred from a server to a client, which was **Code-On-Demand (cod)**. The combination of all the above was **LCODC\$\$\$ (layered–code-on-demand–client-cache-stateless-server)**. REST was finally derived by adding the **U (uniform interface)** style.

The style of uniform interface is essential to REST. It is composed of the following five constraints²⁴.

All important resources are identified by one resource identifier mechanism.

The naming complexity of a system is reduced in this way.

Access methods are the same for all resources. In this way, original servers, clients, and intermediaries can be programmed independently without the risk of being incompatible on the interface level. A specific resource can choose to implement a subset of the access methods according to its requirements.

Resources are manipulated by exchanging representations. A resource is always hidden behind the interfaces. The resource is secure since there is no way for a client to interpret how a well-designed resource is programmed. On the other side, the representation removes the programming coupling between a server and its clients.

Representations are carried by self-descriptive messages. Meta-data included in the message is encoded in an easy-to-interpret way, and contains sufficient information to be processed by a receiver.

Hypermedia works as the engine of application state. Hypermedia or hypertext²⁵ is a non-sequential composition of information and controls. A hypermedia representation sent from a server to a client indicates the current application state, and also the possible state transfers that the client can trigger by using the controls.

A RESTful service is a service whose architectural design is constrained by REST. In this thesis, the term “RESTful web service” refers to a service that utilizes web standards such as URL and HTTP as major elements. A RESTful design only needs to comply with a subset of REST constraints shown in Figure 2.5. A comparison between RESTful services and SOAP-based web services is shown in Table 2.1.

²⁴Fielding listed four constraints — the first, third, fourth and fifth items — originally in his thesis. He added the second item later [37].

²⁵Fielding used the term of “hypertext” instead of hypermedia in some occasions [37].

Table 2.1: Comparison of RESTful web services and SOAP-based web services.

Aspect	SOAP-based web services	RESTful services
Key abstraction	Object/method	Resource
client-server	default	default
stateless	no preference	recommend statelessness
cache	difficult to make use of cache	cache-friendly
layered system	hard to work with intermediaries (see U4 for reasons)	intermediary-friendly
code-on-demand	depends on implementation	depends on implementation
U1: uniform identifier	need identify a method by both URL and messages	default
U2: same methods set	use only POST, and expose arbitrary methods for invoking	default
U3: manipulation through representations	use serialized parameters/objects	encourage different representations of the same resource
U4: self-descriptive messages	suggest to put lots of metadata in SOAP header, and SOAP header entries are not standard	encourage self-descriptive messages, for example, by using HTTP
U5: Hypermedia As The Engine Of Application State (HATEOAS)	have no notion of hypermedia, focus mainly on data, lack of controls	HATEOAS-friendly on the basis of U1-U4
Code generation	default	implementation-dependent

The lack of a code generation mechanism is often a criticism of RESTful services. However, code generation support for SOAP-based web services has its cost, like the tight-coupling between a service and its programming language.

2.9 Interesting properties of web services

As mentioned in Chapter 1, web service technologies were developed to address the compatibility issues of RPC, but failed to fix them. The specification family of WS-I was developed to address this. Besides compatibility, there are other non-functional requirements that are of interest for large-scale distributed applications, such as performance, scalability, reliability, and modifiability.

2.9.1 Performance

For distributed applications of client-server style, performance can be evaluated from two perspectives. From the server's perspective, performance can be measured by the number of requests accomplished in a unit time, or throughput. From a client's perspective, performance is perceived by the time spent to wait for the response. Shorter response time is better performance.

Little's law [70] indicates that

$$N = XR \quad (2.3)$$

if the system is stable²⁶, where X is the throughput, R is the residence time, and N is the number of requests being processed on the server. The maximum that N can reach is the server's *capacity*. The minimum that R can be is a request task's *demand*.

The most natural and straightforward way to decrease the residence time is to use fast processing units. A faster processor can finish a task in shorter time with the identical number of required processor cycles. However, a more interesting question is how to achieve the same by software improvement.

²⁶Strictly, the corresponding stochastic process needs to be stationary.

Response time is the cumulative result of the delays, including the round trip time of request-response and residence time on the server. Basically, there are two approaches to reducing response time:

1. Get a cached response from an intermediary instead of from the origin server.
2. Decrease the residence time by mapping a request to sub tasks and then reducing to get the result.

The SOAP specification did not describe any caching approach. Even worse, the default HTTP method for SOAP requests was `POST` as specified in SOAP 1.1, which made it difficult to take advantage of HTTP's caching capability to reduce the response time of SOAP message exchanges.

2.9.2 Scalability

According to Little's law, a server's throughput can be improved by increasing the number of requests N while keeping the residence time R unaffected. To achieve this, more resources need to be used for processing the additional requests. The resources can be local resources – like Central Processing Unit (CPU) cores, or distributed units – like nodes in a cluster. Such a scale-up often implies two changes in scale: the number of simultaneous requests or clients served, and the number of computational units in use.

Scalability is a system's ability to sustain an acceptable service level for an increasing number of concurrent requests [76, 57, 18]. The service level should be measured from the client side because there is no linear correlation between throughput and response time. With an increased workload, the most common approach is to utilize more computational or operational resources on the server side. Since the amount of resources that can be located in a box is always physically and economically constrained, clustering of distributed nodes with load balancing is more widely applied. Figure 2.6 shows that a web service, programmed in Axis²⁷ and deployed on Tomcat, scales up with the number of threads and stops scaling when all available resources are used up in the system. The parallel workload was generated

²⁷See <http://axis.apache.org/axis2/java/core/>

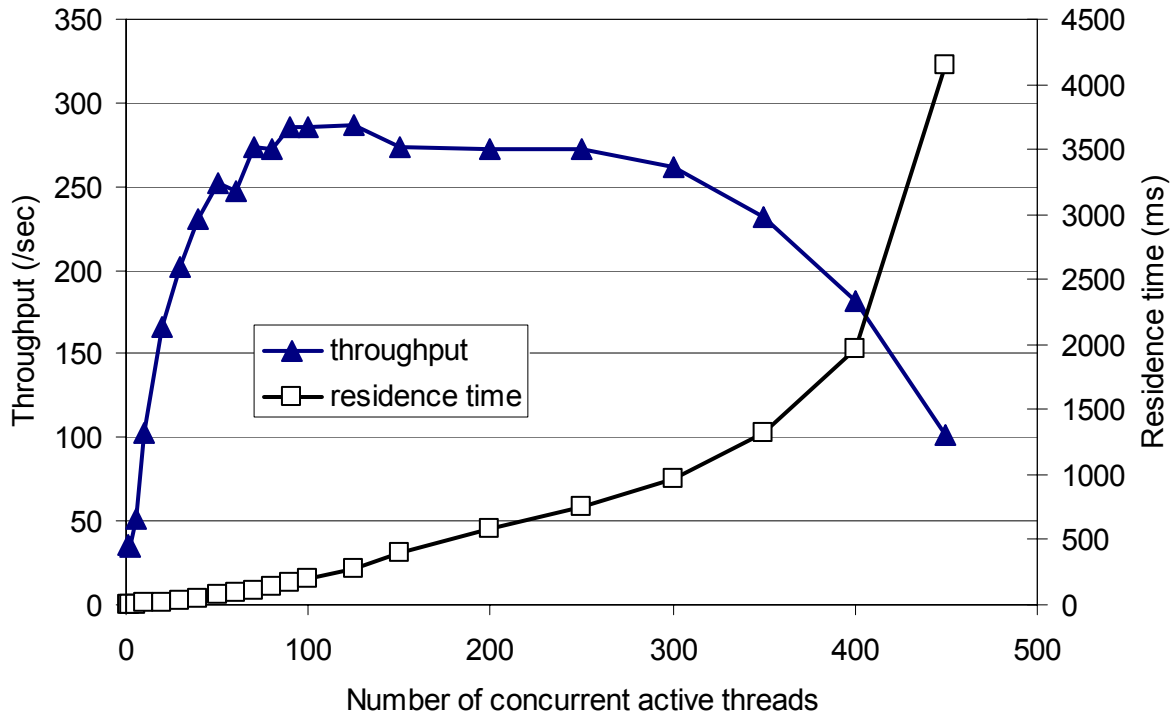


Figure 2.6: The throughput and residence time of a web service versus the number of concurrent active threads.

by JMeter²⁸ running on another machine located in the same WAN. As shown, the average response time increases almost linearly with the number of concurrent active threads in the system before the system thrashes. Similar results have been observed on .NET applications [51] and Java EE (Enterprise Edition) applications [97, 49]. More details of the measurement and models of scalability are discussed in Appendix A.

The server side system should be a layered structure for the purpose of clustering. For better scalability, a message should be efficiently parsed and routed through layers. HTTP messages were designed for pipe-and-filter style processing in a layered architecture. This advantage of HTTP was not used by SOAP that treats HTTP as a transport mechanism. Although pipeline of SOAP processing is also possible, it can never reach the same efficiency as HTTP.

A networked system is more than just the server, and it includes the clients and intermediaries. The approaches to scalability by utilizing the resources on clients and intermediaries are superior to those focusing on only the server side. The more clients a system has, the more

²⁸See <http://jmeter.apache.org/>

“free” resources these approaches can leverage. Caching is such an approach contributing a lot to the Web’s scalability.

Caches on intermediaries and clients push replicated resource representations closer to the clients than the origin server. Since the popularity of web requests follows a Zipf-like distribution [24], caches can be efficient even if only the top portion of requests are cached due to available space. As we discussed, SOAP intrinsically lacks caching capability.

2.10 Lessons of distributed computing

Peter Deutsch pointed out seven wrong assumptions made by distributed application developers when he worked at SUN, and one more item was added to the list by James Gosling [91]. This list of assumptions is known as “Fallacies of Distributed Computing”.

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn’t change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

These assumptions are fallacies because the following facts:

- The developers tend to program distributed applications in the same way as they do for local applications.
- The developers program and test distributed applications in lab environments, and assume the real network environments are the same.

When frameworks/tools allow developers to program a distributed application like a local one, the developers will be misled. This was exactly what happens with RPC. In *A Note on Distributed Computing* [100], Jim Waldo and his colleagues discussed four major differences between local and distributed computing — **latency**, **memory access**, **partial failure**, and **concurrency**. The first two are obvious, while the latter two are not. This list resulted from lessons learned from the NFS project where RPC was created. NFS is a re-implementation of a non-distributed API for a distributed application. Partial failure was one of the major reasons contributing to NFS’s reliability and robustness problem. However, a local API has nothing do to with network partial failures. Therefore, the “reliability of NFS cannot be changed without a change to that interface, a change that will reflect the distributed nature of the application”.

The technology of CORBA, one successor of the RPC approach, did not avoid its fate of falling after rising as a popular distributed technology [52]. One of the technical reasons is its difficulties in working with proxy/firewall and partial failures. Although CORBA introduced a standard IDL, programmers still largely depended on the code generators. The compatibility between implementations of the same interface was determined by the code generators for the stubs.

SOAP technically improved RPC and CORBA in many ways. However, a key characteristic has not been changed from RPC to CORBA to SOAP — the convenience for the developers to program a distributed application as a local one [98], and therefore all the flaws brought by this convenience remain. For example, the root reason for incompatibility problems between different implementations of the same WSDL document is the incompatibility of the `wSDL2code` tools. The developers never read the WSDL document, and only read the generated code stubs.

The misleading way of communication between developers and computers, and furthermore, the ignorance of communication between developers, in my opinion, are also common mistakes in distributed computing. The *Fallacies of Distributed Computing* can be expanded by adding two more items.

9. Tools and libraries remove network complexity.

10. Programmers communicate in code.

2.11 Summary

This chapter reviewed the fundamental technologies of web services. The compatibility, performance, and scalability of web services are discussed and the reasons contributing to the issues are discussed. These issues were better addressed by the Web that implements the REST architectural style. Web services and RESTful services were compared from various technical aspects. Their differences became more clear when reexamining the lessons learned in distributed computing.

The next chapter reviews the technologies for web service composition, their essentials and drawbacks. The properties of performance, scalability, reliability, and modifiability of web service compositions are discussed.

CHAPTER 3

WEB SERVICE COMPOSITION

We can't solve problems by using the same kind of thinking we used when we created them.

—Albert Einstein

Composability is one of the most fundamental principles of Service-Oriented Architecture (SOA) [34]. High-order services can be developed by composing other services, and the service ecosystem can evolve from simplicity to sophistication in this way. Therefore, composability should be considered when a service is initially designed, in much the same way as the reusability of procedures in procedural programming or that of objects in object-oriented programming.

Orchestration and choreography are two orthogonal approaches to compositions. Orchestration depends on a conductor-like central service controlling the workflow execution, while choreography assumes consensus on the sequence of actions and interactions among partner services. Obviously, a choreography instance is more difficult to implement than an orchestration instance of the same capability. Each partner service in an orchestration can have its own authority regarding its service contract, while the partner services in a choreography must have a common agreement on the contract.

3.1 A case study

In order to make the discussion about service composition easy to understand, the following application is used as a case study in the rest of this thesis. Science Studio¹ and ANISE² are

¹See <http://sciencestudioproject.com/about.php>

²See <http://www.anise-project.com/about.php>

two joint projects that expose experimental laboratories in the Canadian Light Source (CLS)³ and high-performance computing capability at the University of Western Ontario (UWO)⁴ as services and, furthermore, enable near real-time collaboration for data collection and data processing [71]. Researchers around the world are able to watch (as a team member) or control (as an experimenter) live sessions of data collection and processing. The precious beamline time can be effectively utilized by researches when they can check processing result during or right after an experiment and adjust their experiment plan on time. The processing time can be reduced by a factor of 10 to 100 by the high-performance processing capability.

A typical scenario of X-Ray Diffraction (XRD) scan and near real-time data processing is as follows.

1. An experimenter starts a scan at the VESPERs (Very powerful Elemental and Structural Probe Employing Radiation from a Synchrotron) beamline of CLS. A scan is composed of a map of scan points.
2. The scan process triggers the Charge-Coupled Device (CCD) detector to collect an image on each scan point. The image is transferred from the Science Studio server at CLS to a server at UWO.
3. The image is then processed by the high-performance computing facilities at UWO.
4. When an image is processed, the result is transferred back to the CLS Science Studio server, and presented to the experimenter.

Other requirements include:

- An experimenter or team member shall be able to view the process of image collection and processing at any time, and also access available raw data, processing configurations, and processing results.
- An experimenter shall be able to pause/resume/cancel a running collection and processing.

³See <http://www.lightsource.ca>

⁴Although it is now commonly referred as Western or Western University, the University of Western Ontario and the acronym UWO will continue to be used. See <http://communications.uwo.ca/brandnew/faq.html>.

- An experimenter shall be able to start a new processing of an existing data set with a processing configuration.

3.2 Web service orchestration

A web service orchestration is a service composition approach such that a central service is responsible for conducting the predefined sequence of interactions with client and partner services in order to fulfill the client's request. The conductor service encapsulates the details of the orchestration, and exposes a standard service interface to its clients. At the same time, the conductor service is also a client for the partner services in the orchestration.

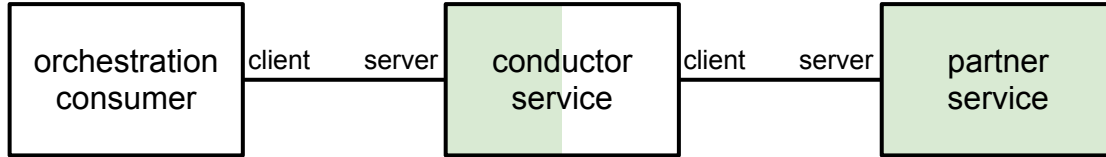


Figure 3.1: Two roles of conductor service in a service orchestration.

A web service orchestration is normally designed from the bottom up. That is, partner services are always designed first. Even if not implemented, their interfaces should be available when an orchestration is designed. A representation of service orchestration is the software artifact that describes the flow of interactions between the conductor service with its client(s) and partner service(s), including how incoming messages are parsed and processed and how outgoing messages are composed. An orchestration representation can be used as design document for developers and an executable program for orchestration engines. Web Services Business Process Execution Language (WSBPEL) is the most widely adopted approach to web service orchestration description.

If the scenario described in Section 3.1 is designed as a service orchestration, the basic partner services to compose the orchestration are described in Listing 3.2 to 3.5. The service endpoint interfaces are described in the same way as those in Web Service Description Language (WSDL) 1.1 [30], except for the notation. The notation in Listing 3.1 is defined based on augmented Backus-Naur Form (BNF) [31].

Listing 3.1: A BNF notation for describing service endpoint interface.

```
endpoint-interface  = "interface" [service-name "::"] endpoint-  
    name "{"  
                    message-exchange  
                    "}"  
message-exchange    = one-way  
                    | request-response  
                    | solicit-response  
                    | notification  
one-way             = input  
request-response    = input "," [CRLF]  
                    output  
                    ["," [CRLF] fault]  
solicit-response    = output "," [CRLF]  
                    input  
                    ["," [CRLF] fault]  
notification        = output  
                    ["," [CRLF] fault]  
input               = "input" ":" tuple  
output              = "output" ":" tuple  
fault               = "fault" ":" tuple  
tuple               = "(" #parameter ")" ; a list of parameters
```

Listing 3.2: The interface of scan service.

```
name      : cls_scan_service  
owner     : CLS  
capacity  : starts an XRD scan with predefined configuration;  
           generates scan data in CLS file system.  
interfaces :  
interface create_scan {  
    input:(scan_configuration),  
    output:(scan_id)  
}  
interface start_scan {  
    input:(scan_id)  
}
```

Listing 3.3: The interface of CLS data service.

```
name      : cls_data_service
owner     : CLS
capacity  : provides access to data in CLS file system.
interfaces :
interface list {
    input:(scan_id),
    output:(list_of_image_id)
}
interface read {
    input:(scan_id,image_id),
    output:(image)
}
```

Listing 3.4: The interface of UWO data service.

```
name      : uwo_data_service
owner     : UWO
capacity  : provides access to data in UWO file system.
interfaces :
interface list {
    input:(processing_id),
    output:(list_of_result_id)
}
interface read {
    input:(processing_id,result_id),
    output:(result)
}
```

Listing 3.5: The interface of UWO processing service.

```
name      : uwo_processing_service
owner     : UWO
capacity  : processes raw data with a given configuration;
           generates processing result in UWO file system.
interfaces :
interface create_processing {
    input:(processing_configuration),
    output:(processing_id)
}
interface process {
    input:(processing_id,image),
    output:(result_id)
}
```

3.2.1 WSBPEL

The language

Web Services Business Process Execution Language (WSBPEL) is an XML-based language to specify business process behaviour using web services [81]. WSBPEL is the most widely used description language for web service orchestration. The services interacting with the process are called partners in WSBPEL. A WSBPEL document describes the message exchanges between a process and its partners, including messaging interfaces (**portType**), message formats (**message**), and Message Exchange Patterns (MEPs) (**partnerLinkType**). A WSBPEL document also describes the business logic for processing messages and conducting message exchanges. The business logic part of a WSBPEL document is composed of scopes and activities. A scope provides context for inside activities. Table 3.1 lists major basic activities and structure activities defined in WSBPEL version 2.0. Note that the WSBPEL specification does not provide graphical notations to describe business processes, although some WSBPEL designers like Oracle WSBPEL process manager and Eclipse WSBPEL editor provide such notations. In practice, WSBPEL documents are normally developed with the help of an IDE. Developers hardly touch the XML directly due to its complexity and poor legibility for human. WSBPEL documents are transformed to executable code by WSBPEL engines for testing and deployment. In this proposal, I use WSBPEL pseudocode to describe processes for the sake of readability. Although it cannot cover all the details available in the corresponding WSBPEL version, a pseudocode description provides a clear high-level view of a process' structure and behaviour. Table 3.1 lists the activities specified WSBPEL, their semantics, and corresponding BNF notations.

A WSBPEL example

The XRD scenario can be implemented by a service orchestration based on the partner services described in Listing 3.2 to 3.5. Listing 3.6 shows the BPEL description of such an implementation.

Table 3.1: WSBPEL Activities and Corresponding BNF notations

Type	Notation	Semantic	BNF notation
Basic activities	<invoke>	Invoke an operation of a partner service. The MEP can be either in-out or in-only.	invoke = "invoke" [service "::"] endpoint "{" one-way request-response "}"
	<receive>	Receive a message from a partner.	receive = "receive" [service "::"] endpoint "{" notification "}"
	<reply>	Reply a message to the client or a partner service.	reply = "reply" [service "::"] endpoint "{" one-way "}"
	<assign>	Update a variable.	assign = target ":@" source
	<wait>	Delay for a period or until a deadline.	wait = "waitFor" ":" "(" ms ")" "waitUntil" ":" "(" time ")"
	<throw>	Signal an internal fault.	throw = "throw" ":" "(" #fault ")"
	<exit>	End the execution of the instance.	exit = "exit"
	<sequence>	The inside activities are executed sequentially.	sequence = "sequence" ":" "{" #activity "}"
	<flow>	Concurrent execution of activities.	flow = "flow" ":" "{" #activity "}"
	<if>	Condition structure. It can be combined with <elseif> and <else>.	if = "if" "(" condition ")" ":" "{" activity "}" #[elseif "(" condition ")" ":" "{" activity "}"] [else ":" "{" activity "}"]
Structured activities	<while>	Repeat the execution of inside activity while condition is true.	while = "while" "(" condition ")" ":" "{" activity "}"
	<forEach>	Execute inside activity for each item in a collection in parallel or not.	forEach = "forEach" item "in" collection ["parallel"] ":" "{" activity "}"
	<pick>	Execute an activity on one messaging or timing event in a set of them.	pick = "pick" "(" #event ")" ":" "{" activity "}"
<pre> process = "process" process-name ":" "{" activity "}" activity = basic-activity structural-activity basic-activity = invoke receiver reply assign wait throw exit structural-activity = sequence flow if while forEach pick </pre>			

Listing 3.6: A service orchestration implementation of the XRD scenario in WSBPEL.

```
process xrd_scan_processing : {
  sequence : {
    receive client::scan{
      output:(scan_configuration, processing_configuration)
    },
    invoke cls_scan_service::create_scan {
      input:(scan_configuration),
      output:(scan_id)
    },
    invoke uwo_processing_service::create_processing{
      input:(processing_configuration),
      output:(processing_id)
    },
    reply client::notify{
      input:(scan_id, processing_id)
    },
    invoke cls_scan_service::start_scan{
      input:(scan_id)
    },
    flow : {
      sequence : {
        finished_image_list := [],
        new_image_list := [],
        finished := false,
        while (!finished) {
          invoke cls_data_service{
            input:(scan_id)
            output:(list_image_id)
          },
          new_image_list := list_image_id -
            finished_image_list,
          if (new_image_list.length > 0) {
            foreach image_id in new_image_list parallel
            {
              reply client::notify{
                input:(scan_id, image_id)
              }
              invoke cls_data_service::read{
                input:(scan_id, image_id),
                output:(image)
              },
              invoke uwo_processing_service::process{
                input:(processing_id, image),
                output:(result_id)
              }
            }
          }
        }
      }
    }
  }
}
```


3.2.2 Essentials of WSBPEL

Although it might be described in various ways, a web service orchestration can always be characterized by two essential aspects. The message exchanges are its behaviours observed from outside; and the concurrency-oriented control flow is its internal structure. In WSBPEL, `invoke`, `receive`, and `reply` are elements of message exchanges, and others are elements of control flow.

Message exchange

An orchestration provides and accomplishes its service capacity through message exchanges with clients and partner services. The message exchanges always happen in certain patterns that reflect the agreed behaviour contract between two endpoints. For web services, such a pattern is called an MEP.

Most SOAP-based web services rely on HTTP as the transportation protocol. It follows that request-response is naturally the dominant MEP for web services [46]. All messages of request-response MEP must appear in pair and in proper sequence. Request-response works perfectly for cases when application state can be easily encapsulated in a message pair, and the time span between two messages is relatively short. However, request-response is quite constraining for describing all the message exchanges in an application.

WSDL 1.1 described four MEPs: one-way, request-response, solicit-response, and notification [30]. The details of these MEPs were described in Listing 3.1. When HTTP is used for message transportation, all these MEPs need to be implemented by request-response.

Publish-subscribe is a widely used message exchange pattern in event-driven systems [58]. An endpoint gets notification for subscribed updates without issuing extra requests in publish-subscribe pattern, which reduces the latency. Publish-subscribe can be implemented by a combination of an initial request-response MEP and following notification MEPs.

Concurrency-oriented control flow

A control flow, or flow of control, is the order of executions of a group of activities. The internal of an orchestration is a control flow that determines the sequence of message ex-

changes and message processing. Most elements of the control flow for service orchestrations, for example, `if`, `while`, and `forEach` in WSBPEL as shown in Table 3.1, are quite similar to those in traditional structured programming.

A significant different between the control flows in service orchestrations and those in structured programming is the native support of concurrency. In WSBPEL, `flow` and `pick` are used to describe explicit concurrent behaviours. The `flow` activity denotes the concurrent executions of inside activities. The `pick` activity denotes the concurrent arriving of events. The `forEach` activity can be configured to be parallel. The `invoke`, `receive`, and `reply` activities describe implicit concurrent behaviours, i.e. concurrent message exchanges. Concurrent activity execution and concurrent message exchanges imply high concurrency level that an orchestration engine needs to handle.

3.2.3 Technical drawbacks of WSBPEL

As discussed in Section 3.2.2, the concurrency-oriented control flow is one of the essentials of WSBPEL. The WSBPEL syntax for control flow was designed based on C style. The basic control flow can be easily mapped to those of structured programming languages by the engine. A flow can be mapped to a process or a thread of the corresponding languages. The concurrency part can be implemented by threading for languages like C and Java. However, multi-threading programming needs a lot of careful engineering in order to avoid the pitfalls of concurrent programming — deadlock and starvation.

The execution of a WSBPEL process is carried out by the engine that it was deployed on. A WSBPEL developer is free to design an orchestration complying with the specification supported by the engine no matter how complex its concurrency structure is. An engine will still be able to compile such an orchestration and execute it correctly. It is inevitable for an engine to sacrifice performance and scalability in such cases. A designer still requires effort to configure the details of an orchestration, for example, the correlations between incoming messages and orchestration instances.

Although WSBPEL was designed as a specification language rather than a programming language, it includes some details that should be kept in a programming language. For example, the `<assign>` activity seems more complicated than the assignment in programming

languages. The type system of WSBPEL relies on the proper interpretations of the WSDL documents of its partner services. A single `<assign>` operation often needs the compatibility of three pairs of `wsdl2code` and `code2wsdl` tools — two for partner services and one for the orchestration itself. The compatibility of WSBPEL orchestrations is even worse than that of normal Web Services.

3.2.4 Other service orchestration approaches

Windows workflow foundation

A workflow describes the series of activities required in order to achieve a goal. It specifies the person or group of persons to perform the activities and the time and logic dependency among them. In many cases, the term “workflow” has little semantic difference from “orchestration” or “process”. A subtle difference between workflow and orchestration is that the former often includes human activities. Although human activities are not explicitly specified in the original WSBPEL specification, BPEL4People was developed for this purpose [66].

Windows Workflow Foundation (WF), a component of the .Net framework, provides a namespace, an in-process workflow engine, and a design environment within Visual Studio for developing workflows interacting with humans and applications [80, 65]. Workflows can be programmed by markup, code, or combination of code and markup. Markup and code are equivalent in development because they are based on the same WF APIs. Extensible Application Markup Language (XAML) is the Microsoft native markup language for WF. Note that an add-on of WF can import and export orchestration descriptions in WSBPEL⁵, and this add-on is not officially supported by Microsoft.

BPMN

The Business Process Model and Notation (BPMN) specifies a set of graphical notations to describe business processes. BPMN was originally developed by the Business Process Management Initiative (BPMI)⁶, which joined OMG in 2005. BPMN aimed to provide understandable

⁵See <http://www.microsoft.com/downloads/details.aspx?FamilyID=6D0DAF00-F689-4E61-88E6-CBE6F668E6A3&displaylang=en> .

⁶See <http://www.bpmi.org/> .

notations for all business process users, including business analysts and technical developers. The business partners in a process are called participants in BPMN. The notations in BPMN can be classified into four categories: flow objects, connecting objects, swimlanes, and artifacts. Flow objects are the nodes representing events, activities, and gateways. Connecting objects are the connectors to link flow objects together. A swimlane denotes the ownership or partition that a group of objects belong to. Artifacts are extra information about the process like data and annotations. The design of BPMN is influenced by state transition diagrams [50] and Petri nets [87]. Table 3.2 lists major notations defined in BPMN version 1.1.

BPMN representation is more intuitive than WSBPEL representation. The graphical notations of BPMN constrains its ability to describe details like MEPs and message schemata. These tasks are no problems for text-based notations like WSBPEL. It is possible to convert an orchestration's description from BPMN to WSBPEL [82]. However, the conversion is difficult to produce executable WSBPEL descriptions.








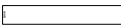


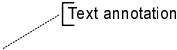
3.3 Web service choreography

Service composition can also be achieved by service choreography. The fundamental difference between a service choreography and a service orchestration is that the former has no centralized point that controls how the interactions among the participants are performed. The participants of a choreography agree on a predefined protocol that specifies their interactions during the choreography.

Service choreography does not constrain any behaviour inside of an individual service. Therefore, a participant service can change its internal implementation without breaking the compatibility with other participants of a choreography as long as the common protocol is followed. This implies a top-down development approach for service choreography, in which the message exchange behaviour of a participant should be developed or adjusted according to the agreed choreography shared by many parties. This often makes the time to market of a choreography longer than an orchestration.

A choreography cannot be owned by any participant because there is no single point of control. A choreography often has no clear rule about which participant should be responsible

Table 3.2: BPMN notations

	Name	Meaning	Notations
Flow objects	Event	The start, stop, or intermediate of a flow. An event can be triggered by message, timer, or others.	
	Activity	An activity can be a task, sub-process, or process.	
	Gateway	A gateway specifies the divergence or convergence of sequence flows	
Connecting objects	Sequence flow	A sequence flow object denotes the sequence of two connected activities. It can also combined with conditions.	
	Message flow	A message flow object denotes the flow of messages between two participants. Its end can be attached to an activity or a lane.	
	Association	An association denotes the relationship between an annotation or data object and an activity or connecting object.	
Swim Lanes	Pool	A pool is the container of a participant's activities in a process	
	Lane	A pool can be divided into several lanes that categorize activities.	
Artifacts	Data object	A data object specifies the data associated with a flow or activity.	
	Group	A group <i>visually</i> divides a set of activities from the rest in a diagram.	
	Text annotation	A text annotation provides more information for the object that it links to.	

for maintenance. The successful execution of a service choreography depends on a shared interest by all the participants. A choreography will be broken once a participant does not share the interest any longer. All these factors make choreographies more frangible than orchestrations since the partner services in an orchestration have their own ownership, responsibility, and interest.

3.3.1 WS-CDL

The Web Services Choreography Description Language (WS-CDL) is a W3C specification for describing choreographies of SOAP-based web services. WS-CDL is an XML-based language that provides a global view of the observed behaviour in terms of ordered message exchanges during P2P collaboration. Note that the development of WS-CDL had not been finished when its working group was closed in 2009⁷.

Since WS-CDL was not designed as an implementation language, it does not provide the capacity to replace an executable composition description language like WSBPEL. On the contrary, WS-CDL can be used as a complement of WSBPEL, for example, to describe the collaboration between two WSBPEL processes or that between a client and a WSBPEL process.

The language

A WS-CDL document contains a *root* choreography that is the only top-level choreography allowed in a WS-CDL document. A root choreography can contain enclosed choreographies that are defined either locally in the enclosing choreography or globally as a separate root choreography. In this way, a choreography can be the composition of other choreographies.

The basic building block of a choreography in WS-CDL is an **interaction**. An interaction is one or more message exchanges between two **participants**. The participants taking part in an interaction play different **roles**. The messages are exchanged through **channels** connecting roles. The sequence of interactions can be constrained by three **ordering structures**: **sequence**, **parallel**, and **choice**. These structures are quite similar to **sequence**, **flow**, and **pick** in WSBPEL respectively. Interaction(s) can be grouped into a container called **work**

⁷See <http://www.w3.org/2002/ws/chor/>

unit. A work unit provides a **guard**, a **repeat**, and a **block** constraint to its body.

Similar to WSBPEL, WS-CDL uses **activities** to describe the performed work. An activity can be an ordering structure, a work unit, or a basic activity. A basic activity is an interaction, a **perform** activity, an **assign** activity, a **silent action** activity, a **no action** activity, or a **finalize** activity. The perform activity is used to perform an enclosed choreography. The assign activity is for copying the value from a source variable to a target variable. The silent action is designed for an unobserved action, and no action for doing nothing. The finalize activity is used to denote the last action to perform before a choreography instance is finished. Such an action can be to confirm, or to cancel the result of the instance based on some condition. Listing 3.7 shows the augmented BNF notation of WS-CDL.

Listing 3.7: BNF notation for WS-CDL.

```

choreography = "choreography" choreography-name ":" "{"
               activity [, finalizer] [, exception] "}"
activity      = ordering-structure | work-unit | basic-activity
finalizer     = "finalizer" : "{" activity "}"
exception     = "exception" : "{" work-unit "}"

ordering-structure = sequence | parallel | choice
sequence          = "sequence" ":" "{" #activity "}"
parallel          = "parallel" ":" "{" #activity "}"
choice            = "choice"   ":" "{" #activity "}"

work-unit = "work-unit" unit-name ":" "{"
            ["guard"      ":" boolean | expression ","]
            ["repeat"     ":" boolean | expression ","]
            ["block"      ":" boolean | expression ","]
            activity
            "}"

basic-activity = interaction | perform | assign | silent | no |
                finalize
perform = "perform" "(" choreography ")"
assign  = target "!=" source
silent  = "silent" "(" role ")"
no      = "no" "(" role ")"
finalize = "finalize" "(" choreography-name [, finalizer] ")"

interaction = "interaction" interaction-name ":" "{"
              "channel"      ":" channel,
              "from"         ":" role,
              "to"           ":" role,

```

```

        "operation" ":" operation,
        #exchange
    }"

exchange = "exchange" exchange-name ":" "{"
    "send"      ":" tuple,
    "receive"   ":" tuple
}"

```

A WS-CDL example

We can use WS-CDL to describe the protocol between the `xrd_scan_processing` orchestration and the `uwo_processing_service` discussed in Section 3.2.1. Listing 3.8 shows the choreography described by the notation introduced in Listing 3.7.

Listing 3.8: A choreography for XRD image processing.

```

choreography xrd_processing_on_demand : {
    sequence : {
        interaction create_processing : {
            channel : processing_channel,
            from: processing_client,
            to: processing_service,
            operation: create_processing,
            exchange create_processing_instance : {
                send : ( processing_configuration ),
                receive : ( processing_id )
            }
        },
        workunit processing : {
            repeat : true,
            sequence : {
                silent (processing_client),
                interaction process : {
                    channel : processing_channel,
                    from: processing_client,
                    to: processing_service,
                    operation: process,
                    exchange process : {
                        send : ( processing_id, image ),
                        receive : ( result_id )
                    }
                }
            }
        }
    }
}

```

3.3.2 Essentials of WS-CDL

Conversation

Service choreography emphasizes conversations more than message exchanges. A conversation is a sequence of message exchanges between two or more participants without a third-party control. A conversation provides to all participants a context within which the messages are directly correlated. An ongoing conversation can yield various possible flows of message exchanges. Note that WS-CDL uses **interaction** for conversation, and only allows two participants in an interaction. The message exchanges in an interaction happen on the same channel.

Modular design

Different from service orchestration, a service choreography does not itself provide a web service interface. It just *describes* the possible interactions among service participants. Therefore, a choreography can be absolutely independent of other choreographies about the same set of web services. This makes it possible to design choreography in a fully modular fashion, as suggested by **perform** in WS-CDL.

The modular design of service choreography benefits its reusability. For micro choreographies can be used to construct macro choreographies without modification. If the implementation of a choreography can keep the modular nature of its design, then the reliability of a composition system can be improved. Most part of a macro choreography will still work properly when a micro choreography inside it fails.

3.3.3 Technical drawbacks of WS-CDL

WS-CDL working group started to work within w3C in 2003. The first version of the specification was published in 2004, and it became a w3C candidate recommendation in 2005. However, there were only very few attempts to implement it. pi4soa⁸ was a joint project by academics and industry. It delivered a graphical editor for WS-CDL as an Eclipse plug-in.

⁸See <http://sourceforge.net/projects/pi4soa/>

The developer can also use the tool to define scenarios of the choreography and simulate it. By scenario simulation, the choreography can be verified empirically. A similar implementation to the pi4soa Eclipse plug-in was programmed in Erlang [40]. So far, there has been no commercial implementation or support for WS-CDL. The W3C WS-CDL working group was closed in 2009 with major documents unfinished⁹.

One of the original goals of WS-CDL was to develop the language as a documentation tool for service choreography. WS-CDL, represented in XML, was designed on the basis of π -calculus [27], a formal process language. Theoretically, a WS-CDL choreography can be formally verified, and be transformed into π -calculus descriptions of end-point service behaviour [26]. Those descriptions, in turn, can help the development of individual services or service orchestrations. Unfortunately, neither XML nor π -calculus was a developer-legible language for documentation.

3.4 Interesting properties of web service compositions

The compatibility, performance and scalability of web services were discussed in Chapter 2. Most of the arguments are still applicable to web service compositions. A service composition, either an orchestration or a choreography, tends to have more complex structures and behaviour than normal web services. This also complicates the analysis of a composition's properties of performance, scalability, reliability and modifiability.

3.4.1 Performance

The performance of a composition service depends on the performance of all its partner services, its structure, and its workload characteristics. Although the overall structures of service orchestration and service choreography are quite different, they share some common flow structures. Table 3.3 shows the shared structures and their corresponding WSBPEL and WS-CDL notations.

⁹See <http://www.w3.org/2002/ws/chor/> and <http://www.w3.org/2002/ws/chor/edcopies/primer/primer.html>

Table 3.3: Common flow structures of WSBPEL and WS-CDL.

Structure	BPEL notation	CDL notation
sequence	sequence, while	sequence, repeat
parallel	flow, forEach	parallel
choice	pick, if	choice, guard

We can apply the method of **hierarchical modelling** [28, 69] to analyze the performance of service compositions with complex structure. The basic idea of hierarchical modelling is to treat a block of network in a model as a black box with input and output, and assume we know its performance measures like residence time. We then decompose the black box into more blocks recursively until we have the performance measures of all the components inside. This process is called **decomposition**. After decomposition, we can put the pieces together and have an approximate result of the whole model's performance. This second process is called **aggregation**.

Lots of work has been done on analyzing queueing network models by aggregation. Most approaches target closed networks. A service composition cannot be modelled as a closed network because the partner services are open to other consumers other than the clients and services of the composition. We can model a service choreography as an open network, where all the partner services are open to external workloads. A service orchestration can be modelled as a mixed network, where the client requests to the orchestration are within a network while partner services are open to external requests. The response time of an arbitrary web service is load-dependent and its distribution is so complicated that even the major service providers like Google App Engine¹⁰ and Amazon Web services¹¹ do not specify a guaranteed response time in their service level agreements. It is both technically and economically difficult to analyze the queueing network model for a service composition. However, the hierarchical modelling approach still can be used to perform approximate boundary analysis of service compositions. For example, the residence time of a parallel structure will have a lower boundary as the maximum of the lower boundaries of all branches, and an upper boundary as the maximum of the upper boundaries of all branches.

¹⁰See <http://code.google.com/appengine/sla.html>

¹¹See <http://aws.amazon.com/ec2-sla/>

The performance of partner services of an orchestration can be assumed to be independent of its workload, i.e. the number of orchestration requests. However, the performance of the conductor service depends on its workload. Furthermore, the performance of the whole orchestration depends on the workload. This is also a reason to model a service orchestration as a mixed network. The conductor service needs to perform multiple message exchanges with partner services and the client for every orchestration instance. Each message exchange will consume resources like network I/O, memory, CPU, and perhaps databases. High concurrency level and tight competition for resources often degrade the performance of the conductor service, and hence of the service orchestration as a whole.

For a long-run process, the clients normally want to be able to retrieve the progress of their requests from time to time. Such requests make the performance of central conductor service worse. For they increase the concurrency level and compete for the resources. Caching can improve the responsiveness by reducing the workload on an origin server.

3.4.2 Scalability

A service orchestration's scalability is largely decided by the conductor service. State and concurrency are the major factors affecting a conductor service's scalability. These two factors are often entangled with each other.

The states in a service orchestration can be grouped into three categories according to their lifetime. The states of first type have short lifetimes. Such states come to exist triggered by an inbound message and disappear when the incoming message is processed or a corresponding message is sent out. The states of second type have relatively long lifetimes. They can span a pair of or more of request-responses. The states of third type have the same lifetimes as an orchestration instance. We name the states instant states, conversational states and lifelong states according to their lifetimes.

In order to allow an orchestration instance to execute correctly and efficiently, states should be managed corresponding to their nature. Implementing a lifelong state as a conversational state will result in incorrectness of execution. On the other side, implementing all states as lifelong states will waste computation resource and make the execution inefficient.

A sequence of interactions with a server with state dependency among them is called a

session in web applications and web services [55]. A session is normally programmed as an object associated with a client on the server side in technologies like ASP, JSP and Servlets. Orchestration states can be implemented by these session technologies as well. There are two ways to achieve session clustering — session affinity and session sharing [29].

By session affinity, all requests belonging to a session are always routed to the same node on which the session was initialized. This requires a front end node to maintain a record of the associations between sessions and back end nodes. A back end node with live sessions needs to be available so that those sessions can be successfully finished. Obviously, sticky sessions will add extra load to front end node(s) and can also bring load imbalance among back end nodes. The dependency between a request and the corresponding pre-determined node will also degrade the reliability of the service and increase management complexity.

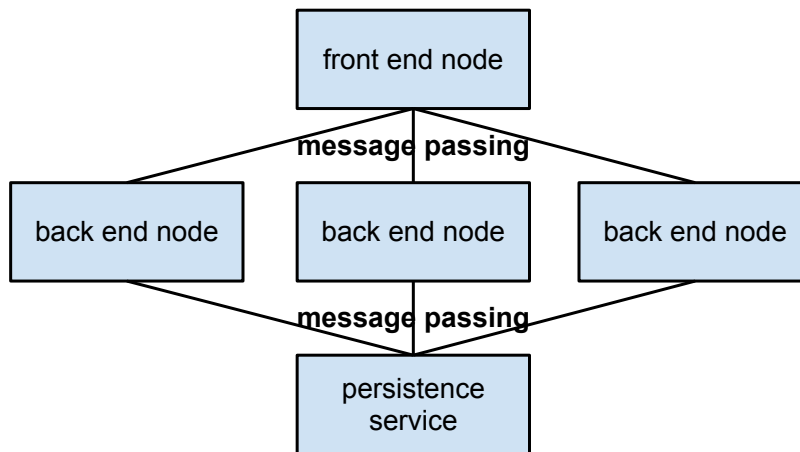
By session sharing, sessions are shared by all the nodes in a cluster, so that a request can be handled by any node. Session state can be shared in two ways: 1) Updated session states are maintained by a persistence service like a database, and later retrieved by any node needing that state. 2) Session states are maintained in the memory that is accessible to all nodes. These two approaches are shown in Figure 3.2.

The first approach is often criticized for moving the bottleneck from the application layer to the persistence layer. For normal relational database, many write operations for each orchestration instance will bring high latency to the database. That will result in longer residence time for all orchestration instances, and the capacity and performance gains brought by clustering can be neutralized. This problem can be partially fixed by putting the storage of the persistence service in memory like Redis¹².

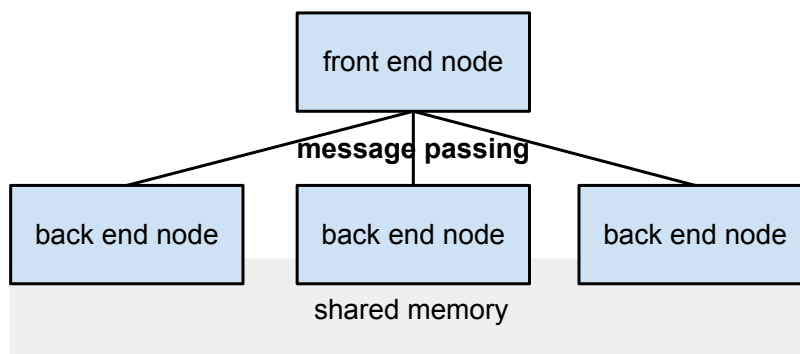
The second approach is known as software-based Distributed Shared Memory (DSM). DSM makes development easier when information in distributed memory can be treated the same as that in local memory. However, DSM introduces coupling between the nodes, and the nodes clustered by DSM need to be more homogeneous than those in the first approach.

Multithreading is implemented in many servers in order to handle simultaneous requests easily, improve performance by increased concurrency, and utilize multiprocessors. When a request initializes an orchestration instance on a multithreading application server, a thread

¹²See <http://redis.io/>



(a) Centralized persistence service



(b) Distributed shared memory

Figure 3.2: Two architectural approaches for session sharing.

will be located for the instance. There are two design options regarding the thread's behaviour: 1) the thread continues working for the instance until the orchestration instance is finished, and 2) the thread stops working for the instance when there is no active processing. If a large portion of the instance's lifetime is not active and waiting for messages or events, the second option will save computational resources and provide better scalability than the first option for a given number of threads.

The second-type and third-type states often need to be shared by threads. The issues of deadlock and starvation are inevitable when states and threads are entangled together. The global variables and `assign` operation in WSBPEL can cause such issues.

3.4.3 Reliability

Reliability of systems is often described and measured as the probability of no failure within a given operating period [94]. The reliability can also be derived via failure rate, which describes the probability that a failure happens instantly after a failure-free period t . The exponential failure distribution is a popular model for reliability, i.e. $R(t) = e^{-\lambda t}$, where $R(t)$ is the reliability of operating period t , λ is the failure rate. Besides the failure rate, reliability can also be measured by Mean Time To Failure (MTTF), Mean Time Between Failures (MTBF), Mean Time To Repair (MTTR), and availability. Availability is the probability that a system is available at any point of time. Simply, reliability can be measured by

$$A = \frac{\text{MTTF}}{\text{MTBF}} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \quad (3.1)$$

A system's reliability is determined by its vulnerable components. There will be a reliability limitation for any component to reach due to physical or cost reasons. Meanwhile, a system's reliability can always be improved by removing single points of failure by introducing redundancy to the system. Note that too much redundancy will still increase the cost and management complexity of a system.

For a network-based application, reliability is the capability to maintain its normal service level or recover from a degraded service level. Partial failures are inevitable for elements like

data, processors, and connectors¹³in network-based applications.

The pursuit of reliability in network-based applications sometimes brings conflicts with other properties like data consistency. In the context of databases, consistency refers to the property that a database remains consistent after a transaction [44]. In the context of data services, a strong sense of consistency means that all processes get the same representation of system state after the state is updated by one of the processes, no matter where a process resides on the network [99]. In other words, if a data service is consistent, then its clients can always get accurate information at the same time. A network partition is an event when the communication between two parts of the network is lost, which is often caused by the failures of network devices.

Eric Brewer made the following conjecture known as the CAP theorem [25].

Theorem 3.4.1 *It is impossible to have a network-based system that has all three properties of consistency, availability, and tolerance to network partitions.*

This conjecture was later formally proved [43]. An implication of the CAP theorem is that a system *cannot* be consistent if it has to be highly available and tolerant to network partitions. For some situations, the availability of incomplete information is still better than unavailability. Vogels reported that availability was more desired than consistency in the design of Amazon web services [99].

For service composition, reliability means the ability to provide service to new requests and recover instances from partial failures. Partial failures for a composition can be the loss of partner services due to either network problems or crashed partner services. WSBPEL and WS-CDL both support the description of fault or exception handling mechanisms. However, the notion of faults is different from that of partial failures. Faults are systemically predictable behaviours that are not success, while partial failures are unpredictable behaviours. In other words, a fault can be caught and handled, while a partial failure can only be recovered from. Recovery from partial failures is also different from compensation that is defined in

¹³In Section 2.3.7 of his dissertation, Roy Fielding wrote “Reliability, within the perspective of application architectures, can be viewed as the degree to which an architecture is susceptible to failure at the system level in the presence of partial failures within components, connectors, or data.” I think what he meant was “the degree to which an architecture is” NOT “susceptible to failure at the system level ...”

WSBPEL to implement transactional data handling. A service composition's reliability can be examined by how fast the composition instances can return to normal operation condition from partial failure. It can also be quantified by the portion of an application that still works when the rest of it fails.

3.4.4 Modifiability

The modifiability is how easy an application can be changed in order to modify a specific component's implementation (evolvability), or adding functionalities to it (extensibility), or reuse some components to achieve new functionalities [38]. A networked application needs to evolve when service hosting systems, client environments, or the network is updated. Such updates might not be the common scenarios for a standalone desktop application to handle, but they are never exceptional for distributed applications running on the Internet like service compositions. When a component is to be updated, the other part of the application should still work as normal. Because the component instances are often distributed across networks, the update should be performed in a gradual manner. When end users have new requirements, an application needs to be extended, which means to add more functions to the application while keeping existing functions working.

For service composition, modifiability can be specified as

evolvability a service composition can replace a partner service with a new one in order to sustain or achieve a better service level.

extensibility a service composition needs to provide new services to the clients according to changed requirements.

reusability Part of a service composition can be used in a new service composition without modification.

These properties are very difficult to achieve by service orchestration. A WSBPEL process can never be modified once it is deployed. A modified process needs to be compiled and deployed again on the process engine.

3.5 Summary

This chapter reviews and compares the approaches for service composition representation. I analyzed the essential characteristics of service orchestration and choreography from their representations. These characteristics contribute to some issues of properties like performance, scalability, reliability and modifiability. The next chapter presents the architectural style of RESTful Service Composition (RSC) and a corresponding programming model.

CHAPTER 4

RESTFUL SERVICE COMPOSITION

Design-by-buzzword is a common occurrence. At least some of this behavior within the software industry is due to a lack of understanding of why a given set of architectural constraints is useful.

—Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures*

You can't connect the dots looking forward; you can only connect them looking backwards.

—Steve Jobs, *You've got to find what you love*

As discussed in Chapter 2 and 3, the traditional web service composition approaches face challenges from the aspects of performance, scalability, reliability and modifiability. The approaches to addressing such challenges on the implementation level, for example, replacing a Java Virtual Machine (JVM) implementation with a native C implementation to improve service performance, have the following disadvantages:

- Such an approach is often application-specific. It is difficult to apply a solution for one application to others.
- Such an approach is often constrained by its dependent environment. For example, the demand of a CPU-intensive task is always limited by the available CPU speed, which, in turn, is physically limited [60].
- Such an approach does not address the group of challenges as a whole. In some cases, the improvement in one aspect can lead to degradation of other aspects. For example, when an application's performance is improved by increasing the concurrency level on

the host system, its reliability is often degraded because the system is less robust to workload fluctuations.

In order to avoid these disadvantages, I choose an architectural approach to addressing the challenges facing service compositions, because it is application-independent, environment-neutral, and able to address the challenges as a whole. For example, such an architecture can utilize event-driven programming for concurrency instead of multi-threading in order to avoid the reliability penalty of increasing concurrency level. I name this approach RESTful Service Composition (RSC). The notion of restful service composition in this thesis is different from those that provide “RESTful” interfaces for traditional web service compositions or service compositions of RESTful services [84, 90]. The questions discussed in this chapter are:

1. How to describe a software architectural style in a systematic way?
2. What is RSC? How is it derived?
3. How to program an RSC-style application in practice?

4.1 Describing software architectural style

The design patterns [42] initiated by the “gang of four” have become a movement in OOP. One of the advantages of their approach was to describe design patterns in a unified way. A software architectural style is quite similar to a software pattern in that they are both abstract and generic software artifacts for specific type of design problems. I adapt the approach to describing design patterns to architectural styles in this thesis. In fact, such an approach was also used for building architectural styles [3], which inspired the work of software design patterns.

A software architectural style is composed of four elements: the problem domain, the proposed solution, one or more application examples of the solution, and the consequences. The problem domain describes the common characteristics of design problems that an architectural style aims to tackle. Such characteristics reflect both functional and non-functional requirements. The non-functional requirements are often more difficult to be addressed than the functional requirements.

Some common non-functional requirements are:

Performance/Scalability A system provides service in a responsive manner, and is able to sustain the service with increasing numbers of users or user requests;

Reliability A system sustains its service in the presence of errors and failures;

Modifiability A system evolves easily when it needs to fulfill new functional requirements.

The proposed solution is an architectural style that is designed for the specific problem domain. As discussed in Chapter 2, a software architectural design is composed of a set of elements, constraints describing the organization and interaction of elements, and the rationale behind the choice of elements and constraints. An architectural style, correspondingly, is composed of a set of *generic* elements, the constraints ruling the *common* way for the generic elements to interact, and the rationale. A generic element can be of a *type* that represents how it is *connected* to other elements. For example, in the architectural style of client-stateless-server (CSS), a server is connected to its clients in a one-to-many relationship. A generic element can also be of a *role* that represents how it *interacts* with connected elements. For example, a server needs to respond to proper requests from clients. A constraint in an architectural style does not restrict the details of interactions between generic elements, but their *principles*. For example, the stateless communication constraint limits a client's dependence on the context state stored on a server.

The consequence is the structural, behavioral and implementation details resulting from the constraints. For the CSS architectural style, the consequence is that a request needs to contain all the information that the server needs to generate the correct response. The rationale behind the client/server abstraction of elements is modifiability, and that behind the stateless constraint is scalability and reliability.

4.2 Defining RESTful Service Composition

A RESTful service composition, in this thesis, does not simply refer to a composition of RESTful services, nor a service composition with so-called RESTful interfaces. The architectural style constrains not only its external interfaces to service consumers and partner

services, but also its internal structure in order to achieve the desired properties. A RESTful Service Composition (RSC) does not have to be developed as a service orchestration nor as a service choreography. Instead, an RSC has a decentralized structure like service choreography but still provides interfaces to initiate composition instance and track its status like service orchestration.

4.2.1 The problem domain

RESTful Service Composition (RSC) describes an architectural style for networked systems where services are the basic building blocks. A networked system is composed of elements distributed across networks. Partial failures are common and not abnormal in a networked system. Therefore, it is a basic functional requirement for a networked system to be able to work with partial failures. A service is an element providing access to information or functions to process information. As far as RSC is concerned, the services can be RPC- or SOAP-based web services, or RESTful services.

The service composition should be able to provide performance as good as its partner services, though by nature, it is more complicated than any of its partner service. The service composition needs to be able to scale with increasing work load or number of consumers. The service composition should still be able to provide service, to some extent, with the existence of partial failures related to partner services or network. The target service should be able to be modified easily in order to deliver new functionalities.

4.2.2 Element types and roles

Elements can be classified into different categories or *types* based on their characteristics. The type represents an element's nature and capabilities inside a system. An element can play different *roles* in various circumstances. There are basically three roles for the elements in a normal service composition — composition consumer, partner service and composition itself. In SOAP-based service compositions — either service orchestration or service choreography — the element types include web service, web service client, and SOAP message. This section discusses the RSC element types, including RESTful service composition client,

RESTful partner proxy, composite resource and functional computation, and RSC element roles, including resource client and relaying service. Some types and roles are introduced so that an RSC-style composition is able to cooperate with existing non-RESTful services. Some types and roles are inherited from REST, and some are specially designed for RSC.

RESTful service composition client

An RSC provides services to its consumers in a RESTful way. This requires composition clients to comply with the REST constraints of cache, stateless, and uniform interface. The cache constraint requires a client to understand cache control metadata in messages and also implement a client-side cache if possible. For service responses of big payloads, cache can dramatically improve the user-perceived performance by reducing the response time. The stateless constraint requires a client to include enough explicit information in each request so that the service composition understands the requests without maintaining the session state with its clients. The uniform interface constraint requires a client to make use of the uniform identification and methods provided by the services, and furthermore, to be able to take part in state transitions driven by hypermedia.

RESTful partner proxy

A partner service of RSC can be either RESTful or non-RESTful. Although some might argue whether a service with RESTful interfaces is truly RESTful, I use “RESTful partner service” to refer to a partner service providing RESTful interfaces. A more straightforward distinction between a RESTful partner service and non-RESTful one is that the latter’s interfaces are developed on the basis of SOAP or RPC.

As discussed in Chapter 2, RPC or SOAP makes a service difficult to work with cache and intermediaries. These drawbacks prevent a service composition of non-RESTful services from achieving the desired properties of performance, scalability, and reliability. The RESTful partner proxy element type is introduced to RSC in order to overcome these drawbacks while still making use of the existing non-RESTful services.

A RESTful partner proxy connects RESTful clients to a non-RESTful service. The proxy communicates with non-RESTful services in SOAP or RPC, and communicates with RESTful

clients through its uniform interfaces. With the help of RESTful partner proxies, a RSC can treat non-RESTful services as RESTful at least at the interface level.

There is no way for obtaining information for building a RESTful interface from the interface description of a non-RESTful service, for example, we cannot get the cacheability and idempotence of an operation by analyzing the WSDL of a SOAP-based web service. Therefore, the implementation details are required in order to develop a RESTful partner proxy for a non-RESTful service.

Composite resource

A resource in the context of RSC has the same sense as that in REST discussed in Section 2.7. A resource is any information that can be named. A composite resource is a resource composed of a set of other resources. An operation of a composite resource is often mapped to corresponding operations of its members. The representation of a composite resource is normally generated by retrieving, processing and combining the representations of its member resources. A common pattern for service compositions is that an operation is first mapped to a set of partner services and then yield the result by reducing. This pattern can be implemented as a composite resource.

Resource client

In REST, the server is viewed as a connector, and the details inside a server are not of interest in architectural design. This makes sense for designers to focus on the relationships between server and other connectors like client and cache. However, from RSC developer's point of view, the internal structure of a server is important for the overall quality of the whole application. For example, Model-View-Controller (MVC) is a widely used architectural style for web application servers. It is also called a multi-tier architecture, and somehow related to the layered system constraint in REST.

An RSC-style composition plays the role of service client as shown in Figure 3.1 when consuming RESTful partner services. It is obvious that the client element needs to follow the REST constraints when interacting with RESTful partner services. An RSC-style composition is also a client when accessing other resources like files or Database (DB). For such resources,

the access is normally conducted through native file or DB interfaces.

The accesses of partner services, files and DB are all related to I/O. Such operations are normally slow. The synchronous way of programming I/O often degrades a system's performance and scalability. In RSC, the resource clients need to follow the constraints that results in an asynchronous I/O operations and programming.

Functional computation

The most important abstraction in RSC is a functional computation. A functional computation is a self-contained computation that can be represented by a function and its inputs. The inputs to a function can be other functions, and the output can also be a function. The evaluation of a function in RSC may depend on its environment, and form a closure. A functional computation can relay the computation to another functional computation with a *future* computation as the callback. Such computation relays are often related to I/O operations that might take a while to finish.

Furthermore, a computational representation can be sent to a capable service where the computational representation can be evaluated to create a computation instance. This makes it possible for a service to finish part of the computation of a composition instance, and then pass it to the other service where the computation can be continued. A decentralized structure will therefore be formed.

Relaying service

When a service finishes part of the requested computation and then passes the rest of the computation to the next service, it plays the role of relaying service. A relaying service can send a computation representation to another service or even to itself. The computation of a composition instance can be carried out by a series of inter-connected relaying services.

During the process of computational relaying, each relaying service needs to decide how to finish the received computation according to its perception of the “baton”. Different from the baton used in a race, a computational baton can be modified during a relay, and it can be branched and merged by a relaying service on the path to the finish. The sender has knowledge of the receiver's capacity to evaluate the relayed computation. This can be achieved by

computational content negotiation. A relaying service always behaves conservatively so that it only executes the trusted part of received computation representation.

4.2.3 Constraints

Staged computation

A service composition is partitioned into loosely-coupled process sections called *stages*. A stage is a scope where computation is carried out within its own context and control flow. It is also a leg that composes the whole process.

A stage normally starts with an I/O event that can be a message, a file event, or a DB operation event. A stage ends with an I/O operation such as sending a message or a group of similar messages, or initiating a file or DB operation that will trigger new events in the future. There is no I/O operation inside a stage.

A stage encapsulates computational state so that there is no shared state between stages. The lifetime of a stage is always shorter than the composition because it does not need to wait from an I/O or timing operation to finish. The lightweight character of a stage makes it computationally cheap to replicate a stage or re-execute a stage.

An existing service composition designed in the orchestration way can be transformed into stages by partitioning it according to stage patterns [72]. This process can also happen dynamically by a relaying service that finishes the leading stage of a received computation and passes the rest to the next relaying service or a resource client.

Uniform computation identifier

Service compositions, composition instances and stages are explicitly identified by one mechanism. Since functional computation is the abstraction for compositions, composition instances and stages in RSC, their identifiers are called computation identifiers. A computation identifier can be considered as a special type of resource identifier, and therefore can reuse the syntax of a resource identifier.

A computation identifier is the *name* of a computation and is the endpoint to manipulate it. The uniform computation identifier makes it possible to expose the computations in

different locations to the same space of inter-connected resources and services.

Access methods are the same for all computations

Similar to REST, RSC requires all computations and computation instances to be accessed by the same set of methods. A specific computation or computation instance may allow a subset of the methods. The semantics of those methods for computations or computation instances are semantically similar to those for resources. This constraint makes it possible for an intermediary service, like a relaying service, to get the basic information of a request just by interpreting the access method and the identifier. For example, we can reuse HTTP methods for access computations in an RSC-style composition. Table 4.1 describes the methods' semantics.

Table 4.1: The semantics of HTTP methods for computation and computation instances in RSC.

Method	Semantics
GET	retrieve the representation of the computation
	retrieve the status of the computation instance
PUT	create a new computation or modify the existing one identified by the URI
	change the state of the computation instance, for example, pause or resume, modify the inputs of an active computation instance
POST	create a new computation instance with given inputs
	provide an active computation instance with more inputs
DELETE	remove the computation
	stop and remove the computation instance

The transitions between stages are driven by baton passing

The loosely-coupled stages of an RSC instance are linked together by *batons*. Each instance stage is initiated by a baton, which is either passed to it from the previous stage or created on the initial composition request. Different from the baton used in race, a computational baton can be modified during a relay, and it can be branched and merged on the path to finish. The sender and the receiver of a baton have mutual trust. The sender has knowledge

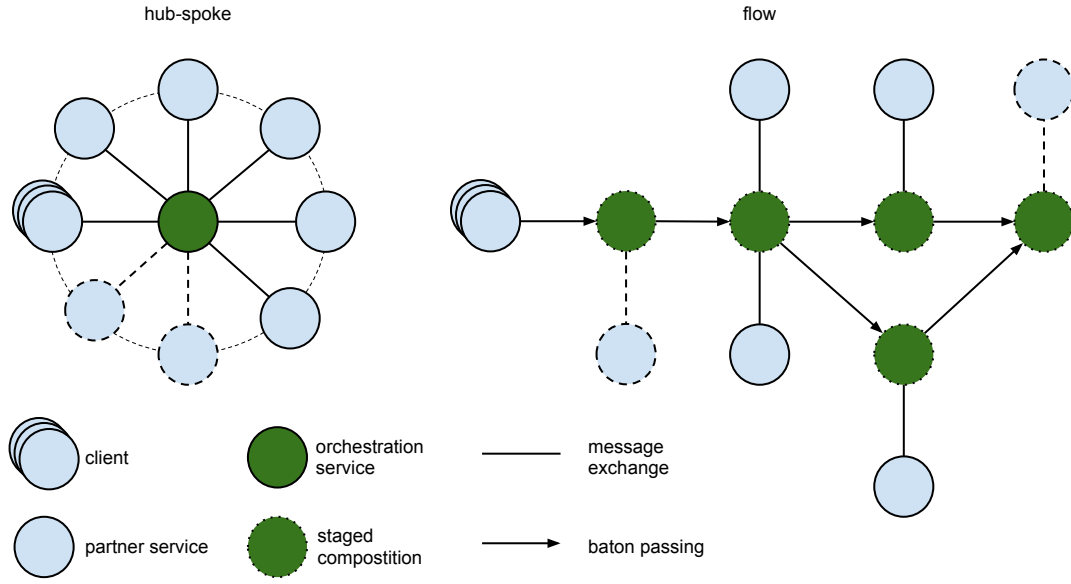


Figure 4.1: The flow structure of RSC compared with typical hub-spoke structure of service orchestration.

of the receiver’s capacity to understand and finish the relayed computation. This can be achieved by computational content negotiation if possible.

A baton can be a message that contains a computation representation or computation identifiers based on which the receiver can construct its stage computation. A baton can also be a callback passed to a resource client. The callback will be triggered to continue the computation when the resource access is finished.

4.2.4 Consequences

Structure

When a service composition is partitioned into computational stages, the typical orchestration’s hub-spoke structure will likely change to a flow structure shown in Figure 4.1. The flow structure makes it possible for a service composition to be deployed in a decentralized way. This improves a service composition’s scalability by removing the major scalability bottleneck — the central hub.

The continuation of a functional computation makes it possible for a service to finish

part of a computation and pass the rest of the computation included in a baton to the *next* service. The computation of a composition can be relayed until it is finished in this way. A service can relay the computation to a different service or to itself. An RSC-style composition can be implemented by the services capable of executing staged computations. Computation negotiation can make the flow structure of baton passing even dynamic since a relaying service can choose to accept or alter the computation to be passed at runtime.

Behaviour

The functional computation abstraction and baton passing require composition stages to support computation negotiation. Computation negotiation makes a service composition more dynamic and flexible than traditional service orchestration or choreography, since each relaying service can choose to accept or alter the computation included in a baton message. The programming of service and client part of a relaying service becomes even more sophisticated because of the dynamics.

Many services use HTTP for message transportation. Request-response is the only message exchange pattern supported in HTTP [14, 36]. Most HTTP-based server applications like servlets can commit responses only when the computation initiated by the request is finished. This model of synchronous processing and messaging brings challenges for RSC's staged computation because a stage can never wait for its successor stages to finish their computation. Asynchronous conversation is one of the consequences for services in order to carry out asynchronous processing. Asynchronous conversation can be implemented with either new MEPs or a series of request-response's.

An asynchronous conversation based on request-response needs the participants to follow simple protocols. For example, endpoint A sends a request to endpoint B, and B replies immediately and tells A to retry in a period of time. Then A is free to do something else until the time for retrying. In this case, A needs to be able to schedule the retry. Another way is as the following. B replies immediately and tells A that A will be contacted when what A just requested is available. In this case, B needs to be able to send messages to A, and A needs to be able to listen. Figure 4.2 shows two asynchronous conversations composed of synchronous messaging. An asynchronous conversation needs more request-response pairs

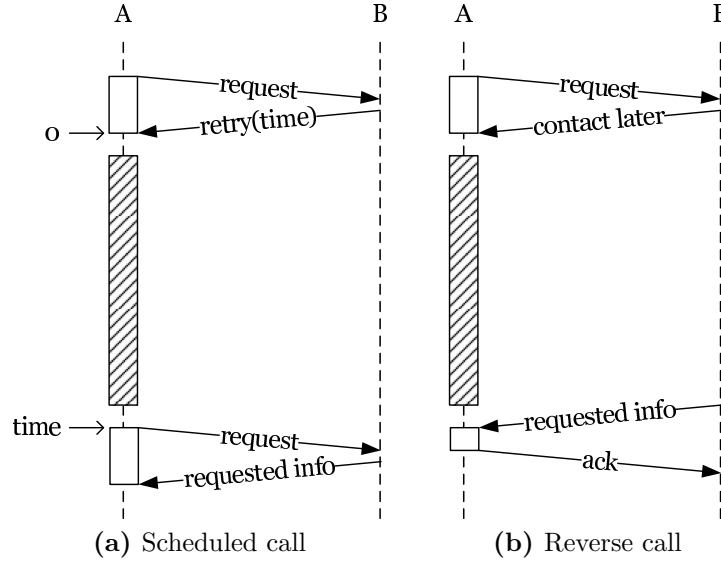


Figure 4.2: Asynchronous conversations composed of synchronous messaging.

than the corresponding synchronous version.

Implementation

The abstraction of functional computation implies that an RSC-style composition needs to be implemented by a language with first-class functions. Popular OOP languages like Java or imperative programming languages like C also can be used for implementation, but the lack of first-class functions in such languages will increase the complexity of programs and the effort of programming enormously. Chapter 5 will have a simple comparison for this.

The computation to be passed to a relaying service needs to be represented by a language that can be converted to or is itself executable in run time by the relaying service. For a static programming language like C or Java, that will be difficult to implement. It will be quite natural to use dynamic languages with virtual machine support like JavaScript for this purpose.

When a stage ends with an I/O operation via a resource client, its computation terminates right after the resource client sends out the resource access request, no matter how long the I/O operation will run. This requires the resource client to be programmed in an asynchronous way. And that in turn requires the system to support Asynchronous Input/Output (AIO).

Table 4.2: File and composite file resource interfaces.

URL	Method	Request	Response
<code>/{{scan}}/?{{query}}</code>	GET	indicates accepted MIME types and cache control in headers	the representation of the composite resource specified by the query
<code>/{{scan}}/{{file}}</code>	GET	indicates cache control in headers	the content of the file
<code>/{{scan}}/</code>	POST	indicates form-data and files in headers and multipart contents in body	the locations of newly created resources like <code>/{{scan}}/{{file}}</code> if present.

4.2.5 An example

As discussed in Section 3.1, a service composition can be developed for near real-time processing of XRD data. Part of the service composition is to transfer raw data from a CLS service to a UWO service when it is available, and similarly to transfer the processing results in the opposite direction. Section 3.2.1 described a service orchestration designed on the basis of web services. The scenario can be implemented as an orchestration based on HTTP-based RESTful services. The key resources in this design are file and composite file whose interfaces are listed in Table 4.2.

Part of the orchestration of these RESTful services will be

Listing 4.1: Part of an orchestration of RESTful services.

```

flow : {
  sequence : {
    finished_image_list := [],
    new_image_list := [],
    finished := false,
    while (!finished) {
      list_image_id := GET(cls_data_service/scan_id?all),
      new_image_list := list_image_id -
        finished_image_list,
      if (new_image_list.length > 0) {
        foreach image_id in new_image_list parallel {
          image := GET(cls_data_service/scan_id/
            image_id),

```

```

        result_id := POST(uwo_processing_service/
            scan_id, {processing_id, image})
    }
}
finished_image_list := list_image_id
if (finished_image_list.length ==
    scan_configuration.mapSize){
    finished := true
} else {
    waitFor : (4 seconds)
}
}
}
sequence : {
    finished_result_list := [],
    new_result_list := [],
    finished := false,
    while (!finished) {
        list_result_id := GET(uwo_data_service/
            processing_id?all)
        new_result_list := list_result_id -
            finished_result_list,
        if (new_result_list.length > 0) {
            foreach result_id in new_result_list parallel {
                result := GET(uwo_data_service/
                    processing_id/result_id)
            }
        }
        finished_result_list := list_result_id
        if (finished_result_list.length ==
            processing_configuration.mapSize){
            finished := true
        } else {
            waitFor : (4 seconds)
        }
    }
}
}
}

```

One of the benefits of REST is that the solution described in List 4.1 can improve performance by caching file resources. Except for the RESTful interfaces, this design has the same structure to that in List 3.6. Such a structure makes both designs suffer two issues:

- Transferring large image files from the CLS service to the service composition and in turn to the UWO service can be slow and resource-intensive.

- The parallel access of many remote resources can slow down the service composition and even lead to race conditions.

A new composition is designed by applying the RSC style. In this new design, the access to remote resources is represented as a computation, and the composition is partitioned into stages.

Listing 4.2: The relaying service of `uwo_data_service`.

```

on(POST, /scans, image_resource_list, processing_id) {
    transfer_stage_id := create_stage(image_resource_list,
        processing_setting_id, callbacks)
    reply(uwo_data_service/transfers/transfer_stage_id)
}
create_stage = function(image_resource_list,
    processing_setting_id, callbacks) {
    transfer_stage_id := gid(image_resource_list)
    register_state(transfer_stage_id)
    forEach resource in image_resource_list {
        http_client(resource/method, resource/url, callbacks(
            transfer_stage_id, processing_setting_id, request,
            response))
    }
    return transfer_stage_id
}
callbacks = [onSuccess, onFail]

onSuccess = function(transfer_stage_id, processing_setting_id,
    request, response) {
    image_id := save_resource(request, response)
    update_state(transfer_stage_id, image_id)
    uwo_processing_service/processings/processing_stage_id :=
        POST(uwo_processing_service/processings,
            processing_setting_id, image_id)
    update_state(image_id, processing_stage_id)
}

on(GET, transfers/transfer_stage_id) {
    return retrieve_state(transfer_stage_id)
}

on(GET, scans/image_id) {
    return retrieve_state(image_id)
}

```

The `image_resource_list` is a computation representation sent from the composition

to `uwo_data_service`. It contains a list of image resources and the way to retrieve them, basically the resource URLs and methods. A transfer stage is started when the `uwo_data_service` received the transfer baton. The transfer stage ends when all the retrieving of resources are handled by `http_client`. The transfer baton is then branched into batons that are represented by callbacks. When a callback is executed, an anonymous stage is started. The anonymous stage saves the image, sends the processing request to processing service, and updates the *persistent* state of the transfer stage and the image. A client can figure out the state of a transfer from `uwo_data_service/transfers/transfer_stage_id`, and also the state of an image from `uwo_data_service/scans/image_id`, and further the state of a processing from `uwo_processing_service/processings/processing_stage_id`.

The cost of sending computation representation of image transfer is much lower than sending the images themselves. For k images of size S , the transportation payload size is reduced from $2kS$ to kS , and the number of required I/O operations in services is reduced from $4k$ to $2k$. This will result an improvement of performance by 100%. The other significant difference between this design and the previous is the usage of `forEach`. For composite resources, `forEach` is the most common construction for performing computation on each resource. As discussed in Section 3.2.1, WSBPEL provides two forms of `forEach`, sequential and parallel. For the sake of performance, parallel execution is often chosen, which requires careful handle of state synchronization in order to avoid deadlocks and race conditions in the normal multi-threading programming model. In the design described in Listing 4.2, there is no need to perform the `forEach` in parallel. Sequential execution can achieve the same level of performance thanks to the style of baton passing. The programmers can benefit from this without worrying about state synchronization. Section 4.3 discusses how to achieve this in detail.

4.2.6 The connections to RESTful Service Composition

As mentioned in Section 1.4.1, I named the architectural style presented in this thesis RESTful Service Composition because REST contains most essentials of the style. It is not a design-by-buzzword, nor a direct application of REST to service composition. RESTful Service Composition is developed on a wide spectrum of theoretical and empirical foundations.

Process and reality

Alfred North Whitehead described a view of the actual entities in his work of *Process and Reality* [105, 92]. In this view, a reality exists in process instances. An actual entity never “changes” because it is nothing but a process instance that is either a concrescence or a transition. A concrescence instance results in positive prehensions of a particular existent. A transition instance results in an original element that constitutes other particular existent when the process perishes. A feeling is the process to map a datum to a subjective form by the subject.

When we think service composition in Whitehead’s way, a composition’s state is the reality of interest. The process of feeling can be considered as a computation that results in a prehension of the initial input or effects a state transition. “There is a becoming of continuity, but no continuity of becoming”. A computation can be always partitioned into stages. Stages connect together to become a continuous flow. A computation’s state can only be “felt” via its representation.

Finite-state machine

A finite-state machine, or a state machine, is a mathematical model to prescribe or describe the behaviour of an abstract machine that could be a simple or complex system. A finite-state machine is composed of a finite number of *states*, a set of *events*, and the transitions between states on specific events. When its next state is fully decided by its current state and a given event, a state machine is deterministic. Theoretically, a nondeterministic state machine can be converted to a deterministic one by powerset construction [59]. Deterministic state machines are more widely used in modelling because of the easiness of implementation.

Mathematically, a deterministic state machine can be defined as quintuple $(\Sigma, S, s_0, \delta, F)$, where Σ is a non-empty finite set of events, S is a non-empty finite set of states, s_0 is an initial state, δ is the state transition function such that $\delta : S \times \Sigma \longrightarrow S$, and F is a subset of S containing the final states.

By definition, a deterministic state machine can be in only one state at any given moment, and its next state is deterministic according to a given event and the current state.

The determinism ensures the state machine can be implemented by a normal single process program. Although very simple, the state machine is so powerful that it can be used to model and design complex systems. Some real software systems like `lighttpd`¹, a high-performance web server, were designed as a state machine.

Can we use a state machine, or more precisely a “distributed state machine”, to model and design distributed applications? At first glance, it might seem to be viable because each distributed node can be modelled as a state machine. However, it has an intrinsic issue — an inability to determine the local state transition triggered by an event sent from a remote node given a local state and vice versa. A deterministic state machine has clear notion of its current state, while it is difficult to tell the state of a remote state machine R given its current state s_R^t and the event that a local state machine L will trigger by sending a message m to R . For the knowledge that L has about R was its *perception* of R ’s state, which could change when m reaches R because of *latency*. In fact, even L ’s perception about R ’s state could be wrong due to network *partitions*. The lack of state consensus prevents a distributed state machine from working as a normal state machine. This theoretically contributes to the technical issues of RPC discussed in Section 2.10.

We can consider each staged computation in RSC as a state machine, and the interconnected stages as a distributed state machine. This is only made possible by the style of baton passing when recognizing the non-existence of state consensus.

Actor

The actor model described a generic model for distributed parallel programming [53, 11]. The actor model extended the dataflow programming model by enabling an actor to create new actors. Each actor has a message box via which messages can be received asynchronously. Message passing is the only way to interact with an actor after it is created. A message can contain the identifier of actor(s). Global state is considered harmful in the world of actors. Because the actor model was purely abstract, there have been only a few direct implementations. In fact, the model is so generic that one can easily see its connections to RPC, SOAP and REST.

¹See <http://www.lighttpd.net/>

The styles of the actor model were carried forward by Erlang/OTP², a concurrency-oriented programming language and libraries originally developed by Ericsson and now open-sourced. An Erlang-based system, the AXD301 switch by Ericsson, achieved 99.9999999% (nine 9's) reliability [10], which is equivalent to about only 3 seconds downtime in a year. It is not by failure avoidance that the system achieved such high reliability but by isolation of partial failures and recovery from partial failures. Joe Armstrong, one of the major contributors of Erlang/OTP, summarized the design principles of Erlang that result in reliable distributed systems in the presence of software errors as follows [9].

- Computation is virtualized as processes.
- Processes running on the same machine are isolated.
- Each process is identified by a unique unforgeable identifier.
- Processes share no state (memory), and asynchronous message passing is the only way for processes to communicate.
- Unreliable message passing is never exceptional. There is no guarantee of delivery.
- The failure of a process can be detected by another process, and the failure reason is described in a failure message.

These principles make the Erlang model much suitable for developing distributed systems that can tolerate partial failures. Asynchronous message passing between isolated computations is the common feature of the actor model, Erlang style and RSC. A significant difference between the Erlang-style concurrency model and the actor model is that the former assumes no guarantee of message delivery while the latter does. The reliable message delivery assumption was a huge simplification of the real distributed systems, and also prevented the actor model to be applied to networked applications. RSC does not assume guaranteed message delivery.

²See <http://erlang.org/>.

SEDA

Welsh proposed an architecture named Staged Event-Driven Architecture (SEDA) for scalable Internet services [102, 101]. A SEDA style application is a network of event-driven stages connected by queues. A stage always follows an event queue and contains a private thread pool. The queues separate the execution environments for stages and provide admission control support and event management. Resource allocation on the stage level can be tuned with private thread pools.

Stage computation is the connection between SEDA and RSC. The stages in SEDA are static and fixed for all the computation tasks in the application. In RSC, stages are request-dependent. Some stages can be determined in design time by the logic of composition. And others are dynamically generated during run time by batons.

ARRESTED and CREST

ARRESTED was an architectural style proposed for distributed and decentralized systems [64]. Distributed systems need to work on a partitioned network where faulty message passing is the norm and asynchronous messaging is required. Decentralized systems need to deal with uncertainty and disagreement of remote resource states. It is almost impossible to achieve consensus of a remote resource's state in distributed decentralized systems. REST is extended to ARRESTED by the following constraints.

- Use **a**synchronous event notification to achieve quick updating of state.
- A message can be **r**outed via a proxy or directly delivered to the destination.
- A **d**elegation can provide a **d**ecision function that ensures **d**istributed consistent access of a resource.
- An **e**stimation function provides the most precise representation of a remote resource base on available local information when the remote resource is not accessible within a given duration.

Although ARRESTED addresses a different problem from RSC, the message routing constraint can be considered as a special case for computational baton passing.

Computational REST (CREST) extended REST by adding the following constraints to REST.

- The representation of a resource is a computation that can be a program, a closure, a continuation, or a computation description and its binding data.
- All computations are context-free. This is corresponding to the stateless constraint in REST.

The computational representation is the key connection between RSC and CREST.

4.3 Programming RESTful Service Composition

Generally speaking, an architectural style should be generic so that it can be applied to a matching application no matter how it is programmed. However, certain language and library can make the implementation easier. For example, HTML is more suitable for representing hypermedia in a RESTful application than JSON. This section discusses the programming paradigm for an RSC-style application.

4.3.1 Functional programming elements for RESTful service composition

On-demand Function

As discussed in Section 4.2.2, functional computation is a key element in RSC. It is quite straightforward to implement such functional computation as functions. The function refers to the same term used in functional programming, not the subroutine in imperative programming. Although a typical imperative programming language like C, C++ and Java can use their own constructions to mimic functions, like anonymous class in Java [12], such languages are more restricted than functional programming languages. In a functional language, functions are first-class constructions that can be passed, returned, assigned to a variable, and stored in data structures. Scala, Clojure and JavaScript are functional languages widely used.

A computational representation needs to be easily executed by a relaying service or a resource client. If the computation is represented by a non-dynamic programming language like Java, Scala or Clojure, a service needs to compile the representation and then execute it in a proper virtual machine. It is also possible to use pre-compiled byte code as the computation representation, but that violates the self-descriptive message constraint of REST. It is more difficult to verify if a byte-code representation is secure and trustful than to verify a plain text representation. Dynamic languages can represent a computation in plain text code, which makes the development of responsive and safe relaying services and resource clients much easier.

Map-reduce

The Parallel structures in most service compositions from an orchestration's perspective can be considered as composition resources in RSC. The typical programming style for such problems is map-reduce. It is different from the MapReduce model proposed by Google engineers [32]. However, they are closely related because they address the same type of problems. The problem is, for a given *list* of elements, to apply a function to each element (map), and then to compose the result based on the output of each function (reduce). Clearly, map can be implemented as a high-order function, like the `forEach` used in Listing 4.2. The map function is implemented by most functional programming languages. How to implement the reduce part is trickier than the map part because the result is shared by all mapped functions.

Continuation passing

A continuation is an abstract representation of computation whose execution is in the future upon a control event. Continuation can be implemented by both functional languages and object-oriented languages like Java. In Jetty³, a continuation is an object that can be used to control the processing of an HTTP request. A processing can be suspended and later resumed. Jetty uses this mechanism to achieve better usage of threads. When a processing

³See <http://wiki.eclipse.org/Jetty/Feature/Continuations>

is suspended, the thread carrying on the processing is freed and returns to the thread pool; when a processing is resumed, a new thread is located from the thread pool to carry on the processing. Therefore, a Servlet can be “asynchronous” so that a thread never blocks when the processing needs to wait for some events. Continuation can be implemented much more easily in a functional programming language merely by a closure.

A closure is formed when a function is associated with its context — in terms of variables — in which the function is created. In functional languages, a continuation is often passed to a function as a callback that becomes a closure by passing the function’s local variables to it. Therefore, it was also called closure passing [5]. When the callback is asynchronous, the function returns instantly without blocking, and the computation in the callback will be executed in the future and still has access to the free variables that are local to the function that already returned. Continuation passing is the most common style that can be found in functional implementations for the map-reduce and resource client scenarios that I have discussed.

Programs with Continuation-Passing Style (CPS) design can often be optimized by compilers to improve their execution efficiency [6, 63]. If a CPS-based program is not optimized, it could cause memory overflow because of deep stacks. However, such a problem does need to be worried about in the RSC style, because a continuation should always appear in the form of asynchronous callback. An asynchronous callback is always executed on a *new* stack.

4.3.2 Infrastructure for programming RESTful service composition

The resource client discussed in Section 4.2.2 needs the service infrastructure to support non-blocking access of files, web resources, and databases. In order to cooperate with I/O events, the resource client needs to be programmed in an event-driven style. Evented I/O is a promising solution for the so-called C10K [62] and RC10K [74] problems.

The original C10K problem studies how to provide reasonable service to 10,000 concurrent clients using a normal server. The essential of the C10K problem is how to support a large number of inbound Transmission Control Protocol (TCP) connections and how to serve the

concurrent requests, which leads to two key design decisions of HTTP servers: I/O and concurrency strategy. The RC10K problem studies how to support 10,000 simultaneous outbound HTTP requests running on a web server. The RC10K problem can be considered as a mirror of the C10K problem, and therefore, the solution of evented I/O is almost identical for both problems.

The I/O models can be divided into two groups: basic I/O and advanced I/O. Basic I/O is synchronous and blocking. An I/O operation is synchronous if the thread initializing the I/O operation cannot switch to other operations until the I/O operation is finished. A function or method is blocking if it does not return until its execution either successfully finishes or encounters an error. There are three ways to implementing advanced I/O [96]. The first approach is to construct a loop to keep trying an I/O option while catching I/O errors until it succeeds. This approach is called polling, and it wastes CPU time. The second approach is I/O multiplexing using `select()`-like system functions. Most operating systems support `select()`, and it is also supported by JVM 1.4 and later versions. The descriptors of connections can be registered on a selector, which calls `select()` to check if there is any I/O event for each of the descriptors. So a thread initializing an I/O operation can delegate the job to a selector and switch to another job. Note that `select()` is blocking until one of the registered descriptors has an event or a timeout occurs. The third approach is Asynchronous Input/Output (AIO), which is both asynchronous and non-blocking. Both I/O multiplexing and AIO enable a server to use a few threads to handle many concurrent connections. Because the native I/O interfaces supported by systems are quite diverse, it is critical for RSC to be implemented on the basis of virtual machines supporting advanced I/O.

4.3.3 Related programming paradigms

Event-driven programming

An event is either the start or the end of a process. In event-driven programming, the flow of a program is controlled by events, which is achieved by 1) detecting an event, and 2) handling the event by executing all actions bound to the event. These two steps are the responsibilities of the event loop that is the main component of the event-driven runtime. It

is the programmer's responsibility to define event handlers and bind them to events. Most event-driven frameworks also allow the programmer to create customized events. Event handlers can be naturally implemented by functions.

Graphical User Interface (GUI) might be the most popular domain where event-driven programming is applied. This is decided by the nature of the problem — user interactions with GUI components are parallel and responsiveness is critical for the user experience. These requirements are also true for most server applications. Many traditional web server implementations use processes or threads to handle concurrent requests, like Apache httpd⁴ and Tomcat⁵. Event-based server implementations have got more and more attention because of their light weight and performance, for example, lighttpd⁶ and NGINX⁷. These implementations were designed to address the C10K problem and outperformed httpd in consistent response time and low memory footprint for a large number of concurrent clients.

Dataflow programming

The dataflow programming paradigm appeared in late 1970's, aimed to address how to program parallel processors [61]. It was later developed as a general purpose programming paradigm. The most well-known dataflow programming languages are LabView and VHSIC (Very-High-Speed Integrated Circuits) Hardware Description Language (VHDL). LabView⁸ is a graphical programming language developed by National Instruments⁹ for virtual instrumentations. A virtual instrumentation is the combination of user-defined GUI and processing functions that mimics circuits. VHDL is a description language for digital and mixed-signal electronic systems like integrated circuits. VHDL is capable of easily describing concurrent systems because of its dataflow nature. The VHDL description of a system can be verified by simulation. Most dataflow programming languages are closely related to hardware, including processors, circuits, and instruments.

A dataflow program is often represented as a directed graph, where nodes are instructions

⁴See <http://httpd.apache.org/docs/2.2/programs/httpd.html>

⁵See <http://tomcat.apache.org/>

⁶See <http://www.lighttpd.net/>

⁷See <http://nginx.org/>

⁸See <http://www.ni.com/labview/>

⁹See <http://www.ni.com/>

and directed arcs are the data flows between nodes. A node is fired once its input arcs have data¹⁰. This makes the biggest difference between a dataflow language and an imperative language. For the instructions are executed one after the other pointed to by the program counter in the machine code of an imperative program, while multiple instructions can be executed in parallel in a dataflow program.

There is no notion of global variables in a dataflow program, which implies that all effects of computation are local [2]. If we consider the arcs as variables in a dataflow program, such variables never change their values once assigned. Two or more arcs forking from one node hold duplicates of the same value, which makes the computation in various nodes connected to those arcs *independent*. In other words, the computation happening in a node has no side effects. A dataflow program is *functional* because of these features.

4.4 Summary

This chapter systematically describes the RSC architectural style. RSC is described as a problem–solution–consequence–example for easy understanding. The connections between RSC and other architectures or architectural styles are examined. A reference programming model is proposed for implementing RSC-style applications. The programming model can help developers not only to understand the architectural design but also to grasp the essential of programming techniques. The major differences between RSC and traditional web service orchestration approaches are

- A RSC-style composition is partitioned into stages that can be developed and deployed independently. On the contrary, a service orchestration can only be developed and deployed as a whole.
- A message represents a computation that can be initialized, executed, modified and relayed in RSC, while it contains the representation of a shared object and its invocation in web service compositions.

¹⁰This is the behaviour of the most widely used dataflow models.

- The interactions between the composition clients and the composition and between different composition stages are all through uniform interfaces in RSC while the interfaces in web service compositions are all based on SOAP.

Next chapter presents the evaluation of RSC in order to prove these differences benefit service composition applications with respect to performance, scalability, reliability and modifiability.

CHAPTER 5

EVALUATION

My friend, all theory is grey, and green
The golden tree of life.

—Johann Wolfgang von Goethe, *Faust*

The impacts of a software architectural design on system properties can be evaluated on a conceptual level. For example, the architectures of the File Transfer Protocol (FTP) and BitTorrent (BT) can be compared by the elements — the clients and the server in FTP vs the peers and the tracker in BT. Furthermore, one can find that BT is more scalable than FTP because the number of a shared file’s replicates increases without boundary with the number of peers in a BT network. The impacts of a software architectural style can be evaluated in a similar way. For example, one can easily conclude that caching can improve a networked system’s scalability and reliability by adding more replicates of data into the system. However, such a conceptual approach has the following limitations:

1. There is no way to quantify an architecture’s or an architectural style’s impacts on the conceptual level.
2. The differences made by some architectural constraints cannot be clearly shown without comparing the implementations.

In order to evaluate RSC’s impacts on systems’ various properties, including performance, scalability, reliability and modifiability, I use four service composition applications for evaluation. The evaluation goals of the applications are listed in Table 5.1.

Table 5.1: The application scenarios and the corresponding evaluation goals.

NO.	Scenario	Evaluation perspective
1	An application in the Science Studio project was developed for transferring XRD files generated during an experiment from CLS to UWO. The application interacts with several underlying services such as the file monitoring service, the file transferring service at CLS and the file transfer service at UWO.	This evaluation studies the impact on file transfer <i>performance</i> introduced by the structural changes.
2	The BMIT beamline at CLS used an application to capture the values of a group of Process Variables (PVs) to generate a snapshot of machine status. In order to make the “exposure” time short, the snapshot service maps individual PV capture requests to a group of services and then reduces the response to generate the snapshot. Some PVs can be unavailable at the time of capturing.	This evaluation studies the <i>scalability</i> of the application for increasing numbers of available and unavailable PVs.
3	A simulation is developed to study the behaviour of a notification application in the presence of network partitions. The notification application is a typical service composition that delivers notifications to a list of services that are distributed in different areas. The connections between areas can be lost due to failures.	This scenario is designed to evaluate if baton passing of RSC can improve an application’s <i>reliability</i> in the presence of partial failures. It also evaluates an application’s <i>performance</i> when failure handling approaches are implemented.
4	The XRD file transfer application has been modified to satisfy emerging requirements when integrating the application with other applications and deploying it to Advanced Light Source (ALS).	The <i>modifiability</i> of RSC is evaluated in this scenario.

5.1 XRD file transfer service

The details of Science Studio [71, 78, 93] have been described in Chapters 3 and 4. This section presents two implementations of the XRD file transfer service, and compares their performance in a real working environment with simulated workload. The first implementation is a typical service orchestration, and the second implementation is an RSC-style composition. A significant difference between them is that an image is only transferred once in the RSC-style implementation rather than twice in the orchestration implementation. In the RSC-style implementation, both the CLS-side service and the UWO-side service are able to initiate a message exchange. The RSC style introduces these structural differences between the two implementations.

The XRD file transfer orchestration creates two HTTP clients instances when there is an image to transfer. One HTTP client sends a GET request to the URL identifying the new image to transfer hosted on the CLS-side partner service. When the image is sent back in the response body, the other client sends a PUT request to the URL identifying the new image to create hosted on UWO-side partner service with that image in the request body. A new image is then created in the UWO-side file system.

In the RSC implementation, the CLS-side partner service sends a PUT request to the URL for the image to be created on the UWO-side partner service when an image is available. Different from the orchestration implementation, the request body does not include the image, but a link to the image. The message sequences of the orchestration implementation and the RSC implementation are shown in Figure 5.1.

5.1.1 Experiment setup

Two implementations are deployed and tested with simulated workload as described in Table 5.2.

Table 5.2: Testing environments and workload for the XRD file transfer service implementations.

Orchestration implemen- tation	A Servlet implementation of the service orchestration described in Section 3.2.1 is developed based on Jetty 7. The partner services are also based on Jetty 7. The service orchestration runs on Scientific Linux 5.3 64-bit servers named srv-ibm-01 with Oracle (Sun) JVM 6. The CLS-side partner service runs on a server named srv-rba-01 with the same software environment as srv-ibm-01 , and they are connected with CLS LAN. srv-ibm-01 has a CPU of 8 cores of 2.4 GHz and 12 GB memory. srv-rba-01 has a CPU of 4 cores of 2 GHz and 2 G memory. The service orchestration polls the CLS-side partner service for new images through HTTP. The UWO-side partner service run on a Ubuntu Linux 10.04 32-bit server named beowulf , which connects to CLS-side servers through high speed research networks ¹ . beowulf has a CPU of 4 cores of 2 GHz and 8 GB memory.
RSC imple- mentation	The CLS-side partner service and UWO-side partner service are designed as described in Section 4.2.5, and developed based on Jetty 7. The CLS-side service runs on srv-ibm-01 , and the UWO-side service runs on beowulf . The CLS-side service polls the file system for new images.
Workload	A serial of XRD scans are simulated for both Setup A and B. The map sizes are 10×10 , 20×20 , 40×40 , and 80×80 . In real beamline experiments, the CCD detector acquires an SPE image every 8 seconds. The simulation program writes a new image to a scan directory every 4 seconds instead. This does not make the test result different from real situations because CPU and I/O resources are underutilized in both cases. The polling interval is set to 2 seconds in both Setup A and B.
Measuremen	In order to obtain the time required to transfer an image from CLS to UWO, the transfer start time and end time are logged. They are the time to send message 1 and to receive message 4 shown in Figure 5.1 for both cases.
Source code	The source code of both implementations is available at https://github.com/dongliu/rsc/tree/master/xrd .

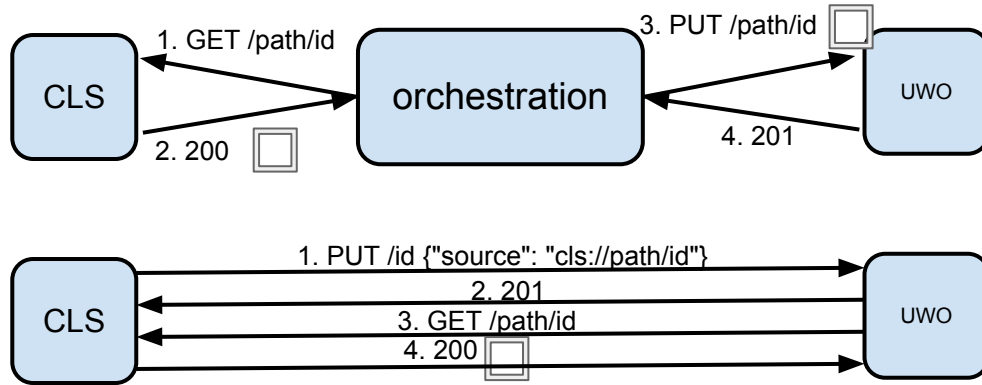


Figure 5.1: The message sequences of the orchestration and RSC-style implementations for XRD file transfer.

5.1.2 Other implementation details

There are three options for the XRD service orchestration’s internal design regarding how to put the image data obtained from the GET response message body into the POST request message body. The first option is to save the image data as a temporary file, and then set the file as the content source of the PUT request. Such a design is obviously resource-consuming, because two extra file I/O operations are needed. The second option is to put the content in memory. The tested orchestration is implemented in this way. The last option is to pipe the stream from the response directly to the request. Obviously, the last one is the most efficient among the three. However, it is also the most challenging technically. The program needs to be able to coordinate the connections and the receiving/sending of HTTP headers and body in two different exchanges. Frameworks like Netty² providing Application Programming Interfaces (APIs) on the socket level can be used for such implementations.

It will be more efficient to use file system events like a file previously opened for writing being closed to trigger the transfer of a new image. Libraries like `inotify` on Linux and `FileSystemWatcher` on Windows can be programmed to monitor such events. However, the monitors must run in the system kernel where the file operations happen. In XRD data acquisition, the images are collected by the CCD detector driver running on a Windows system

²See <https://netty.io/>

on a workstation, and they are written on a mapped drive that is also mounted on `srv-ibm-01` via Common Internet File System (CIFS). Therefore, a service must run on the Windows system to notify the I/O events related to XRD images. Such a design was not implemented because 1) the beamline scientist wanted the Windows workstation to be isolated to other services for the sake of reliability, and 2) we wanted the solution to work with other scans. Instead, the files inside a scan directory are periodically checked in order to find the new images to transfer. It is a little difficult for such a polling approach to tell whether an image is ready for transfer. In the implementations, an image is considered finished when its size does not change within a period³.

5.1.3 Measurement result and analysis

During an XRD scan, the images are generated at a constant rate, and no resources on the servers are overutilized for transferring them regardless of the scan size. Therefore, the transfer time is used as the performance metric rather than throughput. In the orchestration implementation, the orchestration service logs the time to start retrieving an image from the CLS side partner service and the finish time that the image is saved on the UWO side partner service. In the RSC implementation, the start time and finish time of a transfer have to be logged on different services. The start time of a transfer is logged on the CLS side service, and the finish time is logged on the UWO side service. In both cases, the transfer time for an image is the difference of the finish time and the start time. The time records do not need to be adjusted when calculating the difference in the RSC implementation because of the followings.

- The system clocks on the CLS and UWO systems are synchronized with network time servers.
- Ever if the difference between the two system clocks is not negligible, it does not affect the results of comparison between them, because this difference applies to the whole series of measurements.

³This period is the polling interval that is much longer than the time required to finish writing the file.

Figure 5.2 shows the time for transferring an image for RSC and orchestration implementations from CLS to UWO for different scan sizes. Each scan transfer is repeated for 10 times, and the measurement results are averaged. The services on CLS and UWO sides are reset to the same condition before each scan in order to get consistent behaviours. However, because the traffic went through wide area networks, the transfer time could be affected by the network load when the experiments were conducted.

Obviously, the RSC implementation performs better than the orchestration implementation consistently for various scan sizes. The RSC implementation saves about 0.7 seconds for each image. Although the network latency from CLS at Saskatoon to UWO at London, Ont. is larger than that within the CLS intranet, the time cost of exchanges 1 and 2 shown in Figure 5.1 in the orchestration implementation is still higher than the corresponding exchange of the RSC implementation. We can conclude that the structural change introduced by RSC brought the performance improvement by reducing network traffic.

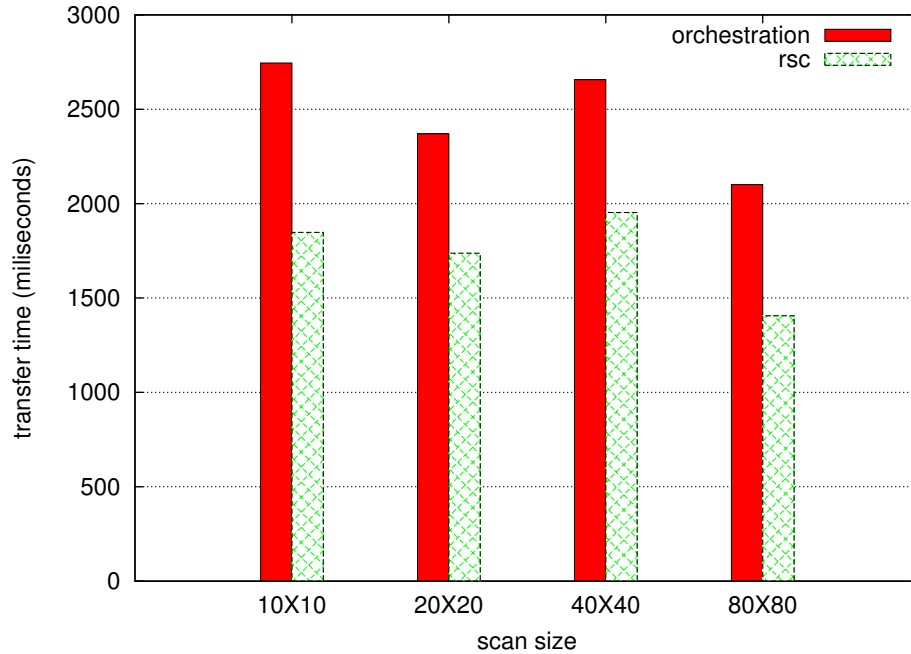


Figure 5.2: The image transfer time from CLS to UWO.

During the tests, the transfer time of the 10×10 scan, especially for the orchestration implementation, is constantly longer than that of other bigger scans. A close inspection of the collected data reveals that the transfer time of the images at the beginning is always

longer than the average of the whole scan. An explanation is that the congestion control mechanism of TCP contributes to the slowness of the leading message exchanges. In the tests, each simulated scan is started from a “cool” state, that is, there is a relative long period between two scans.

5.2 PV snapshot

In many big science facilities like CLS, the control systems are developed on the basis of real-time experimental control software. The Experimental Physics and Industrial Control System (EPICS)⁴ is a popular one for such purposes. The accelerator and beamline control systems at CLS are developed on the basis of EPICS. With EPICS, a control application is composed of a number of Process Variables. A beamline is normally controlled by several applications that can have hundreds of PVs. Logit is a control application developed for the BioMedical Imaging and Therapy (BMIT) beamline at CLS. The major goal is to provide a web-based application that can capture diverse experimental information on the beamline, including instrument states and control parameters available as EPICS PVs, image acquisition parameters, and other manually input records. A feature of Logit is to take a snapshot of a number of PVs. A snapshot should be taken as fast as possible because some PVs change frequently. The snapshot result is composed of the values of a list of PVs. The snapshot application is implemented as a service composition that is composed of services retrieving PV values.

5.2.1 PVs as restful services

A PV is a resource within an Input/Output Controller (IOC) in an EPICS system. The Channel Access (CA) is the application protocol that EPICS implemented to manipulate process variables [56]. Each PV is identified by a unique name like “SMTR1605-1-B10-10:BraggAngle:fbk”. The first part of a PV name separated by colons (:), “SMTR1605-1-B10-10”, is the IOC name. Similar to the slash (/) in a URL, the colon denotes the hierarchical

⁴See <http://www.aps.anl.gov/epics/>

Table 5.3: The semantics of common PV access methods.

Method	Semantics
<code>caget</code>	retrieve the current value of a PV
<code>caput</code>	update the value of a PV
<code>camonitor</code>	subscribe the changes of a PV
<code>cainfo</code>	retrieve the meta information of a PV

structure in a PV name. The example PV name refers to the Bragg angle feedback that is hosted on SMTR1605-1-B10-10. The CA protocol supports two message exchange patterns — request/response and subscribe/notify. The most common methods to access a PV includes `caget`, `caput`, `camonitor`, and `cainfo`. The basic semantics of these methods are listed in Table 5.3. Although the messages in PV access are neither self-descriptive (U4 in Table 2.1) nor hypertext (U5), PVs do follow the first three constraints (U1, U2, and U3) of the uniform interface. Therefore PVs can be considered as RESTful services. PV accesses involve mainly network I/O resources.

The snapshot service receives a snapshot request with a snapshot description via HTTP. The description is translated into a list of PVs in the service. The snapshot task is then mapped to a list of PV `caget` operations. The results from the `caget` operations are reduced to form the result of the snapshot. Obviously, the `caget` operations should be executed in parallel in order to get the snapshot instantly. A question naturally following is how to reduce the parallel `caget` results or errors.

5.2.2 Experiment setup

Five implementations are evaluated in this experiment. The first four implementations are service orchestrations programmed in Python. The fifth implementation is an RSC-style composition. The experiment setup to evaluate different implementations of the PV snapshot application is described in Table 5.4.

Table 5.4: The experiment setup to evaluate different implementations of the PV snapshot application.

Server environment	The application runs on a Scientific Linux 5.4 64-bit server named vsrv-bmitsql-01. vsrv-bmitsql-01 is a virtual machine on CLS VWware cluster. vsrv-bmitsql-01 is configured to run with two 2.2 GHz CPUs and 6 GB memory. It has Python 2.6.8, PyEpics ⁵ 3.2.1, node.js ⁶ 0.6.18 installed. The EPICS version is 3.14.9.
Implementations	Five implementations are developed and tested in the same environment and workload. The first four are based on the PyEpics library — a single-threading PV object-based implementation whose core part is shown in Listing 5.1, a single-threading low-level ca-based implementation shown in Listing 5.2, a multi-threading version of the PV-based one shown in Listing 5.3, and a multi-threading version of the ca-based one shown in Listing 5.4. The fifth implementation is a node.js implementation based on event-driven map-reduce shown in Listing 5.5.
Workload	Three different snapshot tasks are tested. The first task is to get the snapshot of 91 connected PVs. The second is to get that of 91 connected and 4 disconnected PVs. The third is of 91 connected and 15 disconnected PVs.
Measurement	The response time is measured in all the tests.
Source code	The source code of all implementations are available at https://github.com/dongliu/rsc/tree/master/snapshot .

Listing 5.1: A single-threading implementation of the snapshot based on the PV class.

```

for pv_name in pv_list:
    pv = PV(pv_name)
    if pv.wait_for_connection(timeout = 1.0):
        result[pv_name] = pv.get(use_monitor=False, timeout =
                                0)
    else:
        result[pv_name] = 'not connected'

# send the result

```

Listing 5.2: A single-threading implementation of the snapshot based on the ca class.

```

for pv_name in pv_list:
    ch = ca.create_channel(pv_name, connect = False, auto_cb =
                           False)
    result[pv_name] = [ch, None, None]
for pv_name, data in result.items():
    result[pv_name][1] = ca.connect_channel(data[0], timeout =
        1.0)
ca.poll()

```

```

for pv_name, data in result.items():
    if result[pv_name][1]:
        ca.get(data[0], wait = False)

ca.poll()
for pv_name, data in result.items():
    if result[pv_name][1]:
        val = ca.get_complete(data[0])
        result[pv_name][2] = val
    else:
        result[pv_name][2] = 'not connected'

# send the result

```

Listing 5.3: A multi-threading implementation of the snapshot based on the PV class.

```

def get(d, pv_name, size, start, pid):
    pv = PV(pv_name)
    if pv.wait_for_connection(timeout = 1.0):
        d[pv_name] = pv.get(use_monitor = False)
    else:
        d[pv_name] = 'not connected'
    if len(d) == size:
        # send the result
        os.kill(pid, signal.SIGTERM)

# ...
size = len(pv_list)
manager = Manager()
d = manager.dict()
pid = os.getpid()

for pv_name in pv_list:
    p = Process(target=get, args=(d, pv_name, size, start, pid)
    )
    p.start()

time.sleep(30)

```

Listing 5.4: A multi-threading implementation of the snapshot based on the ca class.

```

def get(d, pv_name, size, start, pid):
    ch = ca.create_channel(pv_name, connect=False, auto_cb=
        False)
    if ca.connect_channel(ch, timeout=1.0):
        d[pv_name] = ca.get(ch, wait=True)
    else:
        d[pv_name] = 'not connected'
    if len(d) == size:

```

```
        # send the result
        os.kill(pid, signal.SIGTERM)
# the rest is similar to the PV multi-threading version
```

Listing 5.5: A node.js implementation of the snapshot based on the EPICS `caget` command line tool.

```
size = pv_list.length;
pv_list.forEach(function(pv) {
  ca.exec('caget', pv, function(err, result) {
    complete = complete + 1;
    if (err) {
      results[pv] = {
        name: pv,
        value: 'unavailable'
      };
    } else {
      results[pv] = ca.parseCaget(result);
    }
    if (complete == size) {
      // return the results
    }
  });
});
```

5.2.3 Measurement result and analysis

The response time of the five implementations for three snapshot tasks is shown in Figure 5.3. The legend key “91c+4d” denotes a snapshot of 91 connected PVs and 4 disconnected ones. For the task of catching 91 connected PVs, the single-threading `ca` implementation performs the best. This is because the `ca.create_channel()` and `ca.get()` calls do not explicitly wait for the I/O events to any specific PV, instead, it uses `ca.poll` to get the events for all the PVs⁷. This strategy works fine for connected PVs, since the `ca.connect_channel()` calls to them always return instantly. However, when a PV is disconnected, a `ca.connect_channel()` call can only return when the one-second timeout happens. Because all the timeouts happen serially, the snapshot of 91 connected and 4 disconnected PVs takes about 4 more seconds than that of just 91 connected PVs. Similarly, the snapshot of 91

⁷See <http://cars.uchicago.edu/software/python/pyepics3/advanced.html> for more details

connected and 15 disconnected PVs takes about 15 more seconds.

The single-threading PV implementation behaves like the single-threading `ca` version except for two differences. Firstly, the PV implementation needs to create an object for each PV, which results in about two more seconds of response time than the `ca` implementation for the 91 connected PVs. Secondly, the PV implementation needs two seconds timeout for each disconnected PV, one for `pv.wait_for_connection()` and the other for `pv.get()` when there are disconnected PVs in the snapshot.

A straightforward approach for tackling the accumulating timeouts is to wait for each PV's connection timeout in parallel. Python provides two libraries to achieve parallelism — `threading`⁸ and `multiprocessing`⁹. The evaluated implementations are based on the latter library in order to make the Python implementations comparable with the `node.js` implementation. The `node.js` snapshot implementation achieves parallelism by spawning processes, which is the same as what the Python `multiprocessing` library does. Both the PV and `ca` parallel implementations successfully reduce the time required for waiting for the timeout. However, the cost of spawning a new process for each PV is significant and is proportional to the number of PVs in a snapshot. Two factors contribute to the cost — constructing the objects and manipulating the data created by the server process.

The performance of the `node.js` implementation is not as good as that of the single-threading `ca` implementation in the case of capturing a snapshot of only connected PVs. However, when there are disconnected PVs, the `node.js` implementation outperforms all other implementations in the tests. It performs almost the same for the snapshot of 15 disconnected PVs as that of 4 disconnected PVs. The extra response time caused by disconnected PVs are listed in Table 5.5. The column of “4d” lists the extra response time caused by 4 disconnected PVs, and the column of “10d” list that by 10 disconnected PVs. The scale factors can be calculated by Equation A.14. Although the difference between the scale factors is not big, it can result in significant response time difference for a large number of disconnected PVs, because the scale factor has an exponential effect on the performance metrics. In reality, the scale factor will accelerate degrading for non-scalable systems when the job scale

⁸See <http://docs.python.org/2/library/threading.html>

⁹See <http://docs.python.org/2/library/multiprocessing.html>

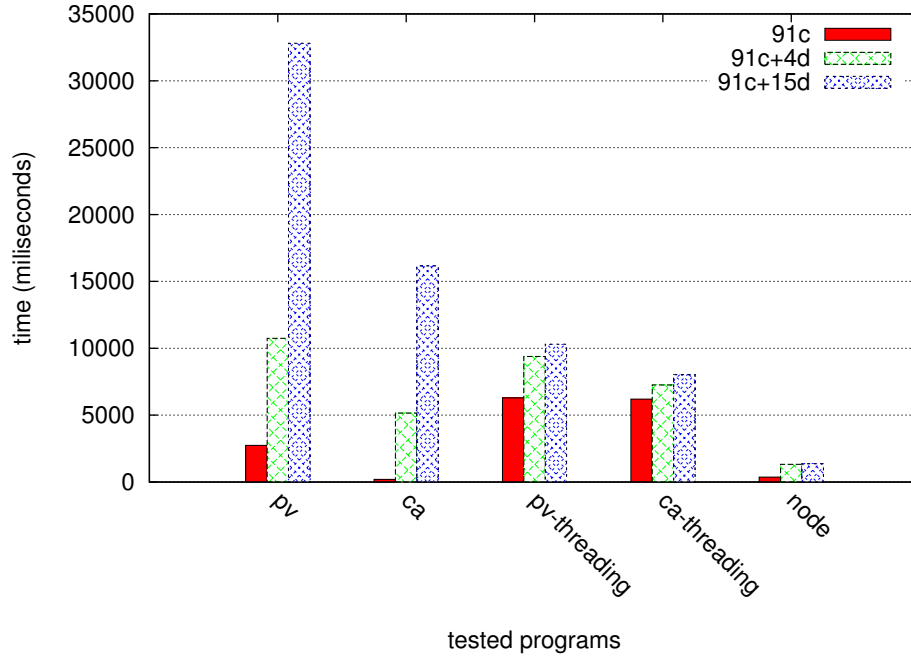


Figure 5.3: The response time of various implementations of the PV snapshot application for three different snapshot tasks.

Table 5.5: The response time in milliseconds for disconnected PVs and corresponding scale factor.

Implementation	4d	10d	Scale factor	15d(calculated)	15d(measured)
PV parallel	3,099	3,287	0.990	3,452	4,015
ca parallel	1,058	1,169	0.984	1,270	1,828
node.js	950	963	0.998	974	993

increases. The column of “15d(calculated)” lists the calculated extra response time caused by 15 disconnected PVs based on the scale factor. The column of “15d(measured)” have the extra response time from measurement. Obviously, the performance of Python implementations degrades faster than that of the node.js implementation.

By Amdahl’s law (see Appendix A), the speedup of a parallel application is bounded by the portion of a job that cannot be paralleled. For a multi-threading or multi-process program like the PV snapshot, the non-paralleled portion includes 1) to spawn threads/processes, 2) to map jobs to the threads/processes. The node.js implementation performs better than the parallel Python implementations because the following reasons:

- The node.js process is lighter than the Python ones. node.js interacts with the process

directly through `stdin`, `stdout`, and `stderr` in a non-blocking way¹⁰. On the other hand, the Python process is wrapped with a Python `Thread` API.

- The node.js implementation maps jobs when spawning processes. That takes shorter time than first creating a process and then starting it in the Python way.

According to Gunther's model (see Appendix A.3.2), the scalability of a parallel application is suppressed by the coherency between the concurrent parts — threads or processes. For map-reduce processing of the computation, the coherency is the reduce part that gathers the results from parallel processes to the master or server process. The RSC implementation in node.js performs the reduce by asynchronous callbacks, and no synchronization is required when manipulating the variable for the reduced results. The orchestration implementations in multi-threading Python have to deal with the synchronization of threads by using a `Manager` object.

5.3 Notification in the presence of network partitions

The top wrong assumption about distributed computing as discussed in Section 2.10 is that the network is reliable. The network can be considered reliable when it is homogeneous within a fully controlled boundary like an organization's intranet. When an application needs to operate across WAN, or some traffic goes through wireless networks, the developers should not assume that the network is reliable in the presence of network partitions. The reliability of service compositions will be more difficult to achieve in such environments than that of normal services. This section uses a simple notification application to show how RSC style can bring better reliability than the normal orchestration approach.

A large number of services belonging to an application are deployed in a network that can be divided into different areas. One of the application scenarios is to synchronize the states of all the services at a specific time. This requires sending notifications of adjusting the state to all services as soon as possible. In order to save time and other resources like network bandwidth and energy, no acknowledge is required. A composition is developed such that

¹⁰See http://nodejs.org/api/child_process.html

the operator needs to send the change state request to the composition, then the states of all services will be changed by the composition. The most straightforward way to implement the composition is a service orchestration that goes through the list of services and sends a notification to each of them. The working model of the orchestration implementation is shown in Figure 5.4 (a).

Sometimes the conductor service of the orchestration cannot send notifications to the services located in a different area when the connection between them is lost, or a network partition happens. In such cases, the notification can still be sent once the connection is recovered before timeout.

In an RSC implementation, the notifications are not sent out from only one centralized service. Instead, a notification task can be passed to a service in a different area, and then the service can send out notifications to other services within the same area, or forward a modified task to another service. This makes it possible for a service located in area A to send notifications to the services in area B, even when the connection between A and B is lost, if meanwhile A is still connected to area C and C is connected to B. The notification task can be passed from A to C then to a service inside B. The working model of the RSC implementation is shown in Figure 5.4 (b).

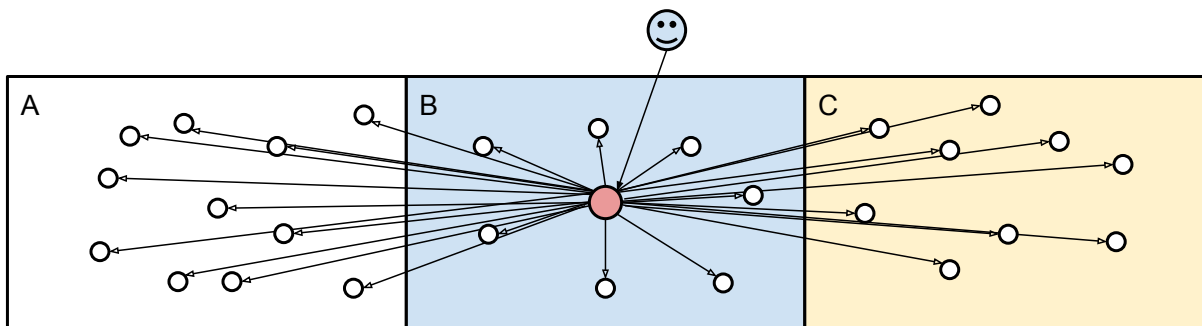
In both the orchestration implementation and the RSC implementation, the state of a notification can be modeled as shown in Figure 5.5. For the orchestration implementation, the “waiting” state refers to waiting for the connection to recover. For the RSC implementation, the “waiting” state refers to waiting for the notification task being routed and executed. This generic state machine is implemented in a simulation program.

5.3.1 Simulation parameters

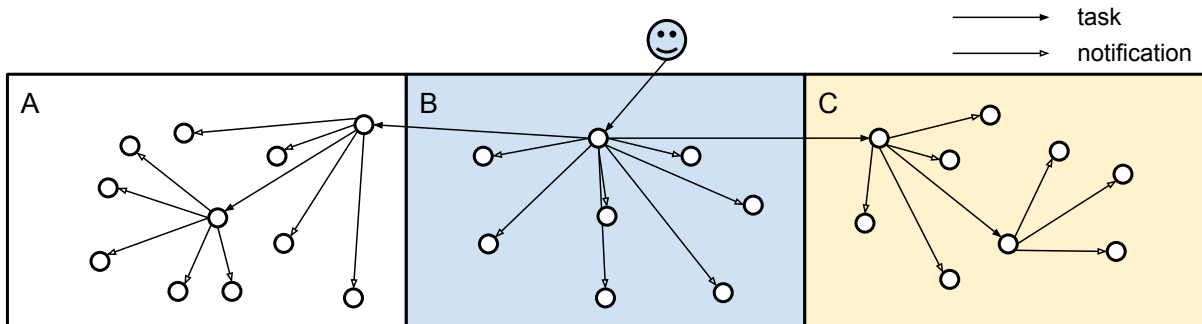
A simulation program is developed in node.js for the notification application in order to evaluate the impact of RSC on reliability. The source code is available at <https://github.com/dongliu/rsc/tree/master/notification>. Some popular simulation frameworks like SimPy¹¹ and MASON¹² can be used to develop the simulation. However, extra efforts of

¹¹See <http://simpy.sourceforge.net/>

¹²See <http://cs.gmu.edu/~eclab/projects/mason/>



(a) Orchestration implementation



(b) Rsc implementation

Figure 5.4: Two implementations for notification.

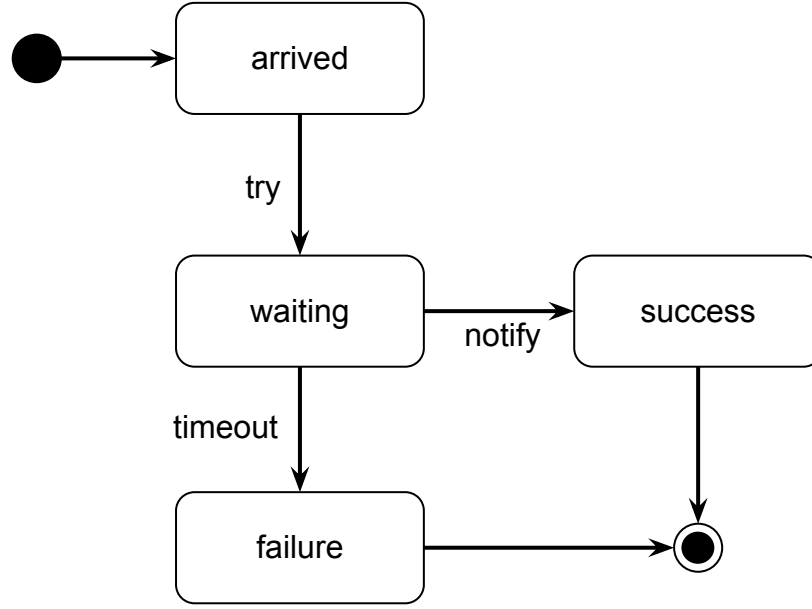


Figure 5.5: The state transitions of a notification task.

programming Python or Java are still required to simulate the behavior of the notification application. The simulation can be programmed naturally in an event-driven way based on the state machine model in Figure 5.5.

The important parameters and metrics of the simulation are described in Table 5.6. Table 5.7 shows the parameter values specified in the simulation.

5.3.2 Simulation results and analysis

The simulation is run with 1 million notification tasks for each availability option. Figure 5.6 (a) shows the failure ratios for two implementations in respect to different connection availability values from high to low. Figure 5.6 (b) shows the average time spent for a notification.

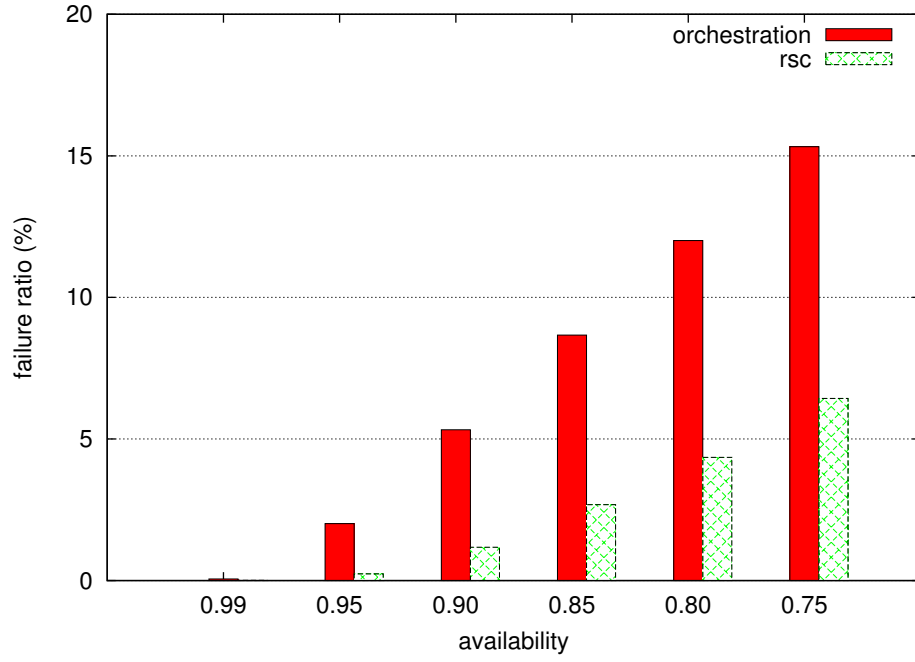
Obviously, the RSC implementation constantly provides higher success ratios than the orchestration implementation for various connection availabilities. This is because the probability of two network partitions is always lower than that of one at any given time. When the connection availability is very high, like 99%, the average time to deliver a notification for the orchestration implementation is shorter than that of the RSC implementation. For two

Table 5.6: The parameters and metrics for notification simulation.

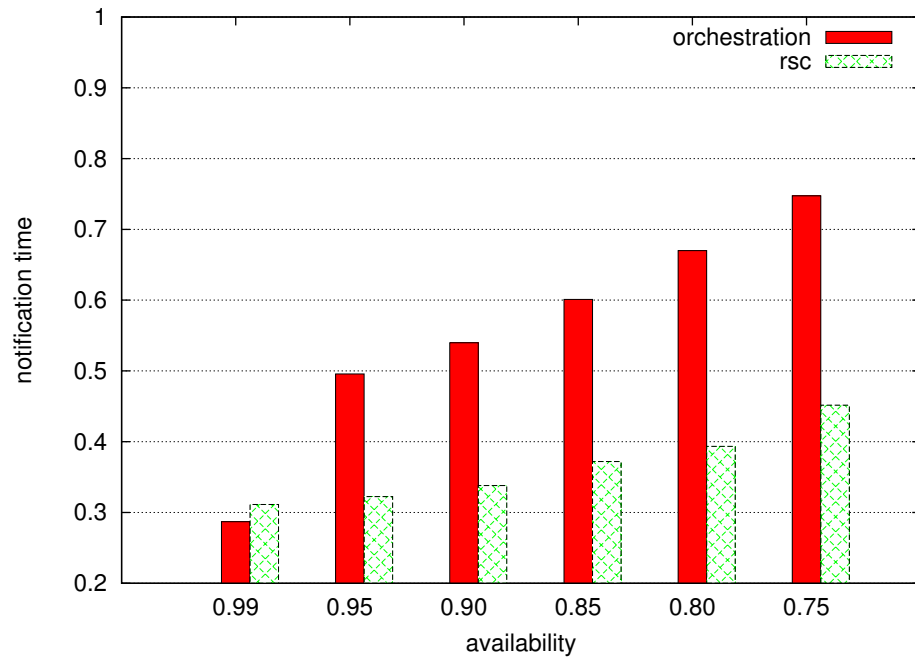
	Name	Notation	Assumptions
Parameters	composition request arrival rate	λ	The inter-arrival time follows exponential distribution with a parameter λ .
	number of areas	p	More areas imply higher probability that a notification can be affected by the unreliable connection between areas.
	number of services in P	n_P	The number of services to notify can be different for area P . In the simulation, they are set to be the same.
	concurrent connections	c_n	A service can set up a number of concurrent connections to other services and send notification in parallel. The service is perfectly scaled for a large number of connections, and the latency is not degraded because of this.
	timeout	T	Timeout occurs when either the notification cannot be delivered or the acknowledgment cannot be received. A notification fails when timeout occurs.
	transportation latency	L_{AB}	The one-way latency from a service in area A to the other in B follows a bounded Pareto distribution $Par(\alpha, L, H)$ [83]. Assume $L_{AB} = L_{BA}$ and $L_{AA} \ll L_{AB}$.
Metrics	connection availability	A_t^s	The connection from area s to area t can be lost because of partitions. By definition, $A_t^s = A_s^t$. The connection within the same area is always available, or $A_s^s = 1$. The availability is derived by MTTF and MTTR as in Equation 3.1. Both MTTF and MTTR are normally distributed $N(\mu, \sigma^2)$ [94].
	failure ratio	f	$f = \frac{failures}{failures+successes}$
	notification time	t	t is the time to deliver a notification. Assume the thinking time for a service to forward a notification task is negligible compared to the transportation latency.

Table 5.7: Parameter values specified for the simulations.

λ	p	n_P	c_n	T	L_{AB}	L_{AA}	MTTF	MTTR
0.25	3	10	30	2	$Par(1, 0.2, 1)$	$Par(1, 0.02, 0.1)$	$N(\mu, 1^2)$	$N(100 - \mu, 1^2)$
$\mu = 99, 95, 90, 85, 80, 75$. Correspondingly, $A = 0.99, 0.95, \dots, 0.75$.								



(a) Failure ratio



(b) Notification time

Figure 5.6: The failure ratio and latency of orchestration and RSC implementations obtained from simulation.

stages are required to deliver a notification to a service located in a different area, the first for the task and the second for the notification. However, when the connection availability gets lower, the penalty for the orchestration implementation to wait for a connection to recover becomes so significant that its average time is longer than that of the RSC implementation.

5.4 Modification of the XRD file transfer service

During the development of Science Studio, the requirements kept changing from GUI to work flow to back end services. These requirement changes were mainly driven by tasks like application integration, and improving performance and reliability. This section reviews the requirement changes related to the XRD file transfer service, and compare the modifiability of orchestration implementation and RSC implementation regarding these requirement changes.

5.4.1 Emerging requirements and corresponding modifications

The emerging requirements are listed according to the time sequence as follows.

1. Retrieve the information of all the finished scans on the CLS service.
2. Transfer a finished scan from CLS to UWO by the image name pattern.
3. Transfer a finished scan from ALS¹³ located at Berkeley, CA to UWO.
4. Verify if a transferred scan is identical to the source, and identify the modified images.
5. Allow patch option when transferring a scan.

Requirement 1

Each XRD image processing requires a set of processing configuration parameters. The parameters need to be tuned in order to find interesting pattern information from the images. The scientists and users wanted to extend the processing capacity to finished scans. The first step for doing that is to list all finished scans hosted on the CLS service. The images are

¹³See <http://www-als.lbl.gov/>

always named in a format like `prefix_num.ext`. The start number of a scan is configured by the user, and it is then automatically incremented by one for every sequential new image during a scan. The information about a scan should include the image name prefix, image type, the start number, the end number, the missing images between the start and the end. For both implementations, the feature should be provided by the partner services.

Requirement 2

When a user wants to process a finished scan on the CLS service, the scan needs to be transferred from CLS to UWO. The transfer request contains the parameters for the name pattern composed of the prefix, start number, end number, missing numbers, and image type. The conductor service in the orchestration implementation needs to construct the list of URLs. The conductor service sends a **GET** request to the CLS service for each image and then **PUT** it to the UWO service. In the case of RSC implementation, the UWO service accepts a **POST** request with the name pattern parameters. When the URL list is constructed, it can then **GET** the images in the list.

Requirement 3

When the XRD processing service at UWO was able to process the images produced at CLS, the scientist and users at a beamline of ALS also wanted to use the service to process their finished XRD scans. In the orchestration implementation, a new partner service needs to be deployed on the ALS server. The network distance between CLS and ALS is much longer than that between UWO and ALS, and transferring an image from ALS to CLS and then to UWO is much slower than from ALS to UWO. Therefore, the conductor service should also be deployed on ALS. For the RSC implementation, a standard service like the CLS one can be deployed on the ALS server.

Requirement 4

An image can be corrupted during transfer, and it can be modified accidentally after being transferred to the destination file system. To ensure the integrity of a scan before the processing, users want to verify that the scan on the UWO server is identical to its source. Comparing

only the size of corresponding files cannot ensure the integrity. The MD5 checksum is used for this. In the orchestration implementation, the conductor service can get two MD5 digests of the same image from two services, and then compare the digests. This requires the partner services to be able to generate MD5 digests for the images in a given scan. For the RSC implementation, the UWO service **GET** the MD5 digests of the image in a scan from the CLS/ALS service, and then do the check locally. Most checksum tools and libraries support both computing and checking the digests of given files.

Requirement 5

When a scan is already transferred, it will waste both the time and network bandwidth to transfer all the images again. Only the missing or different images should be transferred from the source. Obviously, this feature relies on the solution for Requirement 4.

5.4.2 Modifiability analysis

The interface changes and reuse related to the requirements are compared in order to evaluate the modifiability of the orchestration implementation and the RSC implementation. R0 represents the original requirement for real-time processing.

The modifiability of an application can be evaluated by the cost required to modify it for new or changed requirements. The cost is threefold: implementation, deployment and maintenance. The reuse of services can greatly reduce the implementation cost, which is also an essential value of SOA. For the orchestration implementation, only 3 interfaces out of 9 (4 for the conductor service and 5 for the partner services) are reused for the 6 requirements. On the other side, 4 interfaces of the total 8 are reused for RSC. Approximately, that will result in 17% cost saving for the RSC implementation.

In the orchestration implementation, the source code is located in two packages, one for the conductor service and the other for the partner service. They need to be packaged separately and also deployed to different contexts on a server. On the contrary, the RSC has only one package for all the services. In order to enable ALS to use the processing service, both the conductor service and the partner service need to be deployed on their server in the case of orchestration, while only one service is enough for the RSC implementation. When

more and more facilities want to consume the processing service, to deploy and maintain two services will definitely cost more than only one service.

5.5 Summary

This chapter presents four application scenarios to evaluate the performance, scalability, reliability and modifiability improvement brought by RSC to service composition applications. In all these scenarios, RSC style implementations performed better in each evaluation perspective than the corresponding orchestration implementations. RSC's design goals were evidenced by these results. However, in order to obtain such benefits, the developers will need to adapt to programming paradigms like event-driven programming and functional programming. These evaluation scenarios demonstrate how to apply the RSC architectural style and corresponding programming style to real design problems.

CHAPTER 6

CONCLUSIONS

We drive into the future using only our rearview mirror.

—Marshall McLuhan

My discussion of RESTful Service Composition (RSC) began with the review of distributed computing history. RPC, CORBA, and SOAP have been hyped one after the other. Although the popularity of RPC and CORBA has declined, one can still find their tracks in various programming libraries and applications. The SOAP technology led the migration for distributed computing technologies toward so called Web Services and Service-Oriented Architecture.

SOAP took advantage of three successful technologies — RPC, OOP, and XML. RPC provides a convenient way to distributed programming through code generation. OOP brought the power of new languages and libraries to distributed programming. XML makes the messages exchanged between distributed programs explicit. However, these technologies also brought their intrinsic shortcomings to SOAP. Some essential characteristics of distributed computing, like time out and partial failures, can be easily ignored by the programmers when using RPC. OOP adds more complexity to the compatibility aspects. XML messages generated by the machines are beyond human legibility.

An important but often neglected technology contributing to the success of SOAP is HTTP. Without HTTP, SOAP could only be an EAI technology like CORBA or DCOM. HTTP boosted SOAP to the application environment of Internet or the Web. However, HTTP was used just as a transportation mechanism, which is far below what HTTP can do. Similarly, the “web” used in Web Services is far below what the real Web does.

Almost everyone knows the basics of how the Web works like a browser talking to the servers, yet not too many really understand how it was designed. Fielding summarized the

style as Representational State Transfer (REST) based on his experiences of developing the Apache httpd, one of the most popular HTTP servers, and several important web specifications. REST provided a foundation for the developers to review how to design distributed services based on the pros and cons of SOAP. It is also the foundation for my examination of the existing web service composition technologies.

Service orchestration is the dominating service composition technology in Web Services. It overwhelmed the service choreography approach from the beginning because of its simplicity. The w3C choreography committee did not produce a final specification because of the lack of industrial support. The typical hub-spoke structure of service orchestration makes the central conductor service the hot spot regarding performance, scalability, reliability and modifiability. The RESTful Service Composition (RSC) approach was proposed to addressing this problem from an architectural perspective.

The following are the contributions made in this thesis:

- an analysis of the performance, scalability, reliability and modifiability issues of service orchestration from the foundational technologies to high-level specifications;
- RSC, a novel architectural style for developing service compositions and a corresponding programming model to support such a style; and
- an evaluation of RSC by a broad group of scenarios from real-world applications covering all design goals.

This thesis connects the high-level abstract architectural aspects of service composition designs to the low-level practical programming aspects of application implementations. It can help both software architects and also programmers to understand the essentials of RSC. I believe such an approach is a valuable contribution to both research and development related to software architecture.

Since REST was proposed by Fielding, it has been endorsed by more and more parties from both industry and academia. It offers the values of simplicity, transparency, performance, scalability and reliability that make RESTful services popular in the community. Although not all REST constraints are followed, the services tagged with REST generally feature better

simplicity and transparency than the traditional SOAP approaches. Some constraints of RSC, like the uniform identifier and same access methods, are already used in service compositions, including both server-side compositions and client-side mashups. Spring Web Flow¹ is an application framework that supports the notion of staged computation in RSC.

Among all RSC constraints, baton passing is the most difficult one to implement. It requires partner services to be able to interpret a computation representation and execute the representation properly. It is like the case for the hypermedia as the engine of application state constraint in REST, which is considered the most difficult to achieve for a RESTful service². The complexity arises from the fact that the client portion of such an application needs to handle either computation representation or hypermedia. As we know, a web client is more complex than a web server.

In order to overcome such complexities, the community needs to shift the development focus from the server side to the client side by providing more libraries and tools to help the development of powerful clients. Since a function is a key abstraction in RSC, the programmers also need to shift their programming paradigm from OOP to functional in order to implement RSC-style service compositions naturally. The research presented in this thesis can be continued in these directions.

I believe that the complexity and difficulty of distributed computing can be addressed when a technology provides not only an appropriate means for computer-computer communication, but also an appropriate means for developer-computer and developer-developer communication. RSC is an attempt I made in this thesis.

¹See <http://www.springsource.org/spring-web-flow>

²See <http://martinfowler.com/articles/richardsonMaturityModel.html>

APPENDIX A

SCALABILITY

A.1 Performance, capacity and scalability

When a system is stable, or more strictly the corresponding stochastic process is stationary, the mean number of concurrent requests or jobs (N) in the system follows Little's law [70].

$$N = X \times R \quad (\text{A.1})$$

where R is the mean residence time and X is the throughput. When the system is stable, the throughput equals to the arrival rate, or $X = \lambda$. The arrival rate is a measurement of the work load.

Obviously, N will increase when λ increases given that R remains unchanged or increases with N . The maximum that N can reach when the system is still stable is the system's *capacity*.

$$C = N_{max} = X_{max} \times R(N_{max}) \quad (\text{A.2})$$

A simple way to increase a system's capacity is to add a job queue. However, if the server is slow compared to job arrival, then the queue can be overflowed in a short time. Furthermore, the residence time will be increased because of waiting time as a result of increased queue size.

$$R = W + S \quad (\text{A.3})$$

where W is the waiting time that a job spent in the queue before served, and S is the service time.

A system's performance can be measured by either X or R in Little's law. Throughput and residence time represent different characteristics of a system. Throughput is always driven by arrival rate and its upper bound is constrained by capacity. In practice, only stress testing can reveal the maximum of throughput X_{max} . On the other side, residence time is driven by the number of concurrent jobs in a system and its lower bound is constrained by a job's serial fraction of demand. Parallel processing can only reduce the parallel fraction of demand. Residence time R is a more user-centric metric of performance than throughput X . For a multi-user system, an end user's experience is mainly decided by user perceived response time, which is a sum result of residence time and transmission time. Sometime, the residence time can vary a lot while the throughput is about the same. Figure 2.6 shows the relationship between X , R and N in a stress testing of a web service.

Scalability is quite different from performance that can be measured by throughput, residence time, and capacity. It focuses on the *change* of performance along with system resource being utilized. This difference yields the difficulties of scalability measurement, that is, a single scalability measurement needs a series of performance measurements of a system in different configurations. This chapter discusses the mathematical model for scalability and its metrics.

A.2 Definition of scalability

A pure mathematical definition of scalability can be derived from the metrics in Little's law. If we choose the number of jobs in the system, or N in Equation A.1, the scalability can be defined as

$$S = dN/dr, \quad (\text{A.4})$$

where S is the scalability and r is the resource. Considering that most resources are discrete, the equation can be rewritten as

$$S = \Delta N / \Delta r = \frac{N_{n+1} - N_n}{r_{n+1} - r_n}, \quad (\text{A.5})$$

where n is the count of used resource and $n \in \mathbb{N}$. The above equation can be simplified as

$$S = N_{n+1} - N_n, \quad (\text{A.6})$$

when the unit of resource is 1. Similarly, if we choose throughput X , the scalability can be defined as

$$S = \frac{X_{n+1} - X_n}{r_{n+1} - r_n}. \quad (\text{A.7})$$

Note that these two metrics only make sense when the other performance metric R in Little's Law does not change much from R_n to R_{n+1} correspondingly.

When two systems' scalability need to be compared, it is tricky to use the above metrics because one system's initial performance metrics, N_1 , X_1 and R_1 , can be quite different from the other. This issue can be addressed by using relative metrics. Assume a system is perfectly scalable, and every used unit of resource makes the system's performance increased by the same amount.

$$X_{n+1} - X_n = X_n - X_{n-1} = \dots = X_2 - X_1 = X_1, \quad (\text{A.8})$$

or

$$X_{n+1}/(n+1) = X_n/n = \dots = X_2/2 = X_1. \quad (\text{A.9})$$

This can be improved by adding a scaling factor to be more realistic.

$$X_{n+1}/(n+1) = sX_n/n = s^2X_{n-1}/(n-1) = \dots = s^{n-1}X_2/2 = s^nX_1 \quad (\text{A.10})$$

The scaling factor s can be used to measure scalability. s can be calculated when we have any 2 measurements of X , X_m and X_n . For most systems, s is always less or equal to one.

$$X_n/n = s^{n-m}X_m/m \quad (\text{A.11})$$

yields

$$s = \left(\frac{mX_n}{nX_m} \right)^{1/(n-m)}. \quad (\text{A.12})$$

When comparing the scalability of two systems, it is better to choose the measurements of X at same scales to calculate the corresponding scaling factors. When one has k measurements of X , s can be calculated by non-linear regression of the $k - 1$ independent equations. The non-linear regression can be simplified to linear regression by applying logarithm to both sides of Equation A.12.

$$(n - m) \ln s = \ln(X_n/n) - \ln(X_m/m) \quad (\text{A.13})$$

The scale factor can also be calculated by residence time R .

$$(n - m) \ln s = \ln R_m - \ln R_n \quad (\text{A.14})$$

The other relative scalability metric is speedup, which is the ratio of R_1 to R_n for vertical scaling, or the ratio of X_n to X_1 for horizontal scaling.

$$S_n = \frac{R_1}{R_n} \text{ or } \frac{X_n}{X_1} \quad (\text{A.15})$$

A.3 Scalability models

A.3.1 Amdahl's law

The map-reduce approach requires a server to be able to do parallel processing. The demand of a request task can therefore be distributed to many processes that running on different processing units. The optimal result will be the residence time be decreased to $\frac{1}{n}$ of the original, where n is the number of parallel processes. However, by Amdahl's Law [4], the speedup,

$$S = \frac{R_1}{R_n} \quad (\text{A.16})$$

will be always less than $\frac{1}{1-P}$ no matter how large n is.

$$R_n = ((1 - P) + P/n)R_1 \quad (\text{A.17})$$

$$S(n) = \frac{1}{(1 - P) + P/n} \quad (\text{A.18})$$

where P is the portion of a job that can be parallel processed. When n is extremely large, P/n will be close to 0, and S converges towards $\frac{1}{1-P}$.

A.3.2 Gunther's scalability model

Gunther extended Amdahl's Law and proposed a more generic model for scalability [47, 48]. In Gunther's model, a system's capacity is characterized by three factors: concurrency, con-

tention, and coherency. The concurrency is the number of workers that process a job in parallel. A worker can be a thread, or a process, or a cluster node in different implementations. The contention is the portion of a job that cannot be processed in parallel. The coherency is the penalty caused by parallelism, for example, concurrency accesses of the same memory block, database, or a service. When a big number of threads are created on a system, the sharing of memory and thread scheduling service is also coherency.

In Gunther’s model, the scalability of a system is quantified by its capacity.

$$C(p) = \frac{p}{1 + \sigma(p - 1) + \kappa p(p - 1)} \quad (\text{A.19})$$

where C is the capacity, p is the number of parallel processes — concurrency, σ is the contention, and κ is the coherency. When there is only one process in the system, $C = 1$, which is the baseline of the system’s capacity. If there is not coherency, that is $\kappa = 0$, the Gunther’s model is exactly the same as Amdahl’s law, except the notations.

$$C(p) = \frac{p}{1 + \sigma(p - 1)} = \frac{1}{(1 - P) + P/p} \quad (\text{A.20})$$

where $\sigma = 1 - P$. The $\kappa p(p - 1)$ results in a quadratic effect on a system’s behaviour so that the performance or capacity drop when the coherency penalty becomes large enough. Figure 2.6 shows such a behaviour. Concurrency, contention and coherency denote the strategy, constraint, and pitfall when scaling up a system.

REFERENCES

- [1] Fipa abstract architecture specification, Dec 2002. <http://www.fipa.org/specs/fipa00001/SC00001L.html>.
- [2] W.B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, feb 1982.
- [3] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.
- [4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS spring joint computer conference*, 1967.
- [5] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 293–302, New York, NY, USA, 1989. ACM.
- [6] Andrew W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006.
- [7] Architecture Working Group (AWG). Ieee recommended practice for architectural description of software-intensive systems, Sept. 2000.
- [8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.
- [9] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, Dec. 2003.
- [10] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, Raleigh, North Carolina, July 2007.
- [11] Henry Baker and Carl Hewitt. Laws for Communicating Parallel Processes. Technical report, MIT Artificial Intelligence Laboratory, May 1977.
- [12] Abhijit Belapurkar. Functional programming in the java language, July 2004.
- [13] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. Yaml aint markup language (yaml) version 1.2. Web, May 2008. <http://yaml.org/spec/1.2/>.
- [14] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – http/1.0. Web, May 1996. <http://tools.ietf.org/rfc/rfc1945.txt>.
- [15] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax, January 2005. <http://tools.ietf.org/rfc/rfc3986.txt>.

- [16] Tim Berners-Lee. *Weaving the Web*. Harper, 1999.
- [17] Tim Berners-Lee, Tim Bray, Dan Connolly, Paul Cotton, Roy Fielding, Mario Jeckle, Chris Lilley, Noah Mendelsohn, David Orchard, Norman Walsh, and Stuart Williams. Architecture of the world wide web, volume one. Web, Dec. 2004. <http://www.w3.org/TR/2004/REC-webarch-20041215/>.
- [18] André B. Bondi. Characteristics of scalability and their impact on performance. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, New York, NY, USA, 2000. ACM Press.
- [19] Don Box. A young person’s guide to the simple object access protocol: Soap increases interoperability across platforms and languages. *MSDN Magazine*, March 2000.
- [20] Don Box. Code name indigo: A guide to developing and running connected systems with indigo. *MSDN Magazine*, January 2004. <http://msdn.microsoft.com/msdnmag/issues/04/01/indigo/default.aspx>.
- [21] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. web, May 2000.
- [22] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0. web, Nov. 2008. see <http://www.w3.org/TR/2008/REC-xml-20081126/>.
- [23] Tim Bray and C. M. Sperberg-McQueen. Extensible markup language (xml). Web, Nov. 1996.
- [24] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134 vol.1, mar 1999.
- [25] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM.
- [26] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin / Heidelberg, 2007.
- [27] Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. Technical report, W3C, 2006.
- [28] K. Mani Chandy and Charles H. Sauer. Approximate methods for analyzing queueing network models of computing systems. *ACM Comput. Surv.*, 10:281–317, September 1978.

- [29] Vivek Chopra and Sing Li. *Professional Apache Tomcat 6 (WROX Professional Guides)*. Wrox Press Ltd., Birmingham, UK, UK, 2007.
- [30] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Web, March 2001.
- [31] David H. Crocker. Standard for the format of arpa internet text messages. Web, August 1982.
- [32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [33] L. Dusseault and J. Snell. Patch method for http. Web, March 2010.
- [34] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall PTR, 2007.
- [35] Jeff A. Estefan, Ken Laskey, Francis G. McCabe, and Danny Thornton. Reference model for service oriented architecture v 1.0. Web, July 2008. <http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf>.
- [36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999. <http://www.ietf.org/rfc/rfc2616.txt>.
- [37] Roy Fielding. The rest of rest. RailsConf Europe, Sept. 2007. See http://roy.gbiv.com/talks/200709_fielding_rest.pdf.
- [38] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
- [39] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.
- [40] Lars-Ake Fredlund. Implementing ws-cdl. In *proceedings of JSWEB 2006*, Santiago de Compostela, Spain, 2006.
- [41] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part two: Media types. Web, Nov. 1996.
- [42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [43] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

- [44] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.
- [45] Daniel Gross and Eric Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, February 2001.
- [46] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2 part 2: Adjuncts. Web, April 2007. <http://www.w3.org/TR/soap12-part2/>.
- [47] Neil J. Gunther. Unification of amdahl’s law, logp and other performance models for message-passing architectures. In *IASTED PDCS*, pages 569–576, 2005.
- [48] Neil J. Gunther. *Guerrilla capacity planning: a tactical approach to planning for highly scalable applications and services*. Springer, 2006.
- [49] Steven Haines. *Pro Java EE 5 Performance Management and Optimization*. Apress, 2006.
- [50] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
- [51] Jeffrey Hasan and Kenneth Tu. *Performance Tuning and Optimizing ASP.NET Applications*. Apress, 2003.
- [52] Michi Henning. The rise and fall of corba. *Queue*, 4:28–34, June 2006.
- [53] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
- [54] Ian Hickson. Html5 a vocabulary and associated apis for html and xhtml, May 2011.
- [55] Hal Hildebrand, Anish Karmarkar, Mark Little, and Greg Pavlik. The session concept and web services. In *XML 2005*, 2005.
- [56] Jeffrey O. Hill and Ralph Lange. *EPICS R3.14 Channel Access Reference Manual*, July 2011. See <http://www.aps.anl.gov/epics/base/R3-14/12-docs/CAref.html>.
- [57] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.
- [58] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [59] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

- [60] Intel. Excerpts from a conversation with gordon moore: Moores law. Web, 2005. ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf.
- [61] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.
- [62] Dan Kegel. The c10k problem. Web, Sept. 2006. <http://www.kegel.com/c10k.html>.
- [63] Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 177–190, New York, NY, USA, 2007. ACM.
- [64] Rohit Khare and Richard N. Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 428–437, Washington, DC, USA, 2004. IEEE Computer Society.
- [65] Todd Kitta. *Professional Windows Workflow Foundation*. Wrox Press Ltd., Birmingham, UK, UK, 2007.
- [66] Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus von Riegen, Patrick Schmidt, and Ivana Trickovic. Ws-bpel extension for people bpel4people. whitepaper, IBM and SAP, 2005.
- [67] D. Kristol and L. Montulli. Http state management mechanism. Web, Feb. 1997.
- [68] Philippe Kruchten. Architectural blueprints—The “4+1” view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.
- [69] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.
- [70] John D. C. Little. A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.
- [71] D. Liu, E. D. Matias, D. Maxwell, D. Medrano, M. Bauer, M. Fuller, N. S. McIntyre, Y. Yan, C. H. Armstrong, and J. Haley. Science studio: A project status update. In *Proceedings of ICALEPCS2009*, Kobe, Japan, 2009.
- [72] Dong Liu and Ralph Deters. Bust: enabling scalable service orchestration. In *InfoScale '07: Proceedings of the 2nd international conference on Scalable information systems*, pages 1–10, ICST, Brussels, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
- [73] Dong Liu and Ralph Deters. Management of service-oriented systems. *Service Oriented Computing and Applications*, 2:51–64, 2008.

- [74] Dong Liu and Ralph Deters. The reverse c10k problem for server-side mashups. In *Service-Oriented Computing – ICSOC 2008 Workshops*, pages 166–177, 2009.
- [75] Dong Liu, Ralph Deters, and W. J. Zhang. Architectural design for resilience. *Enterprise Information Systems*, 4(2):137–152, May 2010.
- [76] E.A. Luke. Defining and measuring scalability. In *Proceedings of the Scalable Parallel Libraries Conference, 1993*, pages 183–186. IEEE, 1993.
- [77] Stuart E. Madnick. The misguided silver bullet: What xml will and will not do to help information integration. In *Proceedings of the Third International Conference on Information Integration and Web-based Applications and Services, IIWAS2001*, pages 61–72, 2001.
- [78] D. G. Maxwell, D. Liu, E. Matias, D. Medrano, M. Bauer, M. Fuller, S. McIntyre, and J. Qin. Remote access to the vespers beamline using science studio. In *Proceedings of PCaPAC 2010*, pages 118–120, Oct. 2010. <http://accelconf.web.cern.ch/AccelConf/pcapac2010>.
- [79] Nilo Mitra and Yves Lafon. Soap version 1.2 part 0: Primer. Web, April 2007. <http://www.w3.org/TR/2003/REC-soap12-part0-20030624/>.
- [80] msdn. Windows workflow foundation. 2007.
- [81] OASIS. Web services business process execution language version 2.0. Web, Feb 2007. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>.
- [82] Chun Ouyang, Marlon Dumas, Arthur H. M. ter Hofstede, and Wil M. P. van der Aalst. From bpmn process models to bpel web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 285–292, Washington, DC, USA, 2006. IEEE Computer Society.
- [83] Kihong Park, Gitae Kim, and Mark Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Proceedings of the 1996 International Conference on Network Protocols (ICNP '96)*, ICNP '96, pages 171–, Washington, DC, USA, 1996. IEEE Computer Society.
- [84] Cesare Pautasso. Restful web service composition with bpel for rest. *Data Knowl. Eng.*, 68:851–866, September 2009.
- [85] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big web services: Making the right architectural decisions. In *17th International World Wide Web Conference (WWW2008)*, pages 805–814, Beijing, China, April 2008 2008.
- [86] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.
- [87] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

- [88] Paul Prescod. Roots of the rest/soap debate. web, 2002. see http://www.prescod.net/rest/rest_vs_soap_overview/.
- [89] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. Html 4.01 specification, Dec. 1999.
- [90] Florian Rosenberg, Francisco Curbera, Matthew J. Duftler, and Rania Khalaf. Composing restful services and collaborative workflows: A lightweight approach. *IEEE Internet Computing*, 12:24–31, 2008.
- [91] Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. Web, May 2006.
- [92] Donald W. Sherburne. *A key to Whitehead's Process and reality. Edited by Donald W. Sherburne*. Macmillan New York,, 1966.
- [93] N. Sherry, J. Qin, M. Suominen Fuller, Y. Xie, O. Mola, M. Bauer, N. S. McIntyre, D. Maxwell, D. Liu, E. Matias, and C. Armstrong. Remote internet access to advanced analytical facilities: A new approach with web-based services. *Analytical Chemistry*, 2012.
- [94] Martin L. Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. Wiley-Interscience, Malden MA, 2002.
- [95] R. Srinivasan. Rpc: Remote procedure call protocol specification version 2, August 1995. see <http://www.ietf.org/rfc/rfc1831.txt>.
- [96] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional, 1992.
- [97] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 291–302, New York, NY, USA, 2005. ACM Press.
- [98] Steve Vinoski. Convenience over correctness. *IEEE Internet Computing*, 12(4):89–92, 2008.
- [99] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.
- [100] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical report, Sun Microsystems Laboratories, 1994. http://research.sun.com/techrep/1994/sml_i_tr-94-29.pdf.
- [101] Matt Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California, Berkeley, August 2002.
- [102] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.

- [103] James E. White. A high-level framework for network-based resource sharing. web, 1976.
- [104] James E. White. A high-level framework for network-based resource sharing. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, AFIPS '76, pages 561–570, New York, NY, USA, 1976. ACM.
- [105] Alfred North Whitehead. *Process and reality : an essay in cosmology*. Cambridge University Press, 1929.
- [106] WS-I. Basic profile version 2.0. Web, Nov. 2010. See <http://ws-i.org/Profiles/BasicProfile-2.0-2010-11-09.html>.
- [107] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls*. OReilly, 2nd edition, 2000.