Flinders UNIVERSITY
ADELAIDE · SOUTH AUSTRALIA

Continuous Integration for Decentralised Development of Service-Based Software Systems

by

Jameel Almalki

Thesis
Submitted to Flinders University
for the degree of

Doctor of Philosophy
College of Science and Engineering
9th August 2018

Declaration

I certify that this thesis does not include any prior material submitted for a degree or diploma in any university without acknowledgment; and to the best of my knowledge and belief it does not include any prior material published or written by another individual with the exception of where due reference is used in the text.

Jameel Almalki

July 30th, 2018

Acknowledgements

Firstly and most importantly, I want to thank Allah Almighty for giving me blessings, opportunities, strength and guidance to endure this challenging task to the end.

My unreserved appreciation and gratitude go to my parents and my brothers and sisters for their continuous support, prayers and encouragement, which gave me the strength to succeed in my PhD research journey. May Allah bless you all.

My greatest indebtedness is to my wife, Hamdah, for her endless love, support, patience and sacrifice while I have been consumed with this research in recent years. My thanks and love are also extended to my two precious daughters, Diala and Ladan, and my son, Elias, for all the joy and love in my life. Thank you very much, you are indeed my treasures.

I also would like to express my deepest gratitude and appreciation to my supervisor, Dr. Haifeng Shen, for his enthusiastic support, insightful suggestions and valuable guidance throughout this research. Certainly, without his encouragement, invaluable knowledge and support this work would never have been completed.

Last but not least, I would like to thank my friends and PhD colleagues who supported me during the development of my research and who gave me assistance whenever I needed it during the completion of this work.

Abstract

With the rapid adoption of Service-Oriented Architectures (SOA) for the creation of distributed software applications, it is important for software development processes to be supported with relevant methodologies and tools. Traditional approaches to designing and implementing 'monolithic' software architectures fail to address these needs for developing distributed, service-oriented applications. In this light, the discipline of Service-Oriented Software Engineering has emerged to address the development and operation of distributed and loosely-coupled software systems, composed of reusable services often provided by third parties.

A key component the development and operation of SOA systems is Continuous Integration whereby pieces of a software application are integrated and tested as they change. Continuous Integration in SOA-based software systems introduces a whole new level of complexity. For Continuous Integration to work in this context, network-accessible services need to be dynamically and continuously discoverable to be invoked through their APIs to ensure that the individual elements always work together. Enabling Continuous Integration in a centralised software development environment is relatively easy – that is, a dedicated Continuous Integration server can monitor the central software repository and execute integration tests whenever changes are detected. Continuous Integration in SOAs is not so straight-forward, since different parts of a distributed service-based application might be owned and managed by different software development teams and organisations, thereby restricting centralised storage, testing, deployment and execution. In these circumstances, a decentralised approach to Continuous Integration is required to fulfil this practice in the context of API-driven SOA environments.

To this end, this thesis puts forward the following research question: **How can Continuous Integration be enabled in service-based distributed software systems in the absence of a**

central server? We present **Service-Oriented Revision Control** (**SORC**) as an answer to this question. SORC relies on peer-to-peer communication between service providers and consumers to enable discovery and consumption of versions of services, ensuring that the overall service-based software system operates in a stable and reliable manner, even in the presence of frequent service updates and multiple versions. By using three basic commands – i.e., commit, checkout, and update – the SORC approach demonstrates how service consumers can always be informed of compatible and incompatible service versions. To achieve this functionality, the approach uses a distributed reference architecture and implements a framework based on this reference architecture, thereby enabling support for Continuous Integration in SOAs through a novel combination of software versioning and compatibility checking techniques.

The SORC approach presented in this thesis contributes to the domain of Service-Oriented Software Engineering and, more specifically, to Continuous Integration for SOA, as well as to the areas of software versioning and compatibility checking. The main contributions of this thesis include the novel SORC approach itself, which supports Continuous Integration for SOA, the prototype implementation of the SORC system, called as SORCER, demonstrating the viability of the proposed approach, novel versioning and compatibility checking techniques, and a literature survey on relevant state-of-the-art research.

Table of Contents

Declaration **2**

Acknowledgements **3**

Abstract **4**

Table of Contents **6**

List of Figures **9**

List of Tables **11**

Author's Publications **13**

Chapter 1: Introduction **14**

1.1. Traditional Continuous Integration 16
1.2. Service-Oriented Architectures and Service-Based Software Systems 19
1.3. Problem Statement 24
1.4. Overview of the Solution 26
1.5. Research Question, Hypothesis and Objectives 27
1.5.1. Theoretical objectives 28
1.5.2. Technical objectives 29
1.5.3. Experimental objectives 29
1.6. Contributions of the Thesis 29
1.7. Outline of the Thesis 31

2. Chapter 2: Background Theory and Related Technologies **34**

2.1. Software Configuration Management 34
2.2. Version Control 37
2.2.1. Version Control Commands 39
2.3. Classification of Version Control Systems 42
2.3.1. Type of Revision Storage 42
2.3.2. Version Control System Architecture 44
2.4. Service-Oriented Architecture 60
2.4.1. WSDL/SOAP-based services 63
2.4.2. RESTful services 64

2.5. Summary 65

3. Chapter 3. Software Compatibility in SOAs: State of the Art and Existing Gaps....66

3.1. Software Compatibility 66

3.1.1. Software Changes in SOA 68

3.1.2. Service Changes and Versioning 70

3.2. Survey of the State of the Art 74

3.2.1. Related Works on Change Detection 74

3.2.2. Related Works on Service Versioning 81

3.2.3. Related Works on Service Compatibility 88

3.3. Identifying the Gap 90

3.4. Implementing CI for SOA development using SCM systems 94

3.5. Summary 97

4. Chapter 4. Service-Oriented Revision Control in Detail 99

4.1. Service-Oriented Revision Control: an Overview 99

4.1.1. SORC architecture 99

4.1.2. SORC Components 102

4.1.3. Versioning Method 104

4.1.4. Compatibility Problem Addressed by SORC 105

4.2. Threefold Functionality of SORC 107

4.2.1. Commit Operation 107

4.2.2. Checkout Operation 118

4.2.3. Update Operation 122

4.3. Expected Benefits 125

4.4. Summary 127

5. Chapter 5: Case Study and Evaluation 129

5.1. SORC Implementation: the SORCER Framework 129

5.2. Case Study: Comparing SORCER to the Existing Technological Baseline 133

5.2.1. Case Study Implemented by SORCER 134

5.2.2. Case Study Implemented by GitHub/Jenkins 141

5.3.3. Testbed Setup and Methodology of the Experiments 144

5.4. Experiments and Benchmarking 148

5.4.1. Integration Time and Occupied Disk Space with SORCER 148

5.4.2. Integration Time and Occupied Disk Space with GitHub/Jenkins 156
5.5. Discussion of the Results 163
5.5.1. Comparison of the experimental results 163
5.5.2. Discussing the benefits 166
5.5.3. Discussing the shortcomings 172
5.6. Summary 173
6. Chapter 6: Conclusion 175
6.1. Thesis Overview 175
6.2. Discussing Contributions 176
6.3. Future Work 178
6.4. Researcher's view 180
Acronyms 182
References 184
Appendix A: Summary of Related Works 197

List of Figures

Figure 1. Traditional Continuous Integration. 18

Figure 2. Difference between snapshot-based and changeset-based revision control. 43

Figure 3. Architecture of the Centralised Version Control System. 44

Figure 4. Architecture of the Decentralised Version Control System. 53

Figure 5. WSDL/SOAP-based architecture. 63

Figure 6. RESTful architecture. 64

Figure 7. Conceptual schema of a WSDL 2.0 document. 70

Figure 8. Service-Oriented Revision Control System. 101

Figure 9. SORC reference architecture. 102

Figure 10. Old Versioning Method in SORC (Sarib and Shen, 2014). 104

Figure 11. Compatibility problem in SORC. 106

Figure 12. Strict versioning through a new service contract and unique endpoint. 111

Figure 13. Commit message mechanism. 116

Figure 14. Proposed versioning method. 117

Figure 15. Commit diagram. 118

Figure 16. Developer node discovery. 120

Figure 17. Checkout diagram. 121

Figure 18. Update versioned services. 123

Figure 19. Update diagram. 124

Figure 20. A sample of sorcer.disco file. 131

Figure 21. SORCER interface. 133

Figure 22. SORCER workflow sequence diagram. 135

Figure 23. SORCER configuration for the SBS system by DeveloperA. 136
Figure 24. SORCER configuration for DeveloperB's weather forecast service. 136
Figure 25. SORCER configuration for DeveloperC's currency exchange rate service. 137
Figure 26. SORCER files created on all three developers' machines. 137
Figure 27. DeveloperB and DeveloperC commit the first versions of the weather forecast and currency exchange rate services. 138
Figure 28. DeveloperA is able to see both developers' services after running the checkout command. 138
Figure 29. Generated proxy class for the weather forecast service. 139
Figure 30. Generated proxy class for the currency exchange rate service. 139
Figure 31. A sample view of the SBS system that invokes the remote services. 140
Figure 32. GitHub/Jenkins workflow sequence diagram. 143
Figure 33. Sample output of the Jenkins build process. 144
Figure 34. Measuring integration time in SORC. 145
Figure 35. Measuring integration time in GitHub/Jenkins. 146
Figure 36. The SORCER process. 148
Figure 37. The GitHub/Jenkins setup. 156
Figure 38. Integration time. 164
Figure 39. Build time. 165
Figure 40. Total space occupied by the code base. 166

List of Tables

Table 1. Different types of changes in SOA. 71

Table 2. Developer activities using SORC. 101

Table 3. Type of changes in compatibility assessment. 112

Table 4. System specification of PC1 (DeveloperA). 147

Table 5. System specification of PC2 (DeveloperB). 147

Table 6. System specification of PC3 (DeveloperC). 148

Table 7. Weather forecast service commit time in SORCER.. 149

Table 8. Currency exchange rate service commit time in SORCER. 150

Table 9. Weather foreacst service update time in SORCER. 151

Table 10. Currency exchange rate service update time in SORCER. 152

Table 11. Weather forecast service integration time in SORCER... 153

Table 12. Currency exchange rate service integration time in SORCER. 153

Table 13. Total integration time in SORCER. 154

Table 14. Occupied disk space across all three developers. 155

Table 15. Push time for DeveloperB in GitHub/Jenkins.. 157

Table 16. Push time for DeveloperC in GitHub/Jenkins.. 157

Table 17. Build time for the weather forecast service in GitHub/Jenkins. 158

Table 18. Build time for the currency exchange rate service in GitHub/Jenkins....................... 159

Table 19. Pull time for the weather forecast service in GitHub/Jenkins.................................... 159

Table 20. Pull time for the currency exchange rate service in GitHub/Jenkins. 160

Table 21. Total integration time for the weather forecast service in GitHub/Jenkins................ 161

Table 22. Total integration time for the currency exchange rate service in GitHub/Jenkins. 162

Table 23. Total integration time for the overall SBS system in GitHub/Jenkins. 162

Table 24. Comparing SORC and the traditional version control systems. 166

Table 25. Summary of the state of the art in service versioning and compatibility. 197

Author's Publications

1. Almalki, J., Shen, H., (2014). Compatibility in service-oriented revision control systems. Paper presented at the 2014 IEEE 13th International Conference on Autonomic and Trusted Computing (ATC), Bali, Indonesia.
2. Almalki, J., Shen, H., (2015). A Lightweight Solution to Version Incompatibility in Service-Oriented Revision Control Systems. Paper presented at the ASWEC 24th Australasian Software Engineering Conference, Adelaide, Australia.
3. Almalki, J., Shen, H., (2018). SORC: Fulfilling DevOps for API Continuous Integration. The 2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design (CSCWD). (Submitted)

Chapter 1: Introduction

Since the emergence of software development as a discipline in the late 1940s, software systems have gone through many stages of evolution, from performing simple algebraic operations to complex distributed systems, developed and managed by whole teams of IT professionals (Pressman, 2005). Through the whole history of software full of 'ups and downs', developers have been searching for better ways of designing, implementing and managing software applications, resulting in existing approaches being retired and new approaches being adopted. The need for novel approaches became especially apparent in the 1970s and 1980s (Boehm, 1987), when multiple problems in software development became apparent. More specifically, software projects were becoming increasingly complex and large and started running out of budget and/or schedule. Being ineffective and unproductive, many software houses had to close down or were absorbed by larger corporations. This software crisis concerned not only the productivity and the operations of software companies, but also the quality of the software products released to customers. It soon became clear that existing software development practices had to be revisited in order to circumnavigate the emerging software engineering problems and challenges. This is when the first pre-cursors of modern agile methodologies were proposed and theorised. Computer scientists and software engineering pioneers first started advocating an incremental and iterative approach to software development, the basic principles of which later became known as *agile methodologies*. However, it was not until the mid-1990s for the true value of agile methodologies to begin being recognised by the software industry, the wider research community and individual software engineers. With the emergence of the Internet and the increasing role of network communications, software systems grew again in size and complexity. As market requirements pushed companies to release high-quality software products more rapidly, traditional approaches (e.g., the 'waterfall'

model) started being replaced by more flexible, truly agile methodologies. More recently, the software engineering discipline went beyond the traditional notions of applying various methodologies, using different programming languages and frameworks, applying design patterns, etc., and has also embraced the psychological and cultural aspects of the software development process to educate and motivate developers to create correct software. From this perspective, software engineering has truly changed the way end users and developers see and experience the whole process.

Over the last decade, software products have started being delivered to end customers as online services, available via thin software clients over the Internet. This contrasts with the previous state of software being distributed in physical boxes to be installed on end-users' personal computers. In these circumstances, there was a significant gap between the development and the operation phases of the software lifecycle, slowing down the iterative upgrade process. Delivery of software over the Internet enables continuous updating of the software and minimises the gap between the development and operation phases. This environment led to the emergence of Development and Operations (DevOps) as a way of improving efficiency. DevOps is an approach based on *lean* and *agile* methodologies that bring together several IT departments within an organisation, including the development, operations, and quality assurance departments, to support continuous delivery of software products to clients (Sharma and Coyne, 2013). The DevOps model promotes a collaborative and unified method for delivering software and is being increasingly adopted by various sectors of the IT industry. Organisations, aiming to achieve greater revenues, are migrating towards this novel way of organising their internal production and operation activities in a more efficient and productive manner.

However, with the further development of service-oriented computing and increasingly distributed ICT systems, the continuous model can no longer rely on traditional software engineering practices (versioning, continuous integration, and continuous delivery), since they were not designed for distributed, decentralised development scenarios. To address this limitation and facilitate continuous integration in distributed service-oriented scenarios, Service-Oriented Revision Control (SORC) is proposed in this thesis as a set of processes, practices, and a reference system implementation that supports software engineering and continuous integration in distributed, service-oriented software systems.

1.1. Traditional Continuous Integration

Continuous Planning, Continuous Integration, Continuous Deployment, Continuous Testing, Continuous Delivery, and Continuous Feedback are all development and operation practices widely adopted and promoted in the modern software development enterprises. As suggested by their names, the continuous, incremental operation of all involved teams and assets lies at the very core of these practices that aim to decrease the time between applying a change to a system and that change being available in the production environment. Furthermore, maintenance of software quality in terms of both the code and the delivery mechanisms are key elements in the software development process (Bass et al., 2015).

Software systems are becoming increasingly complex and distributed, often involving multi-national development teams that work simultaneously on different components of a common modular project. Ensuring that individual pieces of the overall system fit together, i.e., can be integrated without breaking the stability and operation of the overall software project, is of paramount importance. To achieve this, software vendors rely on Continuous Integration (CI) to automatically build, test, and run software products, thus reducing manual integration effort and

shortening time to market. CI requires that developers regularly commit software changes to a central, shared repository so that newly added updates can be built, tested, and run by an automated CI server. Thereby, *communication* is seen as the primary benefit of CI (Fowler and Foemmel, 2006): implementing an automated building and testing system gives visibility into the current status of a project to all participants, avoiding potential misunderstandings and ambiguities. Another benefit of CI is the reduction of integration time and effort, and the ability to release a working version of the product at any given point in time.

In more traditional models, developers, despite being part of a team, spend most of their time working in isolation, manually merging their code modifications with the master code branch at arbitrary points in time. This batched, infrequent approach to merging accumulated changes is often a difficult, complicated, time-consuming, and error-prone routine. In contrast, CI motivates developers to make more frequent, regular, and fine-grained commits to a shared repository using an existing version control system (VCS) such as Git[1], Mercurial[2], or Subversion[3]. Major discrete updates to the project are replaced by an analogous series of relatively small code updates (Schaefer et al., 2013). In this way, it is ensured that the collaboratively created code works by providing rapid feedback on any possible problem that may be triggered by the committed changes (Humble and Farley, 2010).

---

[1] https://git-scm.com/
[2] https://www.mercurial-scm.org/
[3] https://subversion.apache.org/

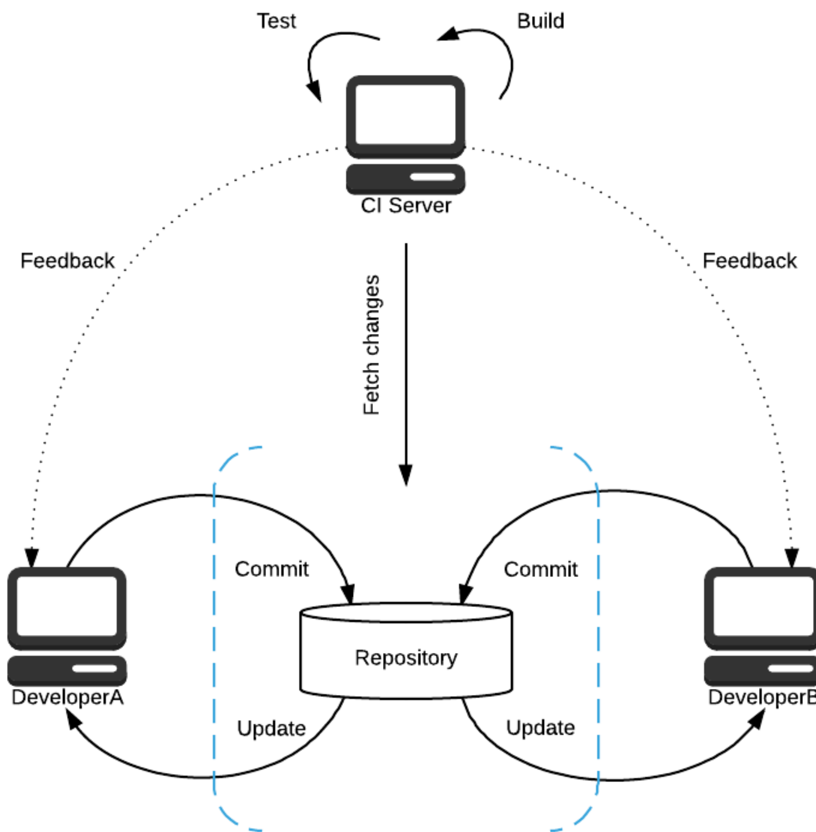

Figure 1. Traditional Continuous Integration.

Furthermore, CI ensures that each team's contribution is continuously validated before being integrated into the common global project. It reduces risks and identifies issues earlier in the software development lifecycle by implementing automated software building and testing (Humble and Farley, 2010). Prior to each commit, developers run unit tests on their code locally as an extra verification step before merging changes with the mainline. Once changes have been committed to a shared repository, the CI server automatically builds and runs unit tests on the updated code to immediately spot any functional (i.e., concerning the functionality of an individual system component) or integration (i.e., concerning the functionality of the overall modular software project) failures (Shahin et al., 2017). Detected errors are then reported to interested

parties (i.e., responsible software developers and/or team leads) to be further addressed in a timely manner.

Figure 1 summarises this established CI practice where developers work locally in an edit/build/test cycle and then integrate their code into a common project in an update/commit cycle. In the update/commit cycle, developers have to update their local codebase by downloading and locally integrating changes made by other developers. Then their own code is committed back to the repository for other developers to use. The edit/build/test cycle is used both when developing code and when merging remote changes into the local workspace prior to committing changes into the mainline (Microsoft Technet, 2014). This development model requires a dedicated CI server that fetches code as it is committed by developers and undertakes the automated integration routine. As a result, standalone CI introduces a delay in the time it takes for changes, new features, and bug fixes to find their way into production. As the frequency of code integration increases (i.e., changes are committed more frequently), the amount of information related to build and test results, generated during the CI process, is expected to grow and potentially slow down overall performance (including generation of reports and alerts to developers) (Shahin et al., 2017). In this light, data representation in a clear, concise, yet informative manner becomes particularly important to facilitate understandable communication of results to developers. This is, however, a minor shortcoming in the light of the promising benefits of CI offered to software engineers.

1.2. Service-Oriented Architectures and Service-Based Software Systems

The industrial and academic communities have been putting their research efforts into investigation of novel approaches to improve the software quality as a potential way of addressing existing limitations in the context of developing complex distributed component software projects, e.g., using Remote Procedure Calls (RPC) (Nelson, 1981) or Common Object Request Broker

Architecture (CORBA) (Otte et al., 1996). More specifically, these limitations include (Hohpe, 2005):

- **Software complexity** – distributed software components communicate between each other by exchanging software objects over a network, thereby bringing another level of complexity to the object-oriented world. One object could control the lifetime of remote objects, pass references, and at the same time rely on polymorphism and inheritance. Given the complexity of network communications and distributed systems, this approach rendered itself to be too complex and unsustainable.
- **Vendor lock-in** – apart from RPC and CORBA, there existed many more proprietary implementations and protocols, making resulting applications not interoperable and portable.
- **Tight coupling** – besides using proprietary technology, distributed components of a larger software system were often highly dependent on each other (i.e., tightly coupled), such that a failure of a single component led to the failure of the overall system.
- **Connectivity** – similarly, tight coupling between distributed components assumed stable, permanent, and reliable connectivity, typically based on wired (e.g., Ethernet) network connections. This contradicted with the rapid growth and increasing role of the Internet and mobile communications in the ICT landscape that aimed to enable global, although not absolutely reliable, connectivity.
- **Call stack** – being able to pause program execution and trace it step by step is an extremely powerful and convenient tool for debugging monolithic applications. However, stopping program execution at a break-point on a local or remote machine in the middle of

information exchange would typically result in unresponsive behaviour of the overall system.

- **Illusion of transparency** – despite the initial intention to hide the underlying communication between distributed components from developers, it only appeared partially true. That is, in tightly-coupled scenarios, a failure of an individual remote component affected the overall system and required developers deal with remote debugging and exception handling. This gave developers a false sense of comfort, resulting in worse software quality.

Taken together, these problems motivated researchers to investigate new ways of avoiding these problems (at least, partially), leading to the emergence of the Service-Oriented Architecture (SOA) paradigm (Erl, 2005; Papazoglou, 2003). Service-oriented Computing (SOC) aims to address the aforementioned limitations by adopting a document-oriented model, in which interaction is based on standardised, technology-agnostic communication protocols. In this context, document-orientation refers to using flexible document formats, explicit contracts, and service registries, and enables an asynchronous interaction style in order to facilitate loose coupling between interacting services (Hohpe, 2005).

Service-orientation supports development of distributed software systems in a loosely-coupled and technology-agnostic manner using basic ‘building blocks’ (Dautov and Paraskakis, 2013; Dautov et al., 2014), i.e., services that are reusable software components and remotely accessible via a standard, well-defined application programming interface (API) (Wei and Blake, 2010). Being loosely-coupled, services are designed and implemented to provide certain functionality without being aware of what technologies underpin the implementation of other elements of a larger SOA ecosystem. In this way, implementation details are hidden and only APIs are exposed to the outer

world, making it possible to seamlessly combine and integrate various services in order to build modular **service-based software (SBS) systems** (Yau et al., 2009; Di Nitto et al., 2008), such that individual elements (i.e., services) can be easily replaced or removed. More specifically, the SOC principles imply that a service is expected to exhibit the following four main properties (The Open Group, 2009):

- By providing a certain level of abstraction, a service acts as a conceptual representation of an underlying business activity with a strictly specified outcome.
- A service is self-contained – that is, it is deployed and runs as an individual independent software asset, and is able to exist even when not being part of a larger service-based composition.
- By abstracting implementation details, a service is essentially seen as a 'black box' by its consumers, who are only allowed to interact with the description of a service and its API. In other words, services typically do not expose their underlying implementation details and are only accessible via APIs that are used to integrate individual services into larger SBS systems.
- As a consequence of the previous property, a service may in fact represent a whole chain of several underlying interconnected services, where one service is coupled with another to provide the final result advertised by the service description.

By correctly following the SOC principles and implementing software in a SOA, software developers are expected to enjoy the following benefits:

- **Reusability** – third-party software assets are remotely accessible for consumption, thus reducing the necessity to re-implement the same functionality.

- **Interoperability** – Web services only expose their APIs by means of standard languages (e.g., Web Services Description Language (WSDL)), whereas actual implementation details are hidden. This enables services to be easily integrated with each other, regardless of implementation technologies of individual services constituting the larger SBS system.
- **Scalability** – since service-oriented software systems are loosely-coupled in their nature, adding or removing a service does not require massive system interruption, such as recompilation or restarting. This way, SBS systems may scale in and out in a seamless and transparent manner.
- **Flexibility** – similarly, by adding or removing individual services, a service-based system may easily adapt to new emerging business requirements with minimum delays.
- **Cost-efficiency** – by exempting software developers from designing and implementing their own software components from scratch and enabling them to reuse already existing and reliable software functionality, services may considerably reduce financial expense on a wide range of redundant software development tasks.
- **Easier integration and management** – since commercially available services are expected to be developed and provisioned by reliable IT professionals, the integration process of putting different services together into an SBS system, as well as its further management and maintenance, is expected to be a straight-forward error-free process.

Taken together, all of the features of SOA make it possible for enterprises to considerably shorten the time required to design and implement a software product, and deliver it to end users as a commercial offering – a key requirement in the presence of stringent market requirements. In this light, the application of CI techniques to service-oriented software projects is a natural development.

1.3. Problem Statement

Continuous Integration for Service-Oriented Software Engineering

With the further development and adoption of SOA, APIs are increasingly becoming the 'glue' of Internet communications that enable software integration and interoperability at a global scale. As IT is becoming increasingly service-oriented, reliable and well-functioning APIs are of paramount importance to SBS systems, making CI one of the main catalysts of the API development process (Wood et al., 2016).

Having introduced a paradigm shift in the way software is delivered to and consumed by end users, the SOA paradigm has also affected the way it is developed and engineered. As a result, Service-oriented Software Engineering (SOSE) has emerged as a reflection of the fundamental differences with traditional software engineering (Breivold and Larsson, 2007; Karhunen et al., 2005; Stojanović and Dahanayake, 2005; Tsai, 2005). SOSE is a software engineering methodology that follows the service-orientation principles and explicitly focuses on software development approaches where reusable services (often provided by third parties) are brought together in a service-oriented fashion (Karhunen et al., 2005). SOSE extensively uses composition techniques to combine services and, therefore, shares many characteristics with component-based software engineering (Heineman and Council, 2001a; Jifeng et al., 2005; Kozaczynski and Booch, 1998) – a generic and well-established methodology for building software systems from reusable components. To this fundamental functionality, SOSE also adds the ability to dynamically discover and invoke remotely located and network-accessible services at run-time, putting the dynamic nature of Web service compositions (i.e., connections between service users and the service providers) at the very core of a service-oriented software system and, therefore, the software development process.

From the software development perspective, however, migrating to SOA introduces a previously-unseen level of complexity. For example, system configuration and deployment become more challenging, as it is now required to consider various network-related issues, such as latency or network interruptions, and potential version mismatches between components (Hohpe, 2005). As it will be explained in the rest of this thesis, loosely-coupled service-oriented systems are better suited for handling minor changes in data formats such as modification of field names or addition of new fields. On the other hand, tighter coupling allows the system compiler to detect potential mismatches between communicating services at an earlier stage – for example, mismatching data types, mistyped method names, or missing function parameters can be detected at compilation time, even before the actual service invocation (Hohpe, 2005). In most cases, loosely-coupled architectures will detect similar problems only at a later (i.e., run-time) stage. In other words, debugging a tightly-coupled distributed software system is easier than debugging heterogeneous, asynchronous, distributed architectures.

CI in these circumstances is seen as a vital instrument to be implemented and applied in order to support the distributed software development process. With SOA, however, CI is not straight-forward, since different parts of an SBS system might be owned and managed by different software development teams and organisations. As individual services, constituting an SBS system, are owned and managed by different stakeholders that do not necessarily provide service source code, testing and building the overall SBS system in one place appears to be infeasible (Richards, 2015). That is, there is no central location (and a dedicated server) where software artefacts can be stored, tested, deployed and run, as it happens with traditional centralised CI approaches. Rather, there are multiple involved parties, each of which independently develops and manages software services that are accessible via APIs (He and Da Xu, 2014). Supporting CI in this context requires

coordination between these loosely-coupled and separately stored systems. In these circumstances, a decentralised approach to CI in the context of API-driven SOA environments is required.

In more practical terms, the discussed motivation and problem statement also reflect the author's personal experience of developing service-based applications working in a geographically distributed team. As the number of updates to some remote services was growing, it became evident that the existing tools that were adopted within the team, such as GitHub, SVN, Jenkins, etc., were unable to support timely automated integration of the frequently updated components into a common service-based software system. From this perspective, the envisioned solution to this outlined problem is expected to assist software developers in engineering and developing distributed software systems in the increasingly service-oriented IT world.

1.4. Overview of the Solution

To address these outlined challenges, this research work presents a novel approach to support CI in the context of service-oriented software development – that is, Service-Oriented Revision Control (SORC). SORC takes into account the distributed nature of SOAs and the inevitable requirement to support integration of individual elements of a shared project in the absence of a single central location for doing this (e.g., a dedicated server with a running CI software).

To achieve this, SORC merges the build/test phases of the edit/build/test cycle into the update/commit cycle, as shown in Figure 2 (which is fundamentally different from the traditional continuous integration process shown in Figure 1). As it is depicted in the figure, only code modifications are done on the local machine, whereas both the update/commit and build/test cycles take place on the network. This way, software developers can modify their individual service components locally, commit them to their personal repositories and servers for deployment, and finally publish service APIs on the network for public access. Next, other developers update their

references to remote services by consuming their respective APIs prior to building and testing. By doing so, during the build/test cycle, developers get immediate feedback on whether individual elements of the overall SBS system are compatible, thereby reducing unnecessary effort in between commits and updates.

To validate the proposed approach, the SORCER framework was developed, which is a prototype solution implementing the SORC reference architecture. By implementing the proposed functionality, it is able to support CI activities in the context of distributed service-oriented software development.

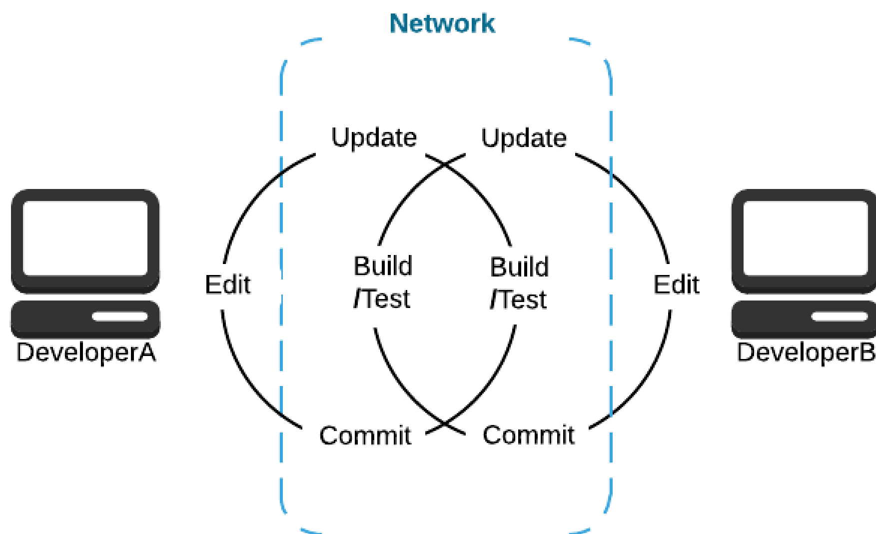

Figure 2. Continuous Integration in SORC.

1.5. Research Question, Hypothesis and Objectives

Taken together, the outlined challenges identify a research and technological gap that is addressed by the work presented in this thesis. Accordingly, this research aims to answer the following main research question:

How can continuous integration be enabled in service-based distributed software systems in the absence of a central server?

To answer this research question, the work proposes a distributed architecture – SORC – and provides a reference implementation that demonstrates a way of enabling support for CI in service-oriented architectures through a novel combination of techniques including software versioning and compatibility checking. The proposed approach is based on the following hypothesis that is also to be investigated and proved in the context of this research effort:

The challenge of enabling continuous integration in the context of distributed service-based software systems can be implemented by combining techniques from software versioning, compatibility testing, push-pull information exchange in a decentralised peer-to-peer manner, where changes made to services are directly propagated to service consumers, without a central registry.

The work to answer the research question and prove the hypothesis is tackled by breaking them down into several theoretical, technical, and experimental objectives. Taken together, these objectives serve as references to present and explain results in a structured and systematic fashion.

1.5.1. Theoretical objectives

- Conduct a literature review on the foundations and the current state of the art in software versioning, change and compatibility detection, with a specific focus on the applicability of the existing approaches to enable CI in SOA.
- Identify an existing gap to be addressed by the proposed research work.
- Propose an approach to address the identified gap.
- Within the identified gap, distil and clearly formulate the main functional and non-functional requirements for an envisioned approach/solution.

1.5.2. Technical objectives

- Design and implement a software prototype of the identified solution (i.e., SORCER framework) to enable CI in a distributed service-oriented software development environment.
- Design and implement a case study experiment to validate the implemented software prototype.

1.5.3. Experimental objectives

- Validate the viability of the proposed approach and the SORCER prototype through a use case with respect to several key criteria.
- Measure and evaluate the results of the validation.

The research question, hypothesis and objectives will be re-visited in the final chapters of this thesis. They serve as the evaluation of the research results. That is, addressing and fulfilling the outlined objectives, as well as proving the hypothesis and answering the research question are seen as the key criteria for this PhD thesis to be considered as a valuable research contribution, advancing the state of the art.

1.6. Contributions of the Thesis

The proposed research effort, as highlighted by the author's personal motivation, can be primarily considered as applied research. It aims to address a practical challenge faced by software engineers when developing distributed service-oriented systems, and, as it will be further explained, does so by combining and extending the existing solutions in a novel intelligent way. Achieving the goals of the proposed approach primarily contributes to the research areas of service-oriented computing and CI by introducing a novel approach to enabling continuous support for software testing and integration in a distributed peer-to-peer environment in the absence of a central party –

functionality that currently has not been proposed or implemented elsewhere. Moreover, the contribution of this thesis also spans several adjacent research fields, such as software versioning and version control, compatibility checking, and software configuration management. More specifically, the main contributions of this research are the following:

- **Literature survey of the state of the art in service compatibility, service change detection, and service versioning**: as part of fulfilling the theoretical objectives, a literature survey has been conducted, identifying existing limitations and gaps in the research field. From this survey, it became clear that the challenging topic of supporting CI in the context of service-based software development is yet to be explored. *Awareness* of the interested research community is seen as one of the first key steps towards advancing the state of the art and, therefore, the conducted survey (and the research in general) is intended to raise overall awareness and attract the attention of the community to this motivating and challenging problem.
- **Functional specification of requirements for distributed CI:** based on a thorough investigation of existing software management systems (employing various versioning schemas, change detection and compatibility testing tools), some existing limitations were identified. These limitations, in turn, led to a list of functional properties that a system implementing CI in service-based software systems is expected to demonstrate. This functional specification serves as the key reference underpinning the design and implementation of SORCER – a software framework implementing the SORC functionality. Moreover, it also contributes to the state of the art in enabling CI for SOAs, as it is expected to be re-used by the wider research community, willing to

engineer their own solutions based on the proposed specification (and thus not "re-invent the wheel").

- **Design of the SORC system:** based on the distilled requirements, we have designed the architecture of the proposed SORC system. This high-level design can serve as a reference model for researchers willing to implement their own similar solutions. Within the overall design of the SORC system, several novel features can be distinguished:
  - Decentralised architecture
  - Novel versioning scheme
  - Novel compatibility assessment mechanism
  - Support for compatible and incompatible changes
- **SORCER framework:** the existing pilot solution implements the SORC design and demonstrates the viability of the whole proposed approach and is available for downloading and experimenting. Using SORCER, one is expected to immediately benefit from the possibility of supporting CI in service-based software systems. Moreover, as we follow an open-source approach to distributing our software, it is also possible for someone to extend the existing functionality of SORCER to implement required emerging features.

1.7. Outline of the Thesis

- **Chapter 2** provides a theoretical foundation of Software Configuration Management and Version Control Systems, and describes how SORC extends the existing technological state of the art. The chapter first explains the motivation behind the emergence of these technologies, existing architectural models, and discusses the main benefits offered to software development teams. Next, the chapter introduces Service-Oriented Computing

and elaborates on the applicability of the existing version control systems to SOAs, and outlines the main differences between the traditional version control systems and the proposed SORC.

- **Chapter 3** introduces the state of the art in the domain of software compatibility in general and in SOAs in particular. It provides some theoretical background to key concepts: service change, change detection, change notification, service compatibility, and Software Configuration Management. Furthermore, using these concepts, the chapter surveys existing approaches, providing an understanding of existing techniques and tools. This survey then highlights research and technological gaps in the state of the art and, thereby, positions the presented work within the overall body of relevant research efforts, as well as outlines the main requirements for a future solution.
- **Chapter 4** looks into the actual design of SORC. It explains the three basic commands, on which SORC is built: commit, checkout, and update. The Commit command is used by service providers to register new versions of a service so that they can be discovered by service consumers. The checkout command is used by service consumers to discover peer developer nodes and the services that are published on them. Finally, the Update command is used by service consumers to migrate from a currently used service version to a new version of the same service.
- **Chapter 5** focuses on demonstrating the viability of the presented ideas and concepts. To do so, the chapter first introduces a use case scenario that serves to illustrate how SORC can be used in the context of service-oriented software development. On the one hand, this case study is intended to highlight existing challenges, and, on the other hand, it showcases how these challenges are addressed by the SORCER framework. Moreover, the

functionality of the proposed system is compared to some existing technologies and further evaluated with respect to several key criteria. The evaluation is followed by a discussion, explaining benefits and shortcomings of the proposed approach.

- **Chapter 6** concludes the thesis by revisiting the main research question, the hypothesis, and the thesis objectives. It summarises and discusses the main contributions identified and introduced during the course of the presented research. The chapter also provides a brief outline of several potential directions for future work, which reflect some ways of further improving and enhancing the presented research effort.
- **Appendix A** contains a table with a concise summary of related works.

2. Chapter 2: Background Theory and Related Technologies

This chapter first provides a theoretical foundation of Software Configuration Management and Version Control Systems – widely adopted technologies underpinning CI activities in enterprise-level software development. The chapter explains the motivation behind the emergence of these technologies, existing architectural models, and discusses the main benefits offered to software development teams in terms of CI. Next, the chapter introduces Service-Oriented Computing, characterised by its distributed/decentralised nature, and elaborates on the applicability of existing version control systems to SOAs. Based on the limitations of the existing approaches to support CI in SOAs, the chapter outlines the main differences between traditional version control systems and the proposed SORC, which is thus seen as an extension to the existing technological state of the art.

2.1. Software Configuration Management

Software development, since its very emergence, has been associated with ever-increasing complexity of the development process itself and the tools used to support this process. Project managers and code developers have been addressing this complexity by means of various automated solutions, ranging from simple version control systems to more advanced configuration management, build management and work-area management software tools. The main problems that arise during the software development process are related to complexity caused by parallel modifications being introduced by different developers at the same time to the same piece of code. Software systems were no longer seen as single-piece, isolated programs, but rather became complex and tangled environments, often spanning across different organisations and administrative domains. Given the increasing number of developers working on a complex software project in parallel from various distributed locations, it became especially important to

ensure that individual parts contributed by various team members could actually ‘fit’ together in a timely, continuous, and automated manner – that is, to perform Continuous Integration.

In this context, to avoid (or at least minimise) software projects failing due to the inability to support increased complexity, there needs to be some form of control in enforcing a common structure, to which developers abide. In these circumstances, it is important to properly structure, control and manage parallel concurrent modifications so as to maintain the quality and timeliness of software products. Among other things, key aspects to be covered in this respect can be summarised as follows:

- **Parallel work on artefacts and simultaneous updates:** the work of one developer can easily be overwritten by some other developer’s work in situations when both developers are working on the same software project separately from each other. Accordingly, there should be an efficient and reliable mechanism to ensure that developers can work in a non-blocking and non-competing manner on the very same software artefacts at the same time (Mukherjee, 2011).
- **Change detection and notification of developers:** when a shared common project is modified by one developer, these newly-applied changes should be communicated to the rest of the development team via change detection and notification mechanisms (Guimarães and Silva, 2012). Otherwise, there may be situations when individual developers keep on working on an outdated version of the project, potentially breaking its integrity and consistency.
- **Multiple versions management:** it is very common to have dependencies between different software components and libraries, which can have multiple versions. In these circumstances, it is important to ensure that the correct versions of software components

are used. Given a considerably large team of developers, constant control over which versions are used by team members is a challenge.

- **Change propagation and synchronisation across different versions:** most enterprise-level software development projects are delivered as a series of evolutionary releases, often following the agile principles (Beck et al., 2001). With each release in different phases of development, any changes (e.g., bug fixes) must be synchronised across all affected versions. That is, if a bug is found in a version already released to a customer, the bug fix should be applied to all consequent versions as well. Similarly, if a bug is found in the most recent release (or even during the development phase), it should also be rectified in all earlier versions that might be affected by it. This, however, is not a trivial task, since enterprise software projects often simultaneously rely on multiple active releases, thereby requiring an extensive team of software developers solely responsible for bug fixing and version synchronisation.

Admittedly, fulfilling these described features goes beyond the manual capabilities of human staff, and needs to be implemented in a reliable, automated manner. A solution to the so called "software crisis" (Sommerville, 2010) – i.e., the failure of software projects due to rapid growth and increased complexity – has become Software Configuration Management (SCM), also referred to as Source Code Management. SCM systems serve to control the complexity of large software development projects, and coordinate seamless concurrent work by multiple developers. This way, developer groups are offered tools that can transparently support collective distributed work during all software lifecycle steps, and facilitate efficient and productive teamwork, supported by CI.

SCM can be defined as a discipline of controlling the evolutionary process of a complex system during its lifecycle. It can also be seen as a software engineering paradigm that focuses on the

management of software evolution and change. The primary function of SCM is to make sure that all changes are efficiently identified and properly managed. An integral part of SCM is versioning, which focuses on assigning unique identifiers to each software artefact (Novakouski et al., 2012). Version control, as it will be explained below, remains the key functionality in the context of rapidly changing complex software systems, widely adopted by enterprises (Baushke, 2015).[4]

2.2. Version Control

A popular approach to managing service changes is versioning (Lublinsky, 2007). Versioning represents a checkpoint in the evolution of a service. Managing service evolution is a critical issue, since changes in services affect service consumption. This suggests the need to deal with multiple versions of a service available at the same time (Andrikopoulos et al., 2012; Papazoglou et al., 2011).

Version Control Systems (VCSs), sometimes also referred to as Revision Control Systems, have long been recognised as an efficient and effective tool to address the aforementioned challenges related to the ever-growing complexity of enterprise software systems. VCSs enable software development teams to handle separate parts of an overall software product as separate identifiable versions, which could be configured accordingly so as to assemble different configurations of the very same component-based software product (Vesperman, 2006). This important ability to version different software components provides programmers with an opportunity to independently work on their own copies, without the necessary requirement to handle potential issues caused by simultaneous concurrent software updates. The main idea underpinning the concept of versioning

[4] Please note that despite the fact that version control is just a single aspect of a broader concept of Software Configuration Management, in the context of this document we might use the terms SCM and version control interchangeably (unless explicitly stated otherwise).

is simple – each time a file is changed a new revision of that specific artefact is created. The artefact evolves as a sequence of multiple revisions, usually indicated by successive numbers (e.g., *foo.1*, *foo.2*, and so on). Any revision in this sequence can further split into two separate lines of changes, thereby establishing a *tree* of revisions. Accordingly, each line in such a revision tree is called a *branch*, which typically inherits the same naming convention (e.g., *foo.2* can further split into *foo.2.1*, *foo.2.2*, etc.).

VCSs serve to support the process of recording and retrieving changes in a software project (Vesperman, 2006), thus making it an essential part of any form of enterprise-level, distributed, collaborative development, where concurrent modifications of the same projects may happen during their development phases (Yeates, 2013). Although there is a number of different VCSs, in general, all these systems serve to record modifications (the terms *change*, *patch*, or *revision* are also frequently used in the literature), such as additions and deletions, being made to a software project by collecting and keeping track of associated meta-data. Based on this meta-data, a VCS can re-construct the state of the software project at any point along its development lifecycle. More specifically, a typical VCS is expected to provide development team members with support for the following key activities (O'Sullivan, 2009a):

- **Provenance and history tracking:** allows a team to track the history of file modifications and states at all steps of the software development project. Users can *i)* see who made a change and when it was made, *ii)* understand why it was made (e.g., by reading a comment), *iii)* analyse the details of the change, and *iv)* re-create the state of the project at the moment when the change was made (i.e., roll back to a particular version).
- **Multi-tenancy and isolation:** enables users to work on independent sub-projects without being disturbed by other users' changes and without affecting the work of their colleagues.

These personal self-contained pieces of software, belonging to individual users, are usually referred to as *branches* (Fischer et al., 2003). Additionally, project branches are also used to control and maintain software releases that are no longer actively developed. For such cases, a branched sub-project is created and developed separately from, and in parallel with, the main project. When the work on the sub-project is finalised, it may be integrated back (i.e., *merged*) into the main parent project.

2.2.1. Version Control Commands

Typically, every VCS is expected to enable interaction between the main repository and local files through the following key operations and commands:

- **Checkout:** The checkout operation is used whenever there is a need to create a new working copy within a repository that already exists. In a broad sense, a working copy is a snapshot of the current working repository, which is used by developers as a place to make changes. As the repository is shared by the whole development team (i.e., multiple users are working with it at the same time), changes directly applied to the repository are not recommended. Instead, each individual developer is expected to work on a dedicated personal copy – that is, a *working copy* – of the repository. The working copy provides him/her with a private personal workspace, where his/her work is isolated from the rest of the team.
- **Commit:** once software developers check out a copy of the working project, they start working and making changes on this local copy. Next, these changes need to be shared with the rest of the team and sent to the central repository for storage – that is, they need to *commit* their changes. A single commit operation may contain multiple changes applied to different parts of the project, including source code files, configuration files, media files,

resources, etc. Whenever there is a change in a project's content that needs to be shared with the rest of the team, this change is included in a commit transaction. A VCS is expected to support atomic commit operations (i.e., similar to atomic database transactions) – that is, it needs to ensure that in the case that any part of a commit operation is not accepted by the repository then the whole commit operation needs to roll back so as to maintain the consistency and integrity of the whole project and related data (Vesperman, 2006). Most VCSs require users to provide a short commit message (also referred to as a commit comment, a commit note, or a commit log) every time a commit is made. Together with other context-related data (e.g., user name, date, time, etc.), this comment is attached to the changed data set. Next, each commit is assigned a unique version number, which is typically a sequential, incrementally increasing number, starting from version 1. An exception here is distributed VCSs, which cannot use this approach, since it is not always possible to synchronise and coordinate sequential numbering across multiple independent repositories. For this reason, these kind of VCSs employ different approaches to assigning unique version numbers that rely either on *i)* pseudo-random number assignment, or *ii)* applying hashing techniques to changes so as to use these hashed values as new version numbers (O'Sullivan, 2009b; Roundy, 2005).

- **Update:** an update operation serves to get a developer's working copy up to date by applying recent changes from the repository. This is done by merging local changes (if any) with the changes pushed to the remote repository by other developers. This synchronises the local working copy with the most recent changes. When the working copy was first created, its contents exactly matched a specific revision of the repository. The VCS is aware of that revision and can identify changes made since that revision. This revision is often

referred to as the *parent* of the working copy; committing changes to the working copy creates a new revision, based on the initial parent revision. From this perspective, updating can be seen as a reverse operation to committing. Both operations transfer changes between the working copy and the repository, but in opposite directions. A commit operation moves changes from the working copy to the repository, whereas an update operation moves changes from the remote repository and applies them to the local working copy.

- **Merge:** as it was previously explained, VCSs support simultaneous collaboration of multiple users, who can access and write to their own working copies of a global shared software project in a parallel manner. In these circumstances, there is always a possibility of a conflict (Vesperman, 2006) – i.e., a situation, when two or more users are working and overwriting the very same pieces of a project, which potentially may contradict each other. In other words, a conflict might occur when two or more different users simultaneously modify same lines of code within the same file. In this case, the VCS is unable to automatically determine the correct version and decide which one needs to be used, or whether the versions need to be combined, or whether neither of the versions has to be selected. To address such situations, manual intervention of software developers is required to resolve the resulting conflict, i.e., by comparing the conflicting code line by line, the developer is expected to choose the right version. As it will be explained and discussed below, in a distributed VCS there is an explicit operation that combines simultaneous edits by two different users, called *merge*. Merging can take place in an automated manner provided there are no conflicts (i.e., changes made by different users to different pieces of code and, therefore, are not conflicting). However, in case there is a conflict, the merging operation typically requests an approval or assistance from the user. In a centralised VCS,

merging happens implicitly every time users invoke the update operation. In general, it is good practice to avoid conflicts, rather than to try to resolve them when they occur. For example, a common established practice to avoid conflicts is to promote frequent sharing of fine-grained changes within the development team. In this way, complicated conflict situations, involving multiple lines and even files, are minimised.

2.3. Classification of Version Control Systems

Typically, there are two criteria with respect to which existing VCSs may differ from each other and classified. These are: *i)* the type of revision storage (snapshot-based and change set-based models), and *ii)* the VCS architecture (centralised and distributed). Both classifications are considered below in more detail.

2.3.1. Type of Revision Storage

Two options for capturing, storing and transferring changes in a VCS repository are (Tichy, 1985):

- **Snapshot-based**: the simplest approach uses *snapshots* that represent a complete copy of the dataset for each version. Also known as the *state-based model* (Conradi and Westfechtel, 1998), this approach is specifically convenient for certain data types and version control scenarios, where determination of atomic data units is often not well-defined (e.g., version control of images or binary media), making it difficult to separate modified and unmodified elements from each other.
- **Delta-based:** in the case of data sets with well-defined atomic data units, a more efficient approach uses *change sets* (also known as *deltas* or *diffs*) that contain only modified atomic data units for each version, i.e., they represent differences between revisions. Similar to snapshots, change sets are captured, transferred, and stored in the repository, but are more light-weight (Dart, 1991).

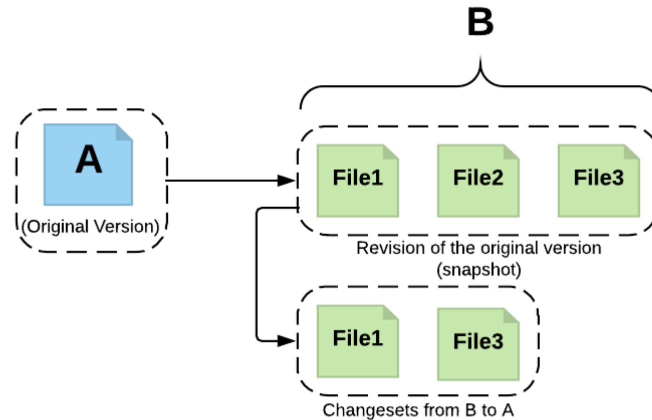

Figure 2. Difference between snapshot-based and changeset-based revision control.

The difference between the two approaches is summarised in Figure 2. Revision B of the original version A contains all the files, including files *File1* and *File3* (which were changed), as well as file *File2*, which was unchanged but was still included in the revision according to the snapshot-based model. On the contrary, the change set-based model assumes storing only the differences between the original version A and the revision B. Thus, only *File1* and *File3* are captured in the changeset for revision B.

Typically, change sets (as used, for example, by Concurrent Version System (Berliner and others, 1990)) provide significant benefits in terms of repository storage space, since there is no need to store excessive amounts of potentially redundant information. In the snapshot-based approach (as used, for example, by SVN), a complete collection of files, corresponding to every version is stored in the repository. This leads to increased disk space that is used for storing revisions, and, as a result, more network bandwidth is required to checkout, update, and commit potentially heavy-weight revisions. On the other hand, the time needed to apply these revisions is minimised, since developers are given all the files and resources at once, and there is no need to correlate original files with modifications, as it happens with change sets. In the latter case, less network bandwidth and storage space are required, which comes at the cost of increased time needed to retrieve a

particular revision, since in order to generate latest revisions, retrieved deltas have to be first applied to base revisions (Tichy, 1985). It is worth noting that as network bandwidth and storage capabilities continuously improve, the majority of existing SCMs, including the popular Git (Loeliger and McCullough, 2012) and Mercurial (O'Sullivan, 2009b) systems, opt for the snapshot-based model.

2.3.2. Version Control System Architecture

Centralised architecture

The first type of the VCS architecture is based on the centralised client-server model. In a centralised architecture, there is always a central (possibly remote) repository, which is responsible for maintaining the revision history of software development projects. As depicted by Figure 3, developers A, B and C are connected to the central server to check out the most recent revision of a project into their individual, locally maintained sandboxes. Once the updated project has been checked out, the developers are ready to work and commit their modifications back to the central repository in the future.

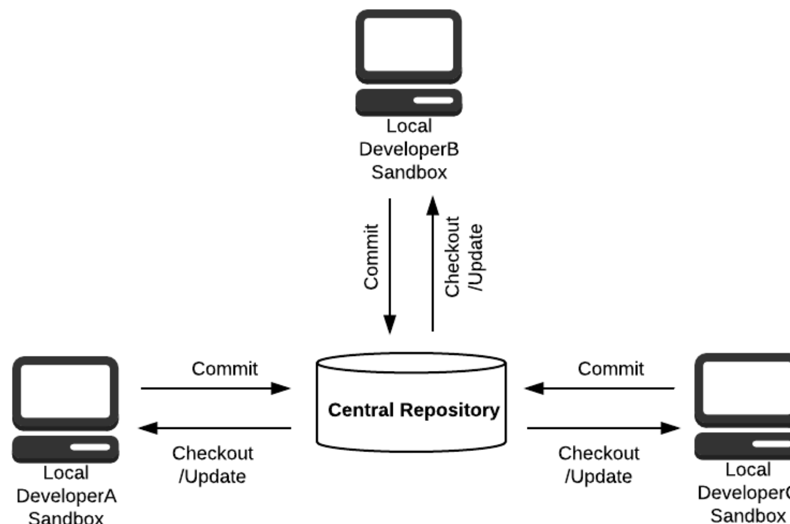

Figure 3. Architecture of the Centralised Version Control System.

There exist multiple centralised VCS solutions on the market, both commercial offerings and open-source freeware. Some of the most prominent ones are considered below in more detail (presented in chronological order).

Source Code Control System

The first centralised VCS was Source Code Control System[5] (SCCS) developed by Marc J. Rochkind (Rochkind, 1975) in the 1970s. His work provided theoretical and practical foundations for a VCS based on so-called `forward deltas', which supported the checkout and commit operations, as well as resource locking to avoid conflicts. SCCS provided support for managing changes to source code files that were continuously modified by multiple software developers at the same time. The functionality of SCCS enabled recording and tracking of applied changes, as well as storing and updating project files. Taken together, these features allowed programmers to retrieve previously committed versions, which were stored on a central server in a dedicated repository. SCCS users were also requested to lock target artefacts, modify them according to their needs, and commit the modifications, thereby creating a new version. Only after all these steps the lock could be released.

Revision Control System

The next step in the evolution of VCSs was made in the 1980s with the emergence of the Revision Control System[6] (RCS). RCS was introduced as a set of UNIX commands to enable users to manage changes in a collaborative software project in an efficient straight-forward manner (Tichy, 1985). The main focus was on source code revision tracking, documentation, and testing-related data – all these were stored in a central repository and could be accessed by developers. RCS

---

[5] http://sccs.sourceforge.net/
[6] https://www.gnu.org/software/rcs/

provided a database-like storage system for project artefact revisions. One of the main design principles of RCS was to store file revisions in a concise, space-efficient manner, aiming to minimise occupied storage space. In the first instance, the revision-related information included the revision date, the revision number, the user who committed the revision, and the log entry provided by the user, describing the updated changes.

To prevent conflicts, RCS also provided the so-called 'strict locking feature' that was supposed to be configured by the VCS administrator. This locking mechanism was required to ensure that the developer locks the file for the check-out operation and modifications in a production environment, after which the working file was expected to be checked-in again to the repository, and the lock would be finally released. As the number of software developers working on the same project was already growing in the 1980s, this feature was particularly useful if/when several developers were simultaneously working on the same file – this feature would protect against overlapping/conflicting changes and file inconsistencies.

The command *Log* was introduced to enable message logging during file check-in operations. This functionality helped developers to keep track and maintain a full list of changes, along with the authors of those changes, made in each revision. Due to memory and storage limitations back in the 1980s, the log file was physically stored only once, whereas other all further revisions were stored in the form of differences to the original file. This mechanism has become known as *deltas* (or *change sets*, as it is also called today). In a broad sense, a *delta* can be defined as an ordered sequence of edit operations applied to an original string, thereby creating a new string. Originally, RCS employed line-based deltas, i.e., allowable edit commands were rather coarse-grained (e.g., insertion or deletion of a whole code line). That is, if a single character within a line was modified, the whole line would be marked as modified. Deltas kept records of file changes and could re-

construct the most up-to-date version by combining the original state with the hierarchy of change records. Unlike SCCS, RCS made use of the so-called *reverse-delta* method for storing revisions. In this case, the system stored only the most recent revision, whereas deltas described the steps to be taken to get back to the original version from the most recent version. The rationale behind this was to shorten the time required to check out the most recent version of a project, rather than spending time on re-constructing it from the original version using multiple deltas. However, the main disadvantage of SCCS and RCS was a limited ability to work concurrently, since programmers were required to lock separate project artefacts, and therefore were not able to work concurrently on multiple files (Koc and Tansel, 2011). As a first step towards addressing this limitation, RCS introduced the novel concept of *merging* (which was called *mergediff*) that enabled parallel software development branches to be combined within a single file. This merging also provided some initial support for detection and prevention of conflicts. Additionally, the widely known VCS terms *check-in*, *checkout*, *branching*, and *update* were introduced by RCS and are still actively used.

Concurrent Versions System

The next step was made with Concurrent Versions System[7] (CVS) (Berliner and others, 1990), which quickly became one of the most popular VCSs, still actively used today. Using CVS, developers are able to retrieve the current state of a repository by copying the entire repository to their local machine from a central server. This local repository copy has become known as the *working copy*. Next, the developers were free to apply any changes to source files in their working copies, and *check in* these newly-added modifications by pushing them to the central repository

[7] http://www.nongnu.org/cvs/

over the network, thereby sharing their changes with other developers. The freshly checked-out working copy is completely identical to the project structure and artefacts of the corresponding remote repository. That is, in the beginning all artefacts in a working copy are exact copies of their respective revisions stored in the central repository. A limitation of CVS, however, was due to the strict requirement to apply changes only to the latest revision currently stored on the central server. If, in the meanwhile, another developer committed changes (thereby creating a new revision), the freshly available latest revision should be first checked out and merged with local modifications. Only after that committing would become possible.

In the CVS terminology, stored software projects are known as modules. Inside those modules, project source files are stored using a delta compression algorithm to enable space efficiency. Even though some modifications may concern the entire module, individual file versions are tracked for each file separately. As a result, matching versions of different artefacts to each other is not feasible (Mukherjee, 2011). More specifically, in CVS it is possible to track the state of a single artefact within a module, but not the state of the working copy of a whole module. This way, CVS raised the version management scope to a project level, where all artefacts of a project can be controlled. Project artefacts are grouped into folders, which can possibly include nested subfolders with files. The top-level repository folder represents the whole software project, managed by the central CVS server as a module.

Another big advantage of CVS compared to its predecessors is that for the first time it allowed checking out of files without locking. This means developers were finally enabled to simultaneously work on the very same file in parallel (Koc and Tansel, 2011). Also, CVS introduced the concept of *branching* to the field of VCSs and contributed to the current state-of-the-art solutions and their underlying branching mechanisms. Another contribution of CVS was

the support for labelling. Labels can be used to define branches or user work areas. In the CVS system, which is built on top of the RCS system mentioned above, these labels are known as *tags*. CVS was also the first VCS solution to allow anonymous read access (also known as Anonymous CVS), which gives users an opportunity to check out a project in a simplified seamless manner, without any kind of previously-needed permissions, restricting only the ability to commit changes back to the repository.

SVN

Apache Subversion[8] (typically referred to as SVN) is a modern full-featured free/open-source VCS, developed and supported by a large community and the Apache foundation (Collins-Sussman et al., 2004). It was originally introduced as a solution to overcome the limitations and shortcomings of CVS. Since then it has developed into one of the most widely used VCSs and has successfully replaced CVS, even though it is still based on the very same initial model, design principles, and APIs. SVN manages software project files and directories, and also enables users to track changes applied to project artefacts (Collins-Sussman et al., 2004). This allows users to recover and roll back to older project versions, and/or follow the history of how the project evolved over time.
SVN can be simultaneously used by multiple developers on different computers connected to different networks. In this way, SVN fosters collaboration and software co-development by providing software development teams an ability to modify and manage the same project artefacts remotely, irrespective of their physical locations. Because the work is versioned, users need not fear that some incorrect change is made to the data, i.e., all changes are recorded, and users can always roll back to the previous state. SVN is also designed to handle source code revision trees

[8] https://subversion.apache.org/

and provide support, specifically tailored to software development, such as, native understanding of different programming languages, coding assistance, and integration with existing IDEs.

ClearCase

IBM Rational ClearCase[9] (henceforth – ClearCase) is a family of commercially available computer software tools that support enterprise-level SCM of various software development assets, including source code (Bellagio and Milligan, 2005). ClearCase enables hardware and software co-development through support for data management of electronic design artefacts. Part of ClearCase is a revision control system that serves to support configuration management in a wide range of enterprise companies, ranging from several software developers to hundreds and even thousands. The functionality of ClearCase is underpinned by two configuration management models: base ClearCase and Unified Change Management (UCM). The former provides some basic infrastructure and functionality, whereas UCM is built on top of this base and extends the basic set of features with out-of-the-box support for specific business cases. Both models are highly extensible and customisable and can be configured to support a wide variety of software development scenarios and business requirements. ClearCase is also highly scalable and is able to accommodate large repository sizes, large number of files, as well as large binary files. Similar to other existing systems, ClearCase supports project branching, labelling, and versioning. More specifically, a non-exhaustive list of ClearCase features is provided below (as outlined in the official web site):

- **Version control and workspace management:** this enables full control over files, directories and other project artefacts throughout the entire software lifecycle.

---

[9] http://www-03.ibm.com/software/products/en/clearcase

- **Advanced parallel development**: this functionality enables multiple users to collaborate on the same project, and is implemented by means of automatic branching, advanced merging and delta differencing technologies.
- **Effective IP security:** ClearCase supports electronic signatures and user authentication to enable access control and security, as well as to address various compliance and governance requirements adopted within a company.
- **Seamless integration with development tools:** a very important feature, given the increasing popularity of development tools and IDEs, such as Eclipse and Microsoft Visual Studio.

Perforce Helix Core

Perforce Helix Core[10] (formerly Perforce Helix) is an enterprise-level VCS and a content collaboration platform (Wingerd, 2005). It is based upon a central database and a master repository to store and keep track of file versions. Helix Core is designed with scalability in mind to accommodate teams and assets of various types and sizes, as well as to support continuous delivery. Helix Core supports the Git functionality, and relies on multiple local and global repositories.

The key features of Helix Core are the following:

- Support for storing of files of any size, ranging from simple text files to huge binary files.
- Support for thousands of concurrent users and tens of millions of committed changes per day.
- Support for cloning existing repositories or starting repositories from scratch.
- Increased performance (it is claimed to be up to 80% faster builds than other Git solutions).

[10] https://www.perforce.com/products/helix-core

- Support for built-in mirroring that allows simultaneously pushing and pulling updates across multiple locations in parallel.
- Works with any Git-supported client application.

Decentralised/Distributed Version Control Systems

The distributed VCS model is based on the maintenance and as-needed synchronisation of several independent repositories. The resulting architecture is decentralised, with no single point of failure. This architecture assumes that every developer has his/her own project repository, through which the project code can be updated, checked out, committed, and transferred in a peer-to-peer manner (i.e., without a central coordinator), as shown in Figure 4. Although the distributed model does not depend on a central repository, its presence is still acceptable, since it can be used as a back-up location for storing project artefacts. In general, the decentralised architecture is expected to be better suited for managing collaborative development in the context of SOA, since there is a requirement to synchronise project artefacts across all interested parties (i.e., developers and their respective repositories). Most prominent decentralised VCSs are considered below in more detail.

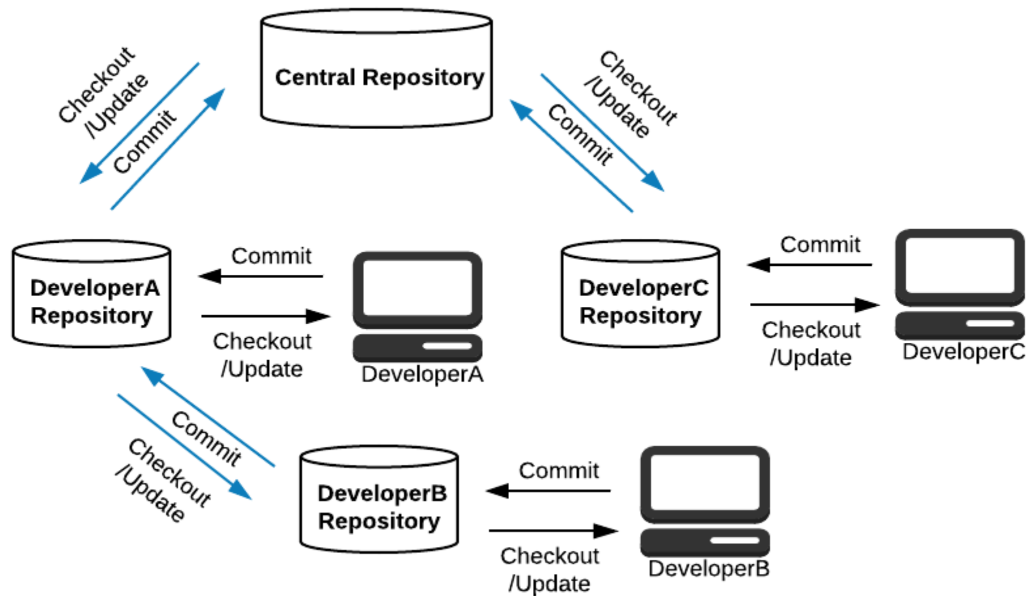

Figure 4. Architecture of the Decentralised Version Control System.

Mercurial

Mercurial[11] is a free, distributed VCS that offers its users support for efficient project management via an intuitive and easy-to-use user interface (O'Sullivan, 2009b). It is advertised as a fully-distributed and loosely-coupled system, where every team member is provided with a local copy of the overall development history. In this way, given the whole set of artefacts to work with, users can work independently of a network connection. Furthermore, in Mercurial, the basic operations of committing, branching and merging are reported to be computationally cheap and fast. More specifically, Mercurial's main features can be summarised as follows:

- **Performance:** Mercurial's data structures are designed and implemented in such a manner that it supports branching and navigation through the revision tree within seconds.

[11] https://www.mercurial-scm.org/

- **Platform independence:** primarily written in Python (with some minor bits implemented in C for better performance), Mercurial was designed to be platform-agnostic. As a result, Mercurial binary releases are available for all major platforms and operating systems.
- **Extensibility:** the core functionality can be extended with a number of official and third-party components. The former may be shipped with a Mercurial distribution or downloaded from the official web site, whereas the latter are contributed by a wide user community. Written in Python, extensions serve to enhance the default basic functionality by adding new commands or modifying the existing ones.
- **Ease of use and usability:** Mercurial was designed and implemented with usability in mind, which makes it – on the one hand – user-friendly, easy to learn and use, and – on the other hand – hard to break and robust to human errors. These features built upon the existing experience of other popular VCSs (e.g., most of the commands are inspired by SVN).
- **Open Source:** as opposed to other commercial solutions, Mercurial is free software, making it a very popular choice for SCM around the globe, and facilitating the rapid growth of the user community.

Git

Git[12] is a free and open-source distributed VCS designed to handle a wide range of tasks, ranging from small to very large projects, with speed and efficiency (Chacon and Straub, 2014). Actively developed and supported by a global community of developers, Git has evolved into an easy-to-learn-and-use tool, which makes it a very popular choice for a VCS. It has minimal overheads and

[12] https://git-scm.com/

network latency with extremely fast performance. Git implements a unique approach to branching (known as *cheap local branching*), where a branch is just a reference to a commit in the local repository. Therefore, no potentially 'expensive' (i.e., bandwidth-dependent) network operations are required. Moreover, Git offers convenient staging areas and multiple workflows. Taken together, all these features enable Git users to easily create and manage multiple local branches, completely isolated and independent from each other. As advertised on the Git official web site, the creation, merging, and deletion operations take seconds. Taken together, this functionality facilitates the following important branching features:

- **Seamless context switching**: Git offers its users a 'sandbox', where multiple branches can be safely created for testing. Developers are free to commit their changes, apply patches, switch between the initial revision and the sandbox environment, and finally merge the two branches.
- **Feature-based workflows and role-based code lines:** Git enables its users to have multiple branches for different features being developed. One of these branches is dedicated to production-ready code, whereas all other branches serve to support everyday testing and development. Eventually, tested and approved changes will be merged to the production branch and pushed to end users.
- **Disposable experimentation:** the above-described features assume that Git users can have multiple local branches for personal use and experimenting. Once no longer needed, these branches can be seamlessly merged and discarded, without affecting anyone else's work.

All these novel branching techniques were introduced to give developers some freedom when choosing what to push to another, possibly public, repository. With Git, developers can keep many of their branches locally and decide what branches should be shared (i.e., a single branch, some of

them, or all of them). This sharing approach is intended to provide developers with more freedom and motivate them to test new ideas without being worried about when and how the locally-developed experimental code will be merged and shared with the rest of the team.

Pastwatch

Pastwatch[13] is a distributed VCS that duplicates most of CVS's functionality, and further extends it with several novel useful features. The default functionality includes the traditional version control features and operations, such as checkout, commit, diff, etc. executed in an atomic manner Thanks to the delta-based model for storing and retrieving version differences, updating a user's working directory and committing are extremely fast, since calculating deltas and applying them take place locally.

As far as novel features are concerned, Pastwatch introduced support for offline repository operations, even when disconnected from the Internet. This functionality is underpinned by a free hosting service, Aqua, that is used for integrated data replication (Yip et al., 2006). Briefly, Aqua is a distributed service, connecting Pastwatch machines distributed around the world in a peer-to-peer manner. Shared projects are replicated on each team member's machine (as well as on the central Aqua server), thereby ensuring that shared projects remain available at all times. Even if some team members (and their machines) disconnect from the network, the remaining members will not be affected. Moreover, disconnected team members are still able to synchronise with the rest of the team using integrated peer backup that can be used to fetch project updates from the nearest available team member.

---

[13] https://pdos.csail.mit.edu/archive/pastwatch/

Comparison of the Two Types of Architectures

There is a considerable difference in how changes are committed and interested parties are updated in the two architectures. A commit operation is two-fold. First, all changes are committed locally, i.e., all file modifications are recorded in the working copy as a snapshot. Next, snapshots are pushed to the global external repository, from where they become available to other developers. There are two strategies for how frequently changes are committed. Usually a user can commit multiple snapshots locally, thus building a fine-grained history of changes and modifications. Snapshots should be committed as frequently as possible to enable fine-grained roll-backs. Committing changes late makes them more difficult to merge, as potentially more edited lines in the artefacts can be conflicting (Estublier and Casallas, 1994). When the latest committed snapshot represents a stable software release, all snapshots, including the latest one, may be pushed to the central repository to become available to others (as in a centralised approach). It is worth noting, however, that shared changes, pushed too frequently to the central repository, should not corrupt the project (e.g., resulting in an uncompilable source code), hindering further work of collaborators. Ideally, commits should not be shared with other developers if they leave the project in an unstable (i.e., uncompilable) state. However, this needs to be balanced with the requirement to frequently synchronise with others in the development team. After a task is completed, all recorded changes should be shared. This is challenging in the context of centralised VCSs, since every commit is automatically shared with all other users. Working in separate branches can solve this problem, but unless all developers work in their own branches, there will always be co-developers affected. Distributed VCSs address this issue by differentiating between local commits and global pushes. Local commits are not shared with other users and can therefore be used to keep track of fine-grained and possibly destabilising snapshots that stop a project from being compiled. A push

operation shares all unshared local commits at once, but can be restricted to sharing with only one participant. A combination of the two strategies enables a flexible and convenient approach to separating personal and global commits and maintaining the stability of the project. In practice, in the software development industry a single person is assigned to collect all contributions and publish them to a central place.

Furthermore, despite the evident benefits of providing support for a wide range of version control activities, the centralised approach to version control may potentially suffer from the following drawbacks:

- **Single point of failure (SPOF):** this is a shortcoming that is intrinsic not only to VCSs, but to a wider range of centralised architectures in general. A SPOF is a part of a larger system that might stop (or at least slow down) the entire system from working if/when it fails (Dooley, 2001). In general, SPOFs are to be avoided in any system that aims to provide high levels of availability and reliability, be it a business process, an industrial system, or a software application, such as a VCS. In other words, if the central version control repository fails for whatever reason, all the dependant users will not be able to perform the version control operations until the repository recovers.
- **Fixed location:** typically, the central repository is situated in a fixed physical location. If the network connectivity to this location is affected, the expected version control functionality service will perform poorly. In an era of global economies, it is common that software development team members reside in different countries, located remotely from the VCS server and, therefore, potentially having to cope with network delays and latencies during data transfer operations (Mukherjee, 2011). In these circumstances, distributing servers across different locations has the potential to decrease the communication delays

for all developers, provided that all geographical locations are covered. However, this, in turn, leads to synchronisation-related issues, since it becomes more difficult to synchronise all copies of the repository.

- **Bottleneck issue and low scalability**: a single central repository may become a `bottleneck' of the overall system. Similar to a SPOF, a bottleneck typically occurs when the stable and reliable operation of a larger system is limited by the capacity of one of its components. Typically, the bottleneck has lowest throughput among all system components involved in computation. In other words, if the central VCS is unexpectedly used by an increased number of participants (i.e., a sudden peak in activity), the server may run out of available resources, which, in turn, may lead to partial unavailability to some users, or to a failure of the whole VCS.

The distributed VCS architecture successfully addresses these limitations of the centralised architecture. The decentralised architecture successfully addresses the SPOF problem, as changes made by a developer or a sub-team of developers are handled within their local repositories, rather than being pushed to and managed by central source repository. From this perspective, it is more convenient than the centralised architecture. On the other hand, it is not yet fully optimised for developing distributed systems, and also suffers from the following potential shortcomings, which primarily stem from the inevitable requirement to synchronise individual copies of a project across all repositories:

- **Requirement for replication:** to keep individual repositories synchronised and consistent, all project artefacts are expected to be continuously replicated across the repositories. This ensures that each developer works on the latest version, thereby avoiding avoid potential conflicts.

- **Bottleneck still exists:** despite the presence of multiple distributed repositories, a single repository may still become a bottleneck whenever there is a need to synchronise a large number of revisions globally.
- **Limitations of the centralised approach may still exist:** a sub-team of developers, all connected to the same developer and his/her repository, may be seen as a smaller-scale version of the centralised architecture, and, therefore, may suffer from the same problems.

Taken together, these limitations prevent the distributed VCS architecture from being universally applicable and widely adopted. There are certain software development scenarios, in which the distributed nature of the decentralised VCS architecture, despite its benefits, may still be not sufficient. An example of such scenarios is service-oriented systems, in which VCSs also play an important role for service revision management, aiming to minimise the impact of changes on service consumers, and improve the overall manageability of services. However, it is an inherently challenging problem because of the distributed nature of SOAs, which typically suffer from the lack of centralised control, emerging problems due to service changes occurring only at run-time, and the overall opacity due to the separation of service APIs from their implementations (Novakouski et al., 2012). As a result, to date there seems to be no holistic and comprehensive solution proposed either by academic researchers or industrial practitioners to address this problem. Moreover, most current efforts seem to have adopted a reverse approach, in which service-oriented software development is tailored towards the existing SCM systems and facilities, instead of developing a truly service-oriented solution, driven by existing requirements (Raygan, 2007).

2.4. Service-Oriented Architecture

The software engineering discipline has gone through several evolutionary steps: after Object-Oriented Programming (OOP) in the 80s (Rumbaugh et al., 1991) and component-based software

engineering (CBSE) in the 90s (Heineman and Council, 2001b), the IT world is now being influenced by Service-Oriented Architecture (SOA). SOA is a style of software design, where distributed software systems are built from network-accessible software components in a loosely-coupled and technology-agnostic manner. It is an evolutionary step in software development that builds upon and further extends the fundamental principles of OOP and CBSE, such as self-description, explicit encapsulation, and run-time functionality loading (Sprott and Wilkes, 2004).

The basic unit of functionality in SOA is a service. In a broad sense, services can be defined as reusable software components, remotely accessible over the network in a loosely-coupled and technology-agnostic manner via a standardised well-defined API (Wei and Blake, 2010). Separate applications expose their functionality as services to other applications via established and standardised communication protocols, making it possible to quickly and easily assemble distributed service-based systems from existing 'building blocks'. Being loosely-coupled, services are designed and implemented to provide certain functionality without being aware of how other components of a larger SOA software system are implemented. By hiding implementation details and only exposing APIs, various services can be seamlessly integrated into a larger service-based software (SBS) system, in which individual elements can be replaced or removed. From this perspective, services can be treated as 'black boxes' that hide away internal implementation details and only expose their APIs, such that they can be discovered and invoked. Services are typically supposed to be mainly static entities, whereas their combinations and configurations are expected to change frequently. This flexibility to dynamically combine and configure SBS systems is supported by well-defined, self-descriptive, standardised and technology-agnostic APIs.

As a result, nowadays SOA represents an established architectural design pattern that promotes providing application functionality as networked services via standard protocols and APIs. SOA

has gained popularity and wide adoption thanks to its enhanced support for interoperability, service abstraction, service discovery, service autonomy, service statelessness, re-usability, and loose coupling.

As a result, more and more enterprises have embraced the SOA approach, paving the way for a whole new paradigm in how ICT products (not just software) are developed and delivered to end users – that is, to Service-Oriented Computing (SOC). SOC assumes that various elements of a distributed SBS system may be owned and managed by different companies under different ownerships and administrative domains (MacKenzie et al., 2006), thereby going beyond the pure ICT field. As a key benefit of adopting SOA, enterprise managers and decision makers see *business agility*, i.e., an ability to rapidly change and adapt existing software systems in response to constantly evolving underlying business requirements and goals. The underlying service-based ICT facilitates loosely-coupled and easily modifiable business processes. As opposed to the traditional 'monolithic' architectures that require extensive manual effort to be modified, loosely-coupled atomic services are much easier to replace and adjust if/when required (Sprott and Wilkes, 2004).

The continuing service-orientation, supported by the emergence of Web 2.0 (Schroth and Janner, 2007), paved the way for the appearance of the so-called Internet of Services (IoS) (Buxmann et al., 2009) – a concept referring to a global architectural model, where ICT resources are increasingly ubiquitous and available through a network connection as remotely accessible services (Bechmann and Lomborg, 2014). Not limited only to software services, the IoS is a much broader concept, which also enables remote provisioning of hardware and networking infrastructures, storage facilities, and development tools. From this perspective, it is closely related to Cloud Computing, which has revolutionised the traditional model for accessing and using

computing, storage and networking resources. By commoditising services, Cloud Computing enables end users to provision ICT resources over the network in a manner similar to traditional utilities, such as gas, electricity and water (Buyya et al., 2013).

2.4.1. WSDL/SOAP-based services

Traditionally, there are two main styles to implement SOAs identified in the literature (Pautasso et al., 2008). These two approaches are currently widely adopted by industry, and are recognised as the two primary technologies to implement enterprise-level Web services.

The approach is based on Web Service Description Language (WSDL) (Christensen et al., 2001) – a World Wide Web Consortium (W3C)-standardised format for describing Web services. Also known as *message-oriented*, it is closely related to the Simple Object Access Protocol (SOAP) (Gudgin et al., 2003), which is used to exchange information in the form of XML messages, usually containing a call to a method and defined according to the WSDL service description. Information about WSDL Web services is typically stored in a special kind of registry, known as a Universal Description, Discovery, and Integration (UDDI) registry (Sleeper, 2002). The architecture of the WSDL/SOAP-based Web services is represented in Figure 5.

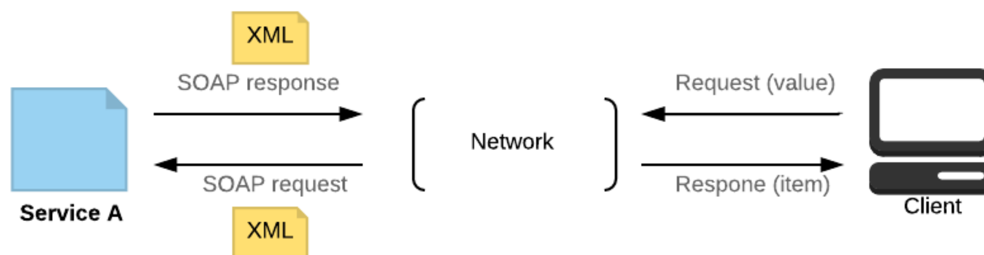

Figure 5. WSDL/SOAP-based architecture.

The roles of each technology in WSDL/SOAP-based services can be summarised as follows. WSDL describes the service and its supported operations, whereas UDDI provides information about what services are available. SOAP serves to describe the format of messages required to

access and invoke a service. Finally, XML is used to tag and structure both WSDL descriptions and SOAP messages. By using XML as a way of structuring data, the WSDL/SOAP-based style is technology-independent and is not constrained by a specific implementation framework, language, or tool.

2.4.2. RESTful services

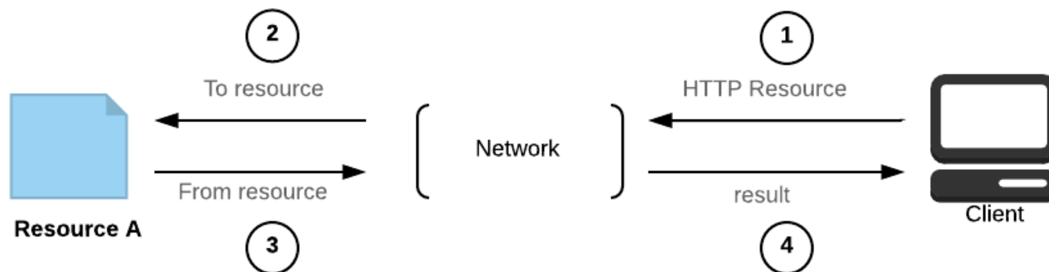

Figure 6. RESTful architecture.

Another way of implementing technology-agnostic Web services is using the Representational State Transfer (REST) architecture and the Hypertext Transfer Protocol (HTTP) to support communication and information exchange with and between services. Every object on the Web is represented by its unique Uniform Resource Locator (URL) that is used to discover, describe, and access that specific object. Object URLs are also used by RESTful services that can invoke the standard HTTP operations (i.e., POST, GET, PUT, DELETE, and HEAD) to access and manipulate Web resources by referring to their URLs. This interaction is depicted in Figure 6. The RESTful style is considered to be simpler, more light-weight, and easier to use and engineer than WSDL/SOAP-based services. On the other hand, REST typically does not have any service description and/or contract (similar to WSDL), which makes engineering a client application a more complicated process. In general, for small-scale APIs and agile development, the RESTful architecture would be a more convenient choice, whereas projects requiring more complicated

APIs and richer functionality (e.g., multiple operations, custom data types, well-defined service description, etc.) are better implemented using the WSDL/SOAP approach (Mulligan and Gračanin, 2009; Wagh and Thool, 2012).

2.5. Summary

The goal of this chapter was to present the fundamentals of Software Configuration Management in general and, more specifically, Version Control Systems as key enabling technologies to support CI. The latter has been recognised as an efficient and effective tool to address the pressing challenges related to the ever-growing complexity of enterprise software systems. VCSs enable software development teams to handle different parts of an overall software product as separate identifiable versions, which could be configured accordingly so as to assemble different configurations of the same overall software product. The chapter then explained and discussed the two existing approaches to implement VCSs: centralised and distributed. The former relies on a single point for controlling and managing software revisions, whereas the latter is typically implemented as a network of distributed servers, each of which serves to perform the version control activities. Potential benefits and shortcomings of the two approaches were discussed to identify their suitability to support version control in modern service-oriented software development, as well as to enable CI for distributed service-oriented software development. As it turns out, however, these existing approaches seem to be not immediately applicable, and there is a need for a novel mechanism. Accordingly, the chapter finally describes service-oriented software development, and discusses its main differences in comparison to the established approaches. This discussion will be further extended in the next chapter, which will survey relevant literature to identify overlaps with the proposed approach and identify existing research gaps in the field of software compatibility specifically in the context of SOAs and associated CI challenges.

3. Chapter 3. Software Compatibility in SOAs: State of the Art and Existing Gaps

The goal of this chapter is two-fold. First, it introduces the state of the art in the domain of software compatibility in general and in SOAs in particular. It provides theoretical background for the key concepts: service change, change detection, change notification, service compatibility, and service versioning. Second, using these concepts, the chapter surveys existing approaches to software compatibility, aiming to provide an overview of existing techniques and tools. This overview identifies research and technological gaps to be addressed by the research presented in this thesis. As such, this survey serves to identify the fundamental requirements of a future solution to software compatibility in SOAs.

3.1. Software Compatibility

In an SBS system, service providers modify their services independently of service consumers. They develop and make changes to their service implementations in accordance with the versions of their service interfaces. Service changes made by different developer teams adhere to disparate guidelines and rules. These changes have a major impact on service compatibility, and an inability to handle compatibility issues can result in inefficient service provisioning or even service disruption for consumers. As compatibility issues cannot be completely avoided, it is important to develop an efficient and manageable approach to the tracking and handling of compatibility issues – a key requirement for enterprise-level CI.

Compatibility is an important concept in service evolution and is closely related to versioning services for the purpose of change management. Service compatibility is typically categorised into two groups – *backward compatibility* and *forward compatibility* (Fang et al., 2007; Orchard, 2007). In backward compatibility, a new version of a service can be introduced without disrupting service consumers. In other words, the major focus of backward compatibility is to continue provisioning

the service to existing consumers, while a new version is being introduced. Forward compatibility, on the other hand, focuses on the design of supportive services for future versions.

- **Backward compatibility:** In backward compatibility, changes to services could be of two types: backward compatible, and non-backward compatible (Lublinsky, 2007; Novakouski et al., 2012).
  - Backward compatible changes: requests from consumers of older versions of the service can be interpreted by new versions without interrupting the service (Parachuri and Mallick, 2007).
  - Non-backward compatible changes: the newer version is unable to interpret data formats of older versions of the service (Parachuri and Mallick, 2007).
- **Forward compatibility:** Forward compatibility means that newer versions of a service can be deployed without causing service interruptions to existing consumers. As opposed to backward compatibility, forward compatibility is much harder to achieve, since the service is expected to gracefully interact with a number of yet-to-come features (Evdemon, 2005). According to Novakouski et al. (Novakouski et al., 2012), forward compatibility is a desired goal in an environment where services experience frequent continual changes. However, implementation of forward compatibility is challenging, as developers have to ensure compatibility with yet unknown future versions of the service (Novakouski et al., 2012).

The issues of compatibility in the process of service evolution are reasonably well known. However, little research has been done on the approaches to avoid or overcome these issues. More importantly, no work has been done in the context of Service-Oriented SCM systems that are used to support concurrent development of SBS systems to ensure they can adapt to concurrent changes,

such as changes in structures, behaviour and policies. Due to this gap, developers are often unable to manage changes and identify compatibility risk factors involved when developing SBS systems. In order to address this gap, this research synthesises various studies in order to help developers identify the causes of compatibility issues and understand their implications in developing services when using Service-Oriented SCM systems.

3.1.1. Software Changes in SOA

It is worth noting that services are typically static and stable entities, whereas their compositions (i.e., SBS systems) are expected to be rather dynamic – i.e., they change and evolve. When emerging business requirements have to be implemented in the underlying IT systems, according to the SOA paradigm, this should not trigger excessive manual changes to the software source code. There are, however, rare situations when the service source code needs to be changed, and this needs to happen with care and caution.

Changes are usually triggered by the need for a service upgrade and can occur at different stages of a service's lifecycle, such as service identification, design, development, deployment and runtime (Parachuri and Mallick, 2007). An upgraded version may or may not be compatible with its previous versions. For example, updates to a service may concern changes to the format of its messages, which potentially affects the communication between the service provider and the service consumer. There are two common aspects in which changes are most likely to occur:

- **Service Implementation:** this aspect deals with the source code of a service. This kind of changes can be handled by an SCM system through revision control.
- **Service Contract:** this aspect deals with the communication between service providers and consumers. These changes can be further categorised into interface changes and policy

changes (Parachuri and Mallick, 2007). The former can be also classified into *changes in service interfaces* and *changes in service policies*.

As far as service contracts are concerned, it is important to understand the common established way of defining and advertising service APIs – i.e., WSDL, which is a standardised XML schema for describing Web services, which allows service consumers to use the functionality of a Web service without explicit knowledge of how the service is implemented.

WSDL 2.0 defines a model for describing Web services in two distinct sections: the abstract functionality and the concrete details (Chinnici et al., 2007). This conceptual schema of the WSDL 2.0 model is shown in Figure 7. At the abstract level (also known as the *functional interface*), the WSDL document describes the general Web service API and defines a messaging model in an abstract manner, without references to specific technologies, protocols and/or encodings. Such a description lists all the operations offered by the service, individual parameters for each operation, as well as abstract data types used by operations. In contrast, the concrete section of the WSDL document provides concrete network addresses, specific protocols, and data formats that can be used to invoke the operations defined in the abstract part of the WSDL document. Frank et al. (Frank et al., 2008) argued that separating the service description effectively divides the API description from the implementation, because service consumers are bound to services only through APIs.

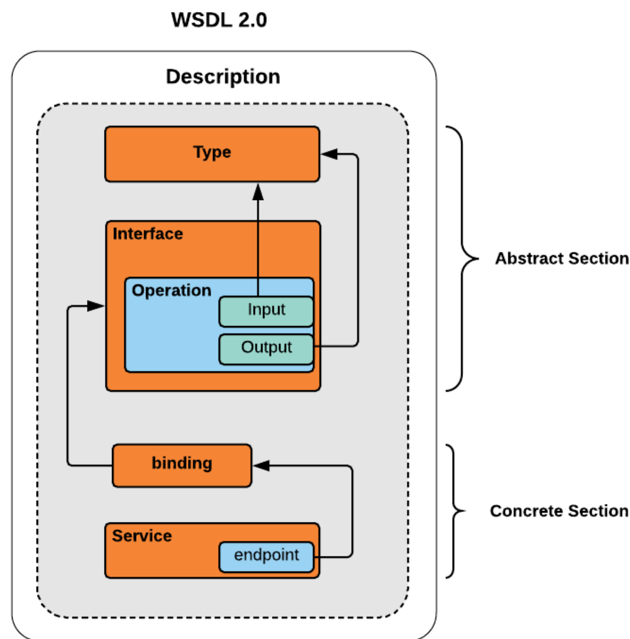

Figure 7. Conceptual schema of a WSDL 2.0 document.

Over time, a service provider will make changes to a service and, thus, release a new version. This new version will be accompanied by a new Web service contract, because the new service version has triggered a modification either in the abstract WSDL description, or in one of the other contract documents, logically connected with the abstract WSDL description (Bharti, 2008). As the WSDL description consists of multiple elements, changes to any of these internal elements result in changes – and, thus, a new version – of the whole WSDL document. A detailed overview of possible changes to service contracts and implementations is presented in the next section.

3.1.2. Service Changes and Versioning

Evolution of service descriptions is the main reason behind service versioning and service compatibility checks. Different schemas and approaches can be used to classify changes in services, among which the classification schema based on compatibility with other versions of the service is the most common one (Novakouski et al., 2012). In this respect, understanding the nature of

changes as well as their effects on the communication between the service providers and consumers can help outline the potential scope for SORC (Borovskiy et al., 2008). Table 1 lists possible changes to services.

Table 1. Different types of changes in SOA.

Scope of Change		Description	Backward Compatible
Service Contract Change	**Service Interface Change**	Adding an operation	Yes
		Removing an operation	No
		Renaming an operation	No
		Changing a binding (SOAP, etc.)	Yes/No
		Adding a binding	Yes
		Adding optional parameters or attributes to message definition	Yes/No
		Adding required parameters or attributes to message definition	No
		Removing an optional/required parameter or attribute from message definition	No
		Changing the definition structure of data type	No
		Renaming and removing composite data type	No

	Service Policy Change	Changing Quality of Service (QoS)	Yes
		Adding an optional assertion	Yes
		Adding a required assertion	No
		Changing security and reliable messaging	No
Service Implementation Change		Fixing bugs	Yes
		Any change affecting the service API	No

In order to track service changes and differentiate between various implementations of the same service, versioning techniques need to be applied. Services are versioned through specific numbering and naming schemes that facilitate the management and identification of versions on the basis of major and minor changes made to the services. Service interface versioning directly deals with the service description and can be defined as metadata of the service. The metadata contains information about the data structures used by the service and the way it can be invoked by consumers.

Service version numbers may follow different formats, among which one of the most widely adopted is the one based on the sequence of major and minor releases which reflect the importance of the applied changes (Erl et al., 2009). For instance, if a service version is numbered 2.3, then 2 indicates a major change to the service, while 3 represents a minor change. Since versions are numbered incrementally, the latest version of a service is allocated the highest number. When the service APIs undergo significant changes, major versions are required, which will probably cause service disruption to consumers as these changes are likely to be backward incompatible. Typically,

minor revisions do not lead to service disruption as these changes are most likely to be backward compatible (Erl et al., 2009; Peltz, 2003).

Another strategy for naming service versions is using release dates. Brown et al. (Brown and Ellis, 2004) and Peltz et al. (Peltz, 2003) provide a comprehensive set of solutions for versioning services, including maintenance of service descriptions in the case of compatible versions and the use of different namespaces in the case of incompatible versions. A new XML namespace is introduced for a major change to a service. In the case of a minor version release, the version identifier is used together with the namespace. Generally speaking, a new XML namespace shows an incompatible version, while a minor version identifier shows a compatible change in the version. As a result, every introduction of a new namespace will cause service disruption to consumers that still use the old namespace. Peltz (Peltz and Anagol-Subarrao, 2004) further suggested the implementation of an intermediate router to translate consumer requests from the older namespace. Both date and/or version stamps are used in the namespace, which follows the W3C schema. A major advantage of this approach over the previously described numbering approach is that it can provide more meaningful information when a new version is released.

An established strategy adopted by many solutions (including the one presented in this thesis) is based on numbering service versions by enumerating version numbers from 1. Version number 0 is a special case used to point to the latest version of a service (Sarib and Shen, 2014). Whenever a new version is released, it always takes number 0 and the previous version takes the highest available version number incremented by 1. Hence, the latest version is always referenced for service consumption. However, this versioning strategy does not consider major or minor changes and should be combined with other versioning strategies (Peltz, 2003) in order to clearly distinguish between compatible and incompatible versions.

3.2. Survey of the State of the Art

This section presents a literature review of related research. This survey has revealed a number of existing relevant works, which can be classified with respect to several important criteria. The following methodology was used to systematise and structure the survey.

Besides relevant papers and authors identified throughout the several years of PhD research, the survey also includes relevant works identified through the well-established online databases and search engines Scopus, Web of Science and Google Scholar. The search was conducted using relevant keywords, such as “versioning”, “WSDL”, “service changes”, “service interface”, “API change”, “service change adaptation”, “service evolution”, etc. The lower limit for the search time frame was set to 2000 – this is when early works on WSDL and SOA started first appearing. As a result, more than 100 papers were identified through the keyword search. Next, by thoroughly browsing through paper abstracts and conclusions, only 42 relevant approaches were retained. Finally, by reading and analysing the remaining papers, 23 were identified as closely-related approaches, and each of which is discussed further below.

3.2.1. Related Works on Change Detection

Consumers of a service can identify changes in a service in various ways. However, there are two main approaches to detect changes, commonly adopted and discussed in the literature: changes are either detected by consumers themselves, or they are advertised by service providers to all interested parties. Hybrid approaches are also possible.

Change identification by service consumers

Service consumers identify service change via a “pull” strategy for information delivery. That is, service consumers are expected to *pull* new updates themselves. In the worst-case scenario, this process is performed in a completely manual manner. That is, service consumers need to regularly

check with the service provider for any changes introduced to the service. Arguably, this approach suffers from insufficient automation and is prone to potential incompatibility problems due to references to outdated services.

A potential solution to this problem is to automate the change detection process to a certain extent, so that the service consumer still remains responsible for detecting service changes, but is able to detect these changes in a timely, reliable, and automated manner. For example, Leitner et al. (Leitner et al., 2008) presented an approach through which consumers of a service make use of service proxies to invoke various versions of the service. The approach enables service consumers to detect changes in service-oriented systems using service version graphs and selection strategies. The consumers are able to query the service provider regarding possible changes, build a service version graph, and dynamically invoke different versions by transparently rebinding service proxies.

Change notifications by service providers

A reverse pattern is also possible – service changes, once implemented, are advertised to all interested consumers by the service provider. From this perspective, these service change notifications follow the "push" strategy for information delivery, and service consumers are expected to be notified of changes in a timely manner.

One of the earliest examples of this strategy was discussed by Peltz and Anagol-Subbarao (Peltz and Anagol-Subarrao, 2004), who presented an approach where a notification is generated whenever a new service version is released for consumers, or an existing version is decommissioned or ceased. For example, when a version is nullified or decommissioned, consumers will receive a notification that the version is no longer supported. The authors proposed using existing design patterns to facilitate the processes of creation and modification of coarse-

grained Web service components. For example, the Web Services Facade pattern (Hogg et al., 2004) provides a framework for maintaining multiple Web service versions simultaneously, which becomes an increasingly complex task as the number of versions grows. This pattern also minimises the complexity of service APIs being exposed to the world, i.e., instead of having multiple access points for several services, a single composite Web service can be designed and provided to service consumers. The facade will become a single entry point for all incoming service calls. Among the benefits of this approach are the loose coupling between the service and its consumers, a single control point for increased manageability, and improved performance of the overall SBS system due to the reduced number of unnecessary network calls.

Change notifications via a third party

Hybrid solutions, where changes are propagated through a third-party mediator, also exist. For example, an approach proposed by Fang et al. (Fang et al., 2007) not only provides a possibility for service consumers to know whether a version is new or deprecated, but also extends UDDI with service version functionality. It allows consumers to subscribe to the notification system of a service by specifying the types of events about which they want to get notified. From this perspective, the approach benefits from using a third-party broker responsible for storing information about service versions and providing this information to interested parties upon request. With this kind of decoupled communication, service providers and consumers are not required to know about each other, whereas all service changes are propagated to consumers via the UDDI registry. The service provider first "pushes" the latest update to the UDDI registry, upon which all subscribed service consumers are expected to be notified to keep up to date with the latest service version.

Another approach that relies on the presence of a third-party registry is reported by Abang (Abang Ibrahim, 2016). In his thesis, the author focuses on provenance and versioning in e-experiments and services and, more specifically, on how different e-experiments can be reproduced in a coherent and deterministic manner. To address this requirement, the author proposed service versioning to be incorporated into provenance to address deficiencies in the existing schemas. Additionally, the author suggested a new Web service versioning method to support experimental reproduction in situations when one of the services constituting a complex multi-step e-experiment needs to be replaced. It provides an alternative service in case the current service is not available or missing. To achieve this, a service with versioning metadata needs to be registered with a service registry that acts as a centralised data repository. Service information includes the service version naming convention and the date and time of service creation. As a result, the proposed approach is able to generate an additional "wasVersionOf" causal dependency to track service versions and associated dependencies. This ability to recommend an alternative service version if the original service is unavailable and select a specific version of a service allows users to reproduce and promote new discoveries in analysing different service versions. The author implemented and evaluated the resulting reproducibility framework and summarised the key theoretical findings in a reproducibility taxonomy.

Other relevant approaches for change detection

Kohar and Parimala (Kohar and Parimala, 2017) looked into changes at the service level, i.e., changes that concern Web service APIs (WSDL descriptions). In order to measure service evolution, a three-dimensional suite of metrics was proposed. The three types of metrics cover: *i)* the evolution of the service itself, *ii)* the evolution of the service client code, and *iii)* the evolution of the service usefulness. Both service providers and consumers are expected to benefit from the

proposed metric suite. On the one hand, the service provider can use the metrics to measure the evolution of a service. On the other hand, with the help of the metrics, the service consumer can measure the impact of service evolution on the client code and the overall usefulness. Among the main benefits of the proposed metric suite are the simplicity to compute and collect, technological independence, and the fact that no unnecessary managerial burdens are introduced to the service provider or the consumer. To empirically validate the proposed approach and the metrics, the paper presents and discusses a number of experiments, conducted on real-world and simulated data. As claimed by the authors, the conducted experiments suggest that there are cases when the service producer can clearly see whether introduced changes benefitted the consumer, as well as cases when such changes are undesirable and, therefore, should be re-considered.

Chauhan et al. (Chauhan and Probst, 2017) introduce the novel concept of Architecturally Significant Requirements (ASRs), i.e., non-functional requirements that can have a significant impact on the architecture of a software system. For example, for a cloud-based application, ASRs may concern scalability, elasticity, interoperability, security, and privacy. In their paper, the authors claim that ASR identification, classification, and change management are pressing challenges to be addressed. The latter requirement is especially pressing, since quality in cloud-based systems depends on clear understanding of how changes in one ASR aspect can impact other ASRs. From this perspective, ASRs can be seen as service contracts similar to WSDL descriptions. To address this challenge, the authors presented a framework for requirements classification and change management, primarily focusing on distributed Platform-as-a-Service and Software-as-a-Service systems, as well as a wide range of other complex service-oriented software systems. According to the proposed framework, ASRs are classified into three classes: *i)* system management requirements, *ii)* communication and collaboration requirements, and *iii)* monitoring requirements.

Each of these requirement classes also includes multiple key classification attributes. The resulting taxonomy underpinned a probabilistic analysis method, which is used to analyse the impact of ASRs on each other, as well as to analyse the impact of change in one of the requirements on other related and dependant requirements. The presented analysis method used five different types of relations (i.e., dependency, composition, complementation, contradiction, and inverse proportionality) to evaluate the impact of changes. The presented probabilistic analysis method can be used to control run-time system configuration to achieve desired quality in a cloud-based system, whenever changes and modifications occur to ASRs.

Kumar et al. (Kumar et al., 2017) focus on the issue of change-prone services, defined using WSDL, and are therefore likely to cause incompatibility problems. To address this challenge, the authors proposed using source code metrics to predict changes in Web service WSDL APIs, using kernel-based learning techniques and prediction algorithms. To validate the proposed approach, the paper presented a case study based on the popular online auction platform eBay. As a result of this empirical evaluation, it was demonstrated that a predictive model, developed using all metrics, resulted in slightly better prediction accuracy, as opposed to models on a selected set of metrics. The predictive power for various source code metrics and the accuracy of predictive models were above 80% for three different kernel types and for five different versions of the metrics. The predictive model developed using the Least Squares Support Vector Machines (LS-SVM) linear kernel demonstrated better results compared to other kernels. Furthermore, the results demonstrated a consistent accuracy above 80% – an indication of the overall effectiveness of the proposed approach.

Sohan et al. (Sohan et al., 2015) focus on service API changes and aim to address situations when changes to the web API of an individual application may impact dependent applications. As

discussed by the authors, this problem may occur due to outdated and deprecated versions of APIs, limited and inadequate documentation required to upgrade to a newer version, and insufficient explanation and communication of applied changes. Accordingly, this research work surveyed these existing challenges, currently faced by service consumers when dealing with evolving Web APIs, outlining differences between the state-of-the-art research and the established industry practices, and, as a result, identifying gaps. The paper reported a case study in which multiple Web APIs from different application domains were chosen, analysed and compared in order to provide a broader view of the current industry practices as far as Web API versioning, documentation and communication of changes are concerned. As an outcome, the paper distilled a list of suggested best practices and recommendations that include *i)* applying semantic versioning techniques, *ii)* using separate releases to differentiate between bug fixes and new features, *ii)* using auto-generated API documentation with embedded cross-links to changelogs, and *iv)* providing automated API explorers in a live mode.

Papazoglou (Papazoglou, 2008) theorised on the topic of service changes and evolution, providing several classifications and categories. First, the author suggested that a service can evolve due to changes in *i)* structure (e.g., attributes and operations), *ii)* operational behaviour and policies (e.g., adding new business rules and regulations), *iii)* types of business-related events, and *iv)* business protocols. Second, the paper distinguishes between two types of service changes: *shallow* and *deep* service changes. Shallow changes refer to changes that are local to a specific service or clients of that specific service. Shallow changes typically belong to the structural level and business protocols but also to so-called “cascading” changes that concern not only local services, but possibly an entire value chain of an SBS system. Deep changes include operational behaviour changes and policy-induced modifications. Next, the author focused on shallow changes and

proposed a structured approach based on a robust versioning strategy to support multiple versions. Focussing on service compatibility, compliance, conformance, and substitutability, the suggested theoretical approach aims to maximise code re-use, provide better management of deployment and maintenance of services and protocols, and facilitate smooth upgrade operations and improvements, while supporting previously released existing versions. As far as deep changes are concerned, these are more difficult to address, as they concern changes to whole process value chains. To address this issue, the author introduces a change-oriented service life cycle methodology, which enables service reconfiguration, alignment and control in parallel to changes. As claimed by the author, this proposed approach provides common tools to reduce cost, minimise potential risks and improve development agility, as well as helps companies to ensure that the right versions of the right processes are made available at all times.

3.2.2. Related Works on Service Versioning

Several approaches to service evolution are used for service versioning. These approaches tend to differ on the basis of the problems involved. Choosing an appropriate approach has a high impact on successfully completing a service versioning task. As a result of extensive research in this area, good practices and guidelines are available for service providers to understand the most optimal and efficient way of versioning their services (Andrikopoulos et al., 2012; Brown and Ellis, 2004; Erl et al., 2009; Evdemon, 2005; Leitner et al., 2008; Novakouski et al., 2012).

Bachmann (Bachmann, 2015) discussed challenges in the field of Web services when changes occur. Although there is no formal mechanism for identifying changes, they can be identified through a detailed analysis of Web service providers and consumers. The author highlights that, depending on the perspective (i.e., developer, service provider, service broker, service consumer), the focus shifts to design, runtime or configuration time. Additionally, the author differentiates

between syntactic and semantic changes. The former typically refers to changes in Web service client APIs, whereas the latter describes changes occurring in service implementations. Taken together, a solution is outlined using a UDDI-based versioning framework for WSDL services, which makes SBS systems more robust to a wide range of service changes, including changing the security configuration, adding mandatory parameters to complex types, deletion of operations, renaming of operations, deletion of parameters of complex types, changing the type of an element, adding optional response parameters, etc. In this surveyed approach, using UDDI allows service consumers both to be notified of any modifications and to poll the registry at regular intervals to keep up to date with recent changes.

Novakouski et al. (Novakouski et al., 2012) discussed the need for a versioning policy in service-oriented systems. The authors claim that all stakeholders must write and agree to policies that govern what to version, how to version it, how to communicate and coordinate changes, and how to manage the life cycle of changes. Similar to the previous approaches, the authors rely on a UDDI registry as a way of advertising service changes to interested stakeholders. Besides these theoretical considerations, the authors also provide an extensive set of practical recommendations regarding Web service versioning in SOA software development. This guidance also provides frequent mistakes and potential pitfalls, which helps understand common problems that occur in object-oriented domains due to versioning.

Evdemon (Evdemon, 2005) in his report surveyed several versioning techniques used by the Microsoft platform. The author also takes into consideration the uses of backward and forward compatibility, and extends them with notions of *extensibility* and *graceful degradation*. Extensibility refers to an architectural design principle, in which an already existing implementation does not prevent future growth of the system. Extensibility enables extensions to

be implemented without impacting current implementations. The latter principle of graceful degradation suggests that a service should react to unexpected circumstances in a safe, proportionate manner. This principle applies to both the behaviour of services and the messages exchanged by services. Another contribution of the report is the differentiation between *message versioning* (focuses on the data that comprises the messages created and consumed by a service) and *contract versioning* (focuses on WSDL descriptions). The report also provides a list of Microsoft's recommendations regarding service versioning, and presents an overview of the future prospects of these versioning techniques and new developments being made, including WSDL 2.0 and WS-Metadata Exchange.

Erl et al. (Erl et al., 2009) aimed to summarise all existing theoretical and practical information to provide a thorough guide to designing SOA contracts and versioning. Among a wide range of topics covering virtually all aspects of SOA contract development, the authors also focus on versioning and discuss the concepts of backward and forward compatibility, version identification strategies, service termination, policy versioning, validation by projection, concurrency control, partial understanding, versioning with and without wildcards, etc. Targeted at SOA practitioners, the book also provides a number of hands-on examples, addressed by problem-solving techniques, to demonstrate the discussed concepts.

Fang et al. (Fang et al., 2007) proposed a version-aware service description model (to describe Web service versions, handle different versions, and select and call specific versions of a service) and a version-aware service directory model (to manage service versions in the service directory). This functionality is implemented by extending the core schema of a WSDL document to include version-related fields and to describe the attributes of the service versions. Next, these enhanced annotations are parsed by an augmented version of the UDDI registry, which is able to incorporate

versions in a service directory with an event-based consumer notification/subscription mechanism. The authors demonstrated a proof-of-concept implementation of the proposed system, a weather forecast Web service prototype, which is able to provide a mechanism for service consumers to be dynamically notified of new service versions.

Becker et al. (Becker et al., 2011) proposed a versioning framework where differences between multiple versions of a service can be automatically identified. This approach emphasises granular or modular implementation of versioning, while keeping loose dependencies between a service and its elements. More specifically, the authors propose an approach to automatically determine when two service descriptions are backward compatible. Based on a case study, the paper discusses version compatibility information in a SOA environment and presents initial performance overheads. A service provider can assess the impact of proposed changes by automatically evaluating service compatibility. The authors also suggest that relevant versioning requirements should be placed within the service consumer's client code at design-time to avoid incompatibilities happening during run-time. Finally, the authors explore compatibility of SOA messages being exchanged between service providers and consumers to ensure that only compatible messages are communicated. The main shortcoming of this approach is that it is based on the concept of Object-Oriented Paradigm, not currently supported by W3C.

This latter shortcoming was rectified by Yamashita et al. (Yamashita et al., 2012), who followed the W3C standards. The authors introduced a novel feature-based versioning approach for assessing service compatibility and proposed a different versioning strategy, where versioning is only applied to a certain fragment of a WSDL/XML schema. First, the objective of this approach is to implement versioning only in those parts, such as operations and data types (referred to as *features*), that have undergone direct or indirect changes. Second, this approach addresses issues

related to finer-grained versioning. Last, this approach allows changes to be tracked. As a result, the proposed versioning model and compatibility assessment algorithm for supporting evolution is characterised by its finer granularity, as opposed to existing approaches using typical WSDL/XML schema service descriptions. The described approach enables the identification of changed (or affected) features in a new service version, and ensures that these modified features do not affect the stability of dependent client applications. The approach was validated against a real service and demonstrated its viability by detecting changes made to service descriptions while ensuring service compatibility. The authors also provided some practical tips on supporting service evolution through maximising version reusability (e.g., client request redirection and load balancing among different existing versions of the same service).

Leitner et al. (Leitner et al., 2008) proposed a versioning approach with a specific focus on addressing potential compatibility issues. As suggested by the authors, a collection of service revisions is represented as a graph for easier access and retrieval. The latter is supported by version selection strategies that are used at service run-time, which enables dynamic and transparent invocation of different versions of a service as service proxies via a UDDI registry. This way, service consumers are able to select the required version of a service among a collection of existing revisions. However, the general applicability of this solution is somewhat limited due to the (in)compatibility issues that may appear at run-time. In these circumstances, it is up to the service provider to first detect changes that have occurred in different service versions and then handle them accordingly. It is worth noting, however, that run-time compatibility evaluation, particularly for complex service descriptions, is a challenging task. It is often performed manually, and is therefore expensive and prone to errors (Becker et al., 2011), which makes the overall approach even more vulnerable.

Raemaekers et al. (Raemaekers et al., 2017) in their research look into the existing versioning practices, by studying the collection of over 100,000 jar files in 22,000 different libraries stored in the central Maven repository that have been maintained there for over seven years. Aiming to check whether versioning conventions are properly followed by the developers, the authors also tried to understand to what extent the semantic versioning schema provides library users with meaningful information about breaking changes in a specific release. In particular, semantic versioning rules were subject to investigation in the context of the following three revision types:

- Major revisions where breaking changes are allowed;
- Minor revisions where breaking changes are not allowed;
- Patch releases, where only backward-compatible bug fixes are allowed.

Based on the hypothesis that semantic versioning was expected to provide developers with a precise set of rules defining the usage of major, minor and patch version numbers, the authors investigated the repository to prove whether these rules were actually applied. It turned out that around one third of all releases, including minor and patch revisions, introduced one or even more breaking changes (i.e., changes with a significant impact on depending client libraries that can potentially lead to compilation errors), which were only supposed to be part of major revisions. As a result, the authors argue that developers typically do not bother much about communicating backward incompatibilities or deprecated methods in their releases in a systematic and structured manner. Eventually, this information is delivered to users through multiple breaking changes in major releases, and is also available in an unstructured (i.e., only human-readable) manner through labelling and comments. A large number of compilation errors in dependent software systems is an inevitable result of this lack of the semantic versioning practices, which need to be widely adopted by the developer community.

Bhattacharya et al. (Bhattacharya et al., 2017) proposed a model to automate service discovery and dynamic choreography from a service version database. The authors argue that while it becomes imperative to maintain various versions of services at the same time, in some circumstances it may not always be cost-efficient to invoke the latest revision of a service. In this light, it becomes important to thoroughly understand and formally represent requirements as to which service versions to use. To this end, the authors devise a comprehensive framework that models requirements in a formal manner and automatically extracts verbs to generate an activity model, which is then translated into Business Process Model and Notation (BPMN), based on a set of transformation rules. Next, the BPMN nodes are mapped to services, and an algorithm for dynamic discovery of an appropriate service version is conceived. The proposed framework automatically translates input requirements to business models. As a result, it helps in discovering correct service version matches based on the business functions extracted from the requirements and matching with service version details, previously stored in UDDI.

Weinreich et al. (Weinreich et al., 2007) were among the first authors to contribute to the topic of service evolution by presenting an approach for a specific kind of SOAs, where services are implemented as Enterprise Java Beans (EJBs), accessible either by native J2EE clients or by Web service clients. Using a novel versioning model, the proposed approach defines units of versioning, a versioning schema and a schema for service addresses. The authors build upon and extend the concepts of component-based software systems and existing Web service versioning approaches. The proposed approach benefits from a universal, “homogeneous” versioning method applicable to both Web service clients and remote J2EE clients. The proposed versioning model is based on a 3-digit versioning schema, able to distinguish between compatible and incompatible changes. As opposed to the majority of existing approaches, the units of versioning refer to subsystems

containing multiple services, rather than individual services. This enables fine-grained versioning and supports parallel development of semantically related (light-weight) services with associated shared resources. Another key feature of the proposed approach is its support for asynchronous client updates, i.e., deciding whether to keep on using an existing and properly functioning service, or whether (and when) to upgrade to a newer version is solely undertaken by the service consumer. Furthermore, the proposed approach supports controlled updates through SOA governance.

3.2.3. Related Works on Service Compatibility

As already discussed above, service versioning is important for the development and enhancement of services. Nevertheless, the existing works on service versioning do not seem to be sufficient to enable rich support for the evolution of services (as it is proposed by SORC). Since service changes may potentially lead to compatibility issues, developers are sometimes restricted from creating and publishing new revisions of a service with significant changes and extensive features. Therefore, it becomes particularly important to create a mechanism to *i)* evaluate service compatibility, and *ii)* ensure that newly introduced service versions are compatible with previous versions and easily adaptable to changes in the SOA environment. To this end, an efficient mechanism that can automatically evaluate, assess and identify changes and associated compatibility issues is required (Andrikopoulos et al., 2012; Becker et al., 2011; Liang and Huhns, 2008; Yamashita et al., 2012). Kaminski et al. (Kaminski et al., 2006) proposed an algorithm for assessing service compatibility via a versioning framework that allows service descriptions to change at different levels. This algorithm automatically checks backward compatibility between two neighbour service versions using an object-oriented service description, and generates a description of the service data types represented as a set of elements. The approach also supports flexible input parameter formats to be placed in request messages. This way, the algorithm contributes with a novel mechanism to

assist the service provider in estimating and understanding the future impact of changes in the evolutionary development process. Furthermore, the approach presents a SOA-based framework, which implements the proposed algorithm, thereby demonstrating benefits for both the consumers and providers willing to enable and benefit from automated compatibility assessment.

Yamashita et al. (Yamashita et al., 2012) proposed another algorithm that recursively evaluates two adjacent service versions, checking features of compatibility. The algorithm is an adaptation of Kaminski's algorithm (Kaminski et al., 2006) and contains details of the compatibility checking mechanism based on the current best practices and guidelines (Andrikopoulos et al., 2012; Leitner et al., 2008). It is capable of identifying and highlighting the changes in the newly developed service version along with their possible negative impact on the service provision. The algorithm was implemented in real services and results were obtained regarding its effectiveness. Combining the two algorithms (Becker et al., 2011; Yamashita et al., 2012) can provide a precise, comprehensive, and fine-grained compatibility assessment in the context of usage profiles.

Andrikopoulos et al. (Andrikopoulos et al., 2012) contributed a service evolution framework based on Abstract Service Descriptions. This framework is capable of generating details about structural, behavioural and QoS-related components of a service. It addresses the compatibility issues in service versions using a T-shaped compatibility model. It further classifies and extends the T-shaped model into two-dimensional scopes: horizontal compatibility and vertical compatibility. Horizontal compatibility (also known as interoperability) focuses on service versions that can interoperate between service providers and consumers. Vertical compatibility (also known as substitutability when assessing from the provider's perspective or replaceability when assessing from consumer's perspective) emphasises that one service version can be replaced by another.

Through the T-shaped model, they also proposed an algorithm to check and assess the compatibility of service versions.

Liang et al. (Liang and Huhns, 2008) proposed a dynamic evaluation framework through a system model that checks compatibility of Web services as well as compatibility of the key operators. With the proposed model, various experiments were performed to measure the performance and efficiency of compatibility checking against various service factors that can be either compatible or incompatible. For the purpose of evaluating compatibility, semi-structured documents such as WSDL and OWL-S are used. This technique is effective in assessing compatible factors and in identifying incompatible factors involved in service APIs and implementations. However, it is also limited as it focuses only on certain types of changes in services. For example, it does not address policy changes in the compatibility assessment method.

Lublinsky et al. (Lublinsky, 2007) presented a compatibility assessment mechanism that emphasises the interoperability among independently evolving Web services. Service interoperability refers to the ability of one service easily replacing another from the consumer's point of view. In order to identify compatibility between applications and non-native services, they used both dynamic and static analysis algorithms and built tools to apply these algorithms. They also introduced "cross stubs" to check incompatibility and facilitate interoperation across multiple versions of the same service.

3.3. Identifying the Gap

This section aims to summarise and critically evaluate the survey presented above to identifying research and technological gaps in the domains of service change detection, service versioning and service compatibility. This will allow us to outline the application scope for our own research effort.

While studying and classifying existing approaches, several observations were made. We now discuss each of these observations in more detail:

1. **Little support for service compatibility**: with the increased trend of expanding Web applications, there is significant demand for distributed collaborative Web programming environments. SCM introduces the concept of controlling and managing software and is capable of resolving many change management issues in the developed systems. SCM controls critical activities and manages changes involved in a project development lifecycle. However, the traditional SCM systems are not suitable for managing concurrent changes in Service-Oriented Web applications due to the distributed nature of SOA, i.e., service engineering within a loosely-coupled decentralised environment where project artefacts are spread across multiple network locations. In these circumstances, one of the major challenges to be addressed is service versioning, as constant development and evolution of services leads to new versions over time, resulting in multiple versions of the same service. Service versioning gives rise to the issue of compatibility, which becomes a crucial issue in the context of collaborative SOA programming, where each developer's code is exposed to their peers only via APIs. This issue, however, seems to be under-explored in the literature, and, as it will be further explained in the next chapters of this thesis, SORC was specifically designed to bridge these gaps by supporting concurrent development of Service-Oriented Web applications.
2. **Lack of service change notifications**: the second gap is the lack of support for automated notifications pushed to service consumers whenever a service change occurs. Currently, service consumers typically do not know about newly released versions and what changes are made to these versions. As described, service consumers might have to manually check

whether there is a new version of a service and determine whether their own software systems are compatible with the new version. By enabling automated notifications containing semantic descriptions of applied changes, service consumers would be given an opportunity to automatically adjust their systems accordingly, based on business rules currently in place (e.g., by parsing a notification it would be possible to take a decision whether to automatically switch to the latest version or keep using the existing one). As demonstrated by the literature survey, existing approaches seem to offer little support for service change notifications. Even if they do support this feature, notifications typically do not contain much meaningful information (which would allow automated reasoning about whether compatibility is retained), and service consumers have to perform manual checks of the new service versions.

3. **Existing approaches are isolated from each other**: the third observation refers to the relative isolation of the surveyed works (albeit, this is a common issue in scientific research). Individual approaches seem to have a narrow focus on specific aspects of the general problem of service evolution and compatibility. Without providing an overarching cross-cutting solution, to a great extent the majority of surveyed works are represented by vertical, `siloed' approaches. Despite providing extensive discussion on how their work is different from the rest, the authors fail to suggest any opportunity for 'cross-fertilisation', re-use, complementation, and collaboration. As a result, state-of-the-art solutions seem to be developed in isolation from each other, thereby leading to numerous competing, potentially overlapping and redundant technologies. Addressing local scientific issues might be acceptable from a research perspective; from a more practical viewpoint, however, these approaches hardly contribute to creating a truly effective holistic solution to be

adopted by a wider community. Furthermore, studied approaches also suffer from lack of extensibility. Succeeding in solving one aspect (i.e., change detection, service evolution, or service compatibility), they are simply not designed with extensibility in mind, to potentially cover new emerging features in the future. Taken together, these considerations suggest that currently there is a need for an overarching solution, which would integrate all three aspects and enable synergies between multiple approaches, and systematically address change identification, service versioning, and compatibility assessment and resolution.

Taken together, these observations outline a list of functional properties and desired features for an envisioned solution that will bridge the identified gaps and contribute to the state of the art in service compatibility and service-oriented revision control. More specifically, the future solution is expected to combine three key features: service change detection, service versioning, and (most importantly) service compatibility. Also, the solution must clearly communicate service changes by means of an automated notification mechanism. The future solution must incorporate multiple perspectives and provide a holistic view of the problem. That is, it has to be built on a set of solutions from both the provider's and the consumer's perspectives, allowing the service developer (provider) to automatically generate a “Commit” message when he/she commits a new version, which is to be pulled by a peer developer (consumer) when updating his/her local proxies to the service without relying on heavyweight service registries. In this context, a “Commit” message represents an automated notification mechanism. Moreover, the main function of any kind of version control is to track changes to source code (history of changes), there has to be support for a sufficiently long history of tracked changes. As service-oriented revision control primarily focuses on service APIs, it has to support historical records of APIs that go through (compatible

and incompatible) changes. For representation and interaction purposes, this history of changes and revisions is expected to be visually implemented as a versioning tree, through which system users can navigate for a required service version. Finally, the expected solution is supposed to be highly automated, so that users would be enabled to perform a wide range of operations in a seamless and transparent manner. For example, for incompatible versions, the consumer should be asked for confirmation of whether the upgrade to the incompatible version should proceed or not.

3.4. Implementing CI for SOA development using SCM systems

Given these limitations and requirements, a potential solution in these circumstances was to attempt to "replicate" the required functionality using existing tools. To a certain extent, it was required to achieve the required functional characteristics (i.e., to be able to exercise CI when developing service-oriented software in a distributed manner) via a combination of several existing tools/technologies. To this purpose, a common practice to enable CI for SOA development adopted by enterprises is to combine the following two technologies:

- **An SCM system for code storage and version control:** The common way of handling a wide range of CI-related issues is to employ a source code hosting and existing revision control system. GitHub (Dabbish et al., 2012) is nowadays seen as the de facto standard for addressing this kind of challenges. It is a social coding site that uses Git (Loeliger and McCullough, 2012) as its distributed revision control and source code management system. It implements a social network where developers are able to broadcast their activities to others who are interested and have subscribed to them. According to the recent report, GitHub currently hosts 67 million repositories maintained by over 24 million registered

users.[14] Developers can participate in multiple projects, and each project may have more than one developer. The GitHub social coding site is advertised as a developer-friendly environment that integrates many functions, including a Wikipedia-like knowledge repository, issue tracking, and code review. As was explained in previous chapters, the Git repositories managed by GitHub support the two basic commands: **push** (i.e., for uploading code modifications to the repository) and **pull** (i.e., for updating local repositories to the newest commit).

- **An automated software build system for CI:** for distributed software development teams, it is important to enable CI of a shared project. **Jenkins** (Jenkins, 2017) is recognised as one of the most prominent tools that supports CI. It is an open-source CI tool written in Java. It is used by teams of all sizes, for projects written in a variety of languages and technologies, including .NET, Java, Ruby, Groovy, Grails, PHP and many more. Jenkins is a self-contained automation server, which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software. Furthermore, to support various benchmarking and time measurement activities, Jenkins can be extended with a **Build Time Blame Plugin**. This plugin scans the console output for successful builds. If a line matches one of the configured RegEx keys, it will store the timestamp generated for that line by the Timestamper Plugin. Then, it generates a report showing how those steps contributed to the total build time. This is intended to help analyse which steps in the build process take too long and, therefore, are good candidates for optimisation.

[14] https://octoverse.github.com/

As a result, using these two tools, developers are able to implement CI in service-oriented contexts. Admittedly, the two tools are highly optimised, efficient, and reliable, which contributes to the overall QoS of the system. On the other hand, however, this approach suffers from some other shortcomings.

SVN (Pilato et al., 2008) and Perforce HelixCore (Wingerd, 2005) adopt a centralised architecture that uses a central repository to maintain the version history of developmental work. This architecture, however, is not suitable for managing collaborative development of Service-Oriented Web applications, as it requires developers to copy all project files. In a decentralised architecture, such as adopted by Git (Loeliger and McCullough, 2012) and Pastwatch (Yip et al., 2006), a project has several repositories. Each developer has his own repository, which is similar to a centralised repository. Developers can update, checkout, commit, and transfer code across repositories. Although a decentralised architecture is more appropriate than a centralised one for managing collaborative development of Service-Oriented Web applications, all project files still need to be replicated between developers.

Krafzig et al. (Krafzig et al., 2005) explain these existing limitations:

- Services are not supposed to belong to a single project and be owned/managed by a single development team or organisation.
- For this reason, the service infrastructure is distributed across all participants of the global SBS system.
- Being loosely-coupled, individual services constituting an SBS system are expected to be controlled and managed (i.e., developed, tested, deployed, etc.) independently from each other.

As a result, with the increasing trend of developing service-oriented applications, there is a significant demand for a tailored SCM and CI system that addresses these limitations and enables collaborative programming of SBS systems, where loosely-coupled components interact according to the standards of interface description languages and communication protocols. The idea behind loose coupling is that there exists a well-defined service API that allows for the integration between the consumers and the provider of the service. Therefore, the SORC model, as it will be explained in the next chapter, was specifically proposed to overcome the shortfalls in both centralised and decentralised version control architectures (Sarib and Shen, 2014).

3.5. Summary

The goal of this chapter was to familiarise the reader with the state of the art in the research areas of software change, versioning and compatibility in the context of SOA systems. The chapter first provided a brief theoretical underpinning for the key concepts, and also discussed why service versioning is important to maintain compatibility. Next, a literature survey of relevant research papers was presented, discussing key contributions and concepts. Several research and technological gaps were identified as a result, providing the scope and high-level requirements for an envisioned solution. As a potential solution in these circumstances, the chapter also explained how developers are forced to implement CI in SOAs using a source code management system (e.g., GitHub) and a software automation tool (e.g., Jenkins). This combination of technologies, however, also cannot handle the distributed nature of SBS systems, and therefore has to be improved. In summary, the envisioned solution should combine service change detection, versioning and compatibility techniques, and provide an automated way of pushing change notifications to service consumers – taken together, these features pave the way for automated CI in SOAs. Additionally, the solution is expected to take into account all interested parties, including service providers and

consumers, so as to create a truly holistic view of the problem. Based on these features, the next chapters explain and discuss the proposed SORC approach in more detail.

4. Chapter 4. Service-Oriented Revision Control in Detail

This section provides the details of the architecture of SORC, which is underpinned by three main commands. Accordingly, the chapter first introduces the general high-level overview of the architecture, and then goes into the details of the three basic commands, following a top-down approach. The Commit command is used by service providers to register new versions of a service so that they can be discovered by service consumers. The Checkout command is used by service consumers to discover peer developer nodes and published services. Finally, the Update command is used by service consumers to migrate from a currently used service version to a new version of the same service. The update process takes into account whether or not the new version is compatible with the previous version. By bringing together the descriptions of the three commands, the chapter aims to provide a holistic view on the proposed functionality and explain how it can address the CI requirements in a distributed SOA scenario.

4.1. Service-Oriented Revision Control: an Overview

SORC is presented in this section in two parts. The architecture of SORC comes first, whereas individual components of SORC are described in more details in the following section.

4.1.1. SORC architecture

As it was highlighted, SOC introduced a whole new level of complexity to be handled by SCM and CI solutions, going beyond the existing capabilities of traditional systems. In this light, to provide support for the collaborative development of Web applications, SORC is proposed, implementing a distributed revision control model (Sarib and Shen, 2014). The distributed nature of this model allows developers to work independently on a web application project without replicating the project source files among themselves. A team of developers can work on a shared service by exposing a developer's code via service APIs. In this way, revision control of the project

is performed at the service level instead of being managed by a contemporary VCS. This eliminates the need for the team to establish a common centralised source code sharing system.

SORC follows a distributed version control model, which was designed to make collaborative programming of SBS systems more manageable and efficient. It does not replicate project files across developers; instead developers' pieces of code are exposed as service APIs to their peers regardless of their geographical proximity. Generally speaking, SORC acts more like a service broker than a file/repository manager. As illustrated in Figure 8, developers have their own workspaces, as well as their own set of Web services and versions. Whenever information is to be shared across participating developer nodes, a descriptor message is sent to other nodes. By doing so, developers notify each other about the services offered by each developer's node, as well as how the services can be consumed by creating local proxy classes required to remotely invoke these services.

Like other VCSs, the SORC system is concerned with managing changes during the service life-cycle, allowing developers to version their services and, at the same time, maintain multiple releases of services. This, in turn, gives other developers in the team the ability to specify the version of the service they want to use. This system is focussed on controlling and managing services through standard APIs. The basic task of SORC is to share services and to preserve a history of versions of a single service (or multiple services). SORC allows a developer to easily compare two historical versions of a service, consume an older version of a service, or review the list of changes made to a service.

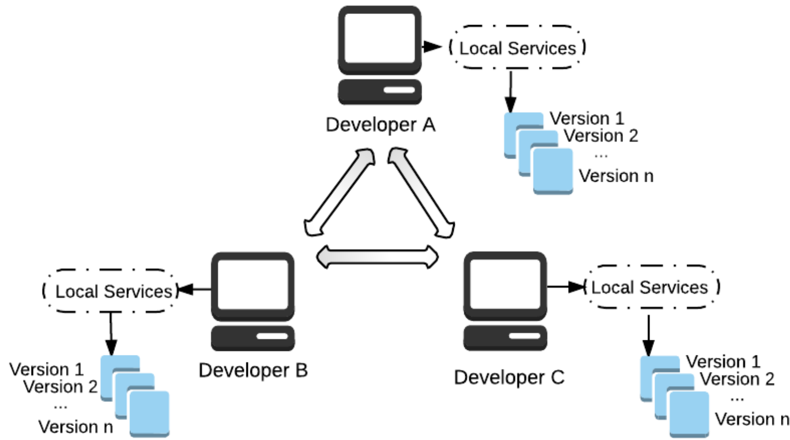

Figure 8. Service-Oriented Revision Control System.

SORC makes use of peer-to-peer connections between developers, as shown in Figure 8. Each developer node functions as a provider and a consumer of services, as described in Table 2.

Table 2. Developer activities using SORC.

Type of Developer	Command	Developer's Activities
Service Provider	Commit	Share services
		Implement service changes
		Perform compatibility assessment
		Identify service changes
		Document the service changes
		Commit new versions
Service Consumer	Checkout & Update	Locate available nodes (developers)
		Discover services and versions
		Consume services
		Get documentation of service changes
		Update service versions

		Adapt version changes
		Display communication graph

4.1.2. SORC Components

The SORC architecture contains the three main elements of a SOA (Papazoglou and Dubray, 2004) – i.e., service consumer, service provider and service directory/registry. As mentioned previously, developer nodes act as service providers and service consumers. There are three basic operations that take place between developer nodes – namely, publication of service descriptions, finding service descriptions, and binding (invocation) of the service through a service API (Yu et al., 2008). The three SORC commands – i.e., Commit, Update and Checkout – assist in implementing these operations.

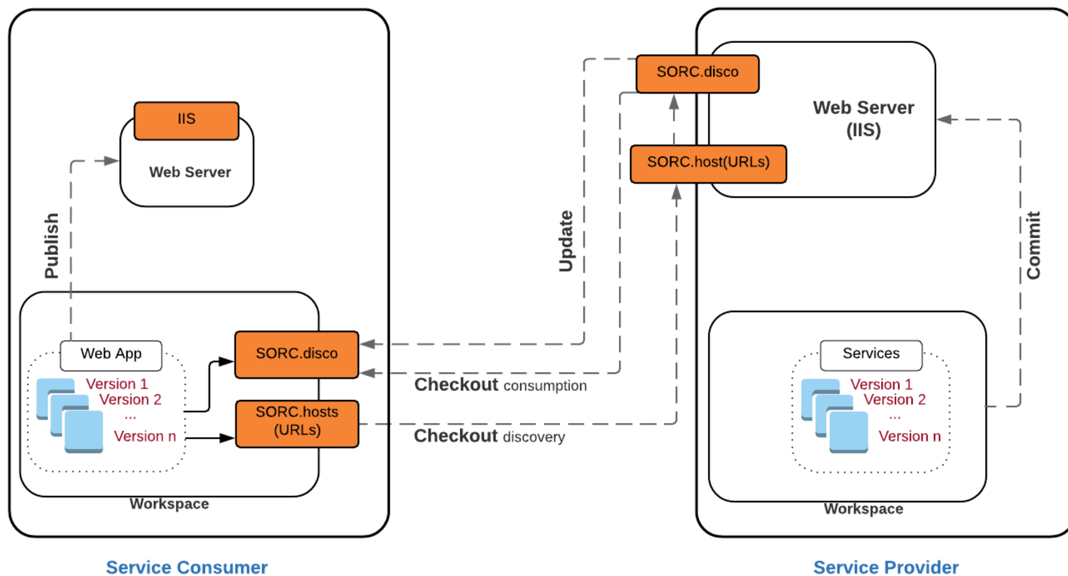

Figure 9. SORC reference architecture.

Developers follow the steps shown in Figure 9 to publish and access services. The Commit command is used to create a service from scratch or release a new version of the service. The Checkout command is used to find nodes that are part of the development team (i.e., same

distributed software project). Next, having discovered peer nodes, it is possible to discover services (and their specific versions) deployed on those nodes and provisioned to peer developers. The Checkout command serves to connect to the remote services with the help of locally generated proxy classes. These classes are associated with service versions and are used to make remote services appear as locally stored components.

The basic elements of the SORC model can be summarised as follows:

- **Service provider** manages the processes for designing, developing, deploying, versioning and publishing services for peer developers.
- **Service consumer** discovers services, understands versioning and service updates and uses these to collaborate with other developers.
- **Discovery file** is a searchable file that stores WSDL documents for the available services and their versions. These WSDL documents are references to individual version files that are available on the developer node. This enables service consumers to look up and consume services, without any other party's involvement.
- **Host file** keeps track of node addresses to allow the service provider and consumers to find each other.
- **Workspace** is the place where source files are edited and services are built, tested and debugged. To support local development, SORC is used to checkout service proxies from other developers as required.
- **Base service** is a mark-up file pointing to a code-behind file that implements a service and provides an API for service consumers. The first invocation of the Commit command to put a service under revision control generates the base service and creates the first version of that service. For example, a developer creates a service named "ServiceA". The Commit

command generates the Base Service "ServiceA" and creates the first version "ServiceA_1_0". Later, any changes made to ServiceA will create new versions, with the second version being "ServiceA_2_0".

4.1.3. Versioning Method

To a great extent, SORC follows an established naming convention, where the *zeroth* version (i.e., ServiceA_0) is automatically assigned to a newly committed service revision, whereas the previous revision is pushed down the versioning sequence and is assigned with an incremented value (i.e., ServiceA_1), as shown in Figure 10. This feature provides service consumers with the ability to automatically be updated every time a new change is introduced by the service provider. As illustrated by the diagram, SORC implements the snapshot-based model, where revisions are stored as individual snapshots. Consumers always reference the latest version of a service by default, unless they explicitly request for an older version (i.e., any service version other than 0). As a result, existing consumers are not explicitly required to be notified of newer versions, even if the provider implements and publishes an updated version, i.e., they will still be referencing the most recent revision.

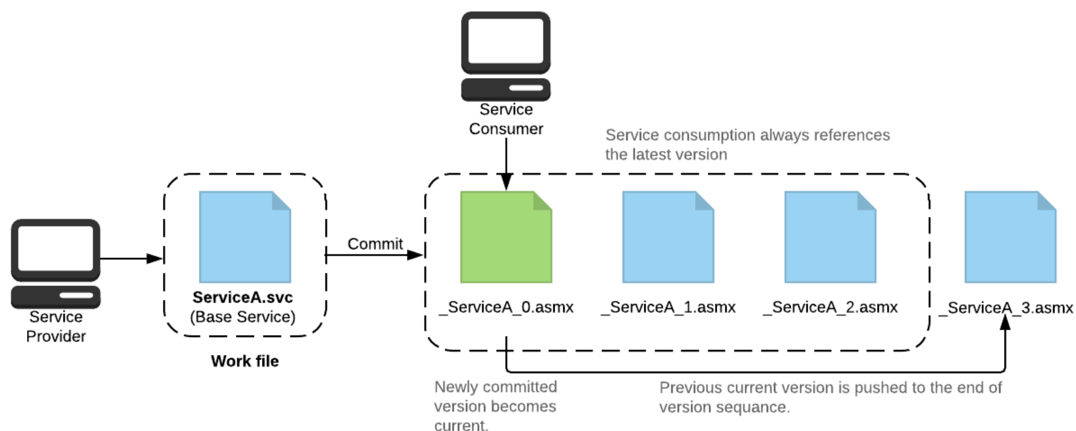

Figure 10. Old Versioning Method in SORC (Sarib and Shen, 2014).

4.1.4. Compatibility Problem Addressed by SORC

In SOC, versioning gives developers the ability to view the history of service versions and compare two variants to recognise the changes between them. Becker et al. (Becker et al., 2011) define service versions from two perspectives: from the consumer's and provider's points of view. From the consumer's perspective, a service version is a "contract" established by the service provider defining service functionality via an API. From a service provider's point of view, however, a version refers to a service implementation and how that implementation changes over time. The versioning approach employed by SORC (Sarib and Shen, 2014) permits consumers to choose and mix different versions of a service. It uses WSDL to store service versions and allows consumers to discover various versions of a service offered by a provider. SORC proposes a change identification method that provides automatic updates whenever a new version is implemented (Sarib and Shen, 2014). Since the proxy class for a consumer always references the latest released version of the service, the existing consumer reference to the service does not need to be changed if it is compatible with the latest service, and the consumer can continue consuming the service without knowing it has changed (Narayan and Singh, 2007). However, if it is incompatible, the service consumption would stop and the consumer may need to explicitly specify an older version of the service.

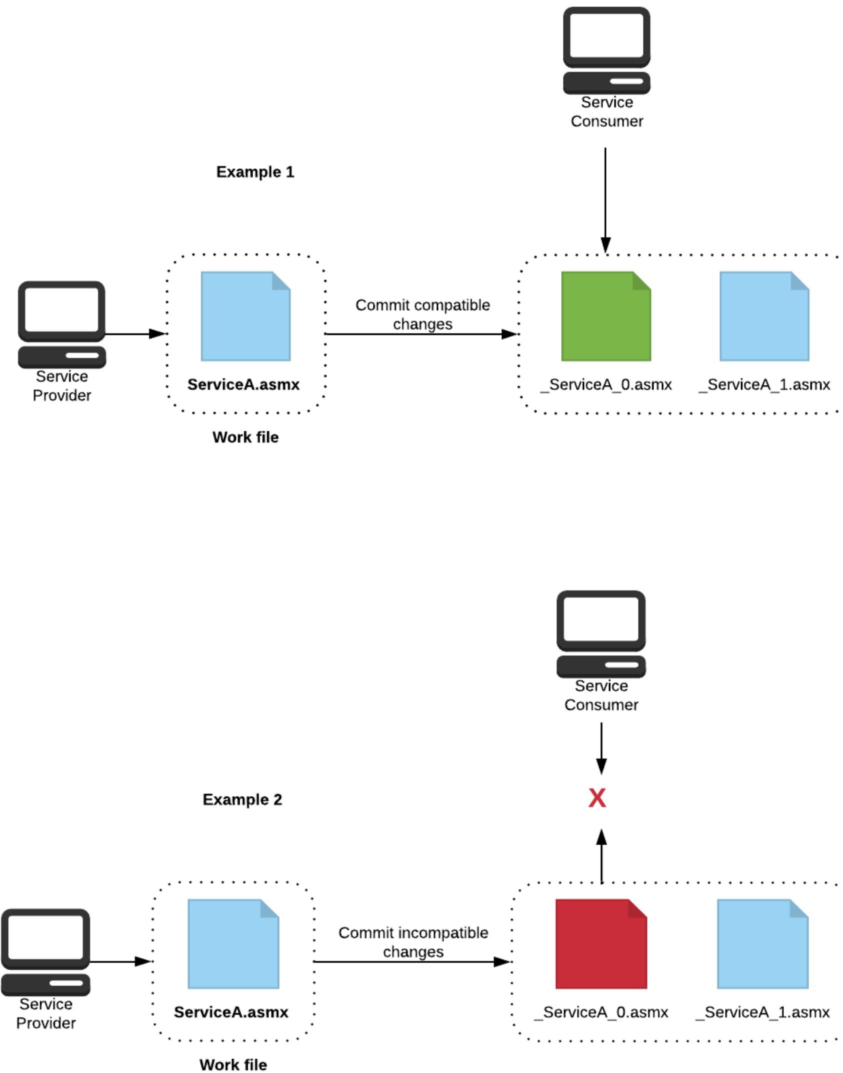

Figure 11. Compatibility problem in SORC.

Considering the inevitability of service changes and the need to constantly develop new versions, it is important to be aware of issues that may occur during service versioning. In particular, it is non-trivial to know whether a new version of the service is compatible with what the consumers expect until a disruption in service is observed at run-time. There has to be a method that validates compatibility before making a new version of a service and providing it to the consumers automatically. Service developers can evaluate the effects of proposed changes by automatically

exploring compatibility information (Becker et al., 2011). If the changes are compatible, as illustrated by Example 1 in Figure 11, Service Consumer B can continue using the newly committed version (ServiceA.0) without any disruption of the service. Otherwise, if the changes are incompatible, as illustrated by Example 2 in Figure 11, the service provisioning and consumption will be disrupted, as Service Consumer C automatically calls the incompatible new version.

4.2. Threefold Functionality of SORC

We now discuss the functionality of the proposed SORC in more detail. The functionality is based on the three primary operations: Commit, Checkout, and Update.

4.2.1. Commit Operation

Traditionally, the commit function in common VCSs is a straight-forward process that submits changes to the developer's repository (Swicegood, 2008). This is different in SORC, which deals with services rather than files. The Commit command in SORC is a process that records service changes in the developer's workspace and publishes the service to a server to be made available to peer developers. Fundamentally, the SORC Commit records changes and releases a new version with a unique, generated version number. The following pseudocode algorithm demonstrates the SORC Commit process, where `Fpath` is the file path of the service, `S` is the service name, `Vnew` is the new version of `S`, and `discoFile` is the discovery file.

Algorithm 1. Commit command.

```
      Require: Fpath, S

      1:   Search for discoFile
      2:   if discoFile is not found
      3:     create new disco file
      4:   end if
      4:   restOrSOAP(Fpath.FullName, S)  // algorithm 2
      5:   compatInfo = versioning(Fpath.FullName, S) // algorithm 3
      6:   discoFile.append(Vnew, compatInfo)// Append new version to
discoFile
```

```
     8:   publishToIIS(projectFileName, publishProfile, visualstudioVersion)
9:   return Vnew // if there were no errors encountered, return the errors
     otherwise
```

The Commit algorithm starts by checking whether a discovery file exists, creating it if it is not found. The discovery file stores details about services and their versions (Lines 1-3). The discovery file of a service provider contains references to service descriptions (i.e., to a WSDL document). This file is managed by service developers to allow them to register versions of their services. **`RestOrSOAP`** (Line 4) checks whether the service is SOAP- or REST-based. The service is then given a version label via the **`Versioning`** function called at Line 5. The **`Versioning`** function, described below, calls an algorithm named **`compatAssess`** to assess whether the service being committed is compatible with its previous version. This is used to generate a new version label. The commit algorithm appends the new version and the compatibility information to the discovery file. Errors during the commit process will result in the commit being cancelled. If there were no errors, then the commit will result in the new version of the service being published on a Web server for public access.

REST or SOAP

Services can be implemented and invoked via different mechanisms, with WSDL/SOAP and REST being the most popular choices. It is important to identify whether a service uses REST or WSDL/SOAP since the versioning algorithm is different for each case. Determining which interface methodology a service uses is accomplished by looking in the service project's `web.config` file to check the type of binding defined. The `web.config` configuration file describes endpoints and binding types supported by services at deployment time, rather than design time. If a service has a `basicHttpBinding` or `wsHttpBinding` then it is considered to be

WSDL/SOAP-based. If the service has a `webHttpBinding`, then it is REST. This is represented by the pseudocode in Algorithm 2, where `Fpath` is the file path, and `S` is the service name.

Algorithm 2. RESTorSOAP.

```
Require: Fpath, S

1:   fetch web.config from Fpath
2:   look for S node in web.config
3:   if S exists
4:     look for descendant nodes of S node
5:     foreach descendant nodes
6:       check value
7:       if it is same as service name then
8:         check its endpoint binding
9:         if binding is basicHttpBinding or wsHttpBinding then
10:          service is WSDL/SOAP
11:        else if binding is webHttpBinding then
12:          service is REST
13:        end if
14:      end if
10:    return S
11:  end if
```

Versioning

Service versioning is not yet covered by any Web service standard (FEDICT, 2014). Although versioning is a very important aspect of a SOA development life cycle, an approach to achieving successful service versioning is still lacking. The SORC system has developed an approach, based on best practices (Andrikopoulos et al., 2012; Brown and Ellis, 2004; Fowler and Foemmel, 2006; Novakouski et al., 2012) to service versioning.

The mechanism for versioning service APIs, as used in the Commit process, is based on three aspects: service versioning method, service versioning strategy, and change identification model (Andrikopoulos, 2010). The versioning method and strategy ensure old and new versions of a

service can exist together in a way that will not impact clients, while the change identification model defines how changes made to services can be identified by consumers (Andrikopoulos, 2010).

Versioning Strategy

Once services have been committed and used in production, some further improvements might be required. Versioning strategies provide a means by which service consumers can avoid disruption caused by service evolution. Versioning strategies are mainly concerned with whether services undergo compatible or incompatible changes (Erl et al., 2011). Different versioning strategies present their own sets of rules (Green, 2008), as described below:

- **Agile Versioning** is dependent on backward compatibility for as long as possible. It avoids creating a new contract or providing new endpoints until compatibility is broken. In the agile approach, service providers make improvements to existing services without versioning them or providing new endpoints. This approach is helpful in situations that require frequent updates to production code (Green, 2008).
- **Strict Versioning** is illustrated in Figure 12 and requires a new service contract and endpoint for any change to a service. The strict approach does not support backwards and forward compatibility (Erl et al., 2011). This approach is suitable in an environment that requires tracking of all changes (Green, 2008). SORC requires the ability to track changes and maintain the history of versions for the purpose of service version identification. We follow this strict approach with the Commit command resulting in the creation of a new contract and endpoint for all changes.

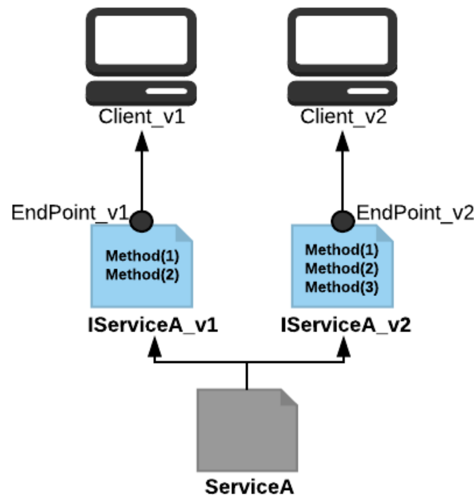

Figure 12. Strict versioning through a new service contract and unique endpoint.

- **Semi-Strict Versioning** lies between the strict and agile versioning strategies. It is useful when only certain types of changes to a service require a new formal contract and endpoint. For example, new operations may be added to a service contract at will, while any changes to existing operations require versioning.

  Versioning Method

Assessing the impact of changes

It is important to understand that services expose their functionality to be bound by service applications. The abstract description within a WSDL specification is the part that establishes the functional interface of a service (Erl et al., 2009). Every operation consists of an operation name, a list of parameters, and an output. The non-functional interface consists of service properties like endpoint addresses, WSDL encoding styles, and policies. This is the concrete section of a WSDL specification and is not included in the compatibility assessment.

A service provider, after making changes to a service API, needs to evaluate whether the new service version is compatible with the previous one. Compatibility in SORC adopts Yamashita's

compatibility checking algorithm (Yamashita et al., 2012), which also extends Becker's work (Becker et al., 2011). It compares two service versions and assesses them based on criteria shown in Table 3.[15]

Table 3. Type of changes in compatibility assessment.

No	Feature Changes	Description	Compatible
1	Adding an operation	Add new operation to service API	Yes
2	Adding a type	Add new type to a new operation or a new type	Yes
3	Adding a type	Add new type to existing operation or type	No
4	Updating a type	Change in description (e.g., cardinality, order or type)	No
5	Removing an operation	Remove operation dependency	No
6	Removing a type	Remove type dependency	No

Yamashita et al. (Yamashita et al., 2012) proposed a compatibility algorithm based on a feature model. The model provides abstract management of an API description and its different parts. It refers to certain parts of a WSDL document, such as operations and data types. They use this model to provide a compatibility assessment based on the guidelines of Table 3. The algorithm evaluates how these features have been affected directly or indirectly by the changes. A direct effect takes place when the feature itself changes. On the other hand, indirect influence happens when there is no change to the feature, but other features that depend on it have changed.

[15] Please note the difference between Table 3, which contains only changes to be checked by the compatibility algorithm, and the more generic Table 1, which contains all possible service changes that are not necessarily checkable by the software compatibility algorithm.

Algorithm 3. Compatibility Assessment.

```
Require: Vpre, Vnew
Ensure: compatInfo

1:   compatInfo:version new
2:   compatInfo:time currentTime
3:   compatInfo:compatible true
4:   compatInfo:changes null
5:   desChange ← evaluateDescriptions(Vpre; Vnew)
6:   if desChange != null then
7:     compatInfo:compatible false
8:     compatInfo:changes = desChange:changes
9:   else if Vpre.hasDependencies() then
10:    depRemoved ← RemovedDependencies(Vpre; Vnew)
11:     if depRemoved != null then
12:       compatInfo:compatible false
13:       compatInfo:changes = depRemoved:changes
14:     else
15:     for each Vpre's dependent version Vdep do
16:       depCompatInfo compatAssess(Vdep; Vpre)
17:       if depCompatInfo.compatible == false then
18:         compatInfo:compatible false
19:         compatInfo:changes += depCompatInfo:changes
20:       end if
21:     end for
22:   end if
23: end if
24: return compatInfo
```

The compatibility assessment algorithm is used to evaluate the compatibility of a new version of a service **`Vnew`** with the previous version of the same service **`Vpre`**. This assessment is conducted during the Commit process and the compatibility assessment can be retrieved by service consumers. Algorithm 3 is used to assess the compatibility of two versions of the same service keeping in mind the rules summarised in Table 3. Changes not mentioned in Table 3 are considered

incompatible. Evaluations of the dependencies and descriptions of the versions are the two main tasks of the compatibility assessment.

Evaluating version descriptions of `Vpre` and `Vnew` is done to see if the two descriptions are the same (lines 5-8). This evaluation returns a complete list of differences, if any, instead of just returning a compatible/incompatible Boolean result.

Evaluation of dependencies via the `removedDependencies` function is an analysis of whether the dependencies of the new version contains the dependencies of the old version (lines 9-13). These dependencies could be, for example, removing service operations or removing different types of an operation. If the evaluation shows a change, then the two versions are incompatible, as per cases 5 and 6 of Table 3.

Finally, the compatibility assessment algorithm compiles an assessment result for the new version in comparison with the previous version. This result contains the version number, the time when the version was committed, the compatibility result and the changes that were detected in the new version.

Version identifiers

A version identifier is a unique label that marks a particular release of a service. The SORC service versioning method assigns service releases a numerical identifier comprising two numbers separated by a period. Version numbers can convey meaning that is related to the changes made to services (Erl et al., 2009). A common approach in this regard is for the versioning method to distinguish between breaking and non-breaking changes (Bechara, 2015). The SORC system makes use of a #Major.#Minor version naming method. Each release of a service has numerical identifiers comprising two numbers affixed to the service name, each of which are separated by "_", for example, "ServiceA_2_1".

The first number indicates a "major" service revision and is increased when a service is changed in a way that is incompatible with the previous version. The second number, called a "minor" revision within a release, is increased when there are minor feature changes or compatible changes in the functionality.

Algorithm 4 below contains the versioning pseudocode, where `discoFile` is the discovery file, `S` is the service, `Vpre` is the previous version, and `Vnew` is the new version. The code first reads the discovery file to identify whether previous versions of the service exist. If not, a first version of the service, ServiceName_1_0, is committed (Lines 1-6). If there is a previous service version, that version is compared with the new version being committed, using the `compatAssess` algorithm (see Algorithm 3), to determine the compatibility between the two. If they are incompatible, the new version number will include an increase by 1 of the major version number from the previous service version (Lines 10-12). Otherwise, only the minor version will be incremented by one (Lines 13-15).

Algorithm 4. Versioning Algorithm.

```
Require: discoFile, S

1:   open discoFile
2:   if discoFile != Null
3:     check Service (S) versions committed
4:     if S is being committed for the first time
5:       Vnew = "S.Name_1_0" // first version of the service
6:       return Vnew
7:     else
8:       Get last version (Vpre)
9:       compResult = compatAssess(Vpre, Vnew)
10:      if compResult = false
11:        Vnew = "v" Vpre.Major++ "." 0
12:        return Vnew
13:      else
```

```
14:         Vnew = "v" Vpre.Major "." Vpre.Minor++
15:         return Vnew
16:       end if
20:     end if
21:   end if
```

Change Identification

It is important in SORC to keep consumers of a service up-to-date with any new versions, compatible or incompatible, committed by the service provider. To support this, SORC makes use of a Commit mechanism that relies on both provider and consumer. Figure 13 depicts the Commit message mechanism. When committing a new version, the service provider generates the Commit message by executing the compatibility assessment algorithm (Algorithm 3). The message contains version number, time, compatibility status and changes made. The message is stored alongside the new version of the service. The consumer updates to a new version of a service using the Update command to pull that version's Commit message from the service provider.

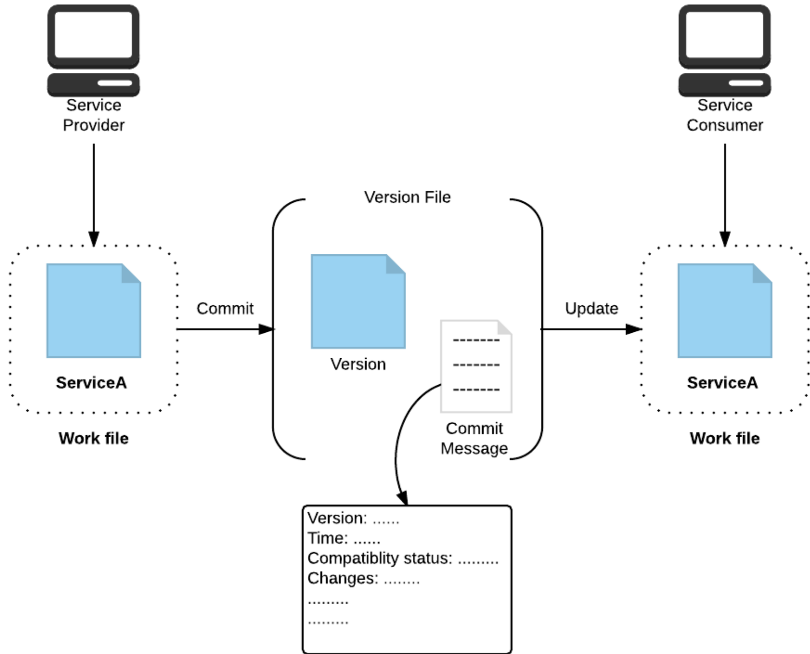

Figure 13. Commit message mechanism.

Committing and Publishing a New Version

The versioning method in SORC, as depicted in Figure 14, takes into consideration the incompatibility issue. A compatible change to a current service, e.g., ServiceA_1_0, is given a version number with the #Minor part incremented (ServiceA_1_1), and this new version is made the current version of the service. The previous version (ServiceA_1_0) is pushed to the second position in the versioning sequence, and so on. In the case of the service provider committing a new version that is incompatible, a new branch of service versions is created by taking the current version's number and incrementing #Major (ServiceA_2_0 from the previous example). Service consumers use the Update command to learn of these committed versions and to generate a local proxy for the current service version. By doing so, the service consumer will be up to date with new service releases without an explicit notification about the changes being made by the provider.

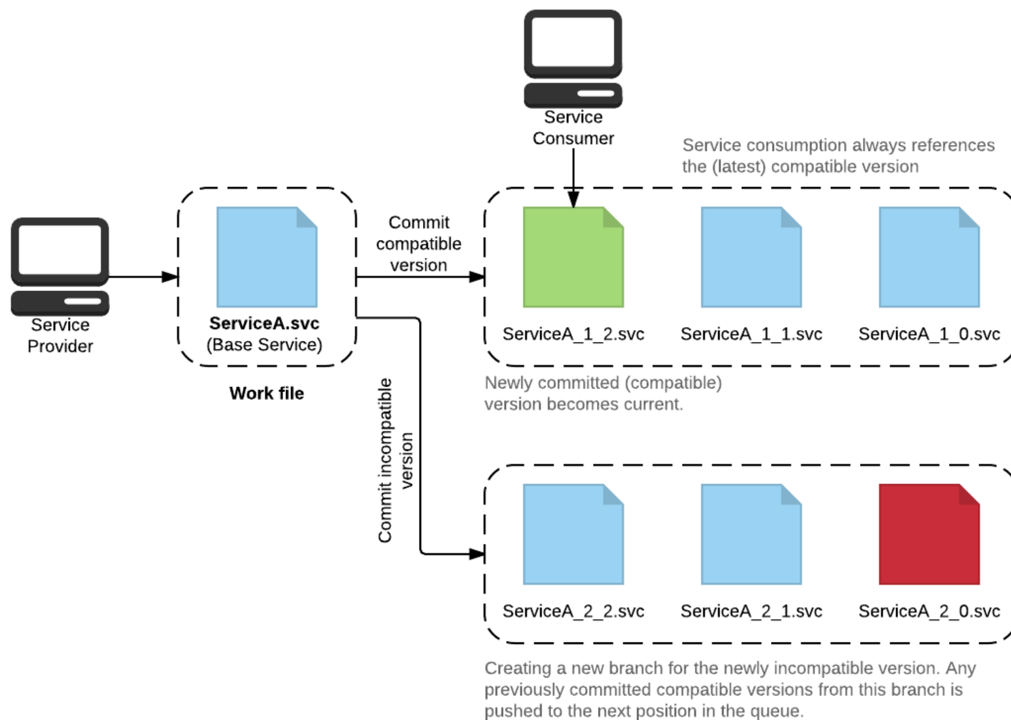

Figure 14. Proposed versioning method.

This versioning method allows service consumers to always reference the latest compatible version committed by the service provider. Service consumers use the Update command to generate a proxy class for the latest compatible version committed by the provider. In the case of incompatible versions, the consumer has the decision of whether they want to adopt a new incompatible version. Figure 15 illustrates the process of a Commit operation.

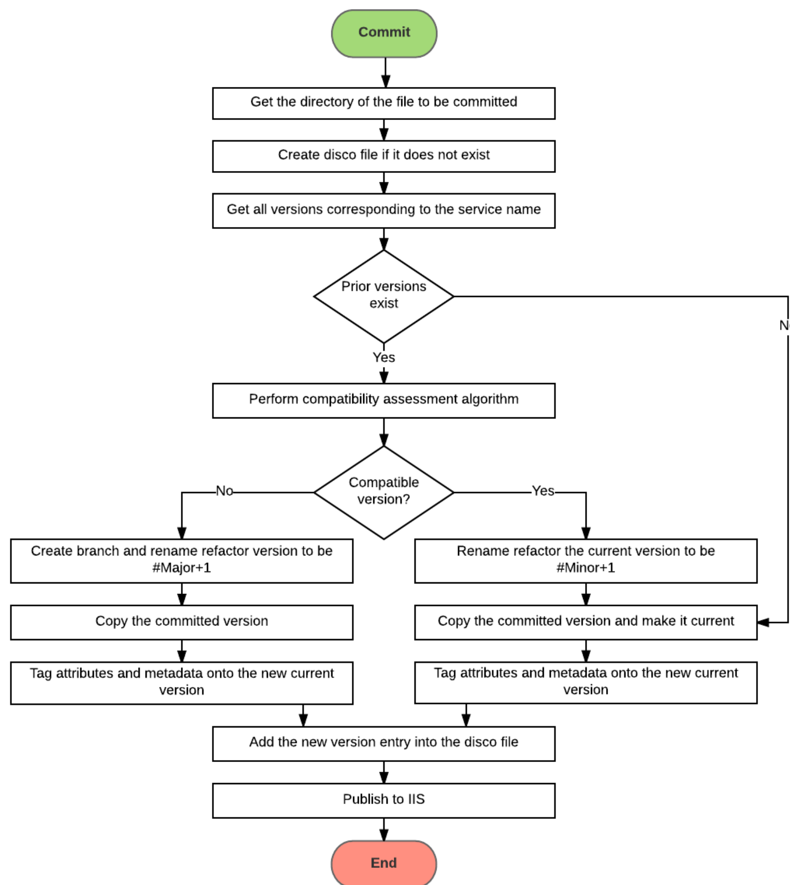

Figure 15. Commit diagram.

4.2.2. Checkout Operation

The Checkout operation can be seen as a three-fold operation that includes three main steps – namely, the discovery of peer nodes, the discovery of available services hosted on the discovered nodes, and the actual consumption of the discovered services.

Node Discovery

Peer discovery allows developers to identify all the nodes in a team project so that they can initiate connections with team members in a peer-to-peer network. Each node in the peer-to-peer network resides on a host connected to a network. Since SORC relies on peer-to-peer connections, developers need discovery mechanisms to find, identify and communicate with other peer developers (Kelaskar et al., 2002). The discovery mechanism in SORC is used to find directly connected peer nodes through which developers learn about each other and make multiple peer-to-peer connections. The auto-discovery mechanism supported by SORC does not require direct discovery of every single node; instead, the developer connects to one node and retrieves a list of other nodes that exist in the team. Once the team nodes have been discovered it becomes possible to retrieve developers' services and their versions.

The Checkout command is used to discover the services offered on a developer node. The Checkout command uses the entries in a special file that contains the addresses of known service providers involved in the project (i.e., *sorc.hosts*). The Checkout process also inspects and traverses the *sorc.hosts* file on the directly connected nodes. This is illustrated in Figure 16. When Node O runs the Checkout command it first goes to its own *sorc.hosts* file hosts and discovers three direct nodes A, B and C. The process then crawls outwards to discover additional hosts by visiting the *sorc.hosts* file on each of these three nodes. These *sorc.hosts* files reveal further nodes (D, E, F and G). Figure 16 illustrates the process of discovering developer nodes.

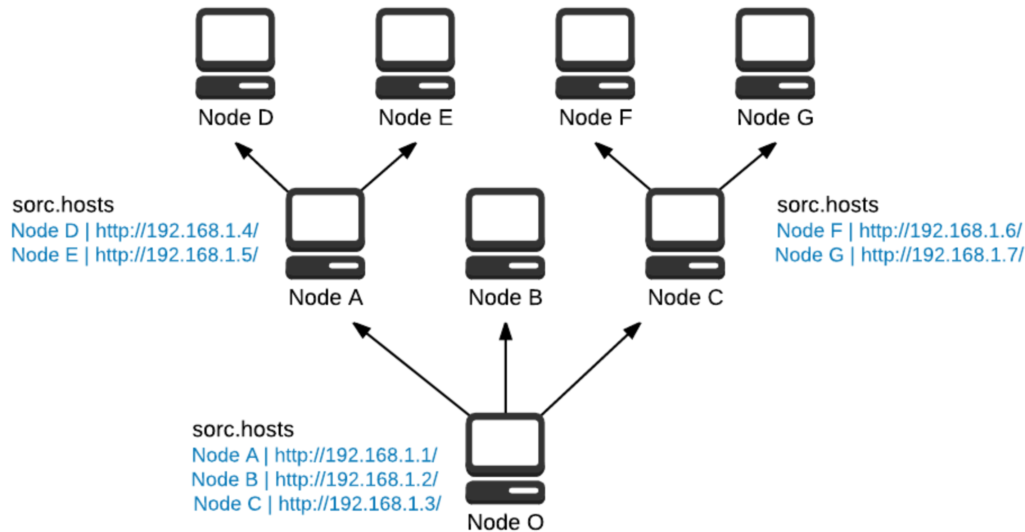

Figure 16. Developer node discovery.

Service Discovery

The service discovery mechanism accesses service directories that store information about services. Once the services and versions are discovered, service consumers can retrieve the specific WSDL documents and know exactly how to use each of the services. They can then specify which service and its exact version they want to use in order to generate proxy classes. The main proxy class represents a proxy to a Base Service, e.g., ServiceA. Sub-classes of the main proxy class represent proxies of the versions of a service, e.g., ServiceA_1_0 and ServiceA_2_0, allowing providers to publish many versions from the Base Service class. On the consumer's side, they can select different versions of the same service in their client applications.

Service Consumption

After the discovery process is completed and service providers are found with their corresponding services and versions, the service consumer performs Checkout to generate a proxy class based on the WSDL description of the selected service and its version. Service consumers can then invoke methods of the proxy class, which communicate with a service over the network by processing the SOAP messages sent to and from the service. Figure 17 illustrates the process of a Checkout

command operation, whereas Algorithm 5 presents pseudocode, where outPath is the path of consumed services, hostfile is the path of the *sorc.hosts* file on the consumer, and list is the list of peer host info.

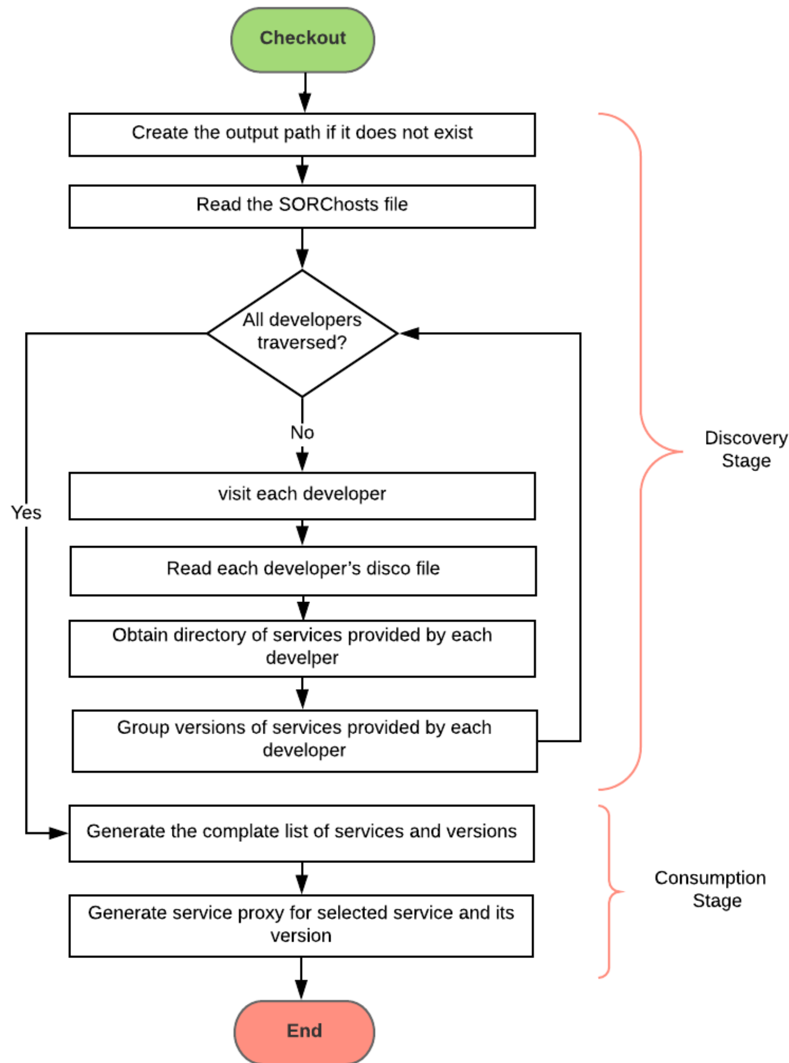

Figure 17. Checkout diagram.

Algorithm 5. Checkout algorithm.

```
Require: outPath, hostfile, list

1:    if outPath does not exist then
2:      create outPath
3:    end if
```

```
4:   get the peer hosts of the consumer from the sorc.hosts file
5:   for each peer host
6:     get the services from the disco file
7:     if services exist then
8:       for each service
9:         create service compilation
10:      end for
11:      for each service compilation
12:        get sorted version list
13:        determine the most recent version
14:        select the service and its version
15:        checkout the selected service
16:        test the service
17:        if pass then
18:          generate the proxy
19:        else
20:          exit
21:      end for
22:    else
23:      exit
24:  else
25:    exit
26:  end for
27:  update config file
```

4.2.3. Update Operation

A service consumer uses the Update command to become aware of changes to a service. The service consumer is in control of the client application and must deal with the evolution of services on which the application depends. However, keeping track of service changes is difficult since the consumer does not have access to the source code of the service. The client cannot have a full understanding of the reasons behind changes, which is crucial for the adoption process of new service versions (Fokaefs and Stroulia, 2012).

To manage this, the consumer must first recognise the changes made to the service. The ultimate task is the generation of a new proxy class for the new version of the service. This is accomplished by pulling the commit message associated with the new service and containing the compatibility assessment performed against the list of applied changes. This process is implemented by the Update command as illustrated in Figure 18. The new proxy is automatically generated upon compatibility assessment. However, client code may need to change to make use of the new proxy.

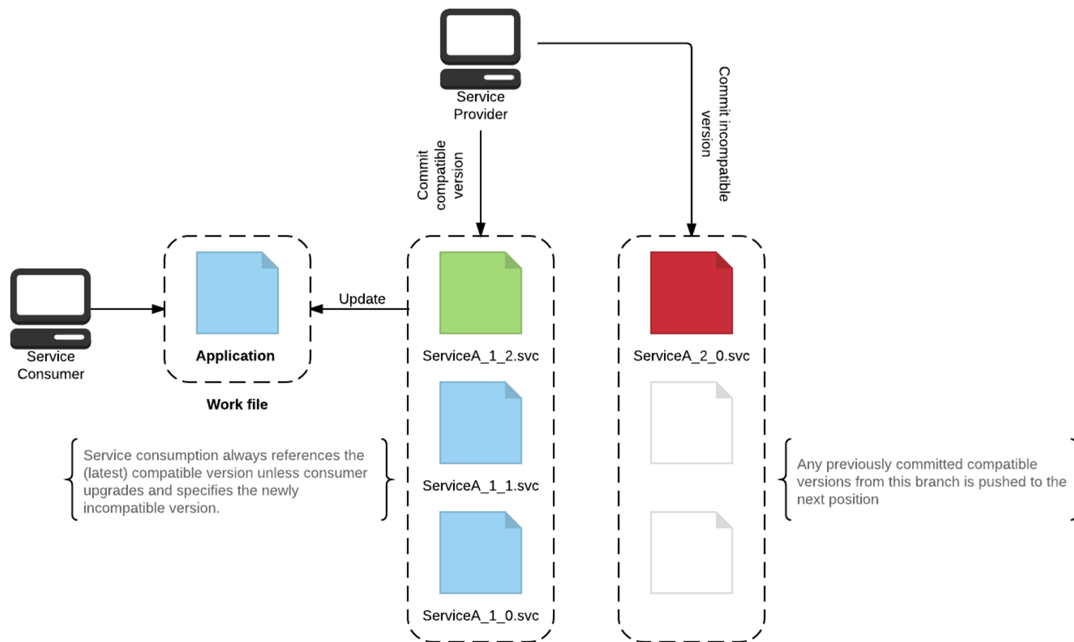

Figure 18. Update versioned services.

Figure 18 depicts the Update command as being based on the compatibility assessment for a new service version. If the new service version is considered compatible, the new proxy for the service will be automatically generated. If the version is incompatible, then the service consumer is asked whether to upgrade to the new version. Figure 19 illustrates the Update process, whereas the pseudocode in Algorithm 6 provides further detail, where outPath is the path of the consumed services, hostfile is the path of the *sorc.hosts* file of the consumer, and list is the list of peer host info.

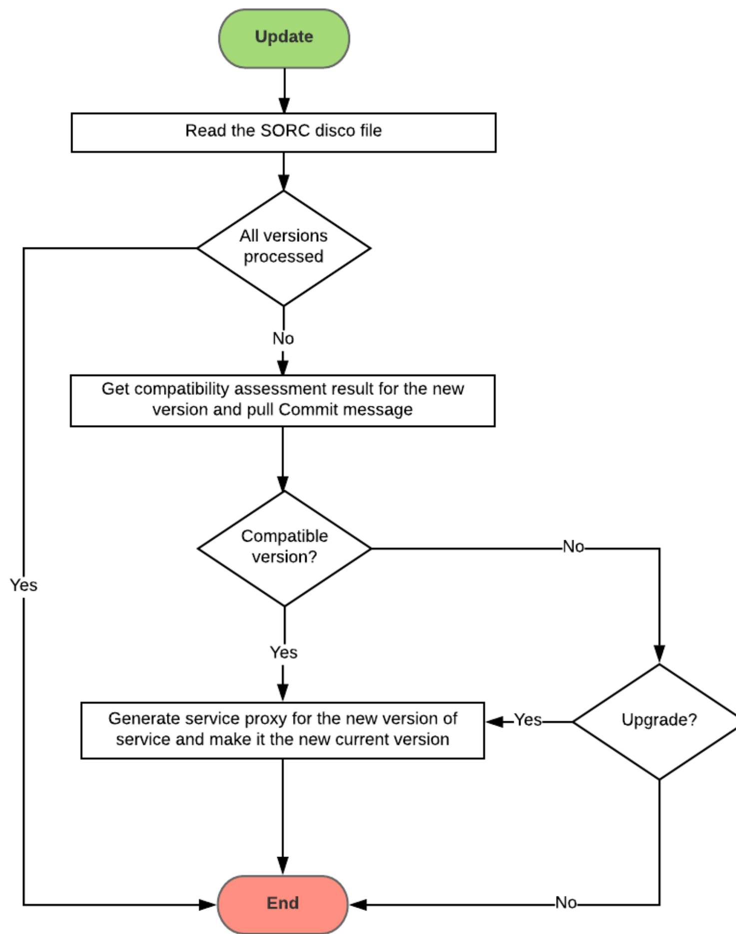

Figure 19. Update diagram.

Algorithm 6. Update algorithm.

```
      Input: outPath, hostfile, list
      1:   if outPath does not exist then
      2:     create outPath
      3:   end if
      4:   get the peer hosts of the consumer from the sorc.hosts file
      5:   for each peer host
      6:     get the services from the disco file
      7:     if services exist then
      8:       for each service
      9:         create service compilation
      10:      end for
      11:      for each service compilation
      12:        get sorted version list
      13:        determine the most recent version
      14:        if version list count >= 2 then
      15:           check compatibility of the recent and the previous
versions
      16:           if compatible then
```

```
17:            test service version
18:            if pass then
19:              generate proxy
20:            else
21:              exit
22:            end if
23:          else
24:            ask for confirmation
25:            if yes then
26:              test service version
27:              if pass then
28:                generate proxy
29:              else
30:                exit
31:              end if
32:            else
33:              exit
34:            end if
35:          end if
36:        else
37:          exit
38:        end if
39:      end for
40:    end if
41:  end for
42:  update config file
```

4.3. Expected Benefits

The correct implementation of the proposed SORC functionality will support the development process of a distributed SBS system by enabling geographically distributed developers to perform CI in a decentralised, loosely-coupled manner. More specifically, developers are expected to benefit from the following features of the SORC implementation:

- **Storage space and network bandwidth**: SORC does not require project artefacts to be physically present on each developer's node locally. Instead, it only generates local proxy classes for remote services, which do not occupy much disk space. This way, SORC avoids redundant code replicated on each developer's machine. Moreover, by minimising the amount of redundant code and artefacts, SORC is also expected to benefit from lower latencies and faster network communication.

- **Performance and speed of updates:** by not checking out project artefacts from the central server, SORC is expected to achieve better performance, as opposed to existing approaches that require downloading potentially heavy-weight SBS system components over bandwidth-limited networks. Instead, SORC only downloads light-weight WSDL descriptions and generates corresponding proxy classes locally.
- **Versioning granularity:** by following a strict versioning schema, SORC supports versioning of all modifications applied to a service, be it a minor or major update. All changes will be detected and assigned a corresponding version label.
- **Real-time collaboration:** thanks to the increased speed of updates, SORC has the potential to support real-time co-development, such that changes applied to a service by one developer almost immediately appear in all other peer developers' workspaces. This is considerably different from the existing approaches that require relatively long time to update project workspaces.
- **Access version metadata at run-time:** with SORC, artefact attributes and meta-data are defined declaratively and are embedded into service versions. As a result, this meta-data can be accessed at run-time, thus enabling dynamic reconfiguration of an SBS system by, for example, replacing one version of a service with another due to incompatibility.
- **Higher level of abstraction and separation of concerns:** SORC a proxy class in SORC does not contain actual implementation details, but rather represents an abstract "stub". Such abstraction provides developers with separation of concerns (i.e., the developers only deal with their own components) and a higher sense of protection of their own code from unauthorised access and modification.

- **Isolated compile-time errors:** In SORC, individual developers and their software components are isolated from each other and are not affected by faulty dependencies. This means that developers are not blocked in case a dependent service is malfunctioning or crashed.
- **Need for integration testing:** as opposed to the existing tools, SORC does not require (frequent and potentially computationally expensive) integration tests, since this kind of validation is already performed during the edit/build/test cycle.

The outlined benefits of the proposed SORC approach will be further revisited and discussed in the next chapter when evaluating a proof-of-concept implementation of the SORC system. Some of these benefits will be quantitively benchmarked and compared to an existing solution, whereas the rest will be demonstrated through a hypothetical discussion. Furthermore, the described benefits of the system will also be analysed in the context of the potential support for handling conflicts and broken builds, as well as the potential of the system to handle changes in external system components (i.e., not Web services).

4.4. Summary

This chapter introduced the main contribution of this thesis – Service-Oriented Revision Control – and explained the key elements underpinning this novel approach. The chapter also briefed the reader on the issues of versioning and compatibility assessment in the context of distributed service-oriented software development and explained how these are addressed by SORC. Next, the chapter discussed the main functionality of SORC, constituted by the three basic commands, Commit, Checkout, and Update, and explained how they are used in a SOA development process. As a next step, Chapter 5 will further extend this discussion with actual implementation details of

the proposed SORC vision. It will present the SORCER framework – a prototype implementation of the SORC architecture, and will evaluate it against a running SBS system.

5. Chapter 5: Case Study and Evaluation

This chapter contains more technical details on how the SORC vision was implemented and validated. The SORCER framework was developed to implement the SORC functionality based on the previously described reference architecture. To evaluate its viability and compare it to existing approaches, a setup with similar functionality was established using GitHub as a version control repository and Jenkins as a CI server. Next, the chapter presents a case study that involves a simple SBS system, which serves to demonstrate how the same features are supported by SORCER and the GitHub/Jenkins setup. This illustration serves to compare the two approaches and is used to evaluate the proposed SORC approach with respect to integration time, build time, and the disk space occupied by the software project. The chapter concludes with an analysis of the experimental results, as well as a discussion of the novel functional properties supported by SORC.

5.1. SORC Implementation: the SORCER Framework

We first start with a description of how the prototype of the proposed SORC system was implemented, based on the design presented in the previous chapter. Below are the key technologies that were used to implement the system:

- **Programming language: C#** was the main programming language used during the implementation phase. Developed and promoted by Microsoft as the central element of the .NET framework, it proved itself to be a simple, general-purpose, high-level, object-oriented programming language (Wagner, 2015). On top of the default object-oriented features, C# also introduced a number of novel features which further promoted the adoption of the language by the developer community:
    - Properties in C# serve as accessors for private member variables;
    - Encapsulated method signatures in C# serve to enable type-safe event notifications;

- C# supports inline XML documentation comments;
- C# supports a special type of attributes that can provide declarative meta-data about types at run-time;
- C# comes with Language-Integrated Query (LINQ) that can be used to query data across a variety of sources within the .NET ecosystem.

The C# code is executed by the .NET framework within a virtual environment called Common Language Runtime (CLR), which also offers a common collection of basic components and class libraries. Being an implementation of the Common Language Infrastructure (CLI), Microsoft's CLR is built to enable an execution and development environment, in which languages and libraries can integrate in a seamless and transparent manner.

- **Web Service framework:** Web services were implemented using the built-in support of the .NET framework. Being at the very core of the .NET ecosystem, **.NET Web services** are extremely easy to create and consume, since minimum number of code lines is required. The .NET framework abstracts most of the internal implementation logic behind services, and takes care of the remote method invocation routine over the network. The default .NET development environment Visual Studio offers rich support for developing, testing, deploying, and calling Web service directly from within the development environment. This way, server-side logic of .NET Web services is made easily and quickly available to client applications.

As a way of discovering and exposing published Web services, the DISCO functionality was introduced in .NET that allows clients to reflect endpoints to discover and publish services and their associated WSDL documents. The DISCO file typically references a

WSDL source that in turn points to the actual Web Service. When developers discover Web Services, they go to the DISCO file which provides the information on particular Web Services. A sample DISCO file, called *sorcer.disco*, is shown in Figure 20. This file is stored in the published Web service folder on a server and points to one or many WSDL documents, which correspond to separate version files available in the current application space. Keeping track of the revision history this way enables service consumers to discover and select specific version of a particular service among several version available on that node. The list of available versions and corresponding WSDL documents, is, however, unavailable, until the *.disco* file is published on the server.

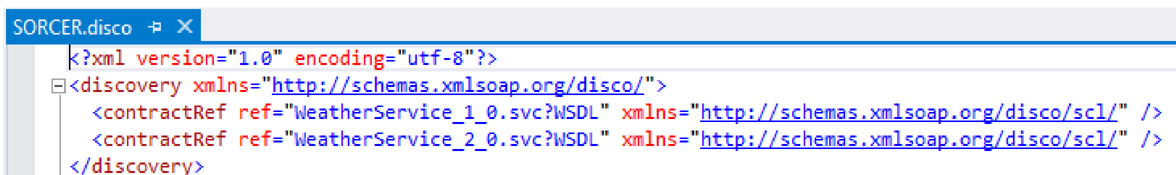

Figure 20. A sample of sorcer.disco file.

- **SOA implementation: Windows Communication Foundation (WCF)** is a .NET technology stack for implementing and deploying complex SOAs (Jones, 2017). Following the principles of distributed computing and loose coupling, WCF supports publishing and accessing WSDL documents, as well as developing client applications for consuming remote services, regardless of the underlying implementation technology. To support this, WCF follows the established Web services (WS) standards, such as WS-Addressing, WS-ReliableMessaging, and WS-Security. In WCF, a client application communicates to a remote service using one or many WSDL end points, which have a unique address (i.e., URL) and a set of binding properties that provide required information on how exactly communication and information exchange will take place. When published, a WCF service is associated with a *.svc* file that specifies the exact .NET class that contains the Web service

programming logic. Apart from the *ServiceContract* annotation in its prefix, this class typically does not differ from any other .NET class implementation.

- **Web server:** to deploy and host developed Web services it was required to run a dedicated web server. To this end, **Internet Information Services (IIS)** (K. Schaefer et al., 2012; Microsoft, 2017), formerly known as Internet Information Server, was used. IIS is a web server distributed by Microsoft, which offers an extensible modular platform for hosting web sites, services, and other applications in a secure, reliable, and manageable manner.

By bringing together the described technologies and tools, it became possible to develop the SORCER framework as a pluggable component for the Visual Studio IDE, as well as to run the experiments and compare them to the existing approaches. A sample screenshot of the SORCER plugin is displayed in Figure 21. The SORCER project settings are stored in the developer's workspace in a file with an extension *.sorcerproj*. The main functions available to the user are the following:

- **Publish Project File** is an XML based file with an extension *.publishproj* that contains a list of properties and supported functionality.
- **Publish Profile** is a name of the publish profile (a *.pubxml* file).
- **Project Location** is the location of the project in the developer's workspace.
- **Public URL** is the IP address of the developer creating the project.
- **Developer Name** is the name of the developer creating the project.
- **Visual Studio Version** is the version of .NET Visual Studio used by the project (supported versions are 2012 and 2013).
- **Peer Developer** is a list of URLs (i.e., IP addresses) of developers involved in the project team.

- **SORCER Project Name** is the name of the project currently being developed.

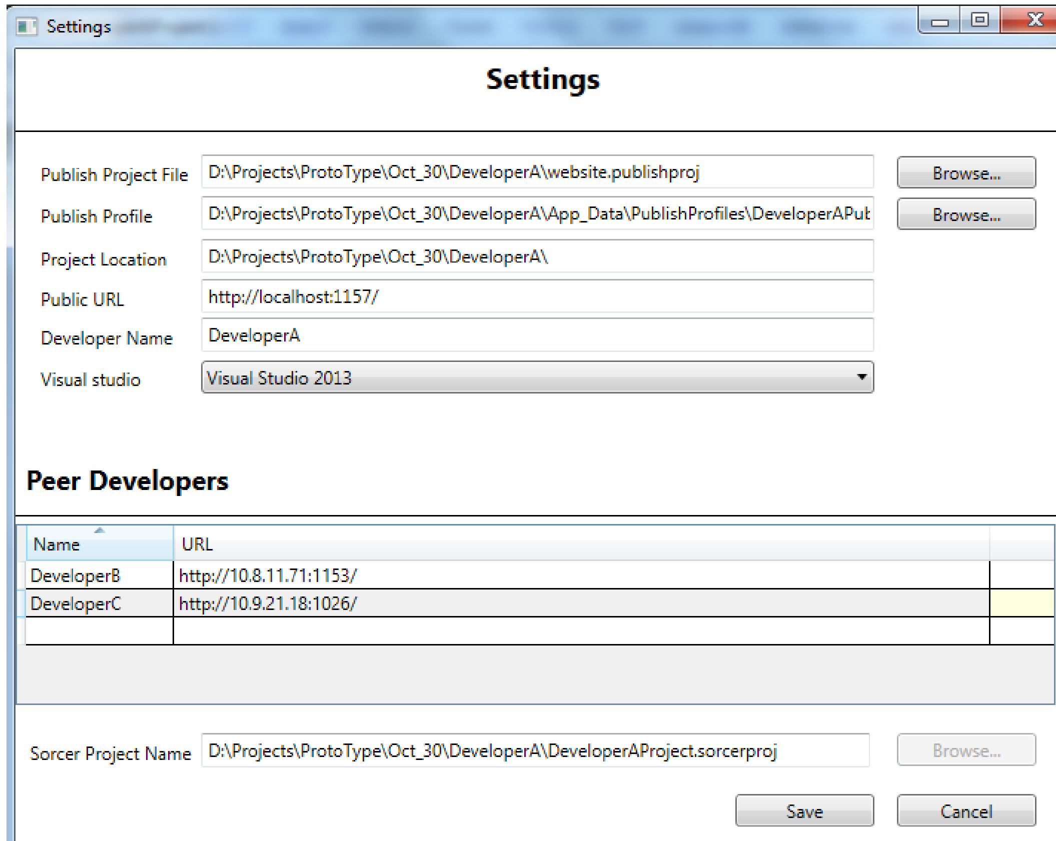

Figure 21. SORCER interface.

5.2. Case Study: Comparing SORCER to the Existing Technological Baseline

The main goal of the case study is to demonstrate the viability of the proposed SORC approach and the implemented SORCER framework, as well as their potential advantages with respect to the current way of implementing CI in the context of service-oriented software development. Accordingly, we first explain the current technological baseline and how CI in distributed service-oriented software development is presently handled.

Admittedly, the novel functionality of SORC goes beyond the supported features of the existing tools, making it impossible to precisely map them to SORCER's functionality. A potential solution

in these circumstances was to attempt to “replicate” the required functionality using existing tools. As it was described in Section 1, a widely adopted solution in these circumstances is to combine two existing technologies to facilitate for SOAs. As a result, in the experiments described below, the SORCER framework will be benchmarked against a setup of GitHub (as a source code repository and version control system) and Jenkins (as a CI engine).

The experiments are based on a hypothetical, yet realistic use case scenario that focuses on a developing an SBS system. The SBS system can be seen as a tourist information application that offers two types of information. First, it provides weather forecast details for a specific geographic location. Second, it also informs tourists on the current currency exchange rates in that specific location. Accordingly, the SBS system is composed of three main components: the main client application itself, a weather forecast Web service, and a currency exchange rate Web service.

Below we describe how this scenario is supported by SORCER and the combination of GitHub and Jenkins. There are three developers involved in this scenario. **DeveloperA** is responsible for the main application that consumes the weather forecast and currency exchange services maintained by the remote **DeveloperB** and **DeveloperC**, respectively. The main methods of the two services to be invoked by the main client application are **getWeatherData** (takes a specific geographic location as an input **parameter) and getExchangeRate** (takes source and target currencies as parameters), respectively.

5.2.1. Case Study Implemented by SORCER

We now put theory into practice and explain how the SORCER framework is supposed to be used by SOA developers to enable CI. This is best illustrated through a sequence diagram in Figure 22 and explained with a few screenshots as follows.

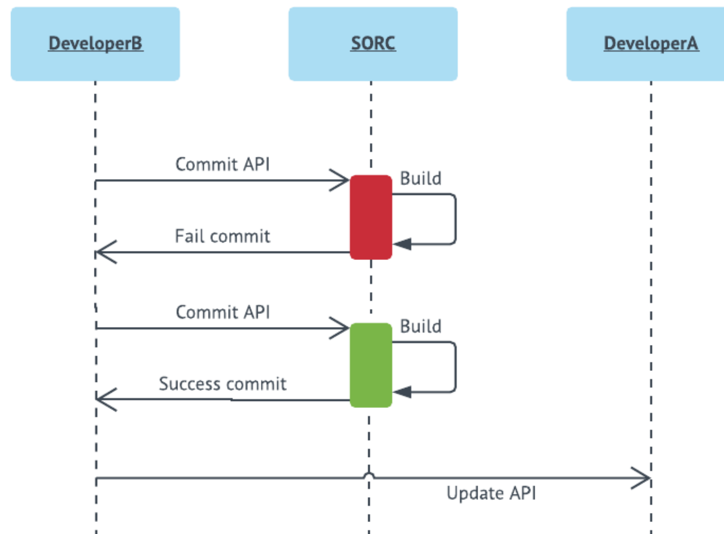

Figure 22. SORCER workflow sequence diagram.

As already said, there are three developers – namely, DeveloperA, DeveloperB, and DeveloperC – that are involved in the SORCER workflow, who are assumed to run the SORCER framework along their IDEs during the SOA development process. DeveloperA is assumed to create an SBS system that consumes the weather forecast service created by DeveloperB and the currency exchange rate service created by DeveloperC.

1. The whole process is initiated when all three developers create their respective projects in Visual Studio IDE, and publish them on a server.
2. Next, using the SORCER plugin, the developers are required to specify configuration for their projects by filling in the required fields, as depicted in Figure 23, Figure 24, and Figure 25. Please note that DeveloperA also needs to specify the URL of DeveloperB's and DeveloperC's working stations, since the SBS system is expected to consume the remote weather forecast and currency exchange rate services. DeveloperB and DeveloperC can omit this field, since their Web services are not dependent on any external services. After

this step, required SORCER files are created on all three machines, as illustrated by Figure 26.

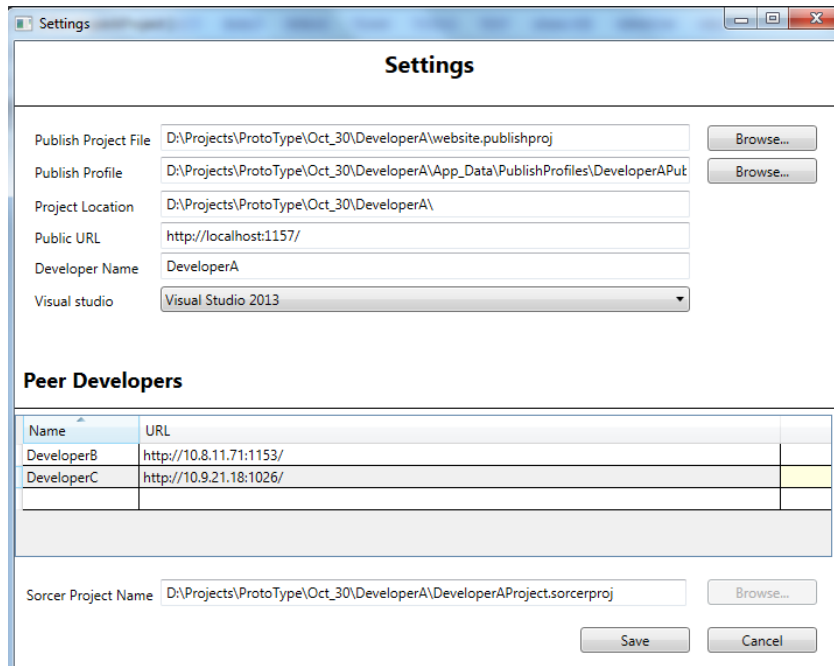

Figure 23. SORCER configuration for the SBS system by DeveloperA.

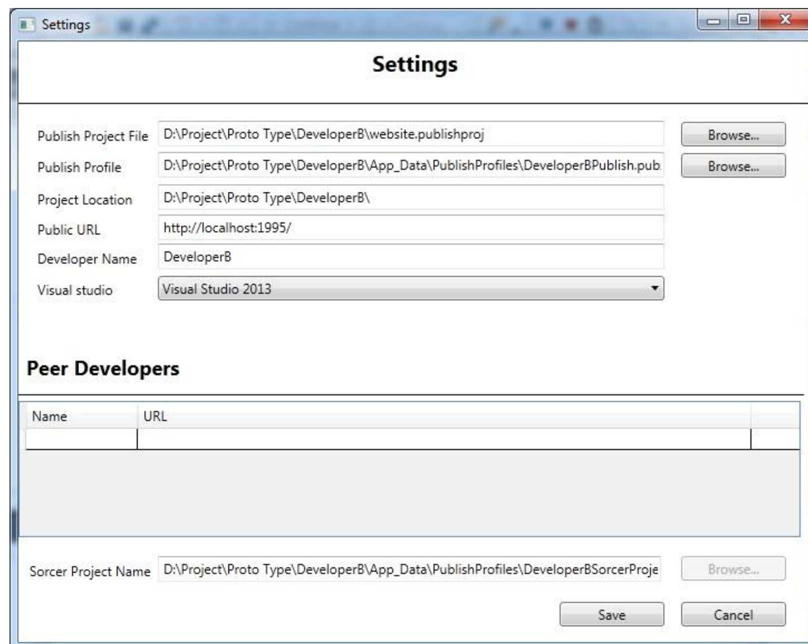

Figure 24. SORCER configuration for DeveloperB's weather forecast service.

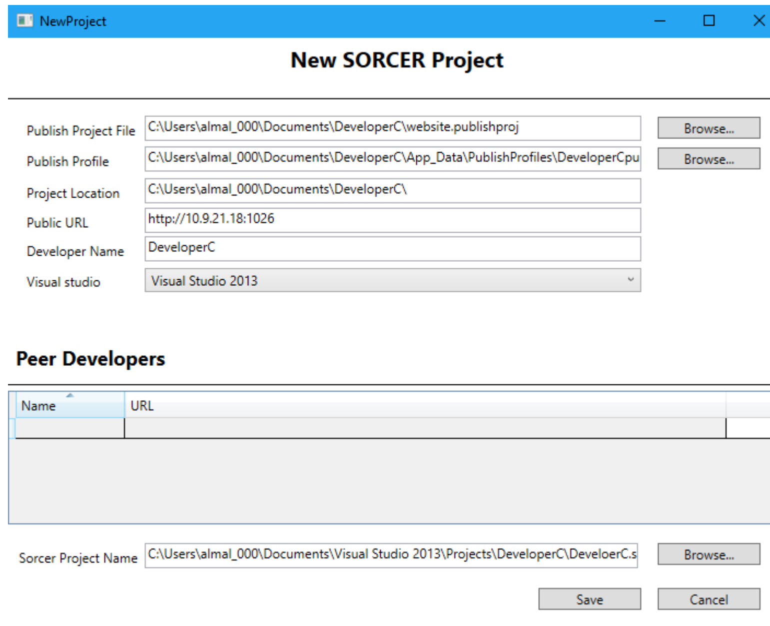

Figure 25. SORCER configuration for DeveloperC's currency exchange rate service.

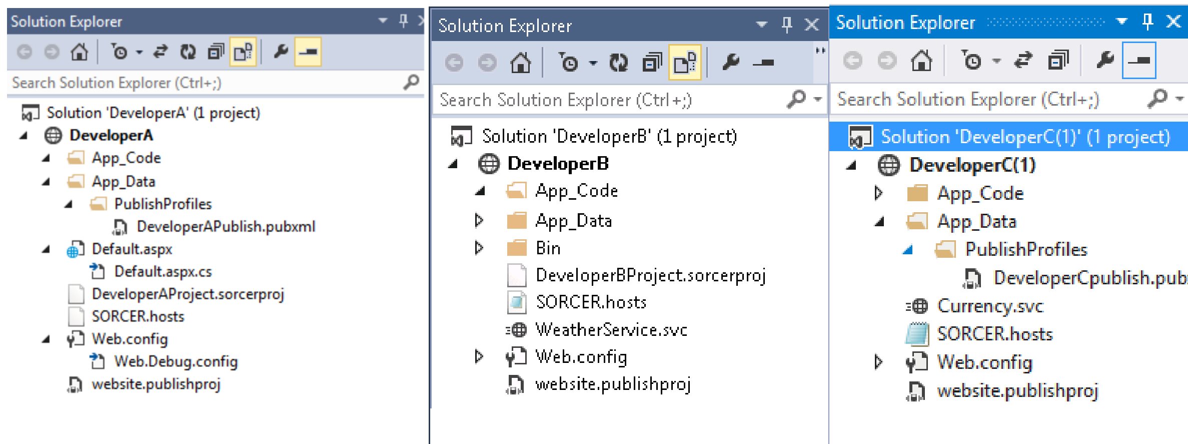

Figure 26. SORCER files created on all three developers' machines.

3. Having configured the tourist information SBS system as a team project, DeveloperB and DeveloperC are now ready to commit the first version of their services (under the version **WeatherService_1_0** and **Currency_1_0**, respectively) through the SORCER tab to be later invoked by DeveloperA.

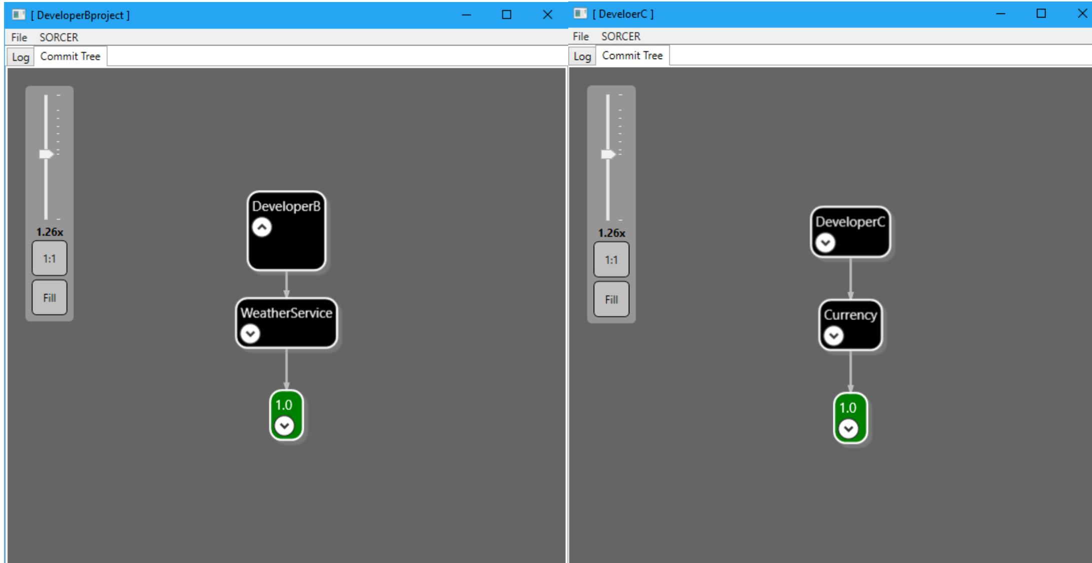

Figure 27. DeveloperB and DeveloperC commit the first versions of the weather forecast and currency exchange rate services.

4. DeveloperA is now ready to run the Checkout command to get updated with the latest changes of the associated services. Since DeveloperA explicitly specified the network addresses of DeveloperB and DeveloperC, SORCER displays available services on those specific remote nodes. Next, DeveloperA is requested to select the required services and specific versions, after which corresponding proxies are generated on the local machine, as depicted in Figure 28.

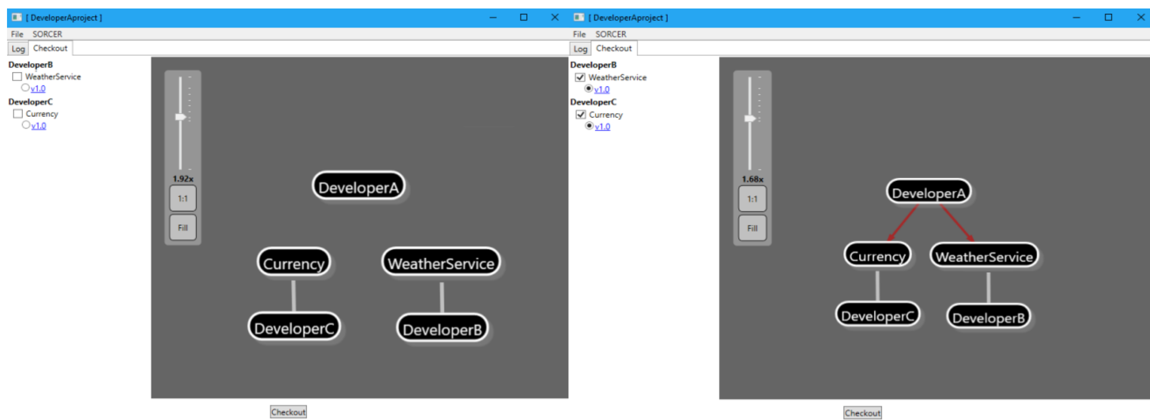

Figure 28. DeveloperA is able to see both developers' services after running the checkout command.

5. Finally, after generating local proxies, the weather forecast and currency exchange rate services appear in DeveloperA's project space with a list of provisioned methods, as illustrated by Figure 29 and Figure 30.

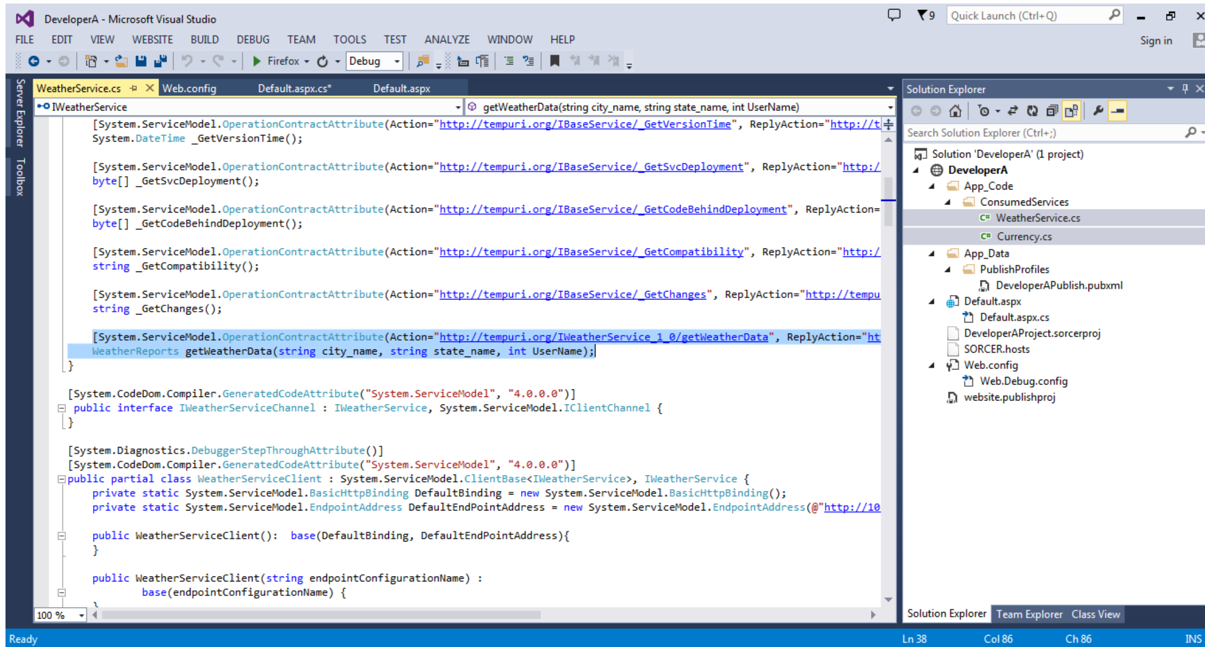

Figure 29. Generated proxy class for the weather forecast service.

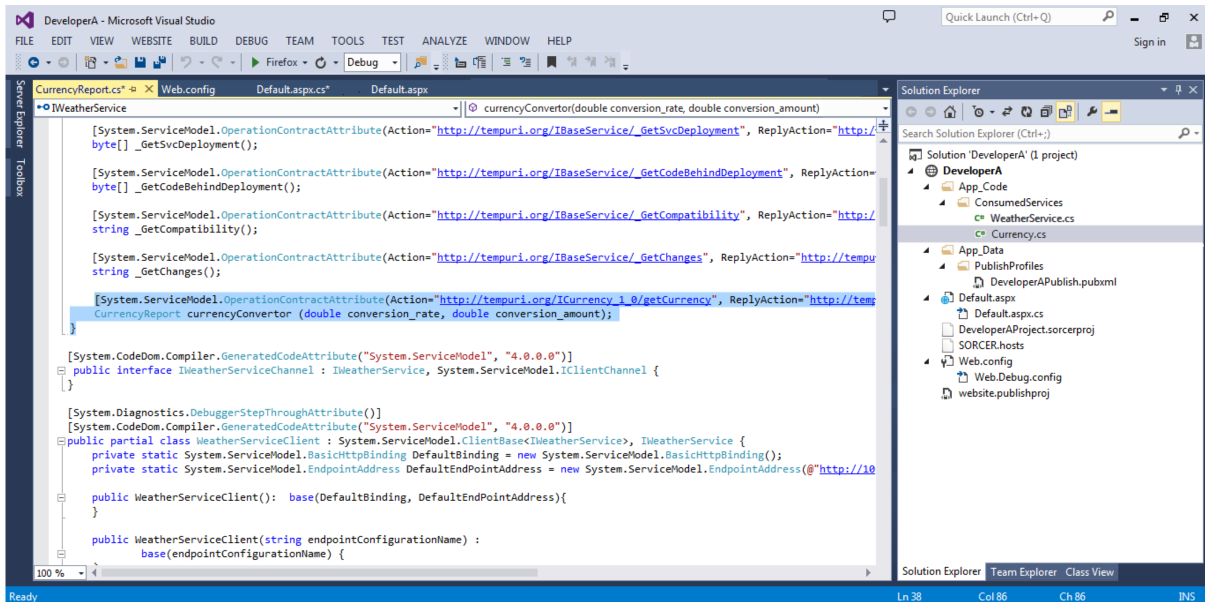

Figure 30. Generated proxy class for the currency exchange rate service.

6. Now, the SBS system created by DeveloperA is able to invoke the **getWeatherData** method of WeatherService and the **getExchangeRate** method of CurrencyService. A sample illustration of the service invocation is depicted in Figure 31. By passing names of a city as a parameter to the **getWeatherData** method, as well as source and target currencies and the amount to be converted, the tourist information application is able to receive corresponding weather forecast information and convert the required amount of money using the current exchange rate.

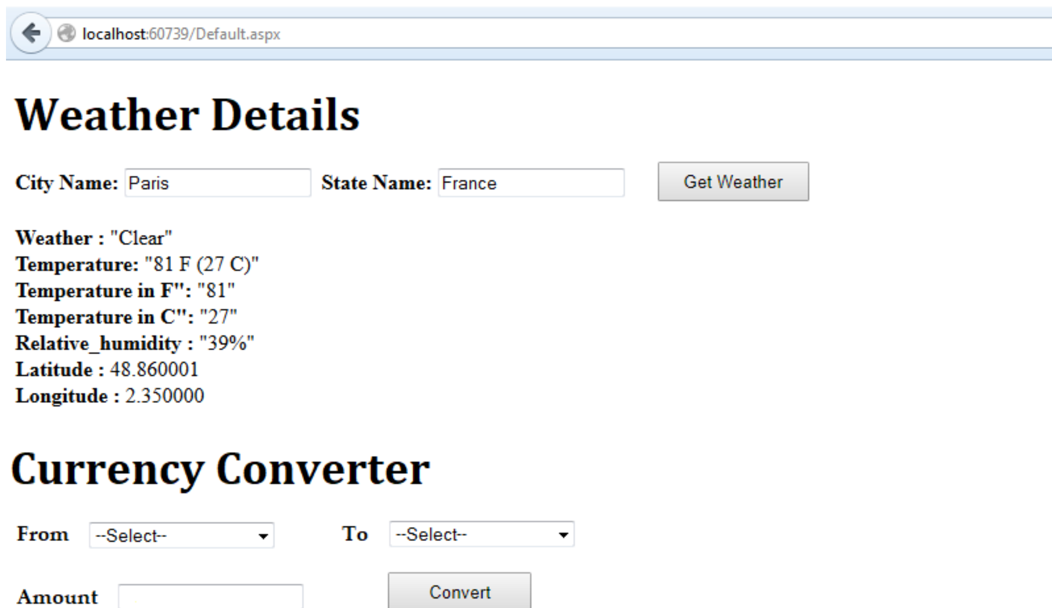

Figure 31. A sample view of the SBS system that invokes the remote services.

After the initial versions of the services have been committed, the next step of this scenario involves consequent updates to one of the Web services. To this end, we assume that DeveloperB commits an updated version of the weather forecast service. At this point, it becomes important to perform compatibility assessment – that is, to check whether a new version of the service is still compatible with the overall SBS system. As previously described, we differentiate between compatible and incompatible changes, each of which is presented below.

Compatible changes

1. DeveloperB makes a compatible change, by adding a new method **GetLatLong** to the weather forecast service.
2. DeveloperB commits the new version. As the change is compatible, the new version will be **WeatherService_1_1**.
3. After the new version is committed by DeveloperB, DeveloperA runs the Update command and automatically generates the proxy class for the new version.

Incompatible changes

1. DeveloperB makes an incompatible change to the weather forecast service, by renaming the original method **getWeatherData** to **getWeatherDetails.**
2. DeveloperB commits the new version. As the change is incompatible, the new version will be **WeatherService_2_0**.
3. After the new version is committed by DeveloperB, DeveloperA runs the Update command. This will result in a message asking for confirmation of whether the upgrade to the incompatible version should be applied.
4. If DeveloperA decides to proceed to upgrade to the incompatible version, a corresponding proxy will be generated.
5. The IDE indicates an error in DeveloperA's code where the method **getWeatherData** is invoked.

5.2.2. Case Study Implemented by GitHub/Jenkins

As it was briefly explained, it is possible to implement the CI functionality similar to the one of SORCER using a combination of the two tools – namely, GitHub as a source code repository and version control system, and Jenkins as the actual CI engine. When implementing this setup, it was

important to ensure that it would actually be possible to compare the two setups (SORCER vs GitHub/Jenkins), and the proposed comparison would be fair and correct. Furthermore, it was also important to agree on some common benchmarking metrics that would apply to both setups and serve to compare them one with another. As explained below, such common shared metrics for both setups were the total integration time and the occupied disk space.

We first explain how the same tourist information SBS system can be developed using GitHub/Jenkins following the CI practices. It is assumed that all three developers have set up GitHub repositories. More specifically, DeveloperA owns three repositories – i.e., one for the main client application, and other two for the weather forecast service (created and managed by DeveloperB) and the currency exchange rate service (created and managed by DeveloperC). Accordingly, DeveloperB and DeveloeprC own single repositories for theit corresponding Web services. As it will be further explained in more details below, the reason for this is that DeveloperA is responsible for building the whole SBS system consisting of both the client application and the two Web services, and therefore is required to have all project components physically present in one place. For this reason, it is required that DeveloperB's amd DeveloperC's Web service code is replicated in DeveloperA's repository as well.

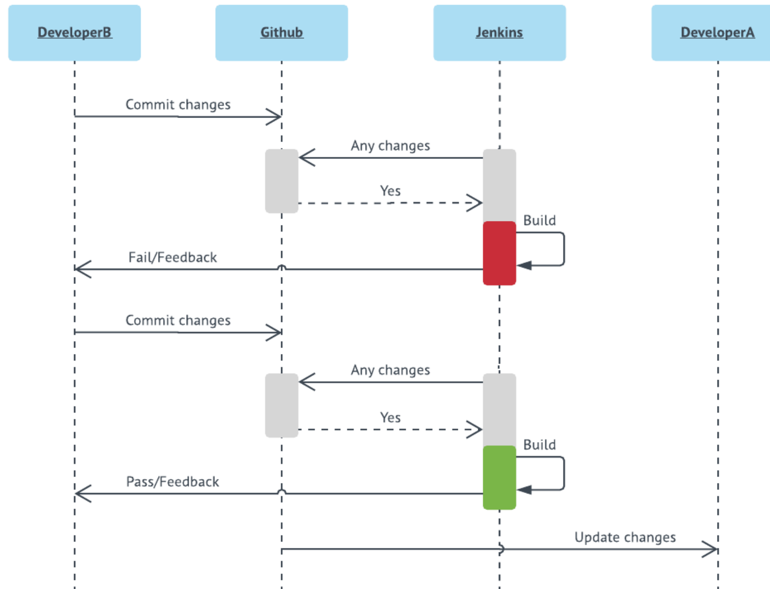

Figure 32. GitHub/Jenkins workflow sequence diagram.

It is also assumed that there is a Jenkins engine (managed by DeveloperA, since he is the owner of the overall SBS system) coupled with the GitHub repository in such a way that the project building routine is triggered every time there are new changes committed either to the client app (by DeveloperA), the weather forecast service (by DeveloperB), or the currency exchange rate service (by DeveloperC). In the latter two cases, DeveloperA's service repositories are first updated with the latest changes so that Jenkins can build the project using the client application and the Web services stored on one physical machine. The whole process is depicted by the sequence diagram in Figure 32,[16] and is described below step by step:

1. DeveloperB creates the **WeatherService** and pushes it to a project on GitHub.
2. DeveloperA clones the **WeatherService** project into a local repository.

---

[16] Please note that similar steps apply to DeveloperC, which is excluded from the diagram for better clarity.

3. DeveloperA develops an application coupled with the **WeatherService** and pushes the application to GitHub.
4. DeveloperB makes changes to the existing service by adding a new method and pushes the updated service to GitHub.
5. Upon detection of the updated version of the **WeatherService**, the Jenkins server tries to build and compile the project to check whether the application still runs with the new version of the service[17] (a sample output is displayed in Figure 33).

Console Output

```
14:23:51 Started by user Admin
14:23:51 Building in workspace C:\Users\vpchauhan\.jenkins\workspace\Jameel_WeatherProject
14:23:51 Cloning the remote Git repository
14:23:51 Cloning repository https://github.com/almalki00/WeatherProject.git
14:23:51  > git.exe init C:\Users\vpchauhan\.jenkins\workspace\Jameel_WeatherProject # timeout=10
14:23:51 Fetching upstream changes from https://github.com/almalki00/WeatherProject.git
14:23:51  > git.exe --version # timeout=10
14:23:51 using GIT_ASKPASS to set credentials
14:23:51 Setting http proxy: 10.8.18.42:8080
14:23:51  > git.exe fetch --tags --progress https://github.com/almalki00/WeatherProject.git +refs/heads/*:refs/remotes
/origin/*
14:23:56  > git.exe config remote.origin.url https://github.com/almalki00/WeatherProject.git # timeout=10
14:23:56  > git.exe config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
14:23:56  > git.exe config remote.origin.url https://github.com/almalki00/WeatherProject.git # timeout=10
14:23:56 Fetching upstream changes from https://github.com/almalki00/WeatherProject.git
14:23:56 using GIT_ASKPASS to set credentials
14:23:56 Setting http proxy: 10.8.18.42:8080
14:23:56  > git.exe fetch --tags --progress https://github.com/almalki00/WeatherProject.git +refs/heads/*:refs/remotes
/origin/*
14:23:59  > git.exe rev-parse "refs/remotes/origin/master^{commit}" # timeout=10
14:23:59  > git.exe rev-parse "refs/remotes/origin/origin/master^{commit}" # timeout=10
14:23:59 Checking out Revision 114bdf709768383c14167aa86f3703bda5a77434 (refs/remotes/origin/master)
14:23:59  > git.exe config core.sparsecheckout # timeout=10
14:23:59  > git.exe checkout -f 114bdf709768383c14167aa86f3703bda5a77434
14:24:01 Commit message: "Change By DeveloperA"
14:24:01 First time build. Skipping changelog.
14:24:02 Finished: SUCCESS
```

Figure 33. Sample output of the Jenkins build process.

5.3.3. Testbed Setup and Methodology of the Experiments

The two metrics to be measured and evaluated in the context of the experiments are the following:

---

[17] It is important to note here that the GitHub/Jenkins combination, as opposed to SORCER, does not perform any compatibility assessment. That is, Jenkins simply takes as input two software components (i.e., the client application and the Web service) and tries to build the overall project. This means that potential project failures will become apparent at a relatively late stage when the incompatible Web service code is already committed and replicated to DeveloperA's repository.

1. **Integration time** is the amount of time required to commit a change to a service and build the project with the new version. Due to the described differences in the two approaches, integration time in SORCER and GitHub/Jenkins setups is measured and calculated differently (albeit they are still aligned and refer to the same non-functional characteristic of the two setups). It is also worth mentioning that the two approaches employ slightly different terminologies to refer to the basic operations. For the sake of consistency, we will follow the common terms "commit" and "update" to refer to both SORCER and GitHub/Jenkins (where the terms "Push" and "Pull" are primarily adopted).

    - **Integration time in SORCER** is calculated as a sum of **Update time** and **Commit time**, as illustrated by Figure 34. After a sufficient number of experiments, the integration time is averaged.

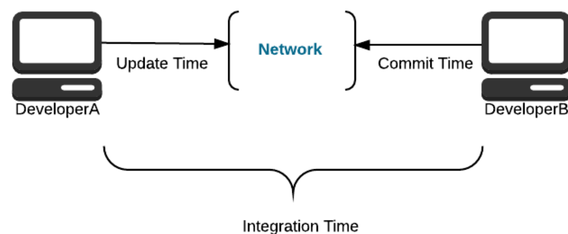

Figure 34. Measuring integration time in SORC.

    - **Integration time in GitHub/Jenkins** is calculated as a sum of **Push time, Build time,** and **Pull time**, as illustrated by Figure 35. Similarly, after a sufficient number of experiments, the GitHub/Jenkins integration time is averaged.

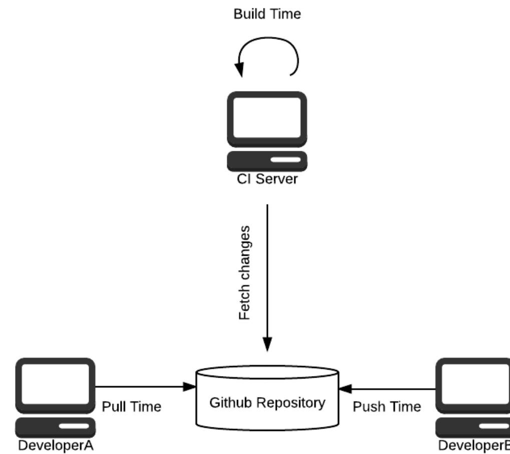

Figure 35. Measuring integration time in GitHub/Jenkins.

2. **Build time** is a sub-metric of the integration time, which highlights the core phase of the CI process. In the GitHub/Jenkins setup, the build operation is undertaken by the Jenkins server, whereas in SORCER the build operation is the second part of the commit process which generally takes place on the provider side (i.e., DeveloperB and DeveloperC).
3. **Occupied disk space** is the total amount of space occupied by the project artefacts. The main difference between the two approaches is that in SORCER there is no need to check out all the project components (i.e., services), whereas with GitHub/Jenkins all artefacts have to be collected (i.e., checked out) in one location to enable project building. The combination of the two technologies requires an excessive amount of disk space to store software project artefacts. It is worth noting that in this setup, GitHub repositories of all three developers remain the primary locations for storing files. Jenkins also requires storage of some temporary files when building the project, but it is assumed to have no impact on the overall occupied space since it pulls files from a related GitHub repository. Therefore, we only measure the space occupied in the GitHub repositories.

To conduct the experiments and measure the integration time (including build time) and occupied disk space, three server machines were used – i.e., by DeveloperA for developing and deploying the client application (in the second set of experiments Jenkins was running on this machine), and by DeveloperB and DeveloperC for developing and deploying the Web services (in the first set of experiments SORCER runs all pre-build checks on these machinee). Table 4,Table 5, and Table 6 contain the hardware specifications of the three servers.

Table 4. System specification of PC1 (DeveloperA).

System Parameters	Specification
Manufacturer	Dell
Model	Dell – OptiPlex 3020
Processor	Intel® Core™ i5-4570QM CPU
Speed	3.20 GHz
Memory (RAM)	4.00 GB
Operating System	Window 7 Professional
IP address	10.8.11.184

Table 5. System specification of PC2 (DeveloperB).

System Parameters	Specification
Manufacturer	Dell
Model	Dell – OptiPlex 3020
Processor	Intel® Core™ i3-3220 CPU
Speed	3.30 GHz
Memory (RAM)	8.00 GB

Operating System	Window 8 Professional
IP Address	10.8.11.150

Table 6. System specification of PC3 (DeveloperC).

System Parameters	**Specification**
Manufacturer	HP
Model	HP Pavilion 15-n245se
Processor	Intel® Core™ i7-4500 CPU
Speed	2.4 GHz
Memory (RAM)	8.00 GB
Operating System	Window 8 Professional
IP address	10.9.21.18

5.4. Experiments and Benchmarking

5.4.1. Integration Time and Occupied Disk Space with SORCER

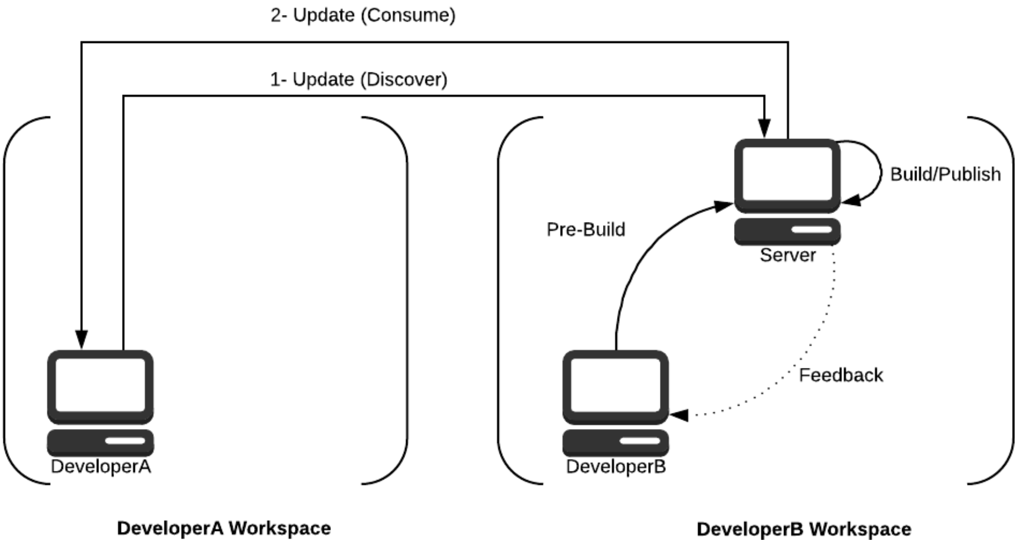

Figure 36. The SORCER process.

The experiments were conducted in a series of 10 iterations, in which service changes were first committed and then updated, following the process diagram in Figure 36. The commit and update operations have further sub-metrics, by measuring which it was possible to calculate the total times for commit and update operations.

More specifically, the **commit time** can be calculated by summing the time required to run the following sub-operations:

- **Pre-Build** covers operations before the service gets published, including compatibility checking, versioning and generating commit messages.
- **Publishing and Building** is the process from building and publishing the service till the developer receives feedback on the status of the project build.

Benchmarking results for commit time in SORCER are summarised in Table 7 (DeveloperB commits the weather forecast service) and Table 8 (DeveloperC commits the currency exchange rate service).

Table 7. Weather forecast service commit time in SORCER.

DeveloperB – Feather Forecast Service				
No. of Changes	**Commit**		**Service Code Size, kB**	**Total Commit Time, ms**
	Pre-Build, ms	**Build & Publish, ms**		
1	64[18]	898	7.23	962
2	538	1192	7.39	1730

[18] Please note the significantly low value for the first iteration. This is due to the fact that the initial commit of the two services did not go through compatibility assessment, since there is not previous version to compare it with.

3	568	938	7.34	1506
4	676	984	7.23	1660
5	632	884	7.41	1516
6	731	928	7.42	1659
7	616	887	7.39	1503
8	648	894	7.33	1542
9	716	890	7.41	1606
10	562	879	7.38	1441
	Avg: 575.1	**Avg: 937.4**	**Avg: 7.5184**	**Avg: 1512.5**

Table 8. Currency exchange rate service commit time in SORCER.

DeveloperC – Currency Exchange Rate Service				
No. of Changes	**Commit**		**Service Code Size, kB**	**Total Commit Time, ms**
	Pre-Build, ms	**Build & Publish, ms**		
1	54	741	5.318	795
2	470	892	5.373	1362
3	516	838	5.355	1354
4	588	917	5.361	1505
5	590	896	5.401	1486
6	649	949	5.413	1598
7	632	892	5.421	1524
8	669	884	5.39	1553

9	706	901	5.418	1607
10	673	927	5.425	1600
	Avg: 554.7	**Avg: 883.7**	**Avg: 5.3875**	**Avg: 1438.4**

The **Update time** consists of the following two steps:

- **Service discovery** is the time required to discover the new version, i.e., a newly committed change from the service providers DeveloperB and DeveloperC.
- **Service consumption** is the time required to generate a proxy of a newly-committed service version.

Benchmarking results for update time in SORCER are summarised in Table 9 (DeveloperA updates the weather forecast service) and Table 10 (DeveloperA updates the currency exchange rate service).

Table 9. Weather foreacst service update time in SORCER.

No. of Changes	**DeveloperA – Updating the weather forecast service**			
	Update		**Code Size, kB**	**Total Update Time, ms**
	Discover, ms	**Consume, ms**		
1	674	464	25.8	1138
2	585	584	26.5	1169
3	593	564	26.8	1157
4	598	540	30.9	1138
5	699	447	36.2	1146
6	564	510	40.2	1074
7	504	584	40.9	1088

8	594	598	45	1192
9	481	540	50.3	1021
10	469	523	52.8	992
	Avg: 576.1	**Avg: 535.4**	**Avg: 37.54**	**Avg: 1111.5**

Table 10. Currency exchange rate service update time in SORCER.

No. of Changes	**DeveloperA – Updating the currency exchange rate service**			
	Update		**Code Size, kB**	**Total Update Time, ms**
	Discover, ms	**Consume, ms**		
1	671	563	27.4	1234
2	635	571	28.1	1206
3	655	574	28.4	1229
4	598	602	32.5	1200
5	667	547	37.8	1214
6	598	581	41.8	1179
7	607	612	42.5	1219
8	668	642	46.6	1310
9	621	593	51.9	1214
10	649	611	54.4	1260
	Avg: 639.9	**Avg: 589.6**	**Avg: 39.14**	**Avg: 1226.5**

Finally, by summing the commit and update times, it is possible to calculate the elapsed time in order to get the total average **integration time** in SORCER. These results are summarised in Table 11 (the weather forecast service) and Table 12 (the currency exchange rate service).

Table 11. Weather forecast service integration time in SORCER.

No. of Integration	Integration Time		
	Commit, ms	**Update, ms**	**Total Integration Time, ms**
1	962	1138	2100
2	1730	1169	2899
3	1506	1157	2663
4	1660	1138	2798
5	1516	1146	2662
6	1659	1074	2733
7	1503	1088	2591
8	1542	1192	2734
9	1606	1021	2627
10	1441	992	2433
			Avg: 2624

Table 12. Currency exchange rate service integration time in SORCER.

No. of Integration	Integration Time, ms		
	Commit	**Update**	**Total**
1	795	1234	2029
2	1362	1206	2568

3	1354	1229	2583
4	1505	1200	2705
5	1486	1214	2700
6	1598	1179	2777
7	1524	1219	2743
8	1553	1310	2863
9	1607	1214	2821
10	1600	1260	2860
			Avg: 2664.9

Finally, assuming that integration of the two services into the overall tourist information SBS system takes place consequently (i.e., remote services are checked out and updated one after another, not in parallel), the total integration time is easily derived by summing up the corresponding metrics for the two services. Accordingly, Table 13 contains the calculated results for the overall integration time using SORCER, when both services are integrated into the SBS system. As a result, the average integration time is 5288.9 milliseconds.

Table 13. Total integration time in SORCER.

Integration Time between DeveloperA and DeveloperB, ms	**Integration Time between DeveloperA and DeveloperC, ms**	**Total, ms**
2100	2029	4129
2899	2568	5467
2663	2583	5246
2798	2705	5503

2662	2700	5362
2733	2777	5510
2591	2743	5334
2734	2863	5597
2627	2821	5448
2433	2860	5293
		Avg: 5288.9

As far as the second important benchmarking metric, occupied disk space, is concerned, it can be split into two parts related to disk space required for integrating the weather forecast service, whereas the second part refers to integrating the currency exchange rate service. Accordingly, the first two columns in Table 14 correspond to these two cases, whereas the third column contains the total aggregated results.

Table 14. Occupied disk space across all three developers.

Service Code Size between DeveloperA and DeveloperB, kB	**Service Code Size between DeveloperA and DeveloperC, kB**	**Total, kB**
33.03	32.718	65.748
33.89	33.473	67.363
34.14	33.755	67.895
38.13	37.861	75.991
43.61	43.201	86.811
47.62	47.213	94.833

48.29	47.921	96.211
52.33	51.99	104.32
57.71	57.318	115.028
60.18	59.825	120.005
		Avg: 89.42

5.4.2. Integration Time and Occupied Disk Space with GitHub/Jenkins

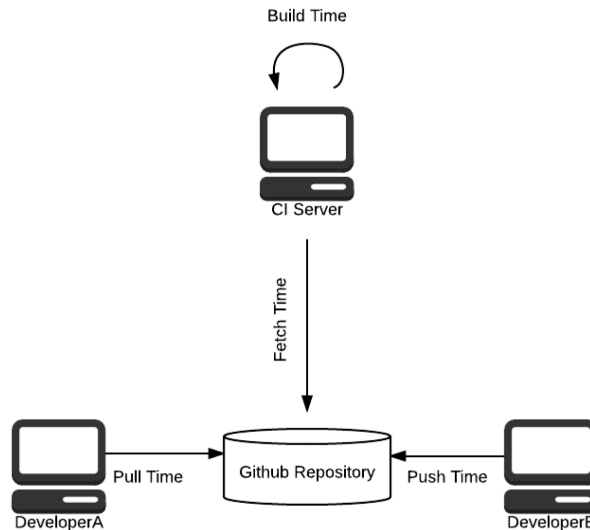

Figure 37. The GitHub/Jenkins setup.

The experiments were conducted in a series of 10 iterations, in which service changes were first pushed to the GitHub repository and then pulled, following the process diagram in Figure 37. As explained, the integration time with GitHub/Jenkins is composed of the three main steps that have to be first measured separately and then summed up – namely, push time, build time, and pull time. The **push time** is measured on the sides of DeveloperB and DeveloperC, as summarised by Table 15 and Table 16, respectively. The tables also include corresponding values for occupied disk space for each commit interation.

Table 15. Push time for DeveloperB in GitHub/Jenkins.

No. of Change	**DeveloperB – Committing the weather forecast service**	
	Commit Time, ms	**Service code size in Repo2, kB**
1	91	21.2
2	86	21.7
3	86	21.8
4	65	22
5	106	22.5
6	76	22.6
7	80	22.8
8	90	23
9	100	23.5
10	76	23.7
	Avg: 88.8	**Avg: 22.48**

Table 16. Push time for DeveloperC in GitHub/Jenkins.

No. of Change	**DeveloperC – Commiting the currency exchange rate service.**	
	Commit Time, ms	**Service code size in Repo3, kB**
1	87	22.5
2	93	23
3	89	23.1

4	71	23.3
5	103	23.8
6	88	23.9
7	97	24.3
8	90	24.8
9	101	25
10	83	25.4
	Avg: 90.2	**Avg: 23.91**

Next, the **build** step takes place on the Jenkins server, and the corresponding benchmarking results are summarised in Table 17 and Table 18.

Table 17. Build time for the weather forecast service in GitHub/Jenkins.

	Jenkins		**GitHub**	
No. of Change	**Fetch time, ms**	**Build time, ms**	**Repo1 (Web app code size), kB**	**Repo2 (Service code size), kB**
1	40	4794	3.75	21.2
2	27	4286	3.75	21.7
3	32	4873	3.75	21.8
4	30	5325	3.75	22
5	33	4781	3.75	22.5
6	31	4800	3.75	22.6
7	33	4898	3.75	22.8
8	34	5161	3.75	23
9	31	5042	3.75	23.5

10	30	4576	3.75	23.7
	Avg: 32.1	**Avg: 4853.6**	**Avg: 3.75**	**Avg: 22.48**

Table 18. Build time for the currency exchange rate service in GitHub/Jenkins

	Jenkins		**GitHub**	
No. of Change	**Fetch time, ms**	**Build time, ms**	**Repo1 (Web app code size), kB**	**Repo3 (Service code size), kB**
1	53	4777	5.48	22.5
2	36	4269	5.48	23
3	42	4856	5.48	23.1
4	40	5308	5.48	23.3
5	44	4764	5.48	23.8
6	42	4783	5.48	23.9
7	47	4881	5.48	24.3
8	39	5144	5.48	24.8
9	50	5025	5.48	25
10	37	4559	5.48	25.4
	Avg: 43	**Avg: 4836.6**	**Avg: 5.48**	**Avg: 23.91**

Finally, the code of the two Web services is **pulled** by DeveloperA. Corresponding benchmarking results are summarised in Table 19 and Table 20.

Table 19. Pull time for the weather forecast service in GitHub/Jenkins.

	DeveloperA pulls the weather forecast service

No. of Change	Update time, ms	Repo1 (Web app code Size), kB	Repo2 (Service code size), kB
1	2535	3.75	21.2
2	2385	3.75	21.7
3	2236	3.75	21.8
4	2499	3.75	22
5	2513	3.75	22.5
6	2299	3.75	22.6
7	2530	3.75	22.8
8	2526	3.75	23
9	2421	3.75	23.5
10	2359	3.75	23.7
	Avg: 2430.3	**Avg: 3.75**	**Avg: 22.48**

Table 20. Pull time for the currency exchange rate service in GitHub/Jenkins.

	DeveloperA pulls the currency exchange rate service		
No. of Change	**Update time, ms**	**Repo1 (Web app code Size), kB**	**Repo3 (Service code size), kB**
1	2692	5.48	22.5
2	2756	5.48	23
3	2521	5.48	23.1
4	2735	5.48	23.3
5	2677	5.48	23.8

6	2620	5.48	23.9
7	2884	5.48	24.3
8	2978	5.48	24.8
9	2812	5.48	25
10	2715	5.48	25.4
	Avg: 2739	**Avg: 5.48**	**Avg: 23.91**

From the above tables referring to push, build and pul benchmarks, it is possible to obtain the total **integration time** and the **disk space** occupied by the project for the GitHub/Jenkins setup. The overall results are summarised in Table 21 (for the weather forecast service) and Table 22 (for the currency exchange rate service), whereas

Table 21. Total integration time for the weather forecast service in GitHub/Jenkins.

No. of Change	Total integration time, ms	Total occupied disc space across the two involved developers, kB
1	7460	71.1
2	6784	72.6
3	7227	72.9
4	7919	73.5
5	7433	75
6	7206	75.3
7	7541	75.9
8	7811	76.5
9	7594	78

10	7041	78.6
	Avg: 7401.1	**Avg: 74.94**

Table 22. Total integration time for the currency exchange rate service in GitHub/Jenkins.

No. of Change	**Total integration time, ms**	**Total occupied disc space across the two involved developers, kB**
1	7613	99.66
2	7147	101.66
3	7505	102.06
4	8148	102.86
5	7591	104.86
6	7521	105.26
7	7892	106.66
8	8251	108.36
9	7987	109.46
10	7387	110.86
	Avg: 7704.2	**Avg: 105.17**

Table 23. Total integration time for the overall SBS system in GitHub/Jenkins.

No. of Change	**Total integration time, ms**	**Total occupied disc space across the involved developers, kB**
1	15073	191.96
2	13931	195.96
3	14732	196.76

4	16067	198.36
5	15024	202.36
6	14727	203.16
7	15433	205.36
8	16062	207.86
9	15581	210.96
10	14428	213.16
	Avg: 15105.8	**Avg: 202.59**

5.5. Discussion of the Results

The SORCER framework offers unique and novel functionality, not provided by currently adopted approaches. In this section we discuss its main benefits in terms of both functional and non-functional characteristics with respect to the currently adopted approaches.

5.5.1. Comparison of the experimental results

As demonstrated by the case study, using existing technologies, such as GitHub and Jenkins to support CI during service-oriented software development, a user needs to put all components in one place. The proposed SORC approach and the implemented SORCER system address this limitation. As indicated by the benchmarking results, the proposed approach is also faster in terms of both integration time and build time, as well as requires less storage space to support CI activities.

Figure 38 depicts the difference in integration time between the two approaches. These charts indicate that the proposed SORCER system provides better performance, being faster than the existing GitHub/Jenkins setup by 32%.

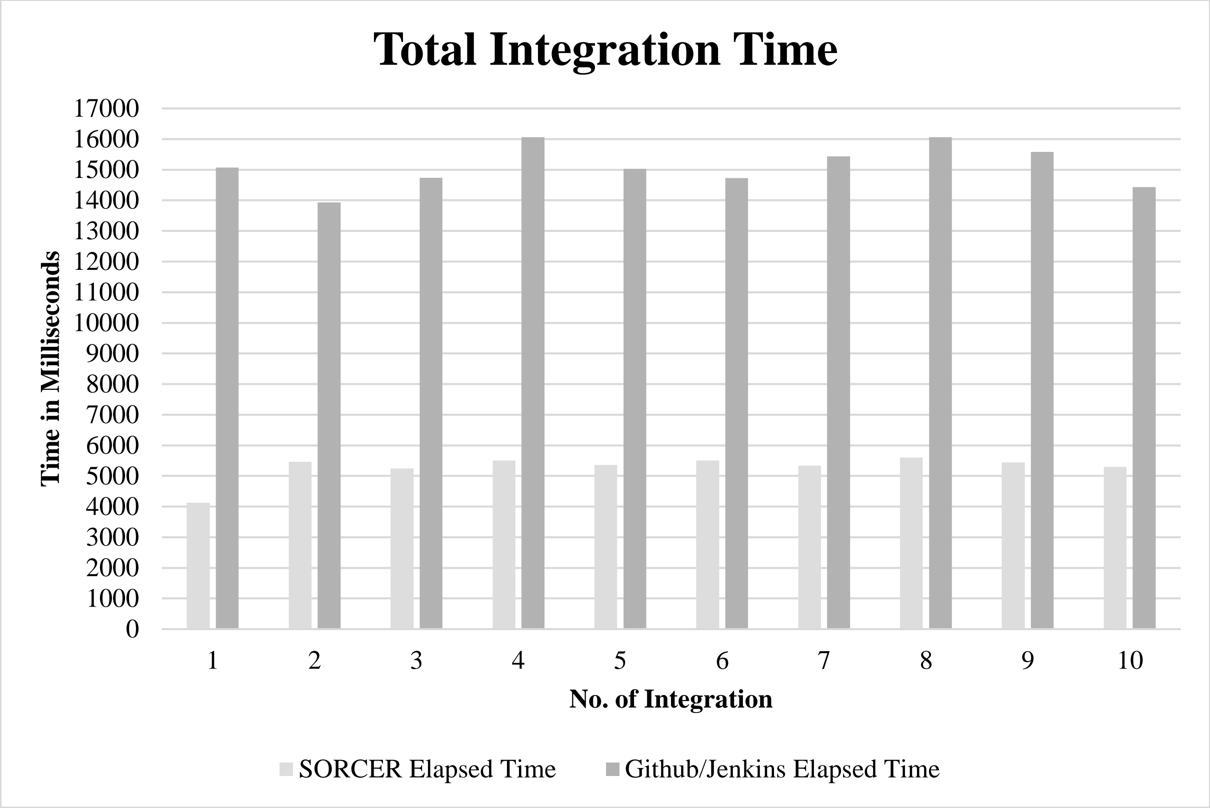

Figure 38. Integration time.

As far as the core phase of the integration process – i.e., project building – is concerned, there is also a clear indication of SORCER's advantages. As illustrated by Figure 39, the build time with SORCER is less than the one with Jenkins by 43%.

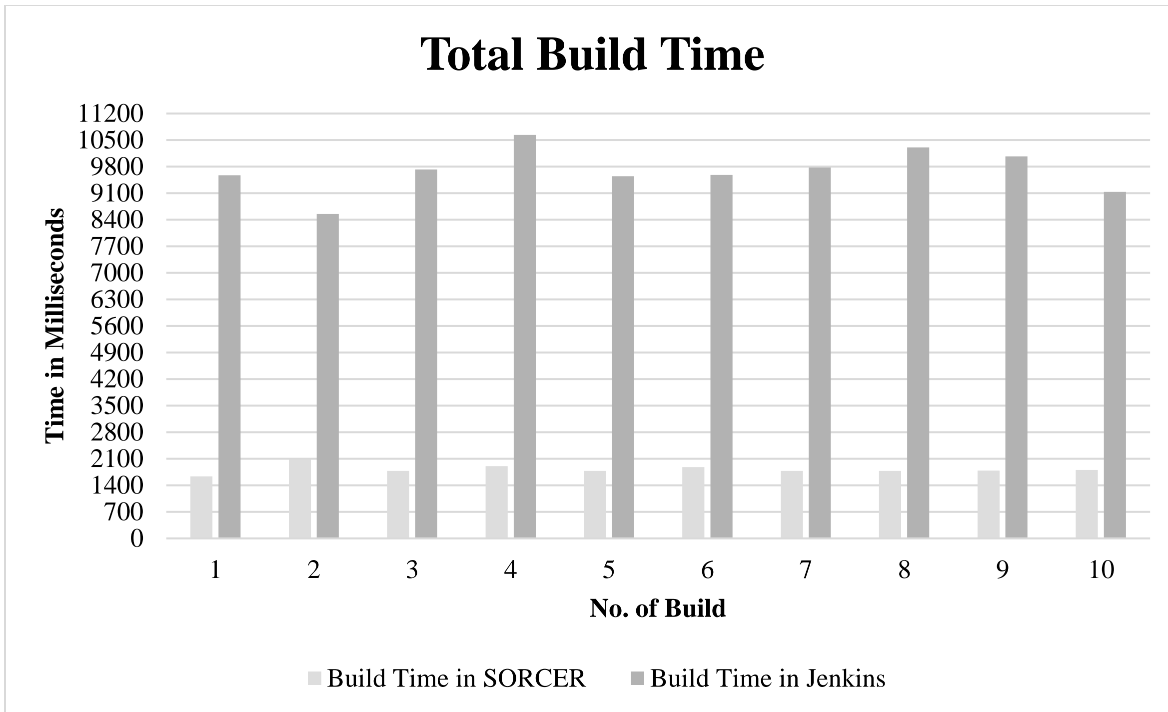

Figure 39. Build time.

Finally, as previously explained, in GitHub/Jenkins, the project code is stored in the main repository, and DeveloperA must have a copy of the whole project in his workspace (two Web services and the main web app), whereas DeveloperB and DeveloperC only maintain their corresponding Web services. As a result, the source code has to be replicated multiple times. In the SORCER system, each developer only has their own code without replicating it to the other developers. In our case study, DeveloperB has the weather Web service code, DeveloperC has the currency exchange rate service code, and DeveloperA has the client application code. Accordingly, the total disk space occupied by the project code in the two experiment setups is the sum of disk spaces, occupied by the SORCER team and the GitHub/Jenkins team, respectively. As demonstrated by the charts in Figure 40, SORCER is able to save up to 28% on average.

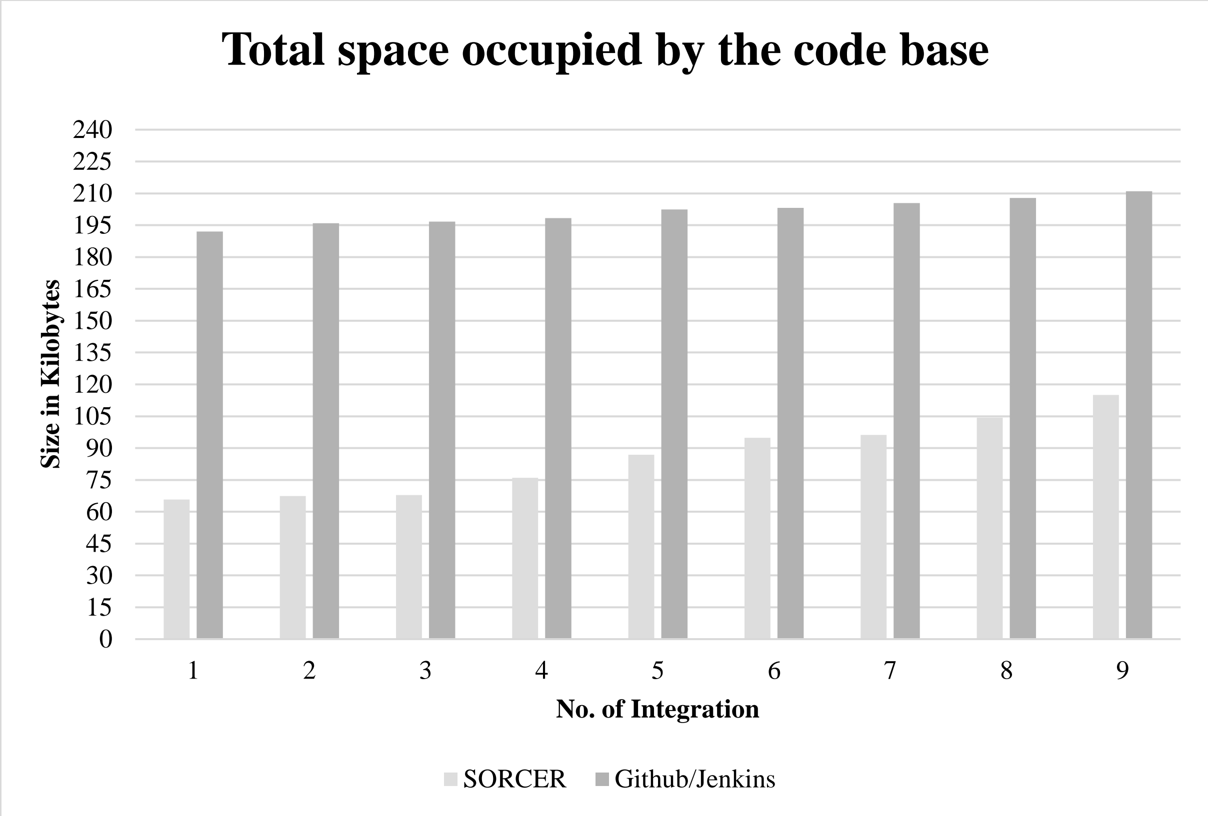

Figure 40. Total space occupied by the code base.

5.5.2. Discussing the benefits

The key differences between the proposed SORC approach implemented as the SORCER framework and the existing version control and CI solutions, such as GitHub and Jenkins adapted for the service-oriented software co-development, are summarised in Table 24**Error! Reference source not found.**, structured around the key comparison criteria (Sarib and Shen, 2014), previously outlined in Section 0. These differences are present in a less explicit form throughout the whole thesis, whereas here they are explicitly summarised in one place.

Table 24. Comparing SORC and the traditional version control systems.

Criteria	Traditional version control	SORCER
Performance and speed of update	Using the traditional systems, users are first	As demonstrated by the experiments, updates in

	expected to download the required project files (or at least changesets) from the repository. Next, it is required to execute some merging operation (Conradi and Westfechtel, 1998; Shen and Sun, 2004), after which, eventually, users can start actually working. Taken together, all these operations result in a very time-consuming update process.	SORCER are extremely fast, since users are only required to collect service descriptors from other nodes to generate proxy classes and start coding.
Storage space and network bandwidth	Traditional systems require that copies of the entire project files are continuously present on each of the developer's machine, and revision snapshots or changesets are exchanged between developer nodes (Shen and Sun, 2002).	As demonstrated by the experiments, SORCER needs minimal amount of storage space and network bandwidth during commits and updates, since SORCER users do not need to store the actual software artefacts, but rather exchange

		relatively light-weight service descriptors.
Mode of collaboration	Traditional systems are only suitable for asynchronous collaboration as updates are costly and are, therefore, not expected to be performed frequently (Shen and Sun, 2004, 2002).	Since SORCER's speed of update is much faster, it is capable of supporting synchronous collaborative programming though frequent periodical updates.
Access version metadata at run-time	Traditional systems utilise some specific tools to tag software version with meta-data, which typically cannot be queried or modified at run-time.	SORCER allows artefact attributes and meta-data to be embedded into software versions. This meta-data can be queried and modified at run-time, thus enabling dynamic reconfigurable applications that involve run-time switching versions of used services.
Higher level of abstraction and separation of concerns	The abstraction level provided by traditional systems highly depends on the project setup and the	Since a proxy class in SORCER does not contain the source code of the actual service, it provides

	practices, adopted by specific software development teams.	developers with a higher sense of *i)* abstraction for other developers' code, and *ii)* protection of their own code from unauthorised access and modification.
Isolated compile-time errors	Traditional systems need to update the entire project tree, which can potentially result in wrongly-committed source files containing compile-time errors copied to all developers' machines. This problem is especially challenging, since none of the users is able to compile the project, until the initial bug is fixed, corresponding changes are committed and all involved users are updated accordingly.	In contrast, SORCER avoids compile-time errors, since a service request will result in an error only when a malfunctioning service is invoked. This way, individual developers and their software components are isolated from each other and are not affected by faulty dependencies.
Need for integration testing	Traditional systems highly depend on multiple	SORCER does not depend on integration tests, since

	integration tests to ensure that individual components fit each other.	this kind of validation is already performed during the edit/build/test cycle.

Furthermore, the proposed approach also has the potential to address some pressing issues, such as broken builds, conflicts, and complexity of changes.

Handling broken builds

Another potentially valuable feature that can be supported by SORC is handling broken code. In version control and CI, broken code, in many cases, is unavoidable. There are best practices that can be applied in order to reduce this issue (Duvall et al., 2007). The ultimate mitigation of this risk can be achieved by having a well-factored build script that compiles and tests the code in a repeatable manner. By making it part of the team's adopted development practice, it is possible to always run a private or independent build before committing code to the version control repository. In this way, developers can commit their services and, provided that the development team is relatively small, the probability of another developer needing to commit while the build is broken is also relatively low. In these circumstances, it makes sense to limit development teams to 4-8 developers per team and to modularise the code base on a per team basis (Marchesi and Succi, 2003). With this kind of isolation, the build process is expected to be fast, and only a few developers from the same team will be affected in case the build is broken.

To support this feature, SORCER can automatically process the build to validate changes, every time a developer commits a new set of modifications. When the build is complete, the server notifies the developers of whether the committed changes were built successfully or not. If not, the commit process will not publish the service. Thus, it guarantees that no broken changes will be committed and therefore other team developers will not be affected.

Handling conflicts

The CI practice encourages developers to frequently commit their changes to the central repository at least on a daily basis. This approach is beneficial to avoid merge conflicts when two developers have been independently modifying the same code, as well as facilitating better communication on what each developer is currently working on (Betz and Walker, 2013). Typically, frequent commits minimise potential sources for conflict errors, making conflict fixing easier and faster (Fowler and Foemmel, 2006). In contrast, long delays between integration testing can potentially lead to code change conflicts that are difficult to locate and track (Laukkanen et al., 2017).

These requirements are perfectly in line with, and are supported by, SORC. In SORC, there is no strict limit on how often commits have to be made and, therefore, common established practices can be applied. Moreover, since compatibility assessment in SORC takes place on a regular basis (i.e., not only at the very end of the development process), the risk of conflicts is minimised (Smart, 2011). Furthermore, since there is no central repository and developers are not expected to work on other team members' components (i.e., external services), in most cases conflicts will be avoided by design.

Handling complex changes applied to external components

Complex changes – i.e., aggregate changes concerning multiple modifications applied to (multiple) source files – are known to be more error-prone (as opposed to atomic changes), and the complexity of changes has already been found to impact software quality (Hassan, 2009; Kerzazi et al., 2014). In these circumstances, ensuring CI becomes especially important. The challenge is becoming even more pressing when some external components, on which the target SBS system is dependent, are modified. Such system components are external to the project (e.g., databases, files, and third-party services) and are beyond control of the development team and,

therefore, modifications made to these components will not trigger the build/integration process. It is important to be able to manually run the integration process once changes to an external component have been applied (e.g., a new record added to the database). Since external components affecting the project can be fixed and modified independently from the common project code, it was important for SORC to support manual initiation of the integration process when/if required. This way, the proposed approach goes beyond the application scope of SOA and can be potentially applied to a wider range of software systems, involving not just services, but other external software components.

5.5.3. Discussing the shortcomings

When talking about limitations, we are mainly discussing shortcomings, i.e., features which are currently beyond our capabilities to be addressed. In contrast, improvements, which can be done but due to certain reasons (e.g., time constraints or being not directly relevant to the described PhD research topic) have not been addressed yet, are included in the concluding chapter as directions for future work.

Lack of real-world implementation/deployment and experiments

We described a case study, aiming to demonstrate how the proposed approach can be potentially applied and used. The case study, however, albeit based on a realistic scenario involving an SBS system, is primarily hypothetical. In these circumstances, the evaluation of the results is somewhat constrained with the assumptions of the described use case scenario, which in reality may not necessarily be the case (e.g. network congestion and latency, firewalls, hardware specification of the server machines). In other words, it may turn out that the simulated use case environment does not reflect the existing enterprise conditions, and, therefore, might not be suitable for validating

the approach and proving the proposed concepts. To address this limitation, a more in-depth and realistic experimentation is included as part of the future work.

Lack of support for integration tests at design-time

As described, the SORC approach relies on generated proxy classes for compatibility assessment during the integration process. This, however, means that there are no actual integration tests executed on different parts of the overall SBS system, and potential errors will only emerge at run-time, when a service consumer will invoke a remote service. Given that software developers tend not to follow the established practices of documenting their changes and versioning their revisions, a minor (non-breaking) change may actually appear to be major (i.e. breaking), which will only reveal itself at run-time.

Lack of visibility into peer developers' source code

Even though this contradicts with the very nature of SOA, sometimes it might be of benefit to look into peer developers' (i.e. belonging to the same organisation or development team) source code to understand the application logic behind, debug the system, and modify the system accordingly. With SORC, however, it is not possible, since software artefacts are not downloaded locally for integration, but rather proxy classes a generated from WSDL files. On contrary, when using GitHub and Jenkins, it is possible to download individual SBS system artefacts and inspect the source code.

5.6. Summary

This chapter presented a prototype implementation of the SORC architecture described in the previous chapter. This SORCER prototype was evaluated in comparison with an existing approach built upon technologies implementing similar functionality on a simple service-based application scenario. The scenario involved an SBS system, where committed changes to the weather forecast

and the currency exchange rate services required to be dynamically picked up to be integrated into the main client application. The primary metrics to evaluate SORCER and compare it to the GitHub/Jenkins setup were integration time, build time, and the disk space occupied by the software project artefacts. As indicated by the benchmarking results, the proposed system is faster in terms of both integration time and build time, as well requiring less storage space to support CI activities. Furthermore, as far as functional properties are concerned, the proposed SORC approach has the potential to handle broken builds, conflicts, and the complexity of changes concerning external components.

6. Chapter 6: Conclusion

This last chapter serves to conclude the whole thesis by summarising the key research findings and contributions. This chapter revisits the introduction and discusses whether the initially outlined research question and hypothesis, as well as the theoretical, technical and experimental objectives have been addressed. Finally, the chapter proposes several potential directions for further extending the presented research work.

6.1. Thesis Overview

The presented research explored the pressing challenge of enabling CI in service-based distributed software systems in the absence of a centralised location to develop, test, deploy, and run an SBS system and/or its individual services. In centralised software architectures, CI serves to validate individual modifications, as well as the overall integration of separate committed components in a common global project. CI reduces risks and identifies issues early in the software development lifecycle by implementing automated software building and testing. Typically, once changes have been committed to a shared repository, the CI server automatically builds and runs unit tests on the updated code to immediately spot any functional or integration issues.

This situation, however, has changed with the emergence and rapid adoption of SOAs. As SBS systems grow in size and complexity, it becomes increasingly important to continuously maintain service compatibility, ensuring that individual services, when integrated into a common project, operate in a stable and reliable manner. With SOA, however, traditional CI is hindered by the distributed nature of SBS systems. As individual services are owned and managed by different stakeholders that do not necessarily provide service source code, testing and building the overall SBS system in one place appears to be infeasible. Given this limitation, a decentralised approach to CI is required in the context of API-driven SOA environments. As traditional approaches to

designing and implementing centralised software architectures are of marginal benefit to the loosely-coupled and dynamic nature of SBS compositions, it is important to address this limitation with a solution that enables service consumers to keep up to date with constantly changing and evolving collection of service revisions, so as to ensure stable and uninterrupted operation of a service-based system. This way, CI for decentralised development of SBS systems – the key focus of this thesis – can be achieved.

Accordingly, the main research question to be addressed by the presented research effort was the following – **How to enable continuous integration in service-based distributed software systems in the absence of a central server.** To address this question, this research introduced SORC as a way of facilitating CI through peer-to-peer communication between service providers and consumers, underpinned by the novel versioning and compatibility checking techniques, in order to enable discovery and consumption of service versions, even in the presence of frequent service updates and multiple revisions. The proposed approach was validated and evaluated with respect to an existing SCM system, thereby demonstrating the overall viability and effectiveness of the proposed concepts and ideas. As a result, it can be concluded that the main research question and the hypothesis have been successfully resolved. Furthermore, all of the theoretical, technical, and experimental objectives established at the beginning of this thesis have been fulfilled. The obtained research results contribute to the domains of Service-Oriented Software Engineering, CI for SOA, as well as to the disciplines of software versioning and compatibility checking. The contributions are discussed in more detail below.

6.2. Discussing Contributions

SORC introduces a novel approach to enable continuous support for software testing and integration in distributed loosely-coupled environments in the absence of a central location for

developing, storing, testing, deploying and executing software components – functionality that is currently beyond the capabilities of existing traditional SCM systems. Furthermore, the presented research effort also contributes to several adjacent research areas, including software version control and compatibility assessment. More specifically, the following key aspects constitute the overall contribution of the thesis:

- **Literature survey of the state of the art in service compatibility, service change detection, and service versioning:** Chapter 3 of this document provided an extensive literature survey (fulfilling one of the theoretical objectives). The survey identified exiting technological limitations and research gaps in the domains of CI, Software Versioning, and Compatibility Checking. More specifically, it was revealed that the existing solutions and approaches seem to provide little support for SOA software development. By including the literature survey in this thesis, we intend to raise the awareness of the interested research community, which is expected to be attracted to collaboratively advance the state of the art.
- **Functional requirements specification for CI in SOA:** the literature survey revealed existing gaps in the state of the art. These gaps, served to distil high-level functional requirements for a future CI system that would fit the SOA domain. As a result, these outlined functional requirements were seen as a primary reference underpinning the design and implementation of the SORC system. This functional requirement specification contributes to the state of the art in enabling CI for SOAs, as it is expected to be re-used by the wider research community. That is, anyone willing to engineer their own solutions to enable CI based on the proposed specification can potentially get inspired from the presented findings.

- **Design of the SORC system**: based on the distilled functional requirement specification, we have designed a conceptual architecture of the proposed SORC system. This high-level design can serve as a reference model for a wider research community looking to create their own solutions. That is, it is possible to implement a similar SORC system using different programming languages and frameworks, but still following the same architectural design.
- **Prototype implementation of the SORCER framework**: the existing proof-of-concept implementation demonstrates the viability of the proposed SORC approach and is available for download and experimentation as an open-source software product. Using SORCER, software developers are expected benefit from the CI facilities , even when developing a distributed loosely-coupled SOA software system. We are also expecting a wider community of researchers and practitioners to contribute to the project by extending the existing code base in the future.

6.3. Future Work

This section of this chapter serves to provide the reader with an outline of future research steps that have the potential to further improve the presented SORC research, address existing limitations, and enhance the benefits.

- **Adding support for RESTful services**: the main focus of the presented research was on WSDL/SOAP-based Web services. This concerns both the conceptual architecture of SORC and the implemented SORCER framework. As described, RESTful architectures represent a widely used approach to implement loosely-coupled SBS compositions. Limiting our work solely to WSDL/SOAP-based services significantly reduces its potential

application scope. RESTful services may also use WSDL descriptions, thereby making the proposed SORC approach potentially suitable.

- **More detailed evaluation with respect to multiple criteria, including usability**: Chapter 5 presented and evaluated a case study, aiming to demonstrate how the proposed approach can be applied and used, as well as how it can outperform existing solutions. The case study, however, is primarily hypothetical and has not been tested in a real working environment, i.e., it has not been evaluated by professional software developers in their everyday work. From this perspective, the evaluation of the results is somewhat limited and can be further extended. More specifically, a potentially promising research direction is to evaluate the SORCER system with respect to its usability. That is, we position our work as a competitor to the existing SCM solutions, such as Git and Mercurial, and, therefore, have to evaluate all possible aspects, not just functionality and performance. Accordingly, working in this direction will involve both implicit (e.g., monitoring and observing the behaviour and emotions) and explicit (e.g., questionnaires, interviews, etc.) evaluation techniques with a goal to see how easy and straight-forward it is for SOA developers to use the SORCER framework, and how it can be further improved.
- **Introducing support for Continuous Delivery:** apart from CI, which remains the main focus of this research, it is also important to consider the challenges associated with Continuous Delivery in the context of service-oriented software development. Continuous Delivery is typically seen as the next step after CI and, therefore, the results obtained so far can provide a foundation for further extending this work. More specifically, using the proposed compatibility assessment mechanism (and provided that the integration was

successful), it is possible to enable automated delivery of fully-built and integrated SBS to end users without manual intervention of software developers and administrators.

6.4. Researcher's view

As a final conclusion to this thesis, we are providing the author's personal view and impressions from the conducted research, obtained results, and further opportunities for extending this work.

The PhD research turned out to be a challenging and exciting experience for the author, full of its 'ups and downs' and involving numerous scientific discussions, arguments, and hypotheses. Despite all the moments of disappointment, anger, and despair, it still turned out to be worthy decision to go down this research path and addressing an interesting and challenging research gap. It should be noted that the research challenges raised and tackled in this work are not hypothetical or theoretical. On contrary, they reflect the author's own experience in professional software development, and – more specifically – in developing service-oriented systems working in a geographically distributed team. As the number of updates to some remote services was growing, it became evident that the existing tools, such as GitHub, SVN, Jenkins, etc. were unable to support timely automated integration of the frequently updated components into a common service-based project.

From this perspective, the research was primarily motivated and driven by the issues personally faced by the author, thus guaranteeing that the initially outlined goals would be eventually accomplished. It is the author's belief that the obtained results, albeit not completely mature yet, have the potential to be implemented as a software product (e.g., as a plugin to existing IDEs such as Visual Studio or Eclipse) either as an open-source or commercial project. In both cases, it can assist software developers in engineering and developing distributed software systems in the

increasingly service-oriented IT world. Furthermore, as a next step towards his vision, the author is planning to be still involved in extending the project after completing this PhD research.

Acronyms

API	Application Programming Interface
ASR	Architecturally Significant Requirements
BPMN	Business Process Model and Notation
CBSE	Component-Based Software Engineering
CI	Continuous Integration
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CVS	Concurrent Versions System
DevOps	Development + Operations
EJB	Enterprise Java Bean
HTTP	Hypertext Transfer Protocol
ICT	Information and Communication Technologies
IIS	Internet Information Services
IoS	Internet of Services
IP	Internet Protocol
IT	Information Technology
J2EE	Java 2 Platform, Enterprise Edition
LS-SVM	Least Squares Support Vector Machines
OOP	Object-Oriented Programming
PC	Personal Computer
QoS	Quality of Service
RCS	Revision Control System

RegEx	Regular Expression
RPC	Remote Procedure Call
SBS	Service-Based Software
SCCS	Source Code Control System
SCM	Software Configuration Management
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
SORC	Service-Oriented Revision Control
SOSE	Service-Oriented Software Engineering
SPOF	Single Point of Failure
SVN	Subversion
UCM	Unified Change Management
UDDI	Universal Description Discovery and Integration
URL	Uniform Resource Locator
VCS	Version Control System
WCF	Windows Communication Foundation
WS	Web Service
WSDL	Web Service Description Language
W3C	World Wide Web Community
XML	Extensible Mark-up Language

References

Abang Ibrahim, D.H., 2016. The exploitation of provenance and versioning in the reproduction of e-experiments.

Andrikopoulos, V., 2010. A theory and model for the evolution of software services. Tilburg University, School of Economics and Management.

Andrikopoulos, V., Benbernou, S., Papazoglou, M.P., 2012. On the evolution of services. IEEE Trans. Softw. Eng. 38, 609–628.

Bachmann, R., 2015. Challenges of Web Service Change Management [WWW Document]. URL https://archive.sap.com/documents/docs/DOC-1321 (accessed 6.4.17).

Bass, L., Weber, I., Zhu, L., 2015. DevOps: A Software Architect's Perspective. Addison-Wesley Professional.

Baushke, M.D., 2015. CVS - Open Source Version Control [WWW Document]. URL http://cvs.nongnu.org/ (accessed 7.25.17).

Bechara, G., 2015. Web Services Versioning [WWW Document]. URL http://www.oracle.com/technetwork/articles/web-services-versioning-094384.html (accessed 10.22.17).

Bechmann, A., Lomborg, S., 2014. The Ubiquitous Internet: User and Industry Perspectives, Routledge Studies in New Media and Cyberculture. Routledge.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., others, 2001. Manifesto for agile software development.

Becker, K., Pruyne, J., Singhal, S., Lopes, A., Milojicic, D., 2011. Automatic determination of compatibility in evolving services. Int. J. Web Serv. Res. IJWSR 8, 21–40.

Bellagio, D., Milligan, T., 2005. Software configuration management strategies and IBM® Rational® Clearcase®: a practical introduction. IBM Press.

Berliner, B., others, 1990. CVS II: Parallelizing software development, in: Proceedings of the USENIX Winter 1990 Technical Conference. p. 352.

Betz, R.M., Walker, R.C., 2013. Implementing continuous integration software in an established computational chemistry software package, in: Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on. IEEE, pp. 68–74.

Bharti, N., 2008. Web Service Contract Versioning Fundamentals Part I: Versioning and Compatibility [WWW Document]. URL https://dzone.com/articles/web-service-contract-versionin (accessed 6.10.17).

Bhattacharya, S., Kanjilal, A., Sengupta, S., Chanda, J., Majumdar, D., 2017. Requirements to Services: A Model to Automate Service Discovery and Dynamic Choreography from Service Version Database, in: Requirements Engineering for Service and Cloud Computing. Springer, pp. 151–179.

Boehm, B.W., 1987. Improving software productivity, in: Computer. Citeseer.

Borovskiy, V., Zeier, A., Karstens, J., Roggenkemper, H.U., 2008. Resolving Incompatibility During the Evolution of Web Services With Message Conversion., in: ICSOFT (SE/MUSE/GSDCA). pp. 152–158.

Breivold, H.P., Larsson, M., 2007. Component-based and service-oriented software engineering: Key concepts and principles, in: Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on. IEEE, pp. 13–20.

Brown, K., Ellis, M., 2004. Best practices for Web services versioning. See Httpwww-106 Ibm Comdeveloperworkswebserviceslibraryws-Version.

Buxmann, P.D.P., Hess, P.D.T., Ruggaber, D.R., 2009. Internet of Services. Bus. Inf. Syst. Eng. 1, 341–342. https://doi.org/10.1007/s12599-009-0066-z

Buyya, R., Vecchiola, C., Selvi, S.T., 2013. Mastering Cloud Computing: Foundations and Applications Programming, 1 edition. ed. Morgan Kaufmann, Waltham, MA.

Chacon, S., Straub, B., 2014. Pro Git, 2nd ed. Apress.

Chauhan, M.A., Probst, C.W., 2017. Architecturally Significant Requirements Identification, Classification and Change Management for Multi-tenant Cloud-Based Systems, in: Requirements Engineering for Service and Cloud Computing. Springer, pp. 181–205.

Chinnici, R., Moreau, J.-J., Ryman, A., Weerawarana, S., 2007. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language [WWW Document]. URL https://www.w3.org/TR/wsdl20/ (accessed 6.11.17).

Christensen, E., Meredith, G., Curbera, F., Weerawarana, S., 2001. Web Services Description Language (WSDL). World Wide Web (W3C).

Collins-Sussman, B., Fitzpatrick, B., Pilato, M., 2004. Version control with subversion. O'Reilly Media, Inc.

Conradi, R., Westfechtel, B., 1998. Version models for software configuration management. ACM Comput. Surv. CSUR 30, 232–282.

Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J., 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository, in: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, CSCW '12. ACM, New York, NY, USA, pp. 1277–1286. https://doi.org/10.1145/2145204.2145396

Dart, S., 1991. Concepts in configuration management systems, in: Proceedings of the 3rd International Workshop on Software Configuration Management. ACM, pp. 1–18.

Dautov, R. and Paraskakis, I., 2013. A vision for monitoring cloud application platforms as sensor networks, in: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference. ACM.

Dautov, R., Paraskakis, I. and Stannett, M., 2014. Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms. Journal of Cloud Computing, 3(1), p.13.

Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl, K., 2008. A journey to highly dynamic, self-adaptive service-based applications. Autom. Softw. Eng. 15, 313–341.

Dooley, K., 2001. Designing Large Scale Lans: Help for Network Designers. O'Reilly Media, Inc.

Duvall, P.M., Matyas, S., Glover, A., 2007. Continuous integration: improving software quality and reducing risk. Pearson Education.

Erl, T., 2005. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Erl, T., Bennett, S.G., Carlyle, B., Gee, C., Laird, R., Manes, A.T., Moores, R., Schneider, R., Shuster, L., Tost, A., 2011. SOA Governance: Governing Shared Services On-Premise and in the Cloud. Prentice Hall Press.

Erl, T., Karmarkar, A., Walmsley, P., Haas, H., Yalcinalp, L.U., Liu, K., Orchard, D., Tost, A., Pasley, J., 2009. Web service contract design and versioning for SOA. Prentice Hall.

Estublier, J., Casallas, R., 1994. The Adele configuration manager. Config. Manag. 2, 99–134.

Evdemon, J., 2005. Principles of service design: Service versioning. Microsoft Corp. Aug.

Fang, R., Lam, L., Fong, L., Frank, D., Vignola, C., Chen, Y., Du, N., 2007. A version-aware approach for web service directory, in: Web Services, 2007. ICWS 2007. IEEE International Conference on. IEEE, pp. 406–413.

FEDICT, 2014. e-gov Architecture Service Interface Guidelines. Directorate General Digital Transformation.

Fischer, M., Pinzger, M., Gall, H., 2003. Populating a release history database from version control and bug tracking systems, in: Proceedings of IEEE International Conference on Software Maintenance (ICSM 2003). IEEE, pp. 23–32.

Fokaefs, M., Stroulia, E., 2012. Wsdarwin: Automatic web service client adaptation, in: Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp., pp. 176–191.

Fowler, M., Foemmel, M., 2006. Continuous integration. Thought-Works Httpwww Thoughtworks ComContinuous Integr. Pdf 122.

Frank, D., Lam, L., Fong, L., Fang, R., Khangaonkar, M., 2008. Using an interface proxy to host versioned web services, in: Services Computing, 2008. SCC'08. IEEE International Conference on. IEEE, pp. 325–332.

Green, R., 2008. Versioning Strategies [WWW Document]. URL https://msdn.microsoft.com/en-au/library/ff384251.aspx (accessed 10.22.17).

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H.F., Karmarkar, A., Lafon, Y., 2003. SOAP Version 1.2. W3C Recomm. 24.

Guimarães, M.L., Silva, A.R., 2012. Improving early detection of software merge conflicts, in: Software Engineering (ICSE), 2012 34th International Conference on. IEEE, pp. 342–352.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes, in: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, pp. 78–88.

He, W. and Da Xu, L., 2014. Integration of distributed enterprise applications: A survey. IEEE Transactions on Industrial Informatics, 10(1), pp.35-42.

Heineman, G., Council, W., 2001a. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley.

Heineman, G., Council, W., 2001b. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley.

Hogg, K., Chilcott, P., Nolan, M., Srinivasan, B., 2004. An evaluation of Web services in the design of a B2B application, in: Proceedings of the 27th Australasian Conference on Computer Science-Volume 26. Australian Computer Society, Inc., pp. 331–340.

Hohpe, G., 2005. Developing Software in a Service-Oriented World. (Whitepaper). ThoughtWorks, Inc.

Humble, J., Farley, D., 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader). Pearson Education.

Jenkins, 2017. Jenkins [WWW Document]. Jenkins. URL https://jenkins.io/index.html (accessed 12.18.17).

Jifeng, H., Li, X., Liu, Z., 2005. Component-based software engineering. Lect. Notes Comput. Sci. 3722, 70.

Jones, M., 2017. What Is Windows Communication Foundation [WWW Document]. URL https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf (accessed 12.18.17).

K. Schaefer, J. Cochran, S. Forsyth, D. Glendenning, B. Perkins, 2012. Professional Microsoft IIS 8. Wiley / Wrox, Hoboken, N.J.

Kaminski, P., Litoiu, M., Müller, H., 2006. A design technique for evolving web services, in: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp., p. 23.

Karhunen, H., Jantti, M., Eerola, A., 2005. Service-oriented software engineering (SOSE) framework, in: Proceedings of ICSSSM ’05. 2005 International Conference on Services Systems and Services Management, 2005. Presented at the Proceedings of ICSSSM ’05. 2005 International Conference on Services Systems and Services Management, 2005., p. 1199–1204 Vol. 2. https://doi.org/10.1109/ICSSSM.2005.1500187

Kelaskar, M., Matossian, V., Mehra, P., Paul, D., Parashar, M., 2002. A study of discovery mechanisms for peer-to-peer applications, in: Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on. IEEE, pp. 444–444.

Kerzazi, N., Khomh, F., Adams, B., 2014. Why do automated builds break? an empirical study, in: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on. IEEE, pp. 41–50.

Koc, A., Tansel, A.U., 2011. A survey of version control systems. ICEME 2011.

Kohar, R., Parimala, N., 2017. A metrics framework for measuring quality of a web service as it evolves. Int. J. Syst. Assur. Eng. Manag. 1–15.

Kozaczynski, W., Booch, G., 1998. Component-based software engineering. IEEE Softw. 15, 34.

Krafzig, D., Banke, K., Slama, D., 2005. Enterprise SOA: service-oriented architecture best practices. Prentice Hall Professional.

Kumar, L., Rath, S.K., Sureka, A., 2017. Using source code metrics to predict change-prone web services: A case-study on ebay services, in: Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Workshop on. IEEE, pp. 1–7.

Laukkanen, E., Itkonen, J., Lassenius, C., 2017. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. Inf. Softw. Technol. 82, 55–79.

Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S., 2008. End-to-end versioning support for web services, in: Services Computing, 2008. SCC'08. IEEE International Conference on. IEEE, pp. 59–66.

Liang, Q., Huhns, M., 2008. Ontology-based compatibility checking for web service configuration management. Serv.-Oriented Comput. 2008 407–421.

Loeliger, J., McCullough, M., 2012. Version Control with Git: Powerful tools and techniques for collaborative software development.

Lublinsky, B., 2007. Versioning in SOA. Microsoft Archit. J. 11.

MacKenzie, C.M., Laskey, K., McCabe, F., Brown, P.F., 2006. Reference model for service oriented architecture 1.0. OASIS Open.

Marchesi, M., Succi, G., 2003. Extreme programming and agile processes in software engineering. Proc. XP.

Microsoft, 2017. Home : The Official Microsoft IIS Site [WWW Document]. URL https://www.iis.net/ (accessed 12.18.17).

Microsoft Technet, 2014. Publish from TFS to Windows Azure Pack: Web Sites [WWW Document]. URL https://technet.microsoft.com/en-us/library/dn499801.aspx (accessed 9.18.17).

Mukherjee, P., 2011. A fully decentralized, peer-to-peer based version control system. Technische Universität.

Mulligan, G., Gračanin, D., 2009. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework, in: Winter Simulation Conference. Winter Simulation Conference, pp. 1423–1432.

Narayan, A., Singh, I., 2007. Designing and versioning compatible web services. IBM Tech Rep.

Nelson, B.J., 1981. Remote procedure call.

Novakouski, M., Lewis, G., Anderson, W., 2012. Best practices for artifact versioning in service-oriented systems. DTIC Document.

Orchard, D., 2007. Extending and Versioning Languages: XML Languages. World Wide Web Consort. W3C Httpwww W3 Org2001tagdocversioning-Xml.

O'Sullivan, B., 2009a. Making sense of revision-control systems. Commun. ACM 52, 56–62.

O'Sullivan, B., 2009b. Mercurial: The Definitive Guide: The Definitive Guide. O'Reilly Media, Inc.

Otte, R., Patrick, P., Roy, M., 1996. Understanding CORBA: Common Object Request Broker Architecture. Prentice Hall PTR.

Papazoglou, M., 2008. The challenges of service evolution, in: Advanced Information Systems Engineering. Springer, pp. 1–15.

Papazoglou, M.P., 2003. Service-oriented computing: Concepts, characteristics and directions, in: Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on. IEEE, pp. 3–12.

Papazoglou, M.P., Andrikopoulos, V., Benbernou, S., 2011. Managing evolving services. IEEE Softw. 28, 49–55.

Papazoglou, M.P., Dubray, J., 2004. A survey of web service technologies. University of Trento.

Parachuri, D., Mallick, S., 2007. Service Versioning in SOA, Part I: Issue in and approaches to Versioning. Infosys Tech Rep.

Pautasso, C., Zimmermann, O., Leymann, F., 2008. Restful web services vs. big'web services: making the right architectural decision, in: Proceedings of the 17th International Conference on World Wide Web. ACM, pp. 805–814.

Peltz, C., 2003. Web services orchestration and choreography. Computer 36, 46–52. https://doi.org/10.1109/MC.2003.1236471

Peltz, C., Anagol-Subarrao, A., 2004. Design Strategies for Web Services Versioning [WWW Document]. URL http://soa.sys-con.com/node/44356 (accessed 6.4.17).

Pilato, M., Collins-Sussman, B., Fitzpatrick, B., 2008. Version control with subversion. O'Reilly Media, Inc.

Pressman, R.S., 2005. Software engineering: a practitioner's approach. Palgrave Macmillan.

Raemaekers, S., van Deursen, A., Visser, J., 2017. Semantic versioning and impact of breaking changes in the Maven repository. J. Syst. Softw. 129, 140–158.

Raygan, R.E., 2007. Software configuration management applied to service oriented architecture, in: Software, Telecommunications and Computer Networks, 2007. SoftCOM 2007. 15th International Conference on. IEEE, pp. 1–5.

Richards, M., 2015. Microservices vs. service-oriented architecture. O'Reilly Media.

Rochkind, M.J., 1975. The source code control system. IEEE Trans. Softw. Eng. 364–370.

Roundy, D., 2005. Darcs: distributed version management in haskell, in: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell. ACM, pp. 1–4.

Rumbaugh, J., Eddy, F., Lorensen, W., Blaha, M., Premerlani, W., 1991. Object-oriented modeling and design [WWW Document]. CERN Doc. Serv. URL http://cds.cern.ch/record/237250 (accessed 2.6.15).

Sarib, A.S.B., Shen, H., 2014. SORC: Service-oriented distributed revision control for collaborative web programming, in: Computer Supported Cooperative Work in Design (CSCWD), Proceedings of the 2014 IEEE 18th International Conference on. IEEE, pp. 638–643.

Schaefer, A., Reichenbach, M., Fey, D., 2013. Continuous integration and automation for DevOps, in: IAENG Transactions on Engineering Technologies. Springer, pp. 345–358.

Schroth, C., Janner, T., 2007. Web 2.0 and SOA: Converging Concepts Enabling the Internet of Services. IT Prof. 9, 36–41. https://doi.org/10.1109/MITP.2007.60

Shahin, M., Babar, M.A., Zhu, L., 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access 5, 3909–3943.

Sharma, S., Coyne, B., 2013. DevOps for dummies. Ltd. IBM Ed.

Shen, H., Sun, C., 2004. A complete textual merging algorithm for software configuration management systems, in: Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International. IEEE, pp. 293–298.

Shen, H., Sun, C., 2002. A log compression algorithm for operation-based version control systems, in: Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International. IEEE, pp. 867–872.

Sleeper, B., 2002. The evolution of uddi. UDDI Org White Pap. Stencil Group Inc 1–15.

Smart, J.F., 2011. Jenkins: The Definitive Guide: Continuous Integration for the Masses. O'Reilly Media, Inc.

Sohan, S.M., Anslow, C., Maurer, F., 2015. A case study of web API evolution, in: Services (SERVICES), 2015 IEEE World Congress on. IEEE, pp. 245–252.

Sommerville, I., 2010. Software Engineering, 9 edition. ed. Pearson, Boston.

Sprott, D., Wilkes, L., 2004. Understanding Service-Oriented Architecture [WWW Document]. URL https://msdn.microsoft.com/en-us/library/aa480021.aspx (accessed 12.5.14).

Stojanović, Z., Dahanayake, A., 2005. Service-oriented software system engineering: challenges and practices. IGI Global.

Swicegood, T., 2008. Pragmatic version control using Git. Pragmatic Bookshelf.

The Open Group, 2009. SOA Source Book. Van Haren Publishing, Zaltbommel.

Tichy, W.F., 1985. RCS—a system for version control. Softw. Pract. Exp. 15, 637–654.

Tsai, W.T., 2005. Service-oriented system engineering: a new paradigm, in: Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop. Presented at the Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop, pp. 3–6. https://doi.org/10.1109/SOSE.2005.34

Vesperman, J., 2006. Essential CVS: Version Control and Source Code Management. O'Reilly Media, Inc.

Virmani, M., 2015. Understanding DevOps & bridging the gap from continuous integration to continuous delivery, in: Innovative Computing Technology (INTECH), 2015 Fifth International Conference on. IEEE, pp. 78–82.

Wagh, K., Thool, R., 2012. A comparative study of soap vs rest web services provisioning techniques for mobile host. J. Inf. Eng. Appl. 2, 12–16.

Wagner, B., 2015. Introduction to the C# Language and the .NET Framework [WWW Document]. URL https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework (accessed 12.18.17).

Waller, J., Ehmke, N.C., Hasselbring, W., 2015. Including performance benchmarks into continuous integration to enable DevOps. ACM SIGSOFT Softw. Eng. Notes 40, 1–4.

Wei, Y., Blake, M.B., 2010. Service-Oriented Computing and Cloud Computing: Challenges and Opportunities. IEEE Internet Comput. 14, 72–75. https://doi.org/10.1109/MIC.2010.147

Weinreich, R., Ziebermayr, T., Draheim, D., 2007. A versioning model for enterprise services, in: Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on. IEEE, pp. 570–575.

Wingerd, L., 2005. Practical perforce. O'Reilly Media, Inc.

Wood, C., Mersch, V. van der, Sandoval, K., 2016. API-Driven DevOps: Strategies for Continuous Deployment. Nordic APIs AB.

Yamashita, M., Becker, K., Galante, R., 2012. A feature-based versioning approach for assessing service compatibility. J. Inf. Data Manag. 3, 120.

Yeates, S., 2013. What is version control? Why is it important for due diligence? [WWW Document]. URL http://oss-watch.ac.uk/resources/versioncontrol (accessed 7.25.17).

Yau, S.S., Ye, N., Sarjoughian, H.S., Huang, D., Roontiva, A., Baydogan, M. and Muqsith, M.A., 2009. Toward development of adaptive service-based software systems. IEEE Transactions on Services Computing, 2(3), pp.247-260.

Yip, A., Chen, B., Morris, R., 2006. Pastwatch: A Distributed Version Control System., in: NSDI.

Yu, Q., Liu, X., Bouguettaya, A., Medjahed, B., 2008. Deploying and managing Web services: issues, solutions, and directions. VLDB Journal— Int. J. Very Large Data Bases 17, 537–572.

Appendix A: Summary of Related Works

Table 25. Summary of the state of the art in service versioning and compatibility.

Approach	Scope	Key contributions
Leitner et al. (Leitner et al., 2008)	Service change detection	• Automated change detection & notification mechanism • Using proxies to dynamically invoke different service versions, service consumers are able navigate through the version graph
Peltz and Anagol-Subbarao (Peltz and Anagol-Subarrao, 2004)	Service change detection	• Automated service change notification mechanism • The Web Service Facade design pattern is used to manage the complexity of multiple simultaneous versions • The level of coupling between the consumer and the service is minimised • A control point for manageability is created • Improved overall performance by reducing the number of network calls
Fang et al. (Fang et al., 2007)	Service change detection	• Extends UDDI to support automated change notifications via the publish/subscribe model • Thanks to the decoupled communication, service providers and consumers are not required to know about each other

Abang (Abang Ibrahim, 2016)	Service change detection	• Focus on provenance and versioning in services to enable coherent and deterministic reproduction of e-experiments • A new Web services versioning method, including temporal dependencies between different version via the "wasVersionOf" causal dependency • Key theoretical findings in a Reproducibility taxonomy
Kohar and Parimala (Kohar and Parimala, 2017)	Service change detection	• A three-dimensional metrics suite was proposed to measure service evolution and WSDL changes: i) a service evolution metric, ii) a service client-code evolution metric, and iii) a service usefulness evolution metric. • Main benefits: simplicity to compute and collect, technological independence, and minimum managerial burdens are introduced to the service provider • The proposed metrics suite is validated against real-world and simulated experiments
Chauhan et al. (Chauhan and Probst, 2017)	Service change detection	• Novel concept of Architecturally Significant Requirements (ASRs) – i.e., non-functional requirements that can have a significant impact on architecture of a software system.

		• A framework for requirements classification and change management, primarily focusing on distributed Platform-as-a-Service and Software-as-a-Service systems proposed • ASRs are classified into three classes including i) system management requirements, ii) communication and collaboration requirements, and iii) monitoring requirements • Based on this taxonomy, a probabilistic analysis method to analyse the impact of the ASRs on each other, as well as to analyse the impact of change in one of the requirements on other related and dependant requirements, is proposed
Santanu et al. (Kumar et al., 2017)	Service change detection	• Focus on change-prone services, defined using WSDL that are subject to potential changes and are therefore likely to cause incompatibility problems. • The authors proposed utilising source-code metrics to predict changes in eb service WSDL interfaces, using kernel-based learning techniques and prediction algorithms. • To validate the proposed approach, the paper presented a case study based on the popular online auction platform eBay.

		• The results demonstrated a consistent accuracy of above 80% – an indication of the overall effectiveness of the proposed approach.
Sohan et al. (Sohan et al., 2015)	Service change detection	• Focus on service API changes and address situations, when changes on a web API of an individual application may break the dependent applications. • The authors presented a case study to distil current industry practices on Web API versioning, documentation and communication of changes. • A compiled summary of approaches to Web API evolution, and a list of recommendations for practitioners and researchers based on API change profiles, versioning, documentation and communication approaches.
Papazoglou (Papazoglou, 2008)	Service change detection	• A classification of service changes and evolution: a) changes in structure/operational behaviour and policies/types of business-related events/business protocols; b) shallow vs deep changes. • Shallow changes: a structured approach and robust versioning strategy to support multiple versions of services and business protocols. • Deep changes: a change-oriented service life cycle methodology, which enables service reconfiguration,

		alignment and control in parallel to occurring changes.
Bachmann (Bachmann, 2015)	Service versioning	• Changes are identified at design-, run- or configuration time, depending on the perspective (i.e., developer, service provider, service broker, service consumer). • A UDDI-based versioning framework for WSDL-based services, which makes SOA systems more robust to a wide range of service changes • Using UDDI allows service consumers both to be notified of any modifications or to poll the registry at regular intervals to keep up to date with recent changes.
Novakouski et al. (Novakouski et al., 2012)	Service versioning	• Advocated for a versioning policy in service-oriented systems – i.e., all stakeholders must write and agree to policies that govern what to version, how to version, how to communicate about and coordinate changes, and how to manage the life cycle of changes. • Relies on a UDDI registry as a way of advertising service changes to interested stakeholders. • Provides an extensive set of practical recommendations regarding Web service versioning

		in SOA development. This guidance also provides common mistakes and potential pitfalls, which helps understand common problems that occur in object-oriented domains due to versioning.
Evdemon (Evdemon, 2005)	Service versioning	• Survey of versioning techniques used by the Microsoft platform. • Focuses on backward and forward compatibility, and extends them with notions of extensibility and graceful degradation. • Proposed the differentiation between message versioning (focuses on the data that composes the messages created and consumed by a service) and contract versioning (focuses on WSDL descriptions). • Provides a list of Microsoft's recommendations on service versioning
Erl et al. (Erl et al., 2009)	Service versioning	• Aims to summarise all existing theoretical fundamentals and practical information to provide a thorough guide to the designing SOA contracts and versioning. • Discusses the concepts of backward and forward compatibility, version identification strategies, service termination, policy versioning, validation by projection, concurrency control, partial

		understanding, versioning with and without wildcards, etc. • Provides a number of practical examples, addressed by problem-solving techniques, to demonstrate the discussed concepts.
Fang et al. (Fang et al., 2007)	Service versioning	• Proposed a version-aware service description model to describe Web service versions, handle different versions, and select and call-specific versions of a service • Propose a version-aware service directory model to manage service versions in the service directory, by extending the core schema of a WSDL document to include version-related fields and describe the attributes of the service versions. • Extended the UDDI service, which is able to incorporate versions in a service directory with an event-based consumer notification/subscription mechanism.
Becker et al. (Becker et al., 2011)	Service versioning	• Introduced a versioning framework where differences between multiple versions of a service can be automatically identified.

		• Proposed an approach to automatically determine when two service descriptions are backward compatible. • Explores compatibility of SOA messages being exchanged between service providers and consumers to ensure that only compatible messages are communicated.
Yamashita et al. (Yamashita et al., 2012)	Service versioning	• Introduced a novel feature-based versioning approach for assessing service compatibility and proposed a different versioning strategy, following the W3C standards. • Versioning is only applied to a certain fragment of WSDL/XML schema. • Enabled identification of changed (or affected) features in a new service version, and ensure that these modified features do not affect the stability of dependent client applications. • Provided some practical tips on supporting service evolution through maximising version reusability.
Leitner et al. (Leitner et al., 2008)	Service versioning	• Presented a comprehensive versioning approach specifically for handling compatibility issues, based on a service version graph and version selection strategies

		• The proposed framework is used to dynamically and transparently invoke different versions of a service through service proxies.
Raemaekers et al. (Raemaekers et al., 2017)	Service versioning	• Investigated versioning practices in a set of more than 100,000 jar files from the Maven Central repository, spanning over 7 years of history of more than 22,000 different libraries, aiming to understand to what extent versioning conventions are followed • Concluded that developers spend little effort to communicate backward incompatibilities or deprecated methods in releases to their target users. • Argued that semantic versioning principles should be embraced more widely by the developer community.
Bhattacharya et al. (Bhattacharya et al., 2017)	Service versioning	• Proposed a model to automate service discovery and dynamic choreography from a service version database. • Devised a comprehensive framework that models requirements in a formal manner and automatically extracts verbs to generate an activity model, which is then translated into Business Process Model and Notation (BPMN), this automatically translating the input requirements to business models.

		• The framework helps in discovering correct service version matches based on the business functions extracted from the requirements and matching with service version details, previously stored in a UDDI registry
Weinreich et al. (Weinreich et al., 2007)	Service versioning	• Focuses on evolution of services, implemented as Enterprise Java Beans (EJBs), which are accessed either by native J2EE clients or by Web service clients. • Proposed a versioning model that defines units of versioning, a versioning schema and a schema for service addresses. • Used a 3-digit versioning schema, which supports distinctions between compatible and incompatible changes. • Support for asynchronous client updates – i.e., the final decision, whether and when to update to a new service or whether to stay with an existing service has to be taken by the client.
Kaminski et al. (Kaminski et al., 2006)	Service compatibility	• Proposed an algorithm to check service compatibility based on a version framework that allows service descriptions to change at different finer-grained levels. This algorithm automatically assesses

		backward compatibility between two service versions using an object-oriented service description and generates a description of the service data type in the form of set elements. It also supports flexible input parameter formats in request messages. The algorithm presents an important mechanism to help the service provider identify and understand the impact of changes in the evolutionary development process. In addition, a SOA-based framework in which the algorithm has been implemented shows several benefits for both the consumers and providers who want to explore automated compatibility assessment.
Yamashita et al. (Yamashita et al., 2012)	Service compatibility	• Proposed an algorithm for recursive evaluation of two adjacent service versions, checking features of compatibility. • Capable of identifying and highlighting the changes in the newly developed service version along with their possible negative impact on the service provision.
Andrikopoulos et al.	Service compatibility	• Proposed a service evolution framework based on the Abstract Service Descriptions, able to generate

(Andrikopoulos et al., 2012)		details about structural, behavioural and QoS-related components of a service. • Addresses the compatibility issues in service versions using a T-shaped compatibility model, extended by the notions of horizontal compatibility (i.e., interoperability) and vertical compatibility (i.e., substitutability/replaceability). • Proposed an algorithm to check and assess the compatibility of service versions, based on the extended T-model.
Liang et al. (Liang and Huhns, 2008)	Service compatibility	• Proposed a dynamic compatibility evaluation framework through a system model that checks compatibility of Web services, as well as compatibility of the key operators. • For the purpose of evaluating compatibility, semi-structured documents such as WSDL and OWL-S were used.
Lublinsky et al. (Lublinsky, 2007)	Service compatibility	• Presented a compatibility assessment mechanism that emphasises on the interoperability among independently evolving Web services. • Used both dynamic and static analysis algorithms and built tools to apply these algorithms.

		• Introduced the notion of “cross stubs” to check incompatibility and facilitate interoperation across multiple versions of the same service.