Dependability for ESB systems in critical environments based on self-healing and checkpointing principles

by

MSc. Mariano Vargas-Santiago

Thesis submitted as a partial requirement for the degree of

Ph.D. in Computational Sciences

at the

Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE)
2018, Tonantzintla, Puebla, Mexico

Advisors:

PhD. Saúl Eduardo Pomares-Hernández
Computational Science Coordination
INAOE
Ph.D. Lius Alberto Morales-Rosales
Faculty of Civil Engineering
CONACYT-Universidad Michoacana de San Nicolás de Hidalgo

A mi familia.

Agradecimientos

Agradezco al Consejo Nacional de Ciencia y Tecnología (CONACyT), al Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE) de México, y también a la Universidad Michoacana de San Nicolás de Hidalgo (UMSNH) por el apoyo proporcionado para y durante la realización de este trabajo de tesis. En particular, al INAOE por permitirme desarrollar esta investigación en sus instalaciones, y a la UMSNH por haber confiado en mí.

Agradezco a mis asesores Dr. Saúl Eduardo Pomares Hernández, Dr. Luis Alberto Morales Rosales, Dr. Hatem Hadj-Kacem, y Dr. Khalil Drira, que tuvieron un papel fundamental en mi desarrollo. Al Dr. Luis Alberto le agradezco por haberme apoyado en todo durante mi estancia en la UMSNH. Y a todos ellos les agradezco de corazón la paciencia y dedicación para conmigo, y por apoyarme incondicionalmente.

Quiero agradecer a mis sinodales: Dra. Angélica Muñoz, Dra. Hayde Peregrina, Dr. René Cumplido, Dr. Gustavo Rodríguez y Dr. Khalil Drira por su tiempo, observaciones y sugerencias realizadas durante el proceso de revisión de esta investigación.

Agradezco a mis padres, Alicia y Fernando, por haberme inculcado el espíritu de estudio, y por ser mis guías en la vida. Agradezco a mi esposa y a mi niña por el esfuerzo que han tenido que hacer para mantenerme junto a ellas, aún cuando no he estado físicamente a su lado. A cada uno de los miembros de mi familia por su apoyo y los sacrificios que han hecho para que yo pudiera cumplir una de las metas de mi vida profesional.

Incluyo además, mi gratitud a los tantos amigos que compartieron conmigo durante esta etapa, a todos aquellos que me apoyaron en las buenas y en las malas.

Abstract

Ensuring dependability for computer systems based on fault tolerance is an open challenge. Due to the complexity and heterogeneity of the interactions and services, offered by distributed systems, they make the administration and management of resources in highly dynamic environments exceed the capabilities of more experienced network administrators. As a consequence, new paradigms emerge (autonomic computing) and the Service-Oriented Architecture (SOA). In this dissertation, on the one hand, autonomic computing is focused on solving the complexity of monitoring and diagnosing the behavior of the systems with low resources and little human intervention. On the other hand, to consider the heterogeneity of the systems, the SOA paradigm and the Enterprise Service Bus (ESB) were used for their integration. Applications and systems are needed to intercommunicate with each other, often in unreliable environments such as the nature of the Internet. There are sophisticated solutions, such as replication of services, rollback recovery, and self-healing, which increase the systems' reliability. However, these approaches have drawbacks; for example, they affect the performance of the system, have high implementation costs and/or can endanger its scalability. On the other hand, to facilitate the self-management of the systems, in this dissertation, we implement the Monitoring, Analysis, Planning and Execution (MAPE) control cycle. An open challenge for the MAPE cycle is to efficiently carry out the diagnostic and decision-making processes, collecting data from which the system can detect, diagnose and repair potential problems, that is, increase the dependability of the systems specifically with fault tolerant mechanisms. A useful tool for this purpose is through the implementation of communication-induced checkpointing (CiC) mechanism. The experimental results support the viability of our proposals.

Resumen

Garantizar la confiabilidad (dependability) para sistemas computacionales basados en tolerancia a fallos es un reto que representa un desafío, aún abierto. Debido a la complejidad y heterogeneidad de las interacciones de los servicios ofrecidos por los sistemas distribuidos, estos hacen que la adminsistración y el manejo de los recursos en entornos altamente dinamicos excedan las capacidades incluso de los adminstradores de red más experimentados. Como consecuencia, surgen nuevos paradigmas: el cómputo autonómico (autonomic computing) y el de la arquitectura orientada al servicio (SOA, Service-Oriented Architecture). En esta disertación, por un lado, enfocamos al cómputo autonómico para resolver la complejidad de monitorizar y diagnosticar con bajos recursos y poca intervención humana el comportamiento de los sistemas. Por otro lado, para considerar la heterogeneidad de los sistemas se utilizó el paradigma SOA y el bus de servicios empresariales (ESB, Enterprise Service Bus) para su integración. Las aplicaciones y sistemas se necesitan intercomunicar entre sí, muchas veces en ambientes no confiables como lo es la naturaleza del Internet. Existen soluciones sofisticadas, tales como: replicación de servicios, regresión en reversión, y auto-sanación, que aumentan la confiabilidad del sistema. Sin embargo, esos enfoques tienen inconvenientes; por ejemplo, afectan el rendimiento del sistema, tienen altos costos de implementación y/o pueden poner en peligro su escalabilidad. En contraparte, para facilitar la autogestión de los sistemas, en esta disertación, implementamos el ciclo de control de Monitoreo, Análisis, Planificación y Ejecución (MAPE). Un desafío abierto para el ciclo MAPE es llevar a cabo de manera eficiente los procesos de diagnóstico y toma de decisiones, recolectando datos de los cuales el sistema puede detectar, diagnosticar y reparar problemas potenciales, es decir, incrementar la confiabilidad de los sistemas específicamente con mecanismos tolerantes a fallas. Una herramienta útil para este propósito es mediante la implementación de puntos de control inducidos por comunicación (CiC, communication-induced checkpointing). Los resultados experimentales respaldan la viabilidad de nuestras propuestas.

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Motivation

Advancements in distributed systems technologies allow practically all devices to communicate with one another. And the way of carrying them out has changed from the traditional client-server design to more complex and sophisticated multi-tiered design model, where these are deployed inside a local organization or across several interconnected enterprises.

There are several solutions, with tendencies to mitigate several communication requirements brought by this new multi-tiered paradigm, some of them are oriented to address quality of service (QoS), scalability, interoperability, and others focus their efforts in system manageability and self-manageability for the communications solutions.

Diverse solutions were proposed at different layers of the communication stack, for instance, to address QoS, many network and transport TCP/IP based solution were proposed. These kinds of proposals mainly present scalability issues and are not widely deployed due to network constraints. Other efforts are proposed to manage integrability and interoperability.

For such purpose the middleware was introduced couple of years ago. It deals with interoperability and integrability requirements for distributed systems. Yet, still present QoS and scalability issues while dealing with high volume transactions, i.e. high number of concurrent transactions need to be supported.

As organizations collaborate on projects efficiently through the vast computing power of large scale distributed systems, these require a mean for diverse applications and programs to intercommunicate information in an adequate and consistent form. Leslie Lamport points-out that "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable" [TVS07]. In other words, you know yourself in a distributed system when a computer you did not know exists fails, and therefore one of your local task cannot be fulfilled. Therefore, dependability arises to address a broad spectrum of system characteristics, and many techniques have been introduced. For such aim, dependability, addresses system reliability, availability and fault tolerance [SS17].

In fact, organizations seek dependability for their computer systems, components and applications, by means of providing fault tolerance. With enough information previously saved, the system can continue rendering services to users even when a set of nodes have failed. Thus, one open challenge for systems' dependability is to offer fault tolerance, especially for low computing power devices in heterogeneous environments. For instance, mobile phones or any other device that support requesting services to a corporate enterprise by means

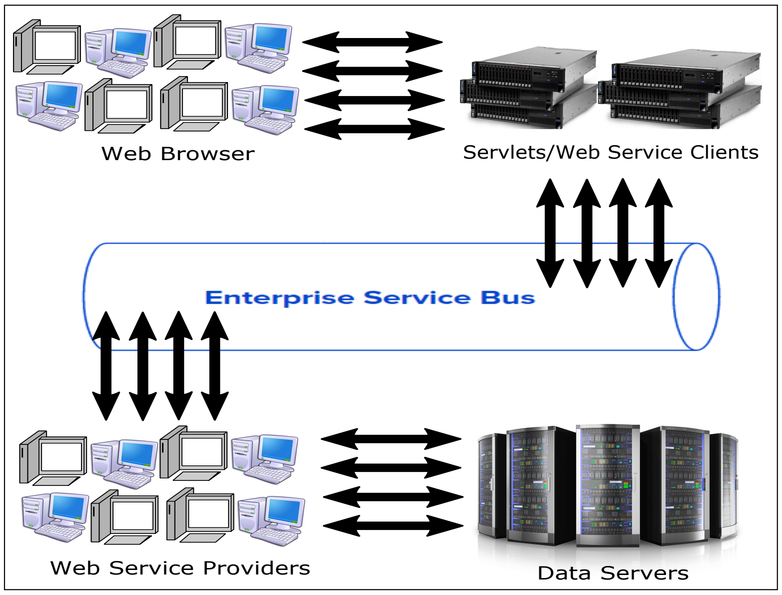


Figure 1.1: Scenario of bank transaction.

of Web services or any other application. Particularly, solving fault tolerance issues like: monitoring, detecting and recovering from runtime failures. To attack the fault tolerance issue present for system dependability efficiently, and considering that distributed systems susceptibility to failures has hampered their vast computing potential, many techniques arise. A promising technique for such purpose is rollback recovery. In this regard, communication induced checkpointing (CiC) is a well-known and efficient technique to pursue rollback recovery addressing fault tolerance [CSPHPC13, SHRK16, VSPHRHK17].

To achieve fault tolerance processes, save its state called checkpoint, this is done during the failure-free execution of the system, and upon a failure computer based systems gain a way to restart from a previously saved state. Reducing the amount of work to be carried out, as the system does not re-execute everything from the beginning. CiC's main goal is to save consistent global snapshots (CGSs), one from each process, free of dangerous checkpointing patterns (z-cycles and z-paths).

For example, Fig 1.1 represents an on-line purchase of some product. However, the users are not aware of that is behind the service presented to them. Fig. 1.1 explicitly shows each of the phases a client has to go through to complete its purchase. Here users start by viewing a product through a Web browser, implicitly accessing Web application server or servers, which in turn present their Web pages. These are implemented with frameworks or servlets, and are accessed through Web Services typically by generating SOAP or REST requests. The Enterprise Service Bus (ESB) interprets incoming messages and performs data processing, only if required, and targets service providers. Web Service providers are based on Web applications servers and also provide required operations or business logic and typically access external databases to check if the product is in stock or not.

1.2 Problem description

Computing systems complexity is overpassing the most experienced systems managers or administrators, needing teams of experts for problem resolution, and being a time consuming task for a human team. To mitigate this situation IBM's vision of autonomic computing paradigm arises [KC03] as a promising solution; specifically tackling issues by implementing self-healing. Furthermore, business processes are implemented broadly over distributed systems where heterogeneity becomes another challenge. To assuage the heterogeneous intercommunications issues between diverse services and applications the Enterprise Service Bus (ESB) is the uttermost common integration approach [KDVSD$^+$14]. Autonomic computing (self-healing) and systems integration (ESB systems) can be better addressed by applying dependability techniques like fault tolerance for instance. Systems' dependability is the ability of such to avoid failures that are severe, frequent and affect the quality perceived by end users. Notwithstanding that there exist sophisticated solutions that suggest dependability, they have drawbacks: in systems performance, implementation costs [YCD$^+$09], others jeopardize scalability [AH11] and a few are for systems' diagnosis only [KDVSD$^+$13] [KDVSD$^+$14]. Definitely, there is a demand for more efficient dependable solutions. Self-healing based on fault tolerance can be achieved through checkpointing mechanisms which have gain a great level of maturity. For a clear understanding of the challenges involved while developing dependability for ESBs systems, having as foundation self-healing and checkpointing mechanism. We have divided this section into three parts. Firstly, we explain why messages ordering is important for systems that use the ESB system as their integration backbone. Secondly, we present the characteristics of checkpointing protocols and self-healing within ESBs systems. Lastly, we expose the problem of synthesizing self-healing and checkpointing protocols for an increased dependable ESB system.

1.2.1 Partial ordering algorithms for ESBs

Before we can implement checkpointing mechanisms some considerations must be contemplated. For example, in distributed and heterogeneous networked environments intercommunications are merely accomplished by business processes when exchanging messages, and the order of how such events happened is well-known. Nonetheless, distributed systems are characterized for not having a common reference of time. As illustrated by Lamport in [Lam78], the concept of one event *happening before* another in a distributed system is shown to define a partial ordering of the events. Partial ordering of events is also called *causal path* or just *causality* between events. In distributed systems there is no global physical time; yet, the causality concept or an event *happening before* another, we could have an approximation of it. The *Happened-Before Relation* (HBR) is described in detail in the background section of this work. Since the appearance of the HBR, there have been a lot of works [Fid91], [RS96] [Ray92] trying to efficiently order events in distributed systems . For example, implementing the notion of logical time using: scalar time, vector time and matrix time; developed by Lamport, Fidge, Mattern and Schmuck, respectively, and the matrix clocks was first informally proposed by Michael and Fischer [KS08].

We need to transfer the HBR concepts to guarantee a partial ordering of events for ESBs systems. Despite, the previous contributions from recognized authors like Lesli Lamport, in practice the HBR is expensive to set up. To obliterate that the HBR is expensive Pomares et al. in [HFD04b] suggested an optimal way to assure causality between events for distributed systems; the Immediate Dependency Relationship (IDR). Since, we want to say whether or not two events are causally related, if so they can not constitute a consistent global snapshot (CGS). The challenge here is to leverage the ESB system structure in order to achieve and reduce at minimum the causal control overhead sent per message in the communication channels. In consequence, having a causal view of the system, eases the task of designing domino effect free checkpointing protocols.

1.2.2 Characteristics of checkpointing mechanisms and autonomic computing for ESBs systems

Checkpointing mechanisms have been used merely for distributed and parallel systems, and have proved to be useful as a fault-tolerant mechanism, and address issues like: software debugging and validation [KS08]. The collection of information in a checkpoint is important for the process of computation recovery. If the information gathered is not useful, then our system only suffered degradation, however, if the checkpointing mechanism is efficient, then we gained a possible recovery point. Hence, the collected data must be useful, always, for the system degradation to be minimum. Only few studies have attempted to systematize the checkpointing paradigm in autonomic computing [OFD06] [CSPHPC13] and suggested it is possible to merge self-healing and checkpointing mechanisms.

On the other hand, client applications usually use unreliable connection protocols when invoking services and have random delays in the communication channels. These protocols do not guarantee message ordering delivery in collaborative environments, sometimes they do not support asynchronous messages exchange. Therefore, a more robust messaging mechanism is needed. Sharing and discovering information in a collaborative context is demanded by the industrial development of dynamic networks. In these scenarios, business partners need to share and modify information remotely. Assuring message ordering is fundamental since all the involved users should have the same view of the system, and data must be presented coherently; also useful to build checkpoints or discarding such. Additionally, messages provide the expected behavior for distributed applications.

1.2.3 Merging autonomic computing and checkpointing mechanisms

In an SOA context large number of concurrent interactions amongst providers and consumers can take place, there is a competition for resources of shared services that can lead to unpredictable conditions or events such as service unavailability, high response time, as consequence not warranting reliability. Implementing SOA within complex business collaborative environments using Web services, are error prone because of unreliable Internet behavior during

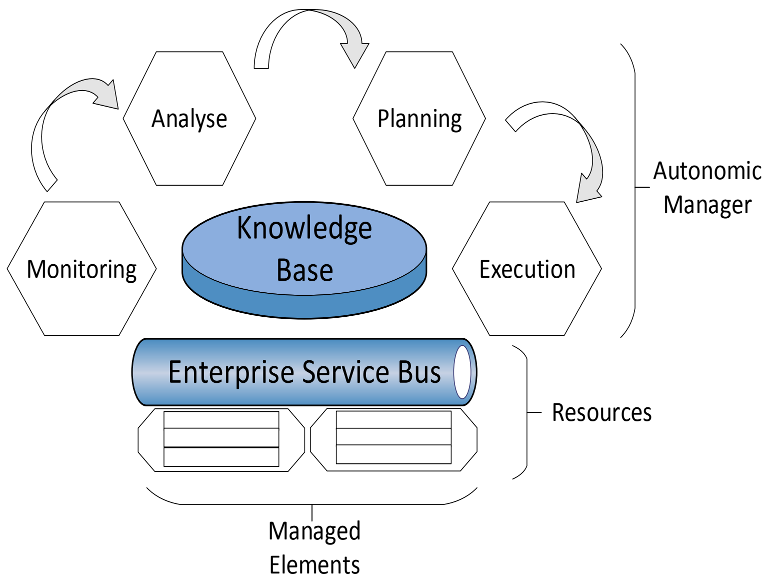


Figure 1.2: Autonomic Service Bus [Dio15].

run-time while they are still required to function correctly and be available on demand.

Failures may lead to terrible consequences such as augmenting execution time, higher costs to run applications, destroyed systems, or system breaches. As a consequence, organizations shall establish a way to make their systems or business processes as dependable as possible before they intend to automate them [VG10]. In anticipating these errors, organizations using core Web Services for their business processes require efficient and seamless solutions. In order to attack the problem of failures presented by Web Services, organizations are extrapolating the autonomic computing paradigm into their business processes as it enables them to detect, diagnose, and repair problems, and therefore improve system dependability.

To improve both performance and therefore dependability such anomalies need to be addressed by proposing efficient approaches, and strategies. Business processes can require a lot of human expertise, time and skills for their configuration, for repairing and management. Therefore it is mandatory avoiding managing systems manually because doing so becomes more expensive and difficult than doing so in an autonomic way. The initiative known as autonomic computing aims at designing and building systems capable of managing themselves, therefore monitoring and evaluating their state periodically for applying changes or taking action to improve their performance, also to recover upon a failure.

Many works have been issued to improve the QoS offered by Web Services by means of autonomic computing properties, i.e. implementing the MAPE cycle. Still, there is no unified or standardized form for implementing this cycle within Web Services, and neither for Web Services composition. Some works treat each one of the MAPE loop phases as an individual Web Service [GZ05, KGM11]. Others only tackle a single feature of the autonomic computing paradigm, in particular self-healing [MSSD06, TZZ$^+$05]. Tian et al., suggest to address the entire MAPE control loop adding other interfaces to confront functional and non-functional Web Services' requirements [TZZ+05].

However, there is a correlation between the failure free execution time and the overhead introduced by autonomic computing and checkpointing mechanisms. As consequence, there is a need to consider it. Organizations that use Web Services for their business processes require efficient approaches that do not jeopardize their throughput, i.e. have small overhead. This can be accomplished by making inferences from information gathered autonomously, and with few to no human intervention at all.

As consequence, we propose to increase ESBs systems' dependability by implementing the MAPE control loop, as shown in Fig. 1.2, building an Autonomic Service Bus.

1.3 Proposed solution

The contributions of this research are threefold. We propose the following, first for collaborative environments where interactions among different parties take place, we consider the order of messages exchanged, and propose a partial ordering algorithm for this matter. Second, we propose to address Web Services fault tolerance by merging autonomic computing with an asynchronous checkpointing mechanism. Finally, we propose an approach based on fuzzy logic under which the amount of checkpoints can be reduced.

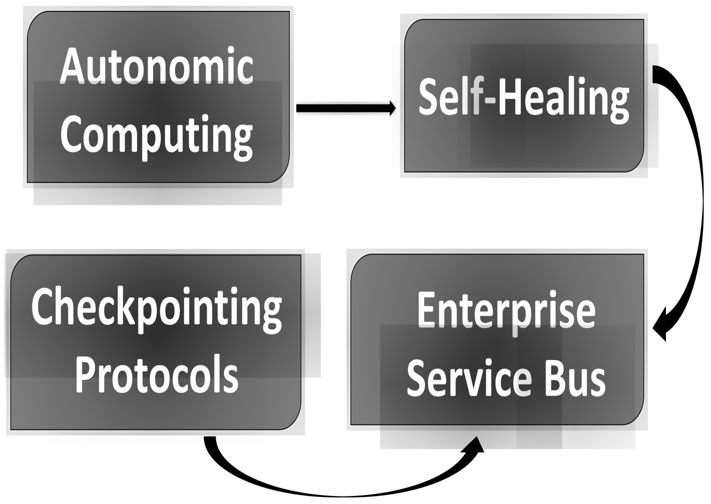


Figure 1.3: Problem Description.

We developed an Autonomic Service Bus (ASB) illustrated in Fig. 1.2. The ASB can adapt to changes in the environment, by supporting fault tolerance.

As illustrated by Fig. 1.2 the managed elements are components which can be monitored: CPU and RAM. A common knowledge base is needed in order to make decisions based on previous experiences, in other words based on the probability of systems' degradations. These decisions are taken by adopting the MAPE loop cycle.

1.4 Dissertation hypothesis and objectives

The following hypothesis is proposed to lead our research:

For ESBs in dynamic critical environments, self-healing features such as: detect, diagnose and repair, can be efficiently implemented with checkpointing mechanisms considering QoS requirements.

1.4.1 Main objective

- To develop dependability capabilities based on QoS requirements for ESBs systems through merging checkpointing mechanisms and autonomic computing.

1.4.2 Specific objectives

1. Identify fault tolerance issues in an SOA context to characterize systems' dependability for collaborative environments.
2. To leverage the ESB system infrastructure and reduce at minimum the causal control overhead sent per message in the communication channels, based on causal ordering.
3. To develop an efficient fault tolerant mechanism based on self-healing and checkpointing for rollback recovery.
4. To implement and deploy the self-healing MAPE control loop along with checkpointing mechanisms to increase ESBs' dependability.
5. To reduce the generation of checkpoints considering QoS parameters for dynamic and collaborative environments based on fuzzy logic and the MAPE control loop.

1.5 Document organization

The rest of this document is organized as follows. In chapter 2 the main concepts about distributed systems, causal ordering, Service Oriented Architecture, Enterprise Service Bus, Autonomic Computing and fuzzy sets are introduced.

In chapter 3 some works related to Web services compositions for both: orchestration and choreographies are presented, we classify and present their advantages and drawbacks. We also illustrate, fuzzy logic for Web services and their applicability from a point of view of autonomic computing.

In chapter 4, the message ordering framework is presented. Firstly, we explain the relevance the order of messages plays for systems consistency. Secondly, based on the Immediate Dependency Relation (IDR), we detail our mechanism. The effectiveness of our framework is also presented by showing performance test.

In chapter 5 we illustrate how checkpointing mechanisms can help out autonomic computing, and then we present the MAPE control loop together with checkpointing mechanism, specifying exactly where to implement.

In chapter 6 the fuzzy consistency system evaluation (FCSE) is defined. Based on the FCSE, which evaluates QoS parameters, a dynamic checkpointing generation approach is presented. The effectiveness of the resulting mechanism is verified by simulations.

Finally, the conclusions and the future work of this research are summarized in chapter 7.

CHAPTER 2

BACKGROUND AND DEFINITIONS

2.1 Software paradigms

2.1.1 Autonomic computing

The notion of autonomic computing systems was proposed by IBM in 2001 [KC03], as a system with monitoring and analysis capabilities of the states of its components. An autonomic computing system can detect problems that arise from failure and continue operation performing maintenance and/or adjustment parameters in QoS degradation without human intervention. In general autonomic computing systems consider four major aspects: self-configuration, self-healing, self-optimization and self-protection, also known as S-CHOP. These four major aspects are best summarized in Table 2.1, showing four aspects of self-management as they are now and how should be under the autonomic computing paradigm.

Many of the big enterprises like HP, Sun, IBM are trying to evolve their system to the autonomic computing paradigm, having as main goal the self-managing of systems. Often autonomic computing systems are being tackled globally; for example Java programs with self-healing and self-optimizing properties studied in [OFD06]. Other approaches are more thorough or detailed for solving the self-healing issues; as examples, a model for diagnosis and adaptation of the self-healing property into the ESB can be found in [KDVSD+13]; the authors argue they transformed such ESB to an Autonomic Service Bus (ASB). But all the computing is done off-line, so the authors basically give a way to diagnose the ESB; predicting how one variable affects another. Another thorough example, an ESB with self-healing capabilities can be found in [AH11], where the authors propose a self-healing architecture for Airports systems integration. We will focus on the self-healing property; which remains still as an open challenge, and it is currently under study by academic researchers and by the industry.

As stipulated by the autonomic computing paradigm, systems must evolve to self-management. To achieve a self-manageable system IBM introduces the notion of autonomic managers and managed elements. Autonomic managers are in charge for the interaction and communication with the outside world, in other words interpreted as human computer interaction and interactions with other elements. Also, as a bridge with the managed elements; which at a time implement the *MAPE (Monitoring, Analysis, Planning and Execution)* control loop. The *Monitoring* phase is in charge of recollecting data, then the *Analysis* phase searches for possible issues, sometimes based on probability, as for the *Planning* phase which must plan ahead, if possible, what actions to take. Finally, the *Execution* phase executes actions regarding previous phases [KDVSD+14]. However, experts in the field must choose

Table 2.1: Four aspects of self-management as they are now and would be with autonomic computing reproduced from [KC03].

Concept	Current computing	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

the autonomic managers and the managed elements.

2.1.2 Service-Oriented Architecture (SOA)

Enterprises' software components or modules (usually deployed as services) running on two or more enterprise's networks, are usually known as distributed enterprise applications. Most of the time, the enterprise network is heterogeneous and it is composed of diverse computers, devices, and operating systems. Since the network is heterogeneous, systems consist of different protocols for data exchange, technologies, and devices distributed across a network. In recent years the industry environment has become increasingly distributed and heterogeneous across multiple organizational and geographical boundaries, there is a strong demand to integrate various distributed applications in order to enhance or increase enterprises' competitiveness. SOA arises from previous technologies that in the past, or in their time, were good integration technologies like: point-to-point and Enterprise Application Integration (EAI) [Cha04]. In a point-to-point style all applications and/or services interfaces are programmed manually to communicate among each other, which increases the costs of maintenance and development [RH08]. To mitigate such complexity and heterogeneity imposed by the traditional point-to-point integration technology the EAI arises, where all the communication passes through a message broker; so programming this interface is only done once. However, this message broker also introduces a known point of failure to the network, what happens when this broker fails, how does the communication among applications take place? Therefore, SOA arises as a new paradigm that most of the organizations, nowadays follow, as applications and systems are becoming more disperse and distributed around the world; mitigating the complexity and heterogeneity of systems. SOA allows to reuse the code of an application facilitating its integration following standards like XML (eXtensible Markup Language), SOAP (Simple Object Access Protocol) to integrate the applications in a standards-based approach instead of a vendor-based approach [UT06].

An SOA separates the monolithic approach of integrating applications or services. The goal in SOA is to make each subsystem of a company present their capabilities through adequate services; mostly by implementing Web services. SOA is an architecture that can build business applications from a set of loosely coupled black box components [HBBK07]. SOA links together business process having an orchestrated well-defined service level. The reuse of existing business applications is done adding a simple adapter to the black box components regardless of how they were built. SOA takes the best software assets used and packages them so that it lets you use them and reuse them, so you do not have to discard software.

Any SOA system is built using different units, services for a specific functionality, to have complex systems. These systems usually involve various physical resources, i.e., network resources, processing components, and of course the logical organization [Men07]. SOA gives support for flexible connectivity and communication among applications by representing each as a service with an interface that lets them communicate readily with one another.

2.1.3 Enterprise Service Bus (ESB)

On the one hand, SOA discovers applications and business functionalities as services through an interface. On the other hand, the ESB can leverage the use of such interface; the SOA paradigm and the ESBs together form complex systems following open standards; which for businesses is of vital importance. An ESB was born as a solution for integration of heterogeneous applications and thus, it has shown to be a powerful integration solution based on open standards and as a basis for complex SOA environments [Men07].

In the context of highly distributed network and loosely coupled applications (given by SOA); the ESB is a new approach to integration, also seen as a middleware layer that allows the integration of heterogeneous applications, using a standards-based approach; in other words, it is a standards-based integration platform. An ESB plays the role of connecting heterogeneous applications and services in an SOA; it handles the transformation, routing, and data mediation/adaptation [Cha04].

2.1.4 Basic functionalities of an ESB

The main and basic functionalities an ESB must integrate are: Virtualization, Mediation, and intelligent Routing these functionalities are known as the core features of an ESB [AP11].

Virtualization: Virtualization, or proxying; in this role, the ESB acts as a proxy for the service provider and handles the requests of the service consumer. The ESB can handle authentication, authorization, and auditing, so the service provider can focus solely on the business logic.

Mediation: Mediation, or message transformation; in this role, the ESB has the capability to take an incoming request and transform the message payload before sending it to the end Web Service.

Routing: In this role, the ESB has the capability to route the incoming requests on a single endpoint to the appropriate service. The ESB can look at a wide array of things like the message content or the message header to determine where the request should be routed.

2.2 Distributed computing

2.2.1 Communication patterns

This section provides the background that characterizes distributed systems for the utilization of the present communication-induced checkpointing (CiC) mechanism, as it is responsible for the generation of checkpoints free of (i) domino effect and (ii) dangerous patterns. Based on these premises, distributed systems have the following characteristics: there is no global notion of time, processes do not share common memory and communicate solely by message passing. In this context distributed computation consists of a finite set of processes $P = \{P_1, P_2, \ldots, P_n\}$. We assume that channels have an unpredictable yet finite transmission delay and are reliable and asynchronous. Two types of events must be considered: *internal* and *external*. Internal events are those that change the processes state, for instance a checkpoint, a finite set of internal events are denoted by E_i. External events are those that affect the global state of the system, for instance *send* and *delivery* events. Let m be a message, $send(m)$ is the emission of m by a process $p \in P$ and $delivery(q, m)$ is the delivery event of m to participant $q \in P$ where $p \neq q$. The set of events associated to M is:

$$E_m = \{send(m) : m \in M\} \cup \{delivery(p, m) : m \in M \wedge p \in P\}$$

Thus the whole set of events is

$$E = E_i \cup E_m$$

2.2.2 Happened-Before Relationship (HBR)

A distributed system consist on various process or applications executed on diverse machines distributed on different parts of the planet to accomplish a task, where the user is not aware of the execution that took place. One of the principal characteristic for a distributed system is that:

- It has no global physical time; as a solution various works, starting with Lamport and the scalar time representation back in 1978 [Lam78], have tried to realize an approximation of it, this is known as logical time.

"As asynchronous distributed computations make progress in spurts, it turns out that the logical time, which advances in jumps, is sufficient to capture the monotonicity property associated with causality in distributed systems [KS08]."

Leslie Lamport defined the HBR trying to totally order events (sending or reception of messages) in a distributed system [Lam78]. Because, the execution of a system may be revealed examining how events took place (knowing if events occurred before, after or concurrently with another event at another process); despite the lack of accurate clocks.

Definition 1 (Lamport, 1978). *The happened-before relation* denoted by $\rightarrow$, is formally defined as the strict partial order on events such that:

- *If events e1 and e2 occur on the same process and the occurrence of e1 preceded the occurrence of e2 then e1 $\rightarrow$ e2.*
- *If e1 is the sending of a message of a process and e2 is the reception of the message by another process, then e1 $\rightarrow$ e2.*

- *If there is an event e3 such that e1 $\rightarrow$ e2 and e2 $\rightarrow$ e3, then e1 $\rightarrow$ e3.*

Other properties for the HBR are that: it is transitive, irreflexive and antisymmetric as illustrated below:

- *Given an event e1, e1 $\nrightarrow$ e1 (irreflexive property)* [1]
- *Given e3 such that e1 $\rightarrow$ e2 and e2 $\rightarrow$ e3, then e1 $\rightarrow$ e3 (transitive property).*
- *Given two events e1 and e2, if e1 $\rightarrow$ e2 then e2 $\nrightarrow$ e1 (antisymmetric property).*

Definition 2(Lamport, 1978). *Concurrent events*: *Two events e1 and e2 are concurrent if e1 $\nrightarrow$ e2 and e2 $\nrightarrow$ e1, denoted by e1 $\|$ e2.*

2.2.3 Immediate Dependency Relation (IDR)

The HBR in practice is expensive since it has to keep track of the relation between each pair of events. In order to avoid that causality is expensive, Pomares et al. [HCR12] have worked on the *Immediate Dependency Relation (IDR).* Which, minimizes considerably the amount of control information sent per message to ensure causal ordering. The IDR is the transitive reduction of the HBR, it is denoted by "$\downarrow$" and it is defined as follows:

Definition 3. (Pomares, 2004) *Two events e1 and e2 $\in$ E have an IDR "e1$\downarrow$e2" if the following restriction is satisfied*:

- *e1 $\downarrow$ e2 if e1 $\rightarrow$ e2 and $\forall$ e3 $\in$ E,* $\neg(e1 \rightarrow e3 \rightarrow e2)$.

[1]In this context, e1 $\nrightarrow$ e2 $\equiv \neg(e1 \rightarrow e2)$ this means that e1 does not happen before e2.

2.2.4 Checkpoint and Communication Pattern (CCP)

It is represented by its distributed computation consists on a set of incoming and outgoing messages and associated local checkpoints.

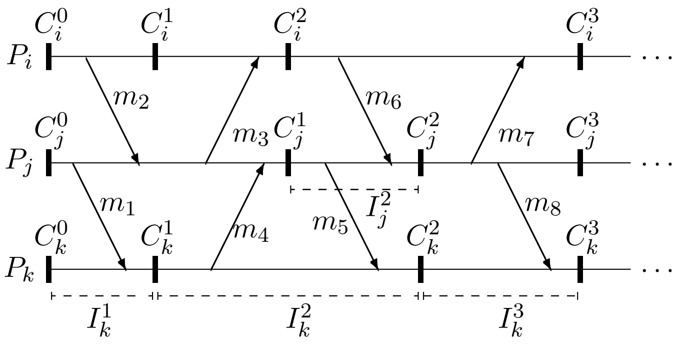


Figure 2.1: CCP [CSPHPC13].

Definition 2.1. *(Netzer, 1995) A communication and checkpoint pattern (CCP) is a pair $(\widehat{E}, E_i)$ where $\widehat{E}$ is a partially-ordered set modeling a distributed computation and E_i is a set of local checkpoints defined on $\widehat{E}$.*

An example of a CCP is shown in Fig. 2.1; showing the *checkpoint interval* denoted I_k^x, with sequence of events occurring at p_k between C_k^{x-1} and C_k^x $(x > 0)$.

Simon et al. introduce an approach based on CiC, it attacks the *overhead* problem as they demonstrate that not all forced checkpoints are necessary as stipulated by solutions proposed so far [LM09]. Simon et al., introduce what they call *save* checkpoint conditions, with those they identified when a forced checkpoint can be removed and/or delayed, details can be found in [CSPHPC13].

The HBR and the IDR are useful for event ordering over distributed systems; with such information consistent global snapshots of the system may be created. So, if we have events ordering we could know which events happened-before others or have a causal path, these form inconsistent snapshots; in contrast if they are potentially concurrent or happened at the same time they form a consistent snapshot [NX95]. Now that we have defined the fundamentals needed to take a consistent global snapshot.

Next we give a brief introduction to checkpointing mechanisms, as these are in charge for building consistent cuts of the system from check-pointed states.

2.2.5 Checkpointing mechanisms

A *checkpoint* is basically information gathered by a processor in a certain time, with such information the processor can return to that checkpoint. A global snapshot consists of the collection of checkpoints taken separately by each processor. A *Consistent Global Snapshot* (CGS) identifies checkpoints that do not have a causal path; they are not related by a message or a sequence of messages. And a *checkpointing algorithm* collects checkpoints during the system computation, so in case of failure the system can recover partially or totally.

Distributed systems are ubiquitous but are not fault-tolerant, yet need a whole lot of computing which makes them susceptible to failures. Many studies try to develop new techniques to add reliability and availability to distributed systems. One of such many techniques is the one known as rollback recovery.

"Rollback recovery treats a distributed system application as a collection of processes that communicate over a network. It achieves fault-tolerance by periodically saving the states of a process during failure-free execution, enabling a restart from a saved state upon a failure to reduce the amount of lost work. The saved state is called a *checkpoint*, and the procedure of restarting from a previously check-pointed state is called *rollback recovery*" [KS08].

In the literature three different checkpoint based rollback recovery techniques can be found: asynchronous or uncoordinated checkpointing, synchronous or coordinated checkpointing and quasi-synchronous checkpointing or communication-induced. Checkpointing algorithms save checkpoints on the stable storage or on the volatile storage depending on the failure scenarios to be tolerated. These were explained briefly, and their advantages as well as their drawbacks are discussed in the related work section of this dissertation.

2.3 Fuzzy logic

2.3.1 Fuzzy logic

Definition 2.2. *(Zadeh, 1965) Zadeh establishes that a set A is defined as a membership function $f_A(x)$ that maps the elements of a domain or universe X with the elements of the interval $[0, 1]$: $f_A(x) : X \to [0, 1]$ representing the degree of membership of x in A.*

Meaning that the closer the value of $f_A(x)$ to 1, the higher the degree of membership of x in A.

A fuzzy set A can be represented as a set of pairs of values: each element $x \in X$ with its degree of membership in A.

$$A = (x, f_A(x))|x \in X$$

Definition 2.3. *(Ross, 2010) Fuzzification is the conversion of a precise quantity to a fuzzy quantity.*

The most used fuzzifier is based on the triangular function:

- **Triangular function:** $f_A(x) = max[min(\frac{x - L}{C - L}, \frac{R - x}{R - C}), 0]$, see Fig. 2.2.

L,C and R are real scalar values that delimit a fuzzy set A, being C the input value that has the largest membership to A.

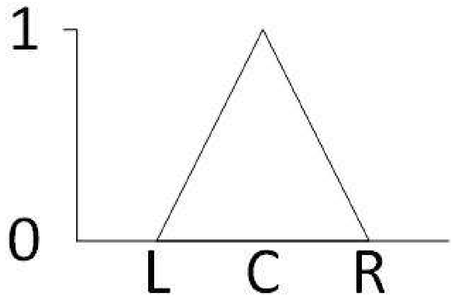


Figure 2.2: Triangular Function.

Definition 2.4. *(Lee, 1990) Defuzzification is the conversion of a fuzzy quantity to a precise quantity.*

Defuzzification can be performed using the *weighted average* method. Such method is one of the more computationally efficient methods. Having as restriction that the output membership functions must be symmetrical. It is given by the following algebraic expression:

$$\frac{\sum f_A(a_c)(a_c)}{\sum f_A(a_c)}$$

where $\sum$ denotes the algebraic sum and a_c is the centroid of each symmetric membership function, see Fig. 2.2. The weighted average method is formed by weighting each membership function in the output by its respective maximum membership value.

Definition 2.5. *(Lee, 1990) Linguistic variables. These are variables whose values are represented using linguistic terms (low, medium, high, very high,etc.). The meaning of these terms are determined through fuzzy sets. A linguistic variable is characterized by (v, T,X, g,m), where:*

- v is the name of the variable.
- T is the set of linguistic terms of v.
- X is the universe of discourse of the variable v.
- g is a syntactic rule to generate linguistic terms.
- m is a syntactic rule that assigns to each linguistic term t its own meaning *m(t)*, which is a fuzzy set in X.

2.3.2 Fuzzy Inference System (FIS)

A fuzzy inference system (FIS) is a way to transform an input space in an output space, using fuzzy logic. The FIS attempts to formalize, using the fuzzy logic, reasoning of human language.

Generally, a FIS has four modules as depicted in Fig. 2.3.

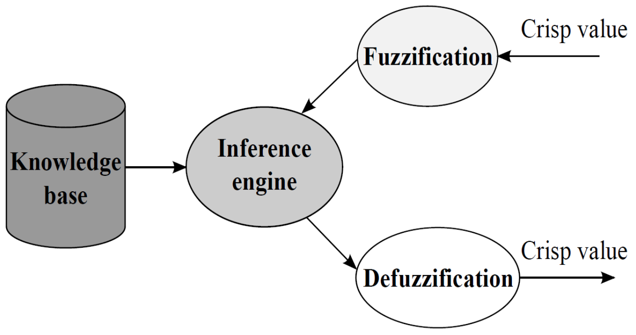


Figure 2.3: Fuzzy Inference System.

- *Fuzzification module:* transforms the system inputs, which are crisp numbers, into memberships to fuzzy sets. This is done by applying a fuzzification function.
- *Knowledge base:* stores if-then rules provided by experts.
- *Inference engine:* simulates the human reasoning process by making fuzzy inference on the inputs and if-then rules.
- *Defuzzification module:* transforms the memberships to fuzzy sets, obtained by the inference engine, into a crisp value.

The most used FIS are the Mamdani type [MA75] and the Sugeno type [TS85].

- In the Mamdani systems the inputs and the outputs of the inference engine are fuzzy
  - *If x is A and y is B then z is V*
- In the Sugeno systems, the inputs of the inference engine are fuzzy and the output is "crisp"
  - *If x is A and y is B then $z = f\ (x,\ y)$*

Chapter 3

Related work

Web Services Composition has emerged as an important computing paradigm to create complex business processes [Pet16]. In distributed heterogeneous environments, individual Web services are used as fundamental elements to support fast and low cost development of a set of interacting services, which form comprehensive businesses functionalities [CDK+02, LDB16]. Thus, according to predefined business requirements, Web Services Composition refers to the process of adaptively composing a set of available Web services into a business process flow. Services composition dramatically reduces the cost and risks of building new business applications in the sense that existing business logics are represented as Web services and could be reused [BHH10].

Web services are presented as a promising technology to implement Service Oriented-Architecture (SOA) [Pas05, FF12, AMH10]. Composing such Web services implies using standard-based languages which interact through Internet-based protocols. Notwithstanding that these technologies readily allow creating large-scale systems, they are, however, prone not only to incoming errors from the dynamic and unreliable Internet, but also to errors that increase proportionally to the number of component counts [MBMS10, GJGT10]. Although Web service composition have been heavily researched, several issues related to dependability still need to be addressed. In this aspect, one primary concern is to provide fault handling mechanisms [ZL10, ZL12, ZL13, ZL15]. Adopting robust fault tolerance mechanisms is necessary because they reduce the risk of faults, and businesses can properly be automated. Therefore, fault tolerant business processes are a necessity, because failures may lead to terrible consequences, for instance, increasing the execution time, elevating the costs of the running applications, destroying or breaching the systems [VG10]. Clearly, organizations need a way to guarantee consumers' needs, meaning, delivering the requested services and delivering what the services are expected to do in a timely manner.

Composing Web services is achieved through integration approaches, such as choreography and orchestration or a combination of these approaches [Pel03, KKM11, RFG12, KGI13]. Over the last decade, diverse works tackling fault tolerance for Web Services Composition have appeared [SQV+14]. Many of them are based on the checkpointing paradigm. Nevertheless, trying to extrapolate the checkpointing paradigm into another paradigm like Web services has proven to be a complicated task due to the dynamic nature under which Web services interact, and even choosing which checkpointing technique is the most appropriate one becomes a complicated task [VM14b, SHRK16]. For example, in the literature we can find four different checkpointing types of mechanisms: asynchronous or uncoordinated, synchronous or coordinated, quasi-synchronous or communication-induced and message logging based checkpointing. Regardless of which checkpointing mechanism is used, rollback recovery increases the reliability and availability of distributed systems [KS08].

3.1 Fundamentals and Web service faults analysis

This section presents the basis for checkpointing Web services composition, giving brief definitions on concepts like: Web services characteristics, Web services composition models, checkpoint, checkpointing, rollback and their applicability to Web services composition, by emphasizing their integration and describing how these two paradigms, i.e. how one can benefit the other. Also, we present an analysis concerning the types of faults in Web services composition.

3.1.1 Web services characteristics

Standardization efforts establish the restrictions for building Web services that exhibit the following characteristics [Man03]:

- Web services are platform-independent and language neutral. They are accessed through a well-known interface. Therefore, Web protocols ensure effortless integration of heterogeneous distributed environments.
- Web services provide an API that can be called by other programs. This interface applies the application-to-application programming technique that can be summoned, for example, by BPEL or any other type of application. The API provides access to the application logic.
- Web services are registered through a Web service *registry*, which enables service consumers and organizations to easily find services that match their needs.
- Web services make interconnections flexible and adaptable because they add a layer of abstraction to the environment. Therefore, Web services support loosely-coupled connections between systems and communicate through their API by exchanging XML messages.

Another aspect considered as non-functional requirement for Web services is:

- *Quality of Service:* Web service composition must agree on the level of *QoS* that has to be met. One open challenge is to take into consideration QoS degradation when applying checkpointing mechanisms.

3.1.2 Web services composition models

Service composition is fundamental in the SOA paradigm. It is oriented towards building complex Web services from smaller components. Composition rules deal with the way in which different services compose a coherent global service. In particular, they specify the order in which services are invoked and the conditions under which a certain service may or may not be invoked. The design of composing Web services is mainly carried out throughout two composition techniques, namely choreography and orchestration.

Orchestration. In this composition model, the involved Web services are under the control of a single endpoint central process (orchestrator). This process coordinates the execution of various actions on the Web services involved in the composition. The Business Process Execution Language (BPEL) [CGK+03] has become one of the predominant standards for Web services composition [JGH09]. BPEL orchestrates the interactions between Web services within a single part that controls and describes a process flow or work-flow. One core feature offered by BPEL is the support for asynchronous communications, which is needed between long-running applications based on Web services. BPEL provides an infrastructure that manages data persistence.

Three basic fault handlers are provided by the BPEL engine: compensation handlers, fault handlers, and event handlers [CGK+03].

- Compensation handlers are used to undo or reverse the effects of a previous activity, specifying the actions to be executed.
- On the other hand, fault and event handlers execute actions at runtime for predefined faults and/or events.

Nonetheless, BPEL only manages predefined faults specified by application designers.

Choreography. This composition model presents an abstract description of protocols. It offers a top view of the management rules which govern the interactions between the involved services in a decentralized application. It differs from orchestration because the former represents control from one member perspective. It allows each involved entity to describe its part in the interaction, thus, being more collaborative. Choreography tracks the message sequences among multiple entities and sources rather than a specific business process that a single orchestrator executes [Pel03]. It is modeled by abstract processes. An abstract process or business protocol specifies the public message exchanges between the different entities.

Choreography uses the Web Service Choreography Interface (WSCI) which defines a collaboration extension to the Web Services Description Language (WSDL). In other words, it defines the overall choreography or message exchange between Web services. The specification supports message correlation, sequencing rules, exception handling, transactions, and dynamic collaboration [Pel03].

3.1.3 Web services composition recovery modes and fault types

In this section, we will first talk about the *recovery modes* that are common in literature for composite Web services, then we categorize faults according to *behavior*, *instant* and *origin* of the faults. Afterwards, we will give the *recovery strategies* based on checkpointing mechanisms.

Recovery modes. There are two types of recovery modes under which composite Web services are currently recovered: global recovery and local recovery mode, described as follows:

- *Global Recovery Mode*: If the overall system rolls back, not only does the failed Web service roll back, but so do all others which are directly or indirectly affected. This is known as global recovery (the overall system recovers). Examples of Web services composition that include this recovery mode are found in [VG10, Vat12, CRA13, ACR13, ARC15, ARM15].
- *Local Recovery Mode*: This occurs when individual Web services fail and attempt to roll back to a well-known point in time where it was working properly, without needing other Web services to perform recovery actions. More efforts have been put into this kind of solution. Some examples can be found in [DMM+02, FLLL07, DTTV09, Zha07b, GUR11, MMBJ09, RCA12, MMJ10, MJ13, VGBH11].

Categories of faults. In this survey the faults are classified according to: behavioral aspects and the moment and origin of the faults. According to behavioral aspects, the faults can be grouped into *permanent*, *intermittent*, *transient* and *byzantine* faults.

Regarding byzantine faults, generally these are processes that may behave arbitrarily; these may disseminate different information to other processes, resulting or constituting a serious threat to the integrity of a system [Zha07a]. Transient faults happen once and then disappear, usually after time the system will behave normally. Intermittent faults happen, then they go away and happen again, and so on, and so forth. These faults behave sporadically and are hard to fix. Permanent faults are caused by system components and do not go away until the component is replaced. Therefore, as found in the literature, byzantine faults are rather cumbersome because no assumption can be made about them. These are difficult to trace and are sometimes even undetectable, for instance not knowing which server/service has failed. Regarding transient, intermittent and permanent faults. In fail-stop faults, a component stops working (temporarily or permanently) but it is assumed that any correct component in the system is able to detect it. Therefore, for simplicity and without loss of generality, in the rest of this work we classify the faults according to their behavior, into only fail-stop faults and byzantine faults.

According to the moment and origin of the faults, Chan et al. in [CBS+09] have introduced a taxonomy categorizing the faults into the following classes:

- Development faults, which occur during system development or maintenance.
- Operational faults that occur during service delivery. For instance, as presented in [RCA12, CRA13] where Cardinale et al. consider faults during the execution process of a Transactional Composite Web Service (TCWS). However, it can be considered as a

fail-stop failure since it is detectable and the authors stipulate that they use replacement of the entire failed Web service.

- Internal faults originating within the system boundary. For example, QoS degradation faults due to a lack of resources [MJ13]. Concerning QoS degradation detection, also grouped into fail-stop fault.
- External faults that originate outside the system boundary and are propagated into the system by interaction or interface. For instance, [VG10], external faults may consider integrity attacks to Web services composition.
- Hardware faults that originate in or affect the hardware. One example of this is a system crash [DMM+02]. Other examples that consider this kind of faults are presented in [Zha07b, MMBJ09].
- Software faults that affect programs or data. An example that considers this kind of faults is presented in [Zha07b].

Fault Recovery strategies. Finally, we present the most common recovery strategies found in literature overview, applied to Web services composition.

- *Backward error recovery*: after a failure occurs, Web services are rolled back to an existing point in time where they were functioning properly. These are commonly based on checkpointing mechanisms [DMM+02, VG10, VGBH11, GUR11, CRA13, Vat12, DTTV09, Zha07b, RCA12, MMBJ09, MMJ10, MJ13].
- *Forward error recovery*: the failed Web services are replaced using substitution and/or replacement [SP13], [IP14] of the failing or a subset of the failing Web services.

  In forward error recovery, the system tries to repair the failure without stopping its execution; some techniques include retry and recovery. For example, Shuchi and Bhanodia use substitution of a subset of Web service that contains one or more failed Web services and replaces such subset with an equivalent subset [SP13]; however, this solution is not based on checkpointing. Only some works tackle Web services composition based on checkpointing using this kind of recovery type [RCA12, CRA13, ARC15].

  There are two well-known techniques for fault tolerance in a distributed system: "active replication" and "passive replication" [LML+11].

  - **Active Replication:** Active Replication means creating redundant application servers. When the system receives a request from the client, the request is forwarded to all replicas, for example concerning Byzantine faults [Zha09].
  - **Passive Replication:** Passive Replication means that only one server acts as the primary one to do the assigned job. If it fails, the backup server takes over, for example [YCD+09].

In [MMSZ07], Monser et al. argue that active and passive replication can be done by means of checkpointing. For example, checkpointing is used by replication strategies but in different ways: passive replication uses checkpointing during normal operation. Active replication, on the other hand, do not use checkpointing during normal execution, but uses it to initialize a new recovering replica. This work describes many other technologies to increase the dependability and security of Web services. Regarding checkpointing mechanisms, they point out ways to apply it to a typical Web services architecture; however, it is a merely descriptive work and no evaluation was detailed.

3.1.4 Checkpointing mechanisms and their applicability to Web services compositions

- *Checkpoint*: refers to the information gathered by a processor in a certain time. With such information the processor can return to that checkpoint [KT87].
- Consistent Global Snapshot (CGS): It identifies checkpoints that do not have a causal path; they are not related by a message or a sequence of messages.
- *Rollback Recovery:* it treats a distributed system application as a collection of processes that communicate over a network. It achieves fault-tolerance by periodically saving the states of a process during failure-free execution, enabling it to restart from a saved state upon a failure to reduce the amount of lost work [KS08].

Considering the above premises, the literature presents four different checkpointing types of mechanisms oriented towards rollback recovery, namely, asynchronous or uncoordinated checkpointing, synchronous or coordinated checkpointing, quasi-synchronous checkpointing or communication-induced and message logging. These can save checkpoints on the stable storage or on the volatile storage depending on the failure scenarios to be tolerated.

Asynchronous or uncoordinated checkpointing

Asynchronous checkpointing consists in each participant taking its own checkpoint, its main advantages are that it eliminates the synchronization overhead imposed by synchronizing, and it has low overhead during normal execution. The main flaw of this approach is that it is susceptible to the domino effect; it is also known in the literature as rollback propagation where processes that should not be rolled back are, in fact, rolled back.

This approach is not suitable for composite Web services because the system may revert to an inconsistent state. For instance, let us suppose a bank business process where the bank or the customer exchange money, the system suddenly fails. One possible inconsistent outcome is that a state of an account A was recorded before the transfer to account B; thus, the system may revert to a state that represents losses for the customer or the bank.

Synchronous or coordinated checkpointing

Synchronous checkpointing solves the domino effect flaw of uncoordinated checkpointing since a process always restarts from its most recent checkpoint; however, all processes must orchestrate their checkpoint activity to form a consistent global snapshot. The storage overhead is reduced because in such technique each process maintains only one checkpoint on the stable storage. Coordinated checkpointing guarantees checkpointing consistency in two main ways:

- *Blocking:* All processes must agree on when to take their checkpoints. An initiator sends a control message to all other processes to take their checkpoints. When receiving such message, the process can no longer send or receive messages, then takes a tentative checkpoint and acknowledges the initiator. For Web services, this is unacceptable, blocking incoming and outgoing messages can result in monetary losses.
- *Non-Blocking:* Based on piggybacked information, processes decide when to take their checkpoints. For Web services composition, this technique is compatible; however, it may not be suitable because of the high overhead and the high control information used.

Quasi-synchronous or Communication-induced Checkpointing (CiC)

In quasi-synchronous checkpointing, any of the involved processes take checkpoints based on the control information piggybacked on the application messages they receive from other processes. Upon the detection of dangerous patterns, like Z-paths, forced checkpoints are taken. CiC is another way to avoid the domino effect since it allows processes to take some of their checkpoints independently [KS08]. CiC is a well-known and studied mechanism which takes into consideration the correlation between recovery overhead in case of failures, and checkpointing overhead, in case of the system failure free execution. In the case of Web services, this can be leveraged because checkpoints can be effectively generated, avoiding non-useful checkpoints.

CiC can be implemented as a transparent mechanism, meaning that it does not require modifications to target applications. In the case of composite Web services, the challenge is to leverage the CiC mechanism in order to achieve and reduce to a minimum the causal control overhead sent per message in the communication channels.

The characteristics that should be exploited by CiC for Web services composition are:

- CGSs can be formed easily as the CiC mechanism avoids dangerous patterns and guarantees consistency by means of forcing checkpoints when needed.
- The system will revert to the last CGS; therefore, it does not overwhelm the system with unnecessary storage.

Message logging based checkpointing

Message logging based checkpointing, oriented towards rollback recovery, consists in saving or recording, by each process in a log, all received and sent messages. The main advantage of this technique is that processes that do not suffer a failure do not need to be rolled back and may continue their execution. However, recording so many messages is expensive in practice; therefore, different alternatives of this technique have been proposed. *Pessimistic Logging, Optimistic Logging and Causal Logging.*

Currently, we have found in literature that most solutions for composite Web services either are based on or implement this mechanism [DMM+02, Vat11, Vat12, AM15a, FLLL07].

To conclude this section, one can argue that asynchronous, synchronous and quasi-synchronous checkpointing mechanism requires all participants or processes to rollback. However, some of their advantages are that they will most likely recover from their last known CGS and guarantee a global recovery within a consistent state, except in the case of an asynchronous mechanism.

3.2 Fault tolerance techniques for Web services composition

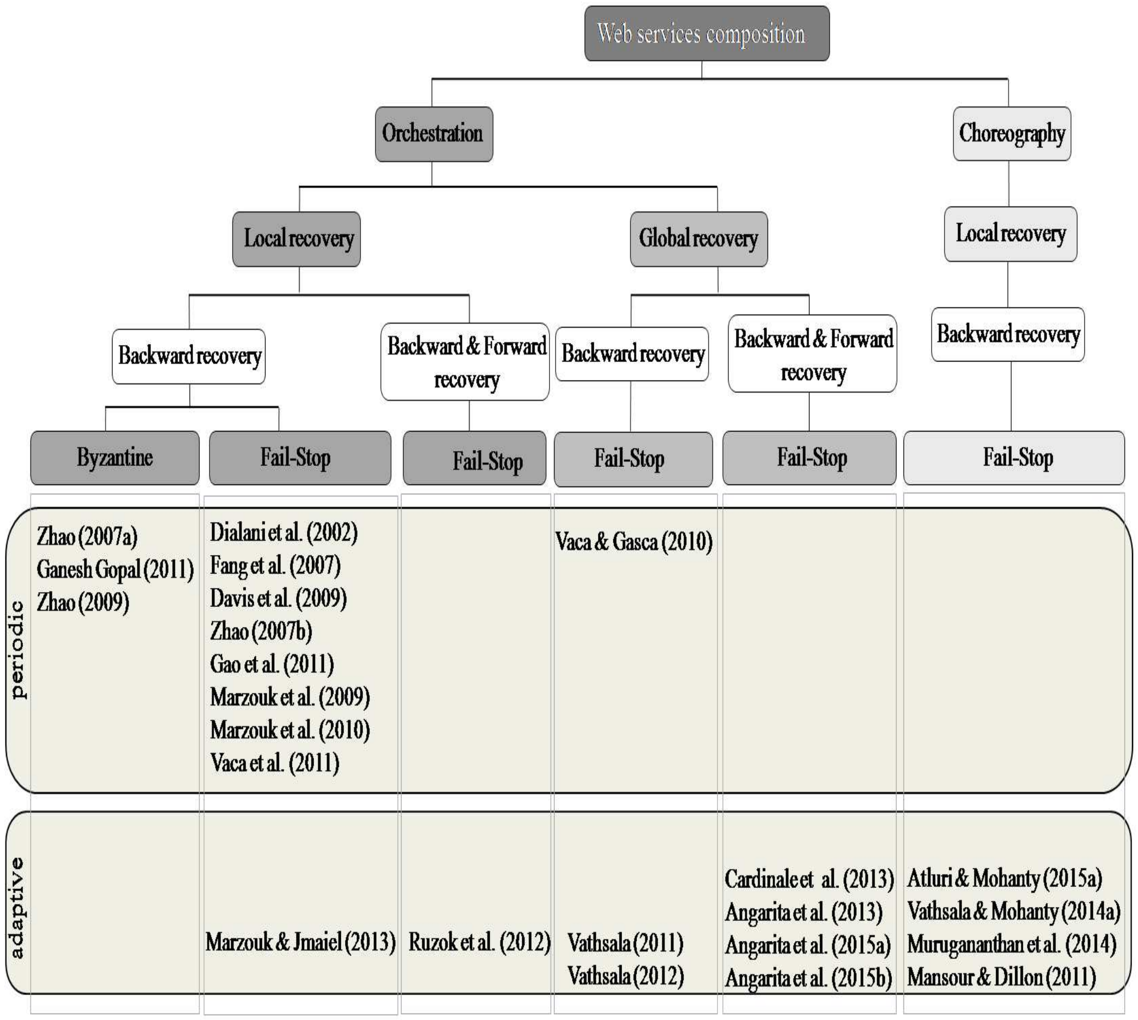
With the proliferation of Web services technology within enterprises, many studies emerged for reliable service composition [IP14] and for composition recovery [GUR11]; nevertheless, there is a need for a new and specific study to classify and give a taxonomy in the realm of fault tolerant Web services that apply checkpointing mechanisms. Therefore, we propose a novel taxonomy that addresses the techniques applied from the perspective of Web services composition paradigms as Fig. 3.1 shows. It is because Web services depend on hardware and software to function properly that the fault tolerance property must be enabled. Fault tolerance is highly desired for Web services composition because it can ensure for long running applications that they are accomplished in a timely manner.

In this section, fault tolerance approaches, drawbacks and issues for many approaches are briefly reviewed in the context of Web services composition for both integration approaches: orchestration and choreography. Fig. 3.1 shows an abstract view of fault tolerance techniques categorized under orchestration and choreography reviewed in terms of a new classification, namely, global and local recovery, as they are the most used fault tolerance techniques found in the literature overview.

3.2.1 Fault tolerance techniques for orchestration

Orchestration has become the predominant standard followed by enterprises for services composition, per se the most followed and applied standard is the Business Process Execution Language (BPEL), although other standards exists like the Business Process Modeling No-

Figure 3.1: Fault Tolerance For Composite Web Services : a Taxonomy.



tation (BPMN), which is the core enabler of Business Process Management (BPM). Both standards specify business rules and the order under which Web services interact to carry out a systems functionality. This section briefly reviews fault tolerance approaches that have checkpointing mechanisms as main core in the context of BPEL and BPMN.

Firstly, we begin by reviewing many works that have local recovery as their main strength .

The first time someone implied that checkpointing was a suitable option for Web services was presented in [DMM$^+$02], where Dialani et al. propose an infrastructure and claim that it is transparent to Web services. Such solution mounted on top of the Web services protocol stack, considering checkpointing based on message logging to restore and/or rollback a single Web service. Yet, the authors also argue that their proposal needs small modifications to recover globally. Nonetheless, such work is descriptive, yet it suggests the use of a local fault manager and global fault manager.

Davis et al. patented a checkpointing technique for long running Web services [DTTV09]. It ensures the survival of Web services when an application server crashes or restart events occur. As stipulated by the author, the Web service state can be “revived" in response to a restart event. Yet, this patent considers individual Web services. They patent the idea that a checkpoint processor can be configured for coupling to individual Web services through a Web services engine. This processor is in charge of running the logic programmed to store and restore the corresponding checkpointed data from each of the failed Web services. Additionally, it manages the corresponding cleaning actions like removing checkpoint data that is no longer needed.

Fang et al. present a framework called Fault Tolerant SOAP (FT-SOAP) based on previous integration middleware as is CORBA [FLLL07]. For interoperability reasons, at the time of writing, the authors examine two implementation approaches: one for SOAP's intermediary, and the other for Axis handler. Checkpointing is based on a logging mechanism which logs incoming requests and checkpoints critical states periodically for backups. However, for the intermediary approach, we found disadvantages like incompatibility, between their SOAP based logging service and other ones that do not implement such service, leading to an inconsistent state after a service is recovered form a crash. Another disadvantage is that, while checkpointing, the primary service is temporarily suspended until checkpointing is completed. Nevertheless, state checkpointing has a great impact on performance. Clients can experience delays while making an invocation to the primary Web services, because of checkpointing its states to its backups.

Wenbing Zhao presents a fault tolerance framework using replication as the main technological approach [Zha07b]. Such work tackles the Web service server side replicating $3f+1$ each client's incoming request. The author proposes to periodically perform garbage collection as not all replicas must be saved all the time. So, when the garbage collection performs its corresponding actions, so does the checkpointing mechanism.

Rukoz et al. illustrate how a checkpoint mechanism can effectively be represented using Petri-Nets [RCA12],providing a fault tolerant recovery scheme. Rukoz et al. propose a three layer architecture: *execution engine, engine thread* and the actual *Web services*, located in the third tier. The execution engine manages the compensation order in case of failure. The engine thread runs a thread; for each peered Web service, it manages the execution control. If Web services fail, their approach is able to monitor and continue execution of the non-fail Web services as far as possible and then resume their execution from the last checkpoint.

Migration and replacement and/or rollback are found in the literature as an attractive way of guaranteeing Web services orchestration fault tolerance as found in [MMBJ09, MMJ10, MJ13].

Marzouk et al. achieve *strong mobility* defined as “enabling a running application component to be migrated from one host to another and to be resumed at the destination host starting from an intermediary execution state called checkpoint " by means of source code transformation [MMBJ09]. They propose transformation rules in order to take checkpoints periodically. They also present three main transformation code aims: to maintain its updated state, to capture and to save the state when a checkpoint position is reached, and to load a checkpoint and to resume the execution starting from it.

Marzouk et al. stipulate that self-adaptivity is needed for applications operating under highly dynamic environments where applications components fail, or sometimes when performance degradations exists causing QoS' degradation [MMJ10]. The authors, identify that other works focus on the unavailability of composite Web services and often use substitution for recovering, causing high overhead. Because other works do not use checkpointing, they have to restart all the orchestration. The authors discuss that their approach pursues the self-healing property; in case of failure, the failed process is migrated to a different server, and in case of a QoS violation, a subset of running instances may be migrated to a new server in order to decrease the initial host load. Marzouk et al. offer a flexible solution at runtime; the checkpointing policy dynamically changes, for instance, whenever the execution context changes an execution manager decides whether to change the checkpointing policy. Nevertheless, a recovery state is built after synchronizing all flow branches. This permits saving a consistent checkpoint. Yet, to our knowledge using synchronization for constructing a consistent checkpoint makes this approach expensive because of the barrier imposed from synchronizing; hence, this solution is slow and lacks concurrency.

The most complete work from Marzouk et al. can be found in [MJ13]. They present both the transformation rules and the *aspects* for strong mobility. They also illustrate that checkpoints can be forced based on policy-oriented techniques. Checkpointing techniques allow saving the state of an orchestration process and roll back to the last checkpoint taken; upon a failure and by making use of *aspects*, source code transformation rules and strong mobility only the non executed code will be resumed and executed. In this work, the authors also take into account the quality of service (QoS), they do so by determining the checkpointing interval based on Markov chains and considering the required QoS of the mobile Web services. This is a sophisticated proposal and the authors present transformation rules, an adaptive dynamic computation of the checkpointing interval, and the selection of the mobility techniques. They use synchronization of parallel branches executed within a BPEL process and their work does not intend to build consistent global states from interacting business processes. Instead, they are able to build a checkpoint state from a single Web service within the BPEL process.

Table 3.1: Checkpointing for Local recovery of Web Services

Reference	Global or Local Recovery	Byzantine or Fail-Stop Faults	Periodic or Adaptive Check-pointing	Backward and/or Forward Recovery	QoS-Awareness	BPEL or BPM
[DMM+02]	Local	Fail-Stop	Periodic	Backward possible Forward	– – ––	– – ––
[FLLL07]	Local	Fail-Stop	Periodic	Backward	– – ––	– – ––
[DTTV09]	Local	Fail-Stop	Periodic	Backward	– – ––	– – ––
[Zha07b]	Local	Fail-Stop	Periodic	Backward	– – ––	– – ––
[GUR11]	Local	Fail-Stop	Periodic	Backward	– – ––	– – ––
[MMBJ09]	Local	Fail-Stop	Periodic	Backward	– – ––	BPEL
[RCA12]	Local	Fail-Stop	Adaptive	Backward and Forward	– – ––	– – ––
[MMJ10]	Local	Fail-Stop	Periodic	Backward	– – ––	BPEL
[MJ13]	Local	Fail-Stop	Adaptive	Backward	Check-pointing based on QoS	BPEL
[VGBH11]	Local	Fail-Stop	Periodic	Backward	– – ––	BPM

Varela et al. argue that companies need to intercommunicate exchanging information between business logics, thus deciding to deploy what is called Business Process Management System (BPSM) [VGBH11]. BPSM helps to automate business processes, but in this context, systems are error prone and cannot guarantee a perfect execution over time. Therefore, a new paradigm called Business Process Management (BPM) arises. It is defined as a set of concepts, methods and techniques to aid the modeling, design, administration, configuration, enactment and analysis of business processes. For the business processes life cycle, the BPM paradigm follows diverse stages: design and analysis, configuration, enactment and diagnosis; however, each stage may introduce different fault kinds. For companies a way to gain dependability in early design stages is indispensable, promoting the reduction of possible faults and risks. In this work, the authors propose to follow traditional or classic fault tolerant ideas such as replication and checkpointing, among others, focusing on the service-oriented business processes context. However, such approach requires the introduction of extra components (sensors) into the business process design, extra time to check each sensor, and the recovery of business process service in rollback.

Table 3.1 summarizes works that are based on checkpointing and are relevant for recovering a single Web service. Thus, it concerns local recovery.

Secondly, we review many works that carry out global recovery, where all participants, in this case Web services, must build and recover from a consistent global snapshot of the system. As an example, one can find that not many works focus their efforts on this kind of solution [VG10, Vat12, CRA13, ACR13, ARC15, ARM15].

Varela et al. identified that while executing business processes, they are susceptible to intrusion attacks, which can be the cause of severe faults [VG10]. Fault tolerance techniques tackle such issues, decreasing risk of faults, and are therefore more dependable, with the aim of achieving dependability before business processes automation. The authors claim that fault tolerance techniques can be applied in order to solve issues related to integrity attacks. Varela and Martínez proposed OPUS, a framework with many capabilities, developed following the Model-Driven Development (MDD) and the Model Driven Architecture (MDA). This framework has four layers: *Modeling, Application, Fault Tolerance and Services.* It is the *Fault Tolerance* layer which is based on checkpointing and rollback recovery. However, the authors do not mention which checkpointing mechanism they use. New and improved checkpointing protocols are proposed in the literature, we believe that the recovery overhead time can be reduced by making use of such improved protocols.

Vathsala et al. propose a way of building global checkpointing of orchestrated Web services [Vat12]. To achieve global checkpointing, they make use of a checkpointing policy. The authors contemplate a global set of checkpoints in order to avoid expensive re-invocation of Web services that are synchronous, and therefore sequentially executed. To generate this global set, the authors compute all possible sequence of calls for an orchestrated Web service. Varela et al. introduce the notion of *Call-based* checkpointing for Web services, thus they employ a set of checkpointing policies. These policies identify the calls within Web services; for instance a *one way* request will checkpoint its state for further use later. Nevertheless, Vathsala et al. address only one instance of the orchestration process, and do not take into account interaction among multi-party orchestration processes.

Cardinale et al. propose a checkpointing approach using colored Petri nets [CRA13]. This work is oriented towards Transactional Composite Web Service (TCWS), which present an atomicity property; such statement establishes an all-or-nothing behavior. In case of failure, their approach relaxes the aforementioned property to a *something-to-all property*. This solution encompasses both forward and backward recovery. This is because a snapshot is taken in by an advanced execution state; however, it must first give a partial result or return *something* to the user. Then for the user to get all later, a possible restart of the TCWS from the last snapshot is executed to complete the result. The main advantage of this work is that checkpoints are only taken in case of failure, therefore the authors claim that they do not increase the system overhead while the execution is free of failures.

Table 3.2: Checkpointing for Global recovery

Reference	Global or Local Recovery	Byzantine or Fail-Stop Faults	Periodic or Adaptive Check-pointing	Backward and/or Forward Recovery	Composite Web Services Approach	QoS awareness
[VG10]	Global	Fail-Stop	Periodic (using integrity sensors)	Backward Recovery	BPM	– – – –
[Vat12]	Global	Fail-Stop	Adaptive	Backward	– – – –	– – – –
[CRA13]	Global	Fail-Stop	Adaptive	Backward and Forward	Petri Nets	– – – –
[ACR13]	Global	Fail-Stop	Adaptive	Backward and Forward	Graphs	Execution Time
[ARC15]	Global	Fail-Stop	Adaptive	Backward and Forward	Graphs	Execution Time, Price, Reputation and Transactional properties
[ARM15]	Global	Fail-Stop	Adaptive	Backward and Forward	Graphs	Execution Time
[Vat11]	Global	Fail-Stop	Adaptive	Backward	– – – –	– – – –

In works [ACR13, ARC15, ARM15] Angarita et al. present a runtime decision-making model that chooses which recovery strategy is best suitable for a Web service within the execution of a Composite Web Services (CWS). The strategies include retry, compensation or checkpointing. In particular [ACR13] presents a preliminary model to select the best recovery strategy in terms of impact on the CWS QoS. The authors extend their work to take into account more QoS criteria to obtain a self-healing model [ARC15], presenting also the impact that different recovery strategies have on QoS and mention that their model chooses the best recovery strategy. Regarding checkpointing, techniques can be implemented to relax the all-or-nothing transactional property and still provide fault-tolerance, allowing users to have partial results and resume the execution later. Finally, in [ARM15] Angarita et al. focus their efforts on providing a general model to support CWS executions, while maintaining required QoS and providing dynamism regarding the selection of fault-tolerance strategies. For all their works, they consider the dynamism of CWS execution, and the QoS's CWS during failure-free execution. Their global solution recovers the entire CWSs. The most recent aim for this kind of solution is to be integrated within dynamic CWS executions while maintaining the required QoS in presence of failures; such solution is automatic and distributed. Fault-tolerant CWS execution is based on transactional properties.

Vathsala et al. aim at providing a way to make Web services orchestration resilient to faults [Vat11]. They propose an adaptive checkpointing policy named "Call Based Checkpointing of Orchestrated Web Services". Using policies the authors adapt the checkpointing rate depending on the mean time between failures and the prediction execution time, a comparison is made and depending on the type of operation carried out during a certain time of the executed orchestration, relying on policies decide whether or not to take a checkpoint. Additionally, reduces the amount of checkpoints. One of the main advantages of this work is that, upon a failure, the entire system does not need to be reseted from the beginning. When Web services within the Web services composition become idle, the latest local checkpoint becomes the global checkpoint of the composed application; and the call-based global checkpoint is defined as a set of latest local checkpoints of each of the Web services that are active during the call. Upon a failure, the application rolls back to the latest global checkpoint and all messages replayed form the message logs. Execution continues without re-invoking the finished constituent Web services.

Table 3.2 summarizes works that are based on global recovery or the overall system recovery that implies using checkpointing mechanisms.

Now we present works that consider a more troublesome kind of faults, specifically those known as Byzantine faults.

Byzantine faults are arbitrary and different users can experience diverse behavior of the system; they are more troublesome than fault-stop. These are only considered by few works [GG11, Zha07a, Zha09] for composite Web services; they implement replicas and fault tolerance mechanism and use checkpointing.

Marimuthu and Gopal consider Byzantine fault tolerance based on replication [GG11]. This kind of works were not feasible due to its runtime efficiency until the introduction of the work presented by Castro and Liskov [CL$^+$99]. Marimuthu and Gopal describe an asynchronous protocol that combines failure masking with imperfect failure detection and checkpointing; however, no implementation detail or performance evaluations are carried out regarding the checkpointing mechanism. This solution encompasses individual requests/responses made to Web services and replicates them $2t + 1$ to mask Byzantine faulty ones; however, this solution does not consider global recovery of the overall system.

In the works [Zha07a, Zha09], Wenbing Zhao presents a fault tolerance framework capable of dealing with Byzantine faults, and not only crash faults. It does so by presenting a framework called BFT-WS that operates on top of SOAP for interoperability reasons and it is based on Castro and Liskov's BTF algorithm for efficiency. The author argues that this framework can overcome Web Services Reliable Messaging (WS-RM) drawbacks. In addition, BFT-WS is backward compatible with WS-RM, and when there is no need to replicate Web service it can run with the default WS-RM. Byzantine fault tolerance is achieved by replicating the server and executing in the same order of all replicas. Regarding checkpointing, it is used for garbage collection, where each replica periodically takes a snapshot of its state. The author adds two additional operations for checkpointing and recovery, namely, *get_state* and *set_state*. In order to update checkpoints while running the BFT algorithm, when a new checkpoint becomes stable, the previous ones along with all the control messages prior to the checkpoint are garbage collected. State restoration is also considered, for instance, when a slow replica has fallen too far behind. Finally, the authors present evaluation of their BFT-WS and claim that it has a low overhead compared to the complexity of this kind of solution. The main difference between [Zha07a] and [Zha09] is that the former and most complete work supports multi-tiered Web services and transactional Web services, while the latter only considers single Web services.

Table 3.3: Checkpointing for Byzantine Faults

Reference	Global or Local Recovery	Byzantine or Fail-Stop Faults	Periodic or Adaptive Check-pointing	Backward and/or Forward Recovery
[Zha07a]	Local	Byzantine	Periodic	Backward
[GG11]	Local	Byzantine	Unknown	Unknown
[Zha09]	Local	Byzantine	Periodic	Backward

Table 3.3 summarizes works that are most relevant for Byzantine faults which rely on checkpointing.

3.2.2 Fault tolerance techniques for choreography

This section briefly reviews checkpointing based fault tolerance approaches in the context of Web services choreography and discusses their advantages and drawbacks.

Composite Web services create complex business processes; however, they are more collaborative than orchestration. Web services are usually provisioned over the unreliable

Internet, and are therefore susceptible to faults, so they must adopt fault tolerance techniques. Nevertheless, in spite of these research challenges, there has neither been much involvement from researchers, nor has it been tackled by the industry.

To make Web services resilient to faults, Vathsala and Mohanty propose recovering Web services, by means of saving checkpoints in message logging [AM15a], considering that only the failed Web services roll back, and it does not cause a chain of reactive services to rollback. Vathsala and Mohanty perform checkpointing of choreographed Web service at three different development stages: design time, deployment time and at runtime.

- At design time, they use the choreography document and introduce checkpoint locations at places where non-repeatable actions take place [VM14a].
- At deployment time, they consider Web services non-functional requirements, such as QoS (response time, reliability, cost of service) and other quantities, like checkpointing time and message logging time.
- At runtime, the authors in a near future will dynamically predict QoS values and dynamic composition of Web services. Therefore, they need response time prediction as presented in [VM12, AM15b].

Table 3.4: Checkpointing for Choreographies

Reference	Global or Local Recovery	Byzantine or Fail-Stop Faults	Periodic or Adaptive Checkpointing	Backward and/or Forward Recovery	QoS-Awareness
[AM15a]	Local	Fail-Stop	Adaptive	Backward	QoS Checkpointing policy
[VM14a]	Local	Fail-Stop	Adaptive	Backward	Unknown
[MVM14]	Local	Unknown	Periodic	Backward	Unknown
[MD11]	Local	Fail-Stop	Periodic	Backward	Unknown

Vathsala et al. identify the most appropriate checkpointing locations by means of their model, where they model the choreography composition as a set of interaction patterns. An introduction to this approach can be found in [VM14b] and details can be found in [VM14a]. Therefore, Vathsala et al. use QoS values obtained from a service, aiming at meeting services execution times and constraint costs. To show the validity of their approach, Vathsala et al. compare checkpointing Web services at design time and deployment time. This proposal aims at introducing minimum overhead during failure free execution, as consequence Vathsala et al. take when possible the minimum number of checkpoints.

Muruganantham et al. address Web services choreography based on an automatic checkpoint algorithm [MVM14]. This approach first locates Web services semantically or based on a semantic search. As a second step, the Web services choreography is composed using AND/OR operators. As third step, Muruganantham et al. developed an auto checkpointing algorithm. Checkpoints are used to mark Web services, if such is executed successfully then the choreography moves to the next operation; otherwise, it restarts from a previous checkpoint. However, this work is merely descriptive and no further details are given. It only presents the system architecture and the rollback-recovery concept to enhance reliability.

Mansour and Dillon propose a new model for Web services modeling the error arrival time as a function of the workload of the server [MD11]. In this work, checkpoints are generated only when the broker realizes that the acceptance testing mechanism is deemed as unacceptable for a Web service of the composite Web services assembly. Checkpoints are associated with initiating a Web service and completion of a Web service. This work considers design errors, hardware server errors and channel transmission errors. The authors are aware that the broker constitutes a single point of failure, to deal with it, they use *Triple Modular redundancy* and *N-version Programming.* Web services choreography is represented by a graph, each node represents a Web service and edges are placed between interacting Web services i.e. i to j. This is done sequentially, meaning that j is executed right after i within the choreography. Using acceptance testing based on positive or negative values of the quantity $E(i, j) = M - R - t$ checkpoints are placed or rollback is executed, where M is the maximum recovery time, R is the actual recovery time and t is the execution time of service j. For instance, if $E(i, j)$ is negative a checkpoint is inserted between Web service i and j and so on for the next sequential task defined in the choreography.

Table 3.4 summarizes works that use checkpointing as their recovery technique within Web services choreography.

3.3 Discussion and open challenges

For this literature survey, we found many papers that describe a diversity of approaches (recovery modes, type of faults considered, etc) highlighting the importance of standardization, since there is no common solution from any of the authors. For instance, fault tolerant mechanisms should be a means by which composite Web services recover: partially, totally, globally and or locally. Not until there is a common agreement among researchers the applicability of the approaches in industry will be hindered. Although all fault tolerance architectures agree on where to place the fault tolerant capability within the Web services protocol stack, a common problem found is that these solutions require special analysis models or familiarity with mathematical models (Petri Nets, Markov Chains).

Despite the fact that taxonomies that classify faults exist, we found in our literature survey that the treated faults are missing in many works.

Therefore, in this survey we propose a novel taxonomy for Web services composition based on the currently most used standards, such as choreography and orchestration. Not many works are currently developed for choreography by means of checkpointing. Those that exist, generate checkpoints automatically and in case of detecting a failure, only one Web service applies a rollback-recovery strategy. Nonetheless, an open opportunity for Web services choreography is global recovery instead of local or individual recovery, contemplating non-functional requirements, such as QoS.

Only few works deal or contemplate QoS while checkpointing Web services composition. In general, in this survey, checkpointing can be carried out periodically; nonetheless, this can lead to inconsistent states or it can be carried out in an adaptive manner. We consider this is the best way of doing so.

Another noteworthy fact is that only one work considers the checkpointing interval as traditional checkpointing mechanisms for distributed systems do. A confusing fact is that most works imply they use checkpointing as means for Web services and Web services composition fault tolerance, but fail to mention which mechanism they use, and whether it is a distributed or a centralized solution. More work needs to be carried out for both orchestration and choreography leveraging quasi-asynchronous checkpointing advantages, such as asynchronous execution and de-centralized nature.

Future trends indicate that there will be a time when choreographies interact against other choreographies. Orchestrations will need to communicate or intercommunicate with other orchestrations and possibly a combination of these aforementioned technologies. Therefore, new and difficult challenges can arise while adopting fault tolerance based on checkpointing mechanisms, which take into account QoS, and checkpointing interval, oriented towards rollback recovery for emerging trends.

Other open challenges include not taking checkpoints at regular intervals of time or periodically, since doing so can revert the system to an inconsistent state. Instead they could depend on the interaction and quality of service among Web services both for existing trends like BPEL and choreographies, as well as for new trends such as interactive BPEL processes. In addition, it is a well-known fact that checkpointing mechanisms have a correlation between recovery overhead, in case of failures, and checkpointing overhead, in case of system failure-free execution. The question that remains as an open challenge involves the quality of service of the involved business processes. Only one work stipulates that they do not incur an overhead during the failure-free execution. Such solution only checkpoints when needed (when faults happen). More challenges include: handling execution programs, partial failures, machine crashes, and conserving data coherency across machines in such situations.

One last note, composite Web services are characterized by their loose coupling, distributed data and distributed components, as well as their asynchronous interactions. However, these are not completely supported by current works. For example, imposing a barrier to synchronize flows inhibits asynchronous interactions among components, which in turn, slows down the system. Another clear example is when works report periodically-saved local checkpoints; this may lead to global inconsistent states. The design of fault tolerant mechanism for Web services composition based on checkpointing presents the following open questions:

To be efficient the following questions arise:

- How often and when must the checkpoints be taken? Most of the works take checkpoints periodically and only some do it adaptively according the system behavior.
- Where or who shall take the checkpoints? Most works rely on proprietary models and there is not a common agreement on such topic.

To be consistent the following question arises:

- Which are the properties that must be satisfied between checkpoints to establish consistent global snapshots? None of the aforementioned works perform a formal verification that they actually rollback to consistent states.

To accomplish the distributed and asynchronous nature of a composite Web service the following question arises:

- Which is the most suitable checkpointing technique that better adapts to the nature of Web services composition? Most of the works are based on checkpointing for message logging; however, message logging has disadvantages like possible rollback to systems' inconsistent global states. None of the previous analyzed works is based on the quasi-synchronous checkpointing technique. Such technique must be explored for fault tolerant Web services composition to leverage their distributed and asynchronous inherent characteristics.

Yet before implementing checkpointing mechanisms, proposed solutions shall have a means of knowing the order under which messages were executed. As distributed systems execution in most cases rely their fault tolerance approaches in such information.

Next section illustrates some of the works or proposals for collaborative environments based on Web services technology.

3.4 Order of messages for Web services based environments

In [WFB$^+$04] the authors propose a collaborative system, a framework based on WS, in particular they implement their solution for conference control integrating various technologies; controlling multipoint audio and video collaborations. In other words, it is a sophisticated way of integrating collaborative applications like H.323, SIP and Access Grid into a single environment. However, they do not take into account that messages need to be properly presented to the end users, since they must have a coherent representation of the data.

Another work which attacks collaborative work environments is presented in [OC09], the authors suggest a framework for the integration of heterogeneous technologies specifically collaborative tools which have the necessity of interacting. Also they want to establish a commonly standardized approach, using Representational State Transfer (REST); an architectural style that specifies constraints applied to WS inducing desirable properties, such as performance and scalability. REST WS aimed at integrating different data models, workflow engines or business rules. However, since the authors use REST, applications run in the World Wide Web using HTTP protocol to transfer data, thus a more robust message ordering is needed for preserving data coherence.

In [HDFD02] the authors propose a coordination protocol for collaborative engineering activities while avoiding erroneous collaboration scenarios in distributed components and applications. Although this work is not based on WS, it is based on causal message ordering, specifically in the IDR reducing the overhead transmitted by each participant. However, WS can provide the interoperability for complex collaborative environments.

Table 3.5: The order of messages for Collaborative Environments

Reference	Aim	Technology used	Environment
[HDFD02]	Avoid erroneous collaborative scenarios	Causality techniques and IDR	Distributed and heterogeneous engineering
[WFB$^+$04]	Integration of multiple collaborative systems, conference (control framework)	Web services	Distributed and heterogeneous video conferences
[OC09]	Integration of collaborative work environments	REST Web services	Distributed and heterogeneous tools

Table 3.5 summarizes the related works. It shows the aim, the technology used and the environment under which the authors propose their solutions. Despite that the different proposals come from the related works, some questions remain open such as: How to integrate Web Services in dynamic environments in an autonomic way without losing the order of the messages otherwise keeping information congruent?

Many other approaches have and still are proposed for message ordering [Sch11], [Cha13], and [Ora14]. However, in [Cha13] to achieve message ordering the author makes use of other software components like a message broker that temporally stores messages and processes them in the order in which they were received. In [Sch11] the author presents a way of message ordering by using labels and proxies; however in communication both ends must agree on the initial label and sequence, then the proxy reads this label and re-orders messages if they arrive out-of-order. The industry has also attacked the problem of message ordering, for example Oracle®[Ora14], proposes a strict message ordering using WebLogic JMS; this is a value added proprietary software. To ensure message order with respect to the processing order for a group of messages, theses are stored grouping them in a single unit called Unit-of-Order.

On the other hand, since there exist a correlation between failure free execution time and checkpointing mechanism overhead, we consider to dynamically generate checkpoints when needed. For such regard, we use fuzzy logic for diagnostic purposes.

Next section underlines fuzzy logic importance within Web services, and illustrates how it is currently implemented by other works. However, we present how fuzzy logic can mitigate practical issues related to autonomic computing, and where it should be placed for this paradigm to properly function.

3.5 Fuzzy logic for Web services

Many Web Services provide similar functionalities, specified in their functional contract Web Service Description Language (WSDL), whereas there are a lot of similar Web services to choose from. However, Web Services non functional requirements are much variable, from users' perspective this can mean getting high or slow responses to their requests, depending on which Web Service each user uses. Fuzzy logic has been a wide area of research it has been used for diverse real world issues, and in Web services one can find different approaches that use fuzzy logic for diverse purposes. On the other hand, the autonomic computing paradigm has many challenges which require to be fulfilled. Each phase from the MAPE control loop has been addressed, by artificial intelligence and soft computing; directly or even as a casualty. For example, the *Monitoring* phase is indirectly addressed in the literature when a set of Web Services that present or share similar functionalities are assessed to automatically choose the best one [KSS15, AAY11, ŞLL10, TT08]; which best meets users' requirements and gives them a better user experience. The analysis and planning phase have also been addressed indirectly, when for example a system is required to adapt to changes in the context, thus the system must react by previously analyzing, and diagnosing the situation. Implicitly, after this the system must plan ahead the corresponding actions in order to adapt to the context, also known as self-configuration/organization [GVAGM10].

On the one hand, fuzzy logic is mainly used for control, on the other hand, autonomic computing deals with systems complexity and aims for self-manageable systems and components. In particular, and for achieving such, the diagnostic process must consider all possible scenarios where uncertainty is a key issue. Thus, more works that deal with the diagnostic process are needed. Therefore, we illustrate how fuzzy logic aids such process. Next, we illustrate related work that cover implicitly the aforementioned phases of the MAPE control loop.

In [TT08] the authors present several works that deal with the Web Services selection problem; Web Services are ranked in order to find the best one that fulfills users' requirements according to QoS presented or perceived. The authors illustrate several techniques for QoS based service ranking, some threat such selection as a composition problem formalizing it as a fuzzy constraint satisfaction problem. Other work uses the QoS based service selection as a fuzzy multiple criteria decision making problem. Last, fuzzy logic is used to evaluate Web Services QoS criteria, where weights are associated to QoS.

In [ŞLL10] the authors propose an architecture and a ranking algorithm for Web Services selection based on ontologies and modifying the Universal Description Discovery Integration (UDDI). The authors store QoS-related semantics in the UDDI data model and make use of that information in the fuzzy selection and ranking process.

In [AAY11] the authors illustrate that from a set of Web Services that share similar functionalities, fuzzy logic can be used for decision model to select the best Web Service among the set; based on the overall performance of the Web Services in the community. Also, they propose an algorithm for ranking Web Services QoS attributes based on dependencies between quality attributes.

In [KSS15] the authors use fuzzy logic for Web Services' selection problem, also considering semantic techniques. Thus they propose a system to dynamically discover and select the best web service that matches the userâĂŹs specified criteria. In other words, the authors incorporate a semantic technology for effectively discovering Web Services based on its functionality, and use fuzzy logic for selecting and ranking the Web Service that were previously discovered semantically.

On the other hand, as autonomic computing has diverse research areas, such as self-configuration, the literature exhibits some works around it based on fuzzy logic. In [GVAGM10], the authors argue that fuzzy logic is used for a middleware that they propose to be self-organized, since it is able to adapt to the environment or context.

Yet, and according to the related work, currently there is no unified or standardized way of addressing autonomic computing, particularly on how to tackle the MAPE control loop. However, artificial intelligence and soft computing principles are commonly used to address problems like detection and diagnosis [KHKS09, HM08]. Also, probabilistic techniques have been used to address the diagnosis process [KDVSD$^+$14, KDVSD$^+$13]; based on a probabilistic model that allows representing the state of the Managed Element from its runtime parameters. The diagnosis can determine when a parameter value is indicating an issue. In these works the authors use a Bayesian network to help define the stochastic properties among the parameters, which is useful to know effects and causes of an issue, besides suggest the adaptive actions required.

Despite the different approaches reviewed from the related work, some questions remain open such as: How to manage scalability of Web Services in an autonomic way to tackle variable service requests? For dynamic environments, how to deal with their non-deterministic conditions?

3.6 Conclusions of the review of fault tolerance for Web services

Although sophisticated solutions exist that merge checkpointing and Web services paradigms and try to tackle Web services fault tolerance, most of these solutions focus merely on a single Web service instance; for example, a Business Process Execution Language (BPEL) process and apply substitution, transformation rules, migration or combinations of diverse approaches. An identified open challenge is to establish consistent global snapshots from check-pointed data for emerging trends such as: interactive business processes (BPEL), interactive choreographies and interactive orchestrations.

We conclude that the price for achieving fault tolerance in some cases affects the scalability of the system or has a negative impact on the systems performance, and oftentimes dynamical environments are not taken into account. In this sense, an open challenge involves the capability of having participants constantly entering and leaving the system. Thus, more work has to be conducted to guarantee Web services composition fault tolerance; which will provide a better quality perceived by the end user and organizations.

Finally, all the above works have been proven worthy of consideration, nonetheless, it would be a milestone for checkpointing mechanisms for Web services composition fault tolerance based on checkpointing if diverse work groups can establish open standards, for instance, regarding when to checkpoint, optimal checkpointing intervals, and QoS driven policies for checkpointing.

CHAPTER 4

FAULT TOLERANCE FOR WEB SERVICES COMPOSITION

Recall, that in this dissertation we look toward dependable ESB system based on autonomic computing and checkpointing mechanisms. At such aim, we address system dependability by means of fault tolerance, however some considerations must be accomplished. First, messages give a view of the interactions that took place, or the execution of how a task was accomplished. It is indispensable to have a consistent view (meaning that the order on how messages were exchanged is important).

Since, checkpointing mechanisms are based on the interactions that took place, they need a means for this ordering of messages to be correct, and coherent. And in practice, causal ordering algorithms have high implementation costs, we follow the immediate dependency relationship (IDR), as it obliterates this notion. Therefore, we focus our effort in efficient checkpointing mechanisms having as basis the IDR, and propose the architecture shown in Fig. 4.1.

We propose to add a fault tolerance layer to the Web service stack, Fig. 4.1. A brief description is given:

- **Discovery Layer**: For Web services there exist a repository called Universal Discovery Description and Integration (UDDI). With this one can find where the service is located.
- **Service Layer**: The service layer usually consists of a Web Service Description Layer (WSDL), which usually describes the Web service functionality.
- **Information Layer**: This layer is in charge of formating the messages. How it should be interpreted by others and in what format.
- **Packaging Layer**: In this layer, the Simple Object Access Protocol (SOAP) is commonly used as it enables cross-platform integration regardless of any programming language. SOAP, defines a modular packaging model and the encoding mechanisms for encoding data within modules.
- **Protocol Layer**: This layer is in charge of the network communications between diverse Web services, they can use different transport network mechanisms: HTTP, SMTP, JMS.
- **Fault Tolerance Layer**: This layer is responsible to deliver high dependability because of the possibility of failures. Fault tolerance techniques usually rely on rollback recovery.

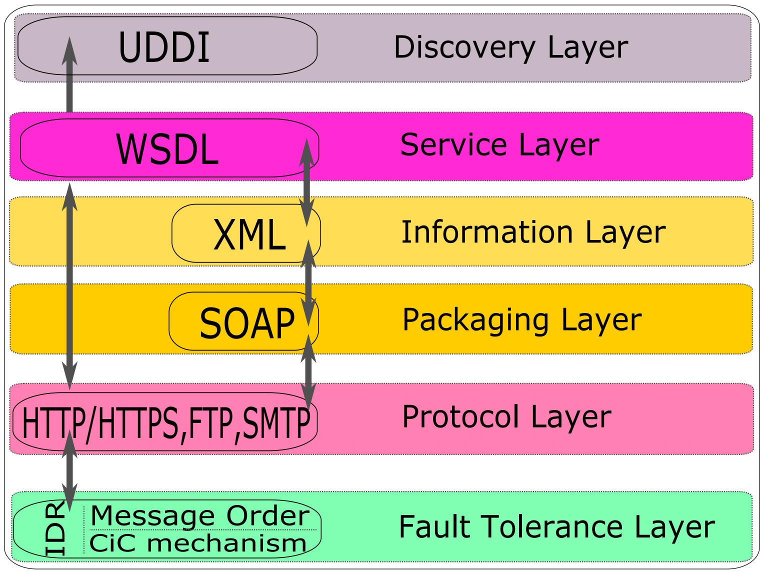


Figure 4.1: Web Service Stack with Fault Tolerance.

This general architecture can be used for any Web service based environment. Regarding the Fault tolerance layer, this relies on rollback recovery mechanism which in turn rely on checkpointing mechanism, and these depend on the order of messages.

As consequence, we propose an Message Ordering Framework, described next.

4.1 Message Ordering Framework (MOF)

Web services are changing the way we see distributed systems since they provide an architecture for integrating applications running in heterogeneous distributed environments; therefore these can be easily integrated, for example by an Enterprise Service Bus (ESB). Today, Web Services are usually applications that describe, publish and are accessed over the Web using open XML standards [CDK$^+$02] [TTMR03], thus, Web Services are the basic compositions for complex business processes. Web Services infrastructure provide a good foundation to build a flexible and extensive exchange protocol [ZJ03]. However, client applications usually use the HTTP as connection protocol when invoking Web Services. But, HTTP does not guarantee message ordering delivery in collaborative environments, plus it does not support asynchronous messages exchange; therefore, a more robust messaging mechanism is needed. In this manner, Web Services can be configured so that client applications can also use the Java Message Service (JMS) as their transport mechanism. JMS can be configured in two different message-based communication styles: point-to-point(P2P) and publish/subscribe. In the P2P style each message is sent to a specific queue from where the receiver extracts their messages. In the publish/subscribe style both publishers and subscribers dynamically publish or subscribe to the content hierarchy. Because of JMS's simplicity, it has become one of the most used solutions for developing scalable collaborative applications.

Collaborative environments and solutions allow users to modify and share knowledge, ideas and information among each other effectively. They have become very popular among organizations because they give greater agility, than those oriented for an specific task within a single enterprise. Also, collaborative environments minimize duplicate efforts and reduce time spent in resolution of issues, giving a better synergy between organizations, and thereby increasing the effectiveness, and efficiency of their collaborations [HDFD02] [WFB^{+}04] [OC09]. Web-based mission critical environments have been a subject of study; in many cases Web Services are used to discover a business functionality, presented as services, that are available through the network and are shared and invoked by corporate partners. Sharing and discovering information in a collaborative context is demanded by the industrial development of dynamic networks. In distributed collaborative scenarios business partners need to share and modify information remotely. For example, consider Fig. 4.2 (showing a product life cycle); a set of aircraft partners need to maintain, support, develop, design and specify aircraft components: gas turbines, engines, cabin, propellers, and wings. These collaborative processes can be represented as Web services and expose their Computer-aided Design (CAD) systems as such.

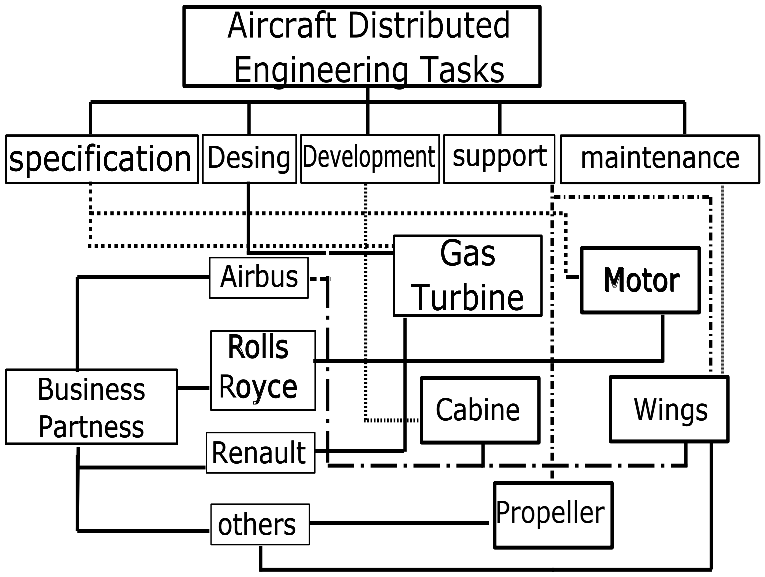


Figure 4.2: Aircraft Distributed Collaboration.

Assuring message ordering for collaborative environments is fundamental since all the involved users should have the same view of the system and data must be presented coherently; additionally the messages provide the expected behavior for the distributed applications. For this purpose, causal ordering protocols are essential for exchanging information, however their implementation is expensive to set up in distributed systems [KS08]. The optimal way of diminishing such overhead is by implementing the *Immediate Dependency Relationship* (IDR). Indeed, the IDR can ensure global causal delivery of messages in group communication and it obliterates the notion that causality can be expensive to implement in distributed systems; it considerably reduces the amount of control information; information carried in each message to preserve data integrity [HFD04a].

4.1.1 Message Ordering Framework for collaborative Web service-based environments

Collaborative environments need a more reliable transport mechanism particularly where the order of the message matters, but JMS alone cannot grant order of messages, at least not for a collaborative environment, for a single process it does. Therefore, the Message Ordering Framework (MOF) is built over JMS, extending its properties. Furthermore, all communications passing through the MOF are enriched with low overhead, consisting on control information, keeping track of the order of messages. MOF uses a messaging middleware, namely a broker, over distributed heterogeneous networks to support the publish/subscribe communication model linking autonomously various publishers and subscribers. MOF does not depend on the Web Services in place; we present the scheme shown in Fig. 4.3 where the ESB system is in charge of the memberships of the users, subscribing and departing them automatically while maintaining causality properties as depicted in Fig. 4.4a for subscription and 4.4b for departure.

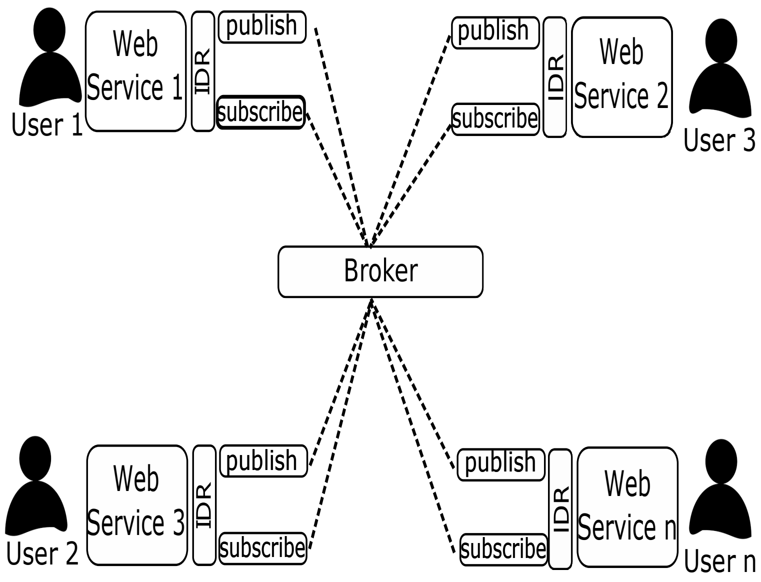


Figure 4.3: MOF Scheme.

Membership subscription

Fig. 4.4a shows the subscription of a new user to the collaborative work environment. When a participant or user wants to join he must send an admission request to the ESB which lets all other users know that a new user is joining with the $join\_request(p_k)$, this being the only non-causal message. Afterwards the new user must wait until a $join\_service(pk, pn)$ is received.

Once received, the new user must wait for the initialization phase $init\_join(p_k, p_i, VT(p_i[i]))$, where he is assigned its vector clock value. Finally, the user must send a $join(p_k)$ to the ESB, which sends it to all other users. Once received, the user is properly integrated to the collaborative environment.

Membership departure

Fig. 4.4b shows the departure of a user from the collaborative work environment. When a user wants to leave the collaborative environment, first he must send a petition request $leave\_request(p_k)$, which is sent by the ESB to all other users announcing that a user will leave the environment. Finally, the user leaving, sends the $leave(p_k)$ message to the ESB; once received by all other participants the user has a proper departure from the collaborative environment and thus causality is preserved among the rest of the participants.

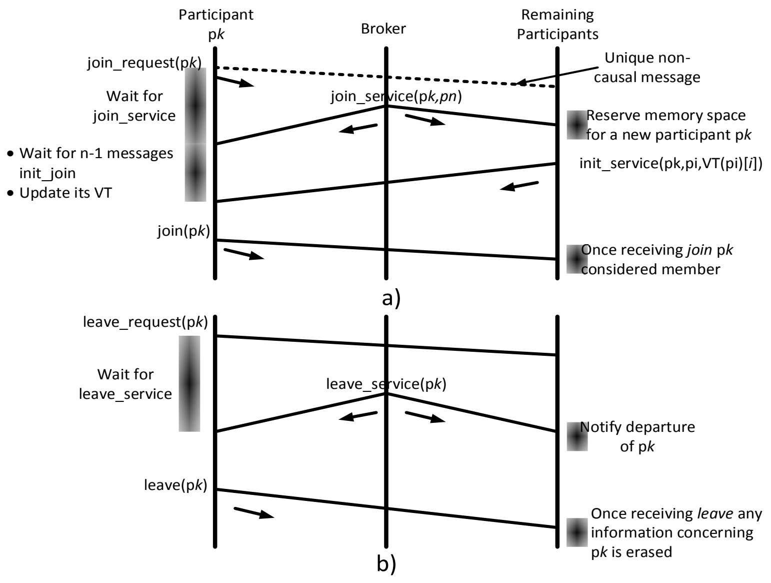

Participant pk
Broker
Remaining Participants
join_request(pk)
Unique non-causal message
Wait for join_service
join_service(pk,pn)
Reserve memory space for a new participant pk
• Wait for n-1 messages init_join
• Update its VT
init_service(pk,pi,VT(pi)[i])
join(pk)
Once receiving join pk considered member
a)
leave_request(pk)
Wait for leave_service
leave_service(pk)
Notify departure of pk
leave(pk)
Once receiving leave any information concerning pk is erased
b)

Figure 4.4: Membership.

4.1.2 MOF's architecture

The aim of the MOF is to propose an extensible framework for the JMS API which can be exploited by Web Services, particularly in collaborative work environments over heterogeneous networks; this is based on the IDR having low overhead and maintaining the information's coherence. A MOF is optimal because it transmits the minimal and necessary amount of control information to completely preserve the causal order among messages or events; also is able to manage interoperability and scalability because it is built for services and the IDR is designed to deal with large distributed systems.

Fig. 4.5a illustrates in detail the communications that take place in a collaborative work environment, where publishing (providers of the service) of a topic/queue takes place and subscribers (consumers of the service) can retrieve collaborative information. In either way publish or subscribe the communication passes through the ESB and then it is delivered to the MOF.

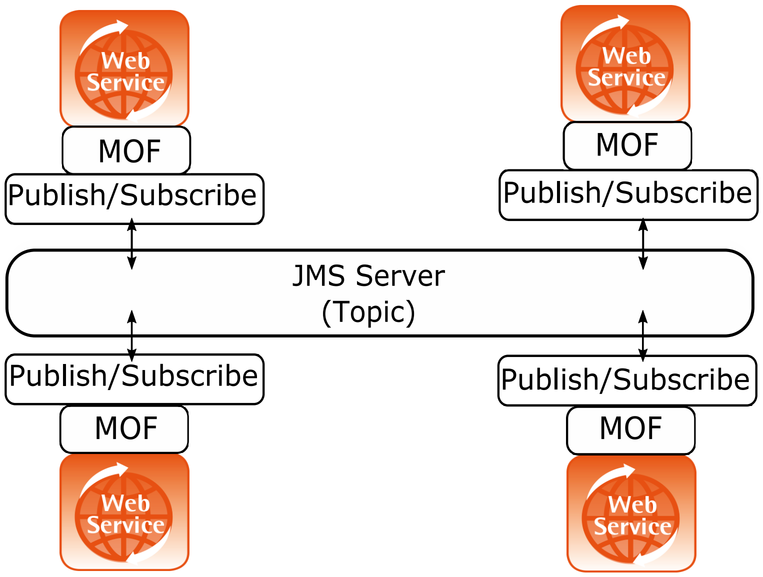


Figure 4.5: Web Services Collaborative Communications

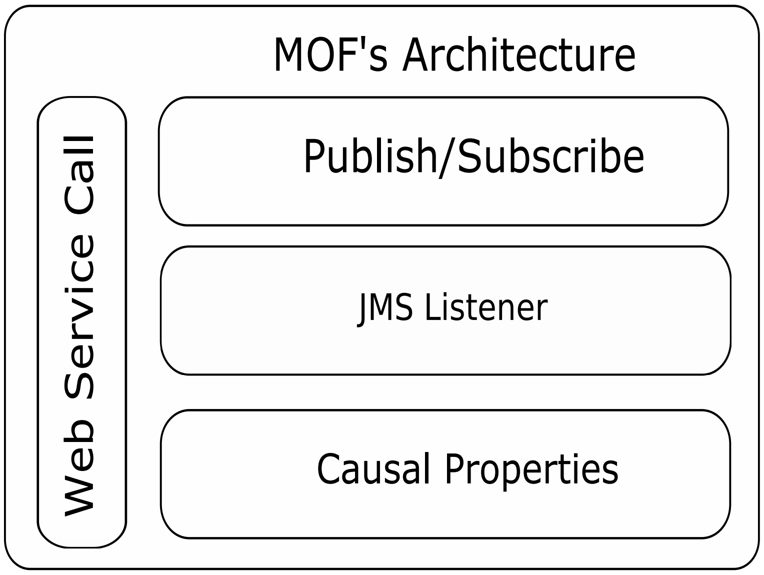


Figure 4.6: MOF's Protocol Stack.

Fig. 4.6 shows the architecture of the MOF. The underlying Web Services are discovered and managed by the ESB, it deploys Web Services as messages to the MOF.

The main actions of the rest of the components are:

- the *Web Services Call* represents that Web Services are usually deployed in remote servers hosted by a third party. The ESB provides the Web Services callback mechanism as a proxy that can be accessed using JMS.
- The *JMS adapter* is needed to communicate with said proxy allowing the exposure of the Web Services as a request that is then passed to the *Causal Properties* component.
- The *Causal Properties* component is in charge of maintaining the order of messages. To achieve such messages requests or replies are enriched with the IDR. To keep it optimal, attaining minimality, timestamped causal information per message corresponds to messages linked by an IDR.
- The *JMS Listener* then listens to events and delivers or queues messages, that is, *subscribes* or *publishes* the message to a queue or topic.

4.1.3 Protocol primitives

JMS API for publishing consists of six steps.

1. Perform a Java Naming and Directory Interface (JNDI) API lookup of the *TopicConnectionFactory* and topic.
2. Create a connection and a session.
3. Create a *TopicPublisher.*
4. Create a *TextMessage.*
5. Publish of messages to the topic.
6. Close the connection, which automatically closes the session and *TopicPublisher.*

The subscription usually consists of seven steps.

1. Perform a JNDI API lookup of the *TopicConnectionFactory* and topic.
2. Create a connection and a session.
3. Create a *TopicSubscriber.*
4. Create a *TextListener* class and registers it as the message listener for the *TopicSubscriber.*
5. Starts the connection, causing message delivery to start.
6. Listen for messages published to the topic.
7. Close connection, which automatically closes the session and *TopicSubscriber.*

Table 4.1: Original JMS interfaces and equivalent.

Original and Equivalent Interfaces	
JMS Interface	Equivalent Interface in MOF
Session	CausalSession

Table 4.2: Original JMS classes and equivalent.

Original and Equivalent Classes	
JMS Interface	Equivalent Interface in MOF
writeMessage onMessage	writeMessageCausal onMessageCausal

For facilitating the task of programmers and for a fast adaptation of antique applications developed, we propose to follow the same dynamic of the JMS API; MOF follows the same structure and philosophy of JMS. The JMS Interfaces that remain the same are: *ConnectionFactory*, *Connection*, *Destination*.

Table 4.1, shows in one of the columns the original interfaces of JMS and the other column the equivalent interfaces provided by our MOF. Table 4.2, shows in one of the columns the original classes and the other column the equivalent classes provided by MOF.

We present the Interface *CausalSession*. It is used for creating two sessions that are used for sending and receiving messages or *publishing* and *subscribing* respectively. JMS imposes some restrictions; one of them corresponds to threading, that is a session may not be worked by more than one thread at a time. Hence in the JMS publish/subscribe style two sessions are mandatory.

Interface CausalSession	
Constructor	
CausalSession(Session session) Creates two separate sessions one for the *publisher* and another one for the *subscriber*.	
Methods	
void	**close**() Closes the causal session
C_TopicSession	**createTopicSession** (boolean transacted, int acknowledgment) Creates two sessions *pubSession* and *subSession*

Now we present the class *writeMessageCausal* and *onMessageCausal*. The main class implements the listener interface, particularly the *MessageListener* which is registered by the *TopicSubscriber*. So when the *TopicSubscriber* created by the *TopicSession* receives a message from its topic it invokes the *onMessge()* method which in our case, invokes the *onMessageCausal()* method.

Similarly the *TopicSession* and the *TopicPublisher* together are used to create the *Message* that is used by the *writeMessage()* method. In our case we invoke the *writeMessageCausal* before publishing a message to a topic.

Class onMessageCausal	
Constructor	
onMessageCausal() Receives messages from the server and verifies causal properties	
Methods	
void	**ProcessMessage**(Message message) Checks for JMS exception
void	**CheckDeliveryCondition** (Message message) Checks for causal order of the arriving message otherwise it queues it

Class writeMessageCausal	
Constructor	
writeMessageCausal() Before sending the message to the server adds its IDR from their neighbors	
Methods	
String	**buildMessage**() Builds a message with vector clock and updated IDR

4.1.4 Mechanism specification for IDR algorithm

Table 4.3: Variable names and type

Variable	Type	Description
i, j	integer	$i, j = 1, 2, \ldots, n$
k	integer	process identifier
t_k	integer	local clock
$message$	char[]	information or data
H_m	integer[]	$H_m = (k, t_k)$
m	char[]	$m = (k, t_k, message, H_m)$
CI_i	integer[]	$CI_i = (k, t_k)$
$VT(p_i)[i]$	vector	vector clock for P_i

Table 4.4: Procedures and description

Procedure	Description
initialization()	All variables are set to zero or empty set.
diffusion(m)	Builds a message m.
send (m)	Sends in broadcast mode a message.
receive (m)	Receives a message from any other process.
queue (m)	Queues a message not received in causal order.
delivery(m)	Users can use data carried by m.

Algorithm 4.1: $TheMinimalBroadcastCausalProtocol(MBCP)$

Input: $message$

Local variables: m is the quadruplet $m = (k, t_k, message, H_m)$ where k is the local process identifier, t_k is the value of the local clock, $message$ is the structure that carries the data, H_m is the immediate history of m.

$VT(p_i)[i]$ is a vector clock, p_i is any process, i, j have values from $1, 2, ..., n$ being n the number total number of processes.

CI represents control information, it is a set of entries $ci_{k,tk} = (k, t_k)$

Output: Causal ordered messages

For more information about variables and procedures, please see Table 4.3 and Table 4.4

```
1  procedure initialization()
2  $VT(p_i)[j] = 0 \;\forall j : 1, 2, ..., n$
3  $CI_i \leftarrow \emptyset$
4  end procedure
5  procedure diffusion($m$)
6  $VT(p_i)[i] = VT(p_i)[i] + 1$
7  $H_m \leftarrow CI_i$
8  $m = (i, t_i = VT(p_i)[i], message, H_m)$
9  Send($m$)
10 $CI_i \leftarrow \emptyset$
11 end procedure
12 procedure receive($m$) for $p_j, i \neq j$
13 $m = (k, t_k, message, H_m)$
14 if not($t_k = VT(p_j)[k] + 1$ and $t_l \leq VT(p_j)[l] \;\forall\, (l, t_l) \in H_m$) then
15     queue(m)
16     else
17         $delivery(m)$
18         $VT(p_j)[k] = VT(p_j)[k] + 1$
19     if $\exists ci_{s,t'} \in CI_j | k = s$ then
20         $CI_j \leftarrow CI_j \backslash \{ci_{s,t'}\}$
21     $CI_j \leftarrow CI_j \cup \{(k, t_k)\}$ $CI_j \leftarrow CI_j \backslash H_m$
22 end procedure
```

Following the IDR principle, proposed by Pomares [Her15], protocols can transmit the minimal and sufficient amount of control information to preserve the causal order of messages.

Messages. A message m exchanged is a quadruplet $m = (k, t_k, message, H_m)$ [1], where:

- k is the local process identifier.
- t_k is the value of the local clock, in the original source.

[1]Each process generates this tuple, when communicating to other process or processes

- $message$ is the structure that carries the data, relevant to each process.
- H_m is the immediate history of m, contains the identifiers of the messages that immediately precede m.

Data structures. The status of a process k are defined by the following structures.

- $VT(p_i)[i]$ is the vector clock established by Mattern and Fidge. The size of $VT_p()[]$ is equal to the number of participants involved in the collaborative environment. The size of the vector is equal to n.
- CI_i is the structure of control information CI_i which contains a set of entries $ci_{k,tk} = (k, t_k)$. Each entry in CI_i denotes a message that is not ensured by participant p_i of being delivered in a causal order.

We present the minimal broadcast causal protocol (MBCP) implemented in the **Algorithm** 4.1. Thus, the MOF extends JMS making it more reliable, and also scalable. Basically the when a user is receiving a JMS message our MOF uses the *onMessageCausal* method and implements the **Algorithm**'s 4.1 steps 12 through 22. When a user sends a JMS message our MOF uses the *writeMessageCausal* method and implements the **Algorithm**'s 4.1 steps 5 through 11.

4.1.5 Experimental results

In order to show that our framework does not have a great impact on the overall performance of the systems, we performed several performance tests. Specifically, we measured response time as a key performance indicator. For performance testing we implement our framework within the following hardware: on a workstation with 16 GB RAM with Windows 7 64-bit as operating system. Two middleware or brokers (Jboss version 5.1 and Glassfish version 4.0) were used, first the WSO2 Application Server was used to deploy to Web services, similarly Glassfish was used to deploy Web services, and for performance tests diverse Java clients were emulated.

We test different Web services capable of interacting with each other, emulating a collaborative work environment see Fig. 4.5. Fig. 4.7 and Fig. 4.8 show quantitatively the variation of the response time; increased according to the number of processes involved, this is to be expected in broadcast mode since the worst case is being evaluated, when all the participants transmit at the same time. In theory the probability of this happening is very low when you have quite a lot processes. This is a problem of the environment in which Web services are collaborating, and working, rather than the proposed approach. In our proposal the response time remains almost constant, as illustrated in Fig. 4.7 and Fig. 4.8, even as the number of processes in the system increases.

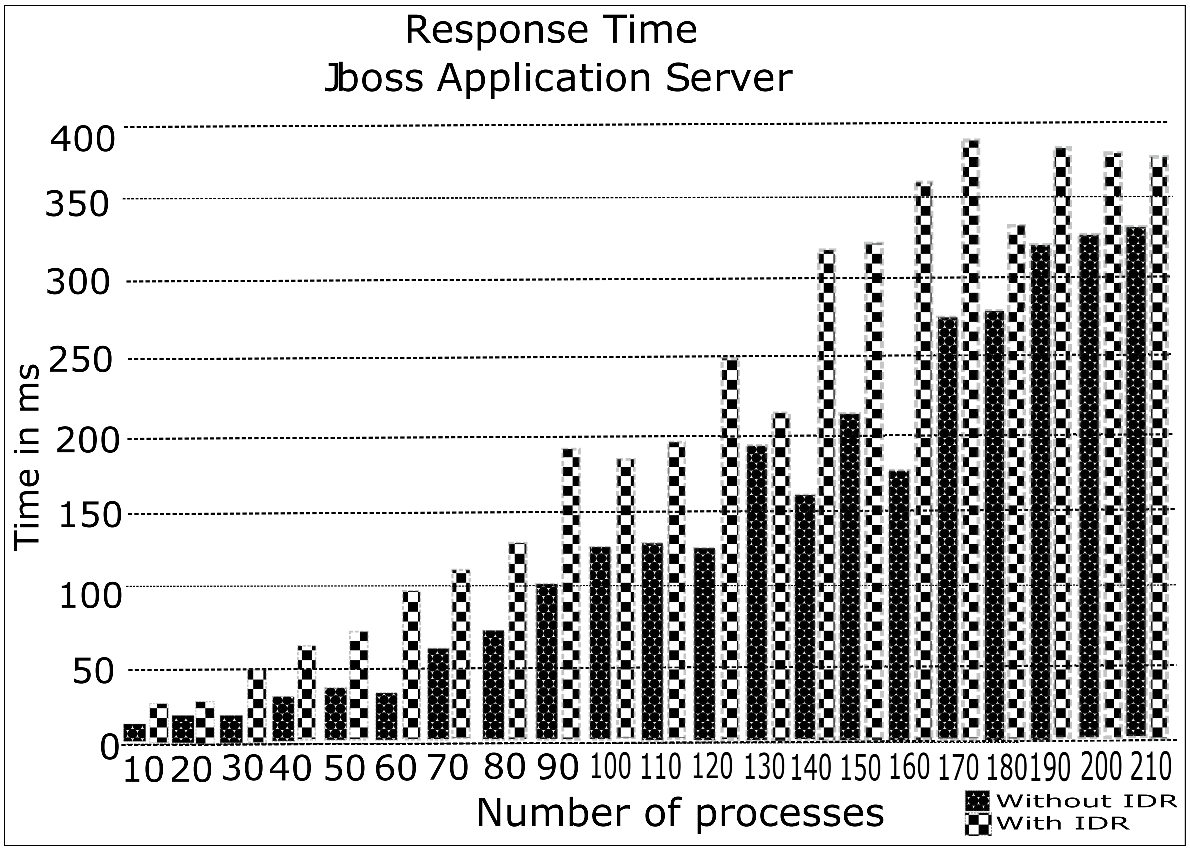


Figure 4.7: Response Time measured when using Jboss Application Server.

4.1.6 Conclusion of the Message Ordering Framework (MOF)

It is vital to respect the order under which messages are exchanged, as these shared functionalities correspond to a common task, by different organizations. The information must respect the order of execution since for distributed systems it represents the behavior of the system.

One of the pioneers in the ordering of messages was Leslie Lamport, he proposes to carry out a strict order between pairs of messages exchanged by different processes. But their the happened before relationship (HBR) is expensive to implement in practice. Therefore different approaches arise, but often they are the company's own software and sometimes they do not always follow standards.

Our approach follows standards, is based on JMS, has interoperability it is oriented to collaborative environments based on Web services. The Web services interface is used to support collaborations between different organizations. In addition, the performance tests conclude that our approach is scalable, since they illustrate a behavior that does not negatively impact the system even when the number of processes involved increases greatly. In other words, the immediate dependency relationship (IDR) becomes acceptable or viable in its implementation. For higher number of processes the system performance is scalable as shown.

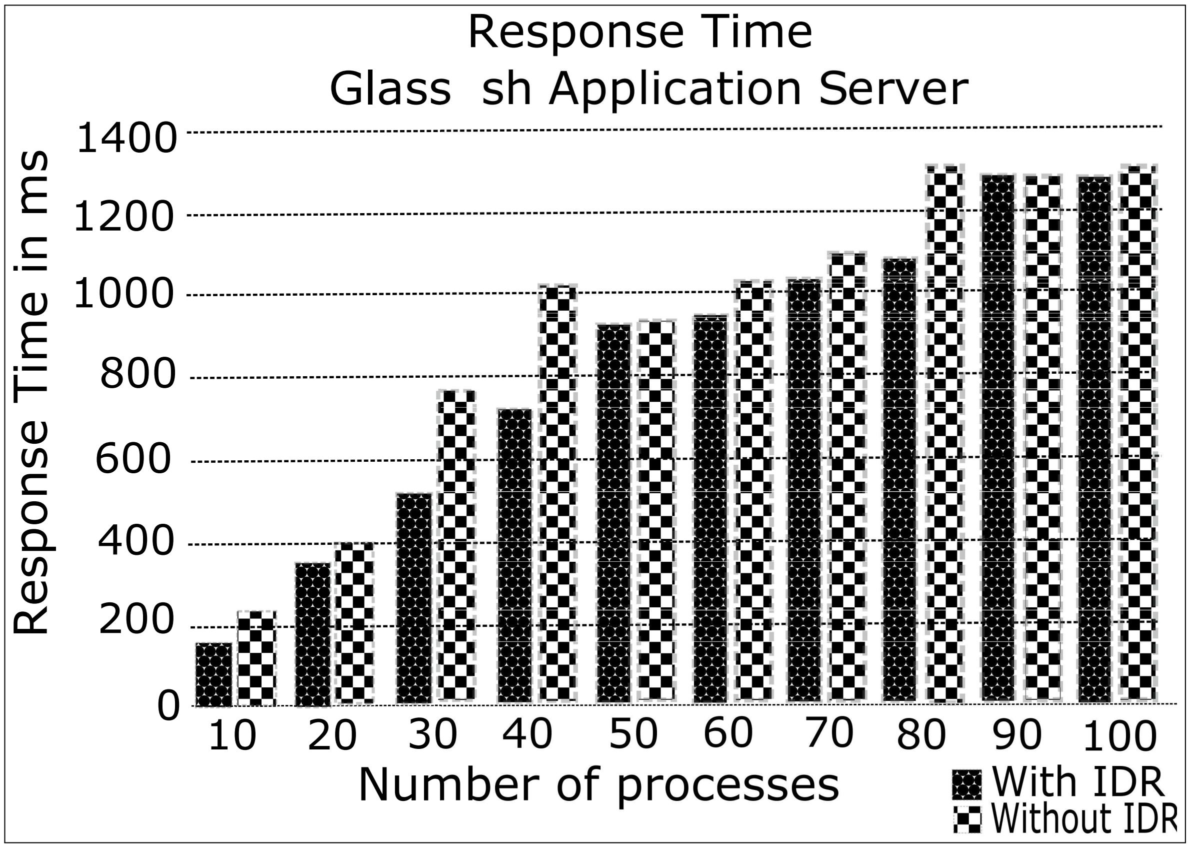


Figure 4.8: Response Time measured when using Glassfish Server.

Recall that the underlying architecture illustrated by Fig. 4.1 is oriented towards individual Web services. On the other hand, in this Section we continue using the same architecture, yet oriented towards a more complex approach, namely, Web services composition.

4.2 Fault tolerance for Web services

With the widespread adoption of the Service-Oriented Architecture (SOA) paradigm most organizations deploy their functionalities and business processes as Web services. Following SOA's architectural model for service composition it allows, even dynamically, the creation and building of intensive distributed software; from a combination of diverse services developed independently [IP14]. These functionalities and services are usually represented by Web services, locally or externally, they can form complex business processes by means of two composition models such as: orchestration and choreography; these arise to mitigate the creation and development of such compositions [RFG12]. For orchestration the predominant standard used is the Business Process Execution Language (BPEL). An important capability of BPEL is that it supports both synchronous and asynchronous communication modes, these communicate among BPEL and/or choreographies at large-scale, situation that is rapidly incrementing both on real world deployments as well as within research community, yet little explored [RFG12]. For example, many BPEL orchestrations may need to communicate with other value added BPEL orchestrations provided by other organizations, as depicted in Fig. 4.9. Such figure illustrates the most basic interaction among two BPEL processes, independently of the internal process each BPEL process carries out, it also illustrates the need for interaction between diverse BPEL processes even if they are implemented on local form or on a external remote place.

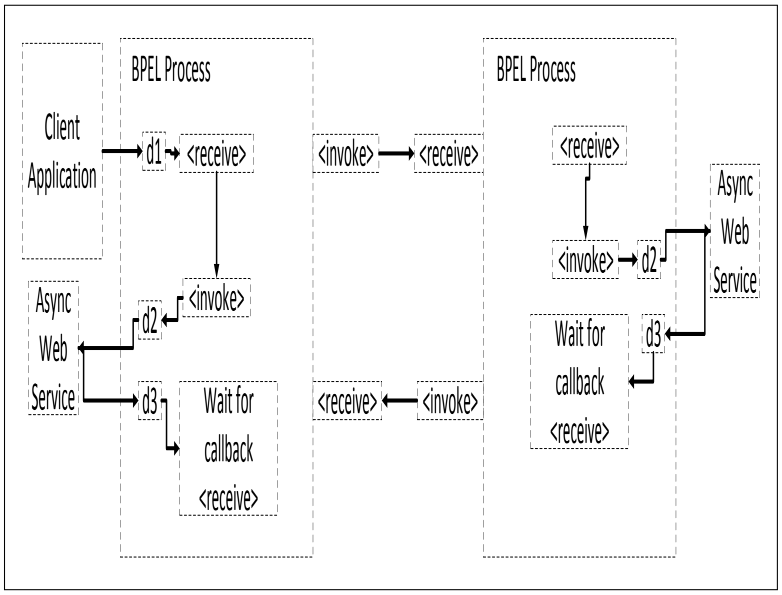


Figure 4.9: Web Services composition Interactions

One can frequently find this kind of interactions on large-scale systems, even when using service compositions models for Web services its complexity due to size and interactive actions is still elevated. Even though Web Services are used within complex business collaborative environments, they are error prone because of unreliable Internet behavior during run-time while they are still required to function correctly and be available on demand. To tackle the fault tolerance problem faced by business processes, problematic organizations require the adaptation of fault tolerant mechanism capable of detecting, diagnosing and repair problems, aiming at increasing systems dependability. Failures may lead to terrible consequences such as augmenting execution time, higher costs to run applications, destroyed systems, or system breaches. As a consequence, organizations must maintain a way to make their systems or business processes as dependable as possible before they intend to automate them [VG10].

Nowadays there are sophisticated solutions for improving system reliability but have some drawbacks, for example, they affect the performance of the systems, they have a high cost of implementation [YCD+09], and/or may endanger the scalability of the system [AH11]. Also, several solutions contemplate only one BPEL process [MMBJ09], [MMJ10], [MJ13] and therefore do not include scenarios where there is interaction; an open challenge to this type of scenario is to increase its dependability, making them more fault tolerant.

We have made a study concerning the checkpointing mechanisms, in order to decide which is the most appropriate on the Web services environment, that suffer faults as: Quality of Service (QoS) violations, inadequate Service Level Agreement (SLA) and temporary software/hardware failures. We found that asynchronous checkpointing mechanisms are not feasible for this context of Web services because this mechanism is highly susceptible to the issue known as the domino effect, i.e. have a high probability that processes may revert to the beginning of their execution [KS08]. The domino effect occurs when cascading rollback happens without finding a consistent point of the system, the worst case is when returning to the beginning of the system execution. On the other hand, there are two known models for synchronous checkpointing mechanisms: blocking and non-blocking. Using blocking checkpointing for Web services represent losses because the ban of request from clients to be processed. Using non-blocking checkpointing to our understanding are not used because of the large overhead introduced and generated. The checkpointing mechanism for interactive Web services environments which supports the described shortcomings, is the quasi-synchronous or communication-induced since it is not prone to the domino effect and generates low overhead, control information, within the system. Such information is used to identify dangerous patterns such as zigzag paths and cycles, in the presence of them the system will not be able to build a consistent global snapshot as stipulated by the authors in [CSPHPC13].

Therefore, we addressed the dependability problem demanded by the new trends, such as interactions between different BPEL processes, interaction between various choreographies or combinations of these techniques, by implementing a fault tolerance layer. Attacking interactive BPEL system fault tolerance, incrementing system dependability, in a distributed and efficient way by using the quasi-synchronous mechanism known as communication-induced checkpointing (CiC). In general, the purpose of CiC is to build consistent global snapshots (CGSs) and to have a checkpoint of each process in a non-volatile storage to survive failures or undo unwanted situations. A CGS can be used within the Web services paradigm for: fault tolerance, software verification and debugging.

4.2.1 Fault tolerance layer based on asynchronous checkpointing

Business infrastructures are more complex and increasingly decentralized among business processes, when they offer a service they are transparent to users who are not aware of all interactions (often asynchronous). This environment, demands robust fault tolerance mechanisms, considering always the consistency of information, that is, scenarios where the order of the messages is important. Hence, it is vital to build a consistent snapshot from which the system can be recovered if a fault occurs. To anticipate these limitations we suggest a robust fault tolerance mechanism, based on the notion of quasi-asynchronous checkpoints where control information enrich messages, in order to keep consistency and avoid dangerous patterns. Each BPEL process has sufficient control information to decide when it is safe to take a checkpoint free of domino effect. Although the implementation of the CiC has a mechanism of generating checkpoints free of domino effect, it is not enough, there is a need also to have a means of collecting checkpoints, i.e., a coherent set of checkpoints to which rollback the system.

Architecture

The proposed approach is suitable for distributed heterogeneous environments, which consists on interactive BPEL processes and need a robust fault tolerance mechanism. We believe that the order and timing of communication events are relevant, therefore we extrapolate the principles of CiC to the compositions of Web services. In general, all communications that pass through the mechanism of CiC are enriched with small overhead, consisting of control information to carry the message order and also for the generation of checkpoints. The architecture is designed to provide interoperability between services and systems having different technologies, through adapters and interfaces using standards-based Web services technology (as shown in Fig. 4.10).

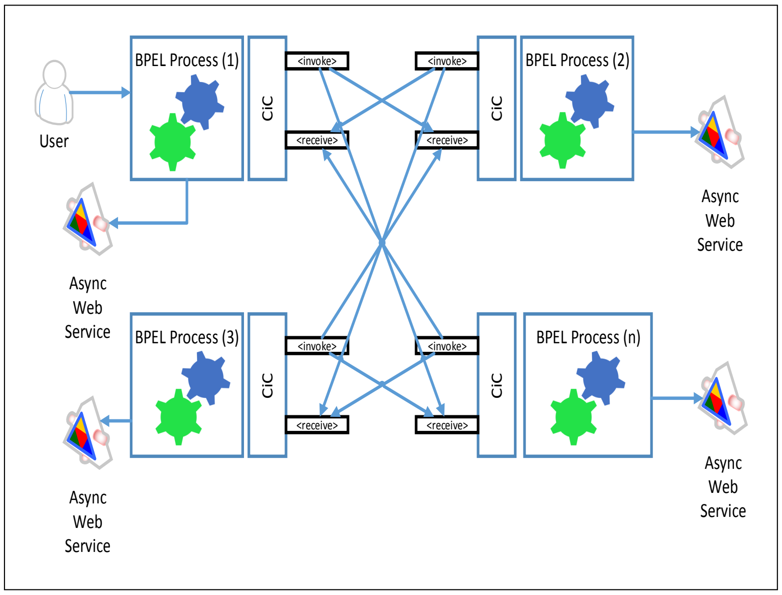


Figure 4.10: Architecture of interactive BPEL Processes.

Furthermore, each BPEL process implements the CiC mechanism, i.e. each request/response made passes through the mechanism CiC, and based on information superimposed within the message exchanged by any BPEL process decide whether or not to take a checkpoint. Because messages allow interaction between various business processes without losing their individual BPEL autonomy, we propose the following architecture, shown in Figure 4.11 at protocol stack level. Where the base is XML, to be independent of the implementation platform. The XML Schema provides the type of system for XML messages. Which they can be transported by any type of communication protocol as it is SOAP. The interface can be represented through WSDL. The behavior for Web services is described by standards such as: BPEL (orchestration) and WSCI (choreography). Finally, it is necessary to have Fault Tolerance mechanism, the former based on checkpointing mechanisms.

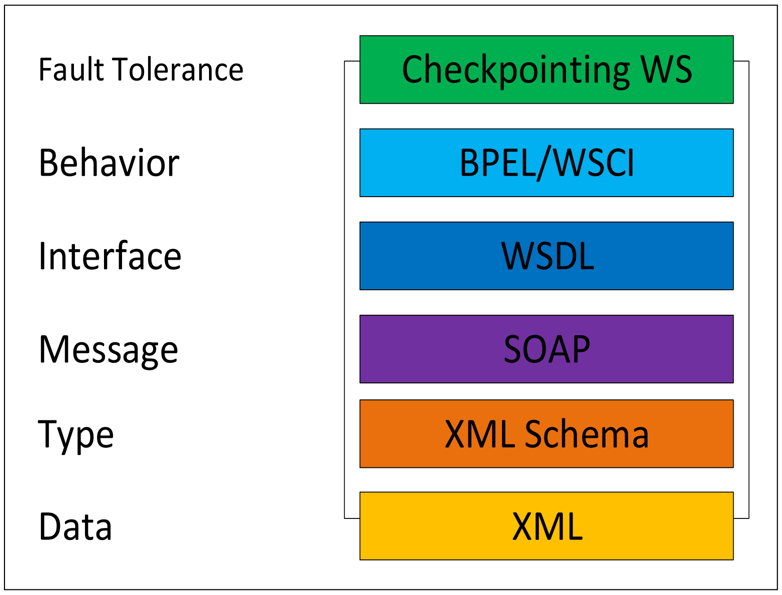


Figure 4.11: Standard BPEL Stack Protocol.

Building CGSs

The checkpoints are properly generated by the mechanism CiC without dangerous patterns as zigzag cycles, zigzag paths and causal paths. Therefore checkpoints are generated without inconsistencies. Now we need a mechanism for collecting a set of checkpoints and capture CGSs, as shown in Figure 4.12. Netzer and Xu [NX95] identified that checkpoints in presence of zigzag paths and causal paths cannot constitute a CGS [NX95].

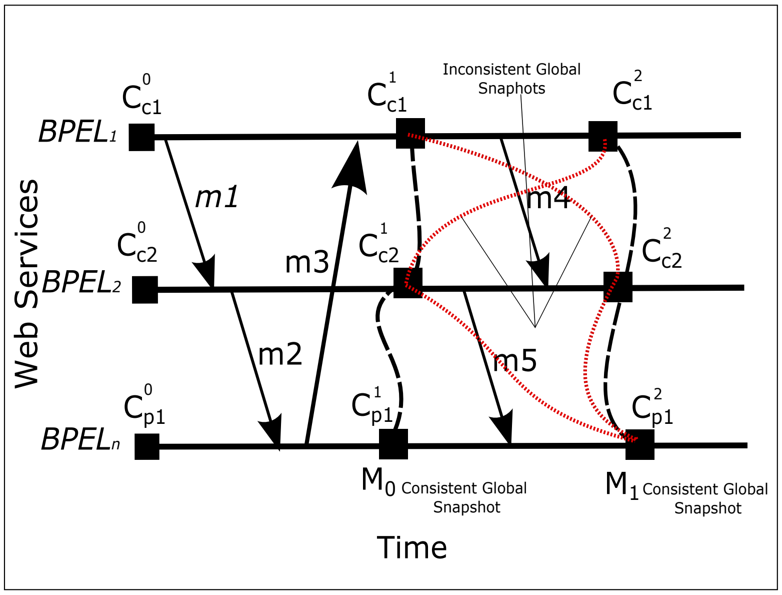


Figure 4.12: Consistent and Inconsistent CGS.

Fig. 4.12 depicts that M_0 and M_1 build CGS while C^1_{c1}, C^2_{c2} and C^2_{p1} cannot be part of a CGS; because of messages $m4$ and $m5$ for instance, although no causal path exists between C^1_{c1} and C^2_{p1} a zigzag path does formed by the aforementioned messages. This means that no CGS can be formed from the checkpoints involved in a zigzag path, in other words no CGS can be built that contains C^1_{c1} and C^2_{p1}. However, the state from which each BPEL process restarts when a fault occurs (after an incident of one or more BPEL processes) must be consistent. Fig. 4.13 shows the interactions made by the BPEL processes, each message passes through the *Web service (CiC)*; within each business process interaction travels a unique identifier. Each message also contains the last checkpoint performed by that BPEL process. To build a CGS, the responsible entity needs at least one checkpoint from each process; in distributed systems checkpoints are collected due to messages exchanged through a communication channel. To extrapolate this concept to BPEL processes, the notion of port will serve the same function as a channel. Through ports, each event (send or receive message) interaction between BPEL processes, is sent in copy to the *Web service (CiC)*.

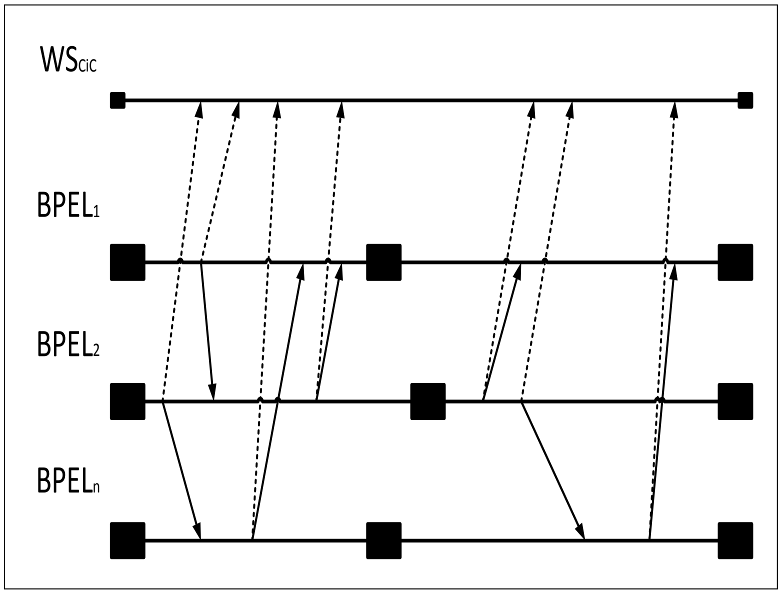


Figure 4.13: Building CGS.

In case that a message is not received by the *Web service (CiC)* from any BPEL processes, represented by $P = \{P_1, P_2, \ldots, P_n\}$ this problem is solved by adding a timer. This timer is triggered after a BPEL process is idle for n seconds, for example process P_i has not exchanged information with any other business process P_j.

4.2.2 Mechanism specification for building CGSs

In order to build consistent global snapshots from previously taken by each process separately, we follow the principles of the IDR and use the following:

Messages. A message m exchanged is a quadruplet $m = (k, t_k, message, C)$ [2], where:

- k is the local process identifier.
- t_k is the value of the local clock, in the original source.
- $message$ is the structure that carries the data, relevant to each process.
- C is the last known checkpoint.

- $VT(p_i)[i]$ is the vector clock. The size of $VT_p()[]$ is equal to the number of participants involved in the collaborative environment.
- $S[]$ is an array that holds n checkpoints, one for each process.
- $Cp_i[i]$ holds the last known checkpoint of a process.

Algorithm for building CGS

It is necessary to reverse the system to a point in time that is consistent, where there are no orphan messages, which are seen as received in a process but not seen as sent.

Table 4.5: Variable names and type

Variable	Type	Description
i, j	integer	$i, j = 1, 2, \ldots, n$
m	char[]	$m = (k, t_k, message, C)$
k	integer	process identifier
t_k	integer	local clock
$message$	char[]	information or data
$Cp_i[i]$	integer[]	checkpoint for P_i
$VT(p_i)[i]$	vector	vector clock for P_i
$S[]$	integer[]	an array that contains a checkpoint for each process
Old_C	integer[]	local variable to save a checkpoint.

[2]Each process generates this quadruplet, when communicating to other process or processes

Algorithm 4.2: Bulding CGSs

Input: $C(p_i)[i]$: is a local checkpoint
Local variables : m is the quadruplet $m = (k, t_k, message, C)$ where k is the local process identifier, t_k is the value of the local clock, $message$ is the structure that carries the data, C is the last known checkpoint, $VT(p_i)[i]$ is a vector clock
Output: S : The consistent global snapshot built from n checkpoints
For more information about variables and procedures, please see Table 4.5 and Table 4.6

```
1  procedure initialization() ;        // All variables are set to zero or empty set
2  $S \leftarrow \emptyset$
3  $VT(p_j)[j] = 0 \, \forall \, j : 1, \ldots, n$
4  $S \leftarrow initial\_CGS(BPEL_1, BPEL_2, \ldots, BPEL_n)$ ;    // Immediately build first CGS
5  end procedure
6  procedure messageDifusion($m$) ;  // A BPEL process uses this procedure before sending m
7  $C(p_i)[i] \leftarrow get\_last\_checkpoint()$
8  $VT(p_i)[i] = VT(p_i)[i] + 1$
9  $m = (k, t_k = VT(p_i)[i], message, C)$
10 diffuse ($m$)
11 end procedure
12 procedure reception($mc$) ;   // A BPEL process uses this procedure for receiving m
13 $mc = (k, t_k, message, C)$ ;              // To ensure causal delivery of m
14 if $\neg(t_k = VT(p_j)[k] + 1)$ then
15     $queue(m)$
16 else
17     $deliver(m)$
18 if $S[i] = \emptyset$ then
19     $S[i] = Cp_i[i]$
20 else
21     $S[i] = UPDATE(C)$
22 if $S == full$ then
23     foreach $i = \{0, \ldots, n\}$ do
24         $S_j[i] = S[i]$
25 else
26     $check\_timer()$
27     $S[i] = ask\_for\_LastCheckpoint()$
28 end procedure
29 procedure UPDATE (C)
30 $Old\_C = S[i]$
31 if $C \in Zigzag$ OR $C \in C\_path$ then
32     $S[i] = Old\_C$
33 else
34     $S[i] = Cp_i[i]$
35 end procedure
36 procedure $ask\_for\_LastCheckpoint$ () ;  // Used when no checkpoint has been received
   from a specific BPEL process
37 $send(m\_checkpoint)$
38 $receive(m\_checkpoint)$
39 $S[i] = Cp_i[i]$
40 end procedure
```

Table 4.6: Procedures and description

Procedure	Description
initialization()	All variables are set to zero or empty set.
initial_CGS($BPEL_1, BPEL_2, \ldots, BPEL_n$)	Ask each BPEL process for their initial checkpoint.
get_last_checkpoint()	Gets last checkpoint known for a BPEL process (because two checkpointing methods: one based on message exchanged and the other periodically)
messageDifusion(m)	Builds a message, adding a vector clock and checkpoint values.
diffuse (m)	Sends a message with its corresponding copy to WS_{CiC}.
reception (mc)	Receives a message from any other process, and verifies causal order is preserved.
queue (m)	Queues a message not received in causal order.
delivery(m)	Users can use data carried by m.
check_timer()	Returns true or false, depending if a timer has expired.
UPDATE (C)	Updates or stays with a checkpoint for a process
ask_for_LastCheckpoint ()	It retrieves a checkpoint for a particular BPEL process.
send ($m\_checkpoint$)	sends a special message requesting a checkpoint value.
receive ($m\_checkpoint$)	receives a special message, that contains a checkpoint value of a BPEL process.

Jointly CiC and the proposed algorithm have the ability to revert the system to the last known CGS and continue operation therefrom.

Algorithm 4.2 can build CGSs from a set of checkpoints taken by the CiC mechanism. The initialization phase, lines 1-5 starts S as an empty set, S stores a set of checkpoints one for each business process. Similarly, the vector time $VT(p_j)[j]$ is set to zero for each process. To preserve causality among exchanged messages, the reception carried out by the *Web service (CiC)* uses lines 12-17.

On the other hand, when a BPEL process sends a message to another BPEL process it also sends this message as a copy to the *Web service CiC*. Which is reflected in lines 6-11, previously both $VT(p_j)[j]$ and $C(p_j)[j]$ are updated, i.e. their vector time and the last checkpoint known from such business process. That is to say, when a BPEL process sends a message it needs first to know which was its last saved checkpoint, line 7. After, for maintaining a causal order between messages exchanged by various processes, each process update their vector time, line 8.

When the *Web service CiC* receives message from any BPEL process first checks that the causality of the system is maintained, lines 11-16. The process for building a CGS, is illustrated in lines 18-28. As a first step, the checkpoint coming from any BPEL process is extracted, as initially S is empty, the checkpoint is immediately assigned, line 19. When S contains a set of checkpoints, one from each process, an initial CGS is built. And when S contains elements the checkpoint value simply updated line 21.

The *UPDATE* procedure, lines 29-35, first verify that the received checkpoint does not belong or is involved in the set of causal paths or the set of zigzag paths, line 31. If the checkpoint is involved in such paths, the variable S keeps the old checkpoint, line 32, otherwise it is updated, line 34.

To verify that a CGS has been built at least one checkpoint from each process is needed, if S is full, line 22, then a consistent CGS is saved as S_j. Being S_j the assembly of CGSs to which the system can be rollback in case of failure. In addition, the situation when checkpoint of a process is not received, is also contemplated; in such case a timer is checked, in the case that the timer runs out, the involved process is requested to send its last known checkpoint, lines 36-40.

4.2.3 Conclusion of the fault tolerance layer approach

We presented an approach on how to construct consistent global snapshots for collaborative environments where there are intercommunications (interactions) among different BPEL processes. It is vital to have robust fault tolerance mechanisms, since without them sometimes to alleviate a wrong situation it is necessary to re-execute the entire business process, which entails high execution times. Checkpointing mechanisms are not directly applicable to Web services and much less to compositions, at the time of writing this work, the need to use the CiC checkpointing mechanisms for interactive BPEL processes had not been considered. The proposed approach does not represent a negative impact on system performance, as it is a scalable, distributed, asynchronous, and low-cost implementation solution.

We can modify our approach to be adaptive, for example, by making use of Quality of Service (QoS) parameters for generating checkpoints. Therefore, it will be an approach that would be activated only when it is detected that the system is not functioning properly, resulting in reduction of dependence between overload during error-free execution and generation of checkpoints.

Chapter 5

Autonomic computing and asynchronous checkpointing

In Section 4, we introduced asynchronous checkpointing for Web services compositions, in this Section we discuss its integration with another paradigm such as autonomic computing. Autonomic computing relies on the MAPE control loop. We identify what parts can be tackled by it, and what parts can be addressed by an asynchronous checkpointing mechanism.

5.1 Autonomic Web services based on asynchronous checkpointing mechanism

5.1.1 Architecture

The proposed approach is suitable to increase system dependability for distributed heterogeneous environments; it leverages the Enterprise Service Bus (ESB) infrastructure which provides an integration backbone for systems integration. Additionally the ESB ensures interoperability and offers several features such as: service discovery, intelligent routing, message processing and service orchestration assuring proper format between service providers and consumers, no matter which programming language they are written on [Cha04].

In general the *functional properties or functional contract* of a Web service are exposed by the Web Service Description Language (WSDL), in other words how the service must behave. Whereas the *non-functional properties* of a Web service are represented by the Quality of Service (QoS) parameters, also for Web service composition, which must be monitored and analyzed in order to conclude if the service is behaving in an adequate form or not. Monitoring an individual Web service and global Web services compositions is challenging because of the particularity presented in each case; since each Web service is unique. Performance parameters can be monitored under diverse scopes, for instance by the client side obtaining parameters like latency, throughput and error rate to name a few.

The architecture is designed to provide interoperability between diverse services and systems having different technologies through standards-based adapters and interfaces that use Web services technology (as shown in Fig. 5.1). Furthermore, each Web service follows the MAPE loop from the autonomic computing paradigm. The service layer represents the invocation or the execution of a required task or service from which performance information will be extracted.

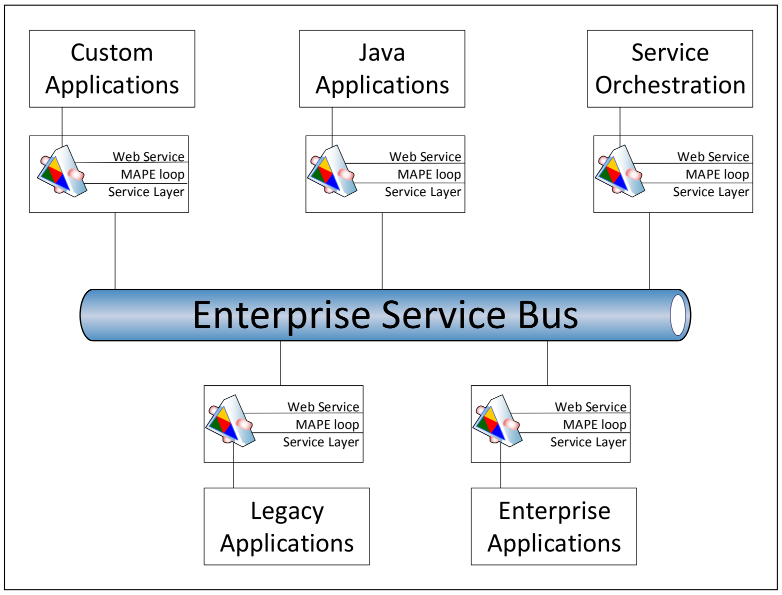


Figure 5.1: ESB with MAPE loop Architecture.

5.1.2 Performance measurements

We proposed an asynchronous checkpointing mechanism to support fault tolerance, therefore, we measure systems' performance before and after implementing the CiC mechanism system architecture.

We have chosen the performance measurements pertinent to network traffic to evaluate the performances before and after applying CiC architecture. We measured the average response time α_t and the throughput β_t, in terms of Transactions Per Second (TPS), for measuring the application services and CiC performance.

Average response time (AVG_{RT})

It is defined as the average time taken by Web service from the time of sending request by a client till the time of receiving the reply from the Application Server. To calculate the response time we use two timestamps, when the client sends a request (t_1) and the time when the response is received (t_2). Then the response time is calculated as:

$$RT = t_2 - t_1 \tag{5.1}$$

This is done for each request/response in the system in play. Latter we average all exchanged messages in the system, as consequence obtaining AVG_{RT}.

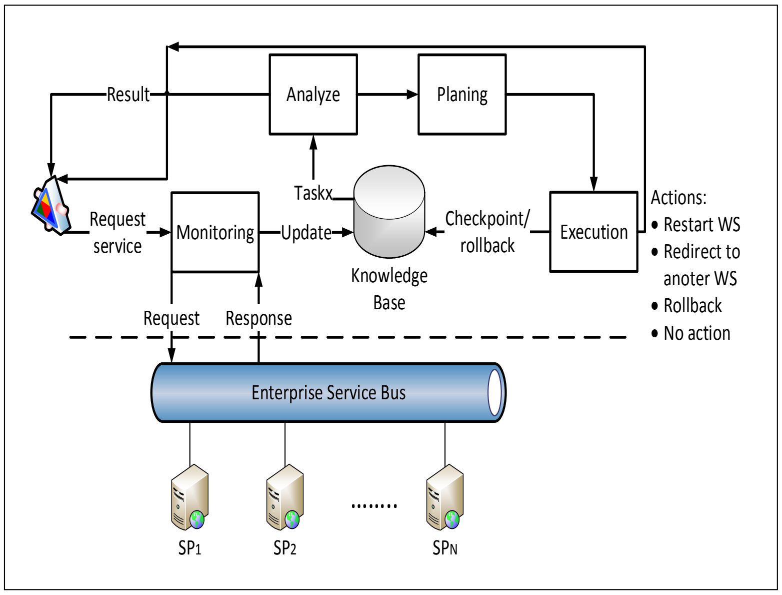


Figure 5.2: Autonomic Web services based on CiC protocols.

Throughput (β_t)

For interactive systems, the system's throughput is defined as the ratio of total number of request to the total time, which has a correlation with response time. We define the Web services systems performance β_t as the amount of data processes by a Web service in a given time interval.

5.1.3 MAPE cycle

Business processes must be dependable so they are available when requested; solutions that suggest augmenting Web services dependability must also be scalable and even autonomic [TZZ+05]. In this work we propose an approach which follows the autonomic computing MAPE cycle based on CiC protocols (as shown in Fig. 5.2).

As illustrated in Fig. 5.2, the *Monitoring* module will initiate the petition, sending a request through the Enterprise Service Bus system; which is in charge of routing, adapting or mediating the request if necessary, in other words the service layer is called. Then the *Monitoring* module computes the QoS parameters as are response time and the performance. These events shall be converted into XML-based messages and stored in a common knowledge base, shared by all Web services. The *Analyze* module uses a *Diagnosis Engine* that checks the extracted information to decide if the behavior of the Web service is normal or if it suffers any anomaly or fault. In other words, it identifies patterns in the logs looking for specific problems that occurred [GZ05]. If the Web service presents normal parameters then such Web service is immediately returned from *Analysis* module to the Web service invoking a service. However, the message is forward to the analysis process that checks the new aggregate values with the aim of predicting the immediate future state of the system, based on Hidden Markov Model (HMM) [HGDJ08], Bayesian Networks [KDVSD+14, KDVSD+13] or any other method that supports prediction. When the QoS parameters are predicted as abnormal, an alert will be sent to the scheduler who will schedule the next forced checkpoint(s). So that later the *Execution* module realizes them. After that if the degradation happens and the system can not continue then we would enter the rollback recovery stage.

5.1.4 Mechanism specification for autonomic Web services composition

The proposed autonomic Web service composition fulfills at runtime the following tasks: first, from the previously built consistent global snapshots it detects when to checkpoint and follows the MAPE control loop for this purpose. Second, it detects when the Web service composition is functioning properly and when it is not.

Our mechanism is specified by the *Autonomic Web service composition* algorithm, based on messages exchanged by individual Web services. A brief description of the main components of the algorithm is provided:

Data structures. The status of a process k are defined by the following structures.

- $WS[i]$ is the structure that holds monitored data about each individual Web service.
- $FWS[i]$ is the structure used for symptoms presented by each Web service, $\emptyset$ means the Web service is functioning correctly.
- $D[]$ is the structure that stores diagnostic information about a Web service.
- $P[]$ is the structure that stores plans for a Web service, based on predefined policies.

5.1.5 Algorithm

We present Algorithm 5.1, that aims at implementing the autonomic computing paradigm and integrating checkpointing mechanism. The aim of our algorithm is to tackle Web services composition.

Table 5.1: Variable names and type

Variable	Type	Description
$WS[]$	integer[]	monitored data
CPU	integer	$0-100$ CPU usage
$Memory$	integer	$0-100$ RAM usage
RT	integer	response time in ms
$FWS[i]$	char[]	stores symptoms of a Web service
$D[]$	char[]	stores diagnostic information P_i
$P[]$	char[]	stores plans for an individual Web service
D_G	char[]	stores global diagnostic for Web service composition
P_G	char[]	stores global plan for Web service composition.
$wsid$	integer	Web service identification
pid	integer	process identification
$wsdescr$	document	Web service description (WSDL)
$wspolicy$	document	Web service policy
$processpolicy$	document	Policy associated to a process
$policy$	document	Set of constraints

Table 5.2: Procedures and description

Procedure	Description
Initialization()	All variables are set to zero or empty set.
Initial_CGS($WS_1, WS_2, \ldots, WS_n$)	Ask each Web service for their initial checkpoint.
MAPE $WS_1, WS_2, \ldots, WS_n$	Each Web service runs the MAPE control loop.
CheckPolicies()	Checks Web services and global policies, and returns symptoms.
Get_Last_CGS ()	Gets the last known consistent global snapshot.
Find_Best_Diagnostic ($FWS[i]$)	Queries the knowledge base for the best diagnostic, that match a set of symptoms S_x.
Classify_Symptoms ($FWS[i]$)	Updates the knowledge base with a set of symptoms and generates a new diagnostic *D*.
Generate_Plan ($D[]$)	Creates a plan *P* for a diagnostic *D*.
Find_Best_Plan ($D[]$)	Queries the knowledge base searching for the best plan.
Build_Global_Diagnostic ($D[]$)	Queries the knowledge base searching for a set of diagnostics and builds a global diagnostic.
Find_Best_Global_Plan($DG[] \cap P[]$)	Queries the knowledge base searching for a set of plans and builds a global plan.
Execute ($action, P_G$)	Executes each action in the global plan set.
Get_Ws_Description($wsid[e]$)	Returns a Web service description document.
Get_Ws_Policy($wsdescr$)	Returns a policy for a Web service.
Get_Process_Policy($pid[e]$)	Returns a policy for a process.
validate ($policy, wsdescr$)	Returns a set of symptoms if a problem is detected by comparing the system and Web service behavior.

Algorithm 5.1 initializes all variables (lines 1 to 4), for instance line 3 builds and assigns to C the systems' initial consistent global snapshot (CGS). The MAPE loop is represented by lines 5 to 24; for each Web service the algorithm starts the *Monitoring* by checking the Web service policies and process level at the time of calling the *CheckPolicies*. Checking that Web service satisfies the constraints stipulated in their policies. During any time of this computation period a new CGS can be build. *Validate* is used to detect a set of symptoms returning a non-empty set when the Web service specifications and policies are not met. For the best case, when the entire composite is functioning correctly a "no problem" is returned

Algorithm 5.1: Autonomic Web Services Composition

Input: WS_i : Web services monitored data such as: $CPU, Memory, RT$,
Local Variables: $FWS[i]$ functioning of a Web service, Sx sympthoms for a Web service, D set of diagnostics from analysis, P set of plans, $wsdescr$ is the Web service functional requirements, $wspolicy$ current policy associated to a Web service, $processpolicy$ rules a specific Web service, $policy$ global policy for Web service composition
Output: P_G : Recovery plan for the Web service composition.
For more information about variables and procedures, please see Table 5.1 and Table 5.2

```
1  procedure Initialization ()
2  $Sx \leftarrow \emptyset$, $P \leftarrow \emptyset$, $FWS \leftarrow \emptyset$ $D \leftarrow null$
3  $C \leftarrow Initial\_CGS(WS_1, WS_2, \ldots, WS_n)$
4  end procedure
5  procedure MAPE($WS_1, WS_2, \ldots, WS_n$)
6  foreach $WS[i],\ i = \{1, \ldots, n\}$ do
7      $FWS[i] \leftarrow CheckPolicies()$
8  if $FWS[i] == 0\ \forall\ FWS[i],\ i = 1, 2, \ldots, n$ then
9      return "No problem found for Composite WS"
10 else
11     Get_Last_CGS(())
12 foreach $\forall FWS[i], i = 1, 2, \ldots, n$ do
13     if $FWS[i] \neq 0$ then
14         $D[] \leftarrow Find\_Best\_Diagnostic(FWS[i])$
15         if $D[] = NULL$ then
16             $D[] \leftarrow Classify\_Symptoms(FWS[i])$
17             $P[] \leftarrow Generate\_Plan(D[])$
18             else
19                 $P[] \leftarrow Find\_Best\_Plan(D[])$
20 $D_G \leftarrow Build\_Global\_Diagnostic(D[])$
21 $P_G \leftarrow Find\_Best\_Global\_Plan(DG[]) \cap P[]$
22 for $action \in P_G$ do
23     $Execute(action, P_G)$
24 end procedure
25 procedure CheckPolicies()
26 $wsdescr \leftarrow Get\_Ws\_Description(wsid)$
27 $wspolicy \leftarrow Get\_Ws\_Policy(wsdescr)$
28 $processpolicy \leftarrow Get\_Process\_Policy(pid)$
29 $policy \leftarrow processpolicy \cap wspolicy$
30 $Sx \leftarrow Validate(policy, wsdescr)$
31 return $(Sx)$
32 end procedure
```

(lines 8 and 9).

Otherwise in order to have a consistent view of the system and to have a proper verification and diagnostic the last known CGS is retrieved from the common *Knowledge Base* (KB), line 11.

Then the vector containing the set of symptoms is *Analyzed* to find the best known diagnosis, retrieved from the KB, this is done for each individual Web service of the composition. When no diagnosis is found, line 15, the symptoms are classified, line 16, and a new diagnosis is generated as well as a new plan, line 17. Contrarily, when the diagnostic is found, line 19, a *Plan* is retrieved from the KB. This is suitable for individual Web service, however concerning a composite Web service the system must build a global diagnostic of how the overall composite is behaving, line 20. The same solution is applied to individual plans, i.e. from a set of individual plans a global plan is generated that constitutes the overall system, line 21. Finally, each action must be executed from the overall global plan carrying out a series of actions, lines 22 and 23, like rollback, restart a specific Web service etc.

5.2 Results and discussion

In order to show that autonomic Web services based on asynchronous checkpointing (specifically communication-induced checkpointing) do not incur in a negative effect on the overall performance of systems, we performed several performance tests. Specifically, we measured response time and transactions per second as a key performance indicators. For testing response time and performance (throughput) we implement our proposed solution using the following hardware: a workstation with 16 GB RAM with Windows 7 64-bit as operating system. The WSO2 Application Server was used to deploy Web services, and for performance tests diverse concurrent Java clients were emulated, in order to have an approximate real world deployment.

5.2.1 Experimental results

Fig. 5.3, Fig 5.5, Fig. 5.4 and Fig. 5.6 show the behavior of the system when evaluating its performance. For this purpose several iterations of the scenarios were performed (in particular each scenario was executed 100 times). That is, for 20 consumers, 2,000 samples were collected, for 30, 3,000 were collected in increments of 10 to reach 200 where 20,000 samples were collected. Subsequently, their average was obtained for the response time and for the transactions per second. The response time measures the time from when the customer sends a request to the credit approval service until the customer receives back his response. Transactions per second measure how many transactions are executed over a certain period of time. We also, illustrate the respective standard deviation for response time and TPS.

From Fig. 5.3 in a quantitative way, it can be observed that although the average response

Table 5.3: Response Time

#of Processes	$RTnoCiC$	$RTCiC$	$Inc\%$	$STDnoCiC$	$STDCiC$
20	11.89	18.89	59%	1.81	3.21
30	11.64	18.91	62%	2.56	6.24
40	12.42	19.36	56%	3.37	10.19
50	13.0	19.82	52%	4.74	18.10
60	14.08	20.83	48%	6.61	16.22
70	15.05	22.12	47%	8.05	20.04
80	15.91	21.52	35%	9.01	21.03
90	16.25	23.65	46%	11.25	28.14
100	16.79	23.58	40%	12.16	30.01
110	17.41	24.70	42%	14.12	24.75
120	18.66	29.52	58%	19.26	29.74
130	21.17	30.15	42%	20.75	24.78
140	23.00	32.14	40%	21.47	31.54
150	22.92	25.81	13%	25.78	27.25
160	22.33	25.34	13%	25.92	28.98
170	22.00	25.19	15%	26.12	28.94
180	19.91	26.26	32%	29.15	29.95
190	24.49	25.92	15%	30.12	32.20
200	23.18	26.16	12%	31.36	33.01

time AVG_{RT} is increased, approximately 30%, this increase is maintained for both 20 customers requesting the credit approval service and 200 clients. We observed during the initial emulation with low number of consumers, the average response time is slightly higher than the existing solution without the CiC mechanism. However, with increasing clients' requests, more users using Web services concurrently, our CiC mechanism performs better with reduced average response time. Interactive environments present diverse challenges, one of them has to due with their resource usage, yet our approach remains constant even when the number of consumers increases.

The emulation results for the systems' throughput (β_t), in terms of Transactions Per Second (TPS), are shown in Fig. 5.4. Its behavior is quite similar to the one exhibit by the average response time. In the early stage of the emulation we observe not much enhancement in β in comparison to Web services that do not implement the CiC mechanism. Nevertheless, when many clients are in play a maximum gain is observed in throughput. Generally, the enhancements in all the performance measurements were observed by applying our approach on the existing Web services considering many concurrent consumers.

The aforementioned figures give guide to argue that our approach is scalable and with low implementation cost (in terms of performance impact). Since the values recommended by ITU G.1010, which attempts to standardize the use of Web services, are not violated for the response time. ITU stipulates the following values: preferred $= 0 - 2$ seconds, acceptable $=$ $2 - 4$ seconds and unacceptable $= 4$ to infinity

Table 5.3 shows the incremented percentage for the response time while adopting or not the CiC mechanism. When 200 clients use concurrently the same scenario, case most seen in real life, saving snapshots from the system incurs some performance degradation. However, we addressed systems' fault tolerance feature by means of an asynchronous checkpointing

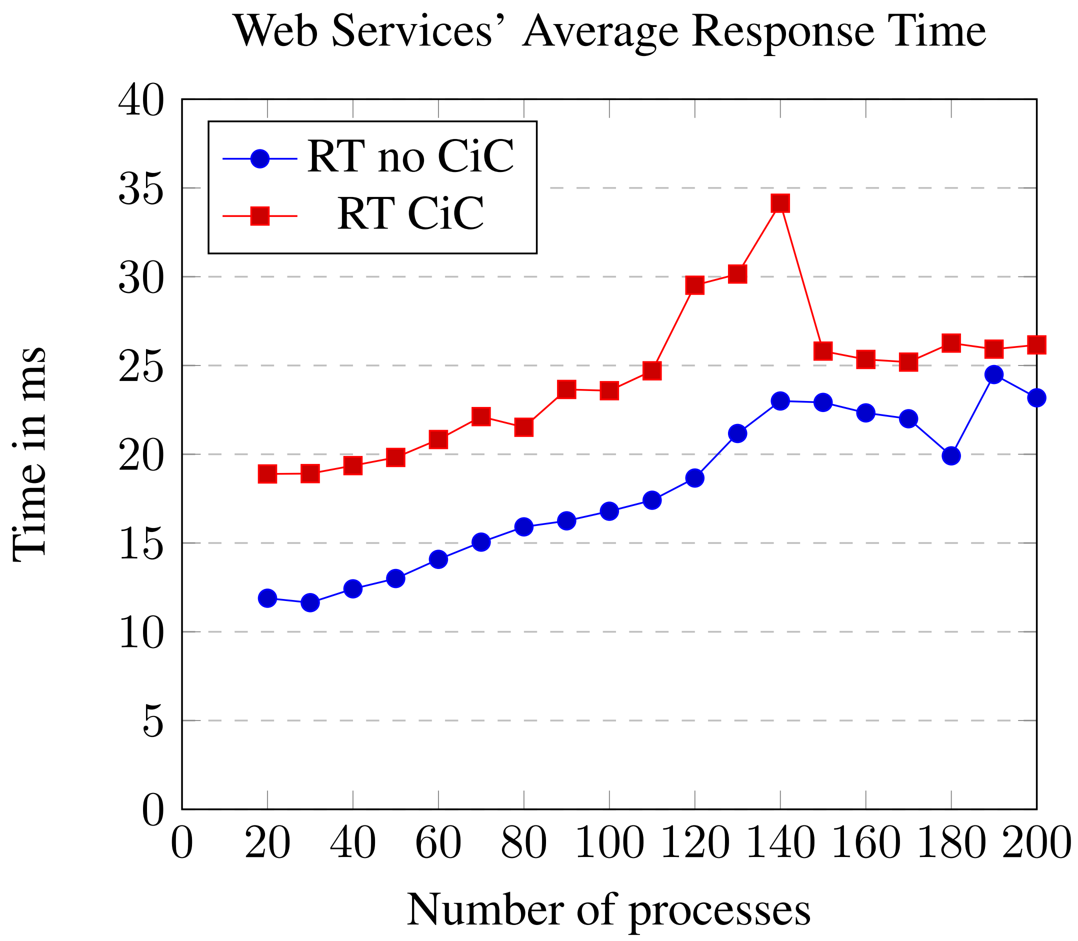


Figure 5.3: Response Time Measurement for the system implementing and without implementing CiC.

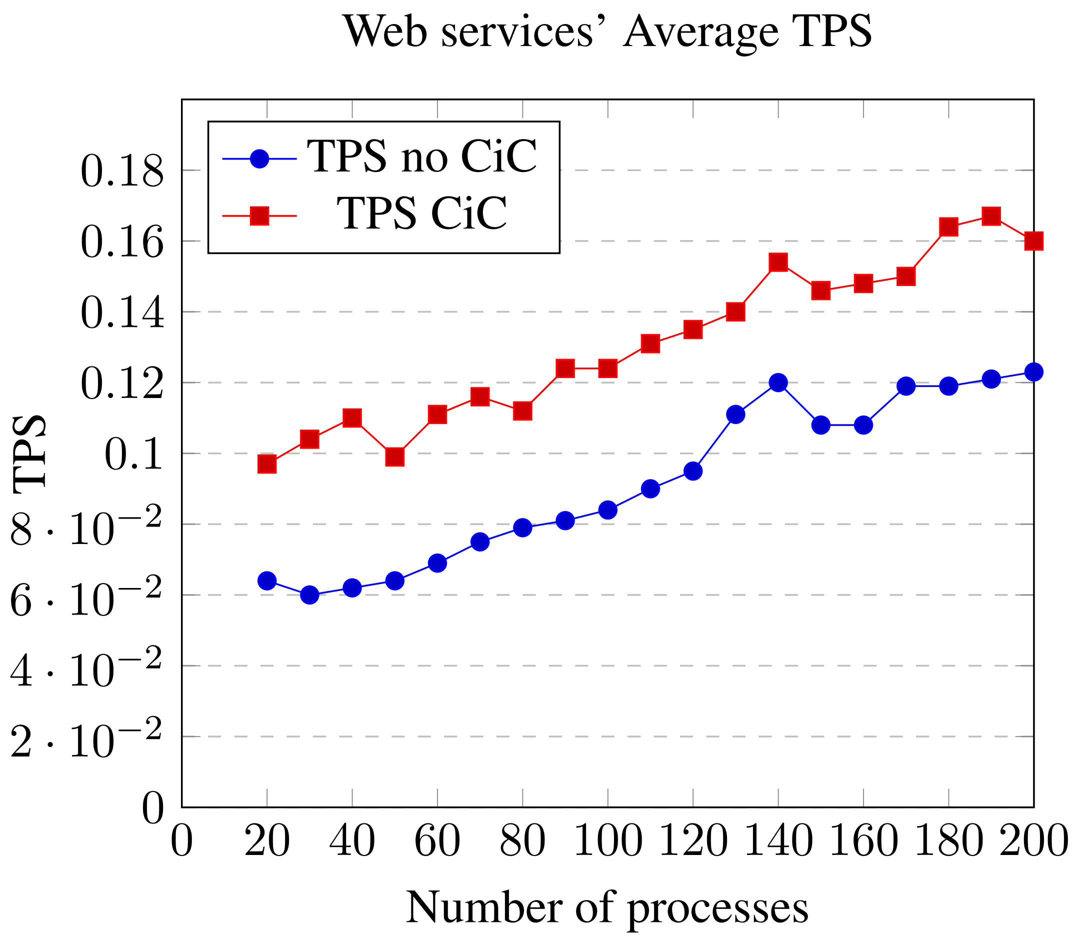


Figure 5.4: Transactions per second Measurement for the system using and without using CiC.

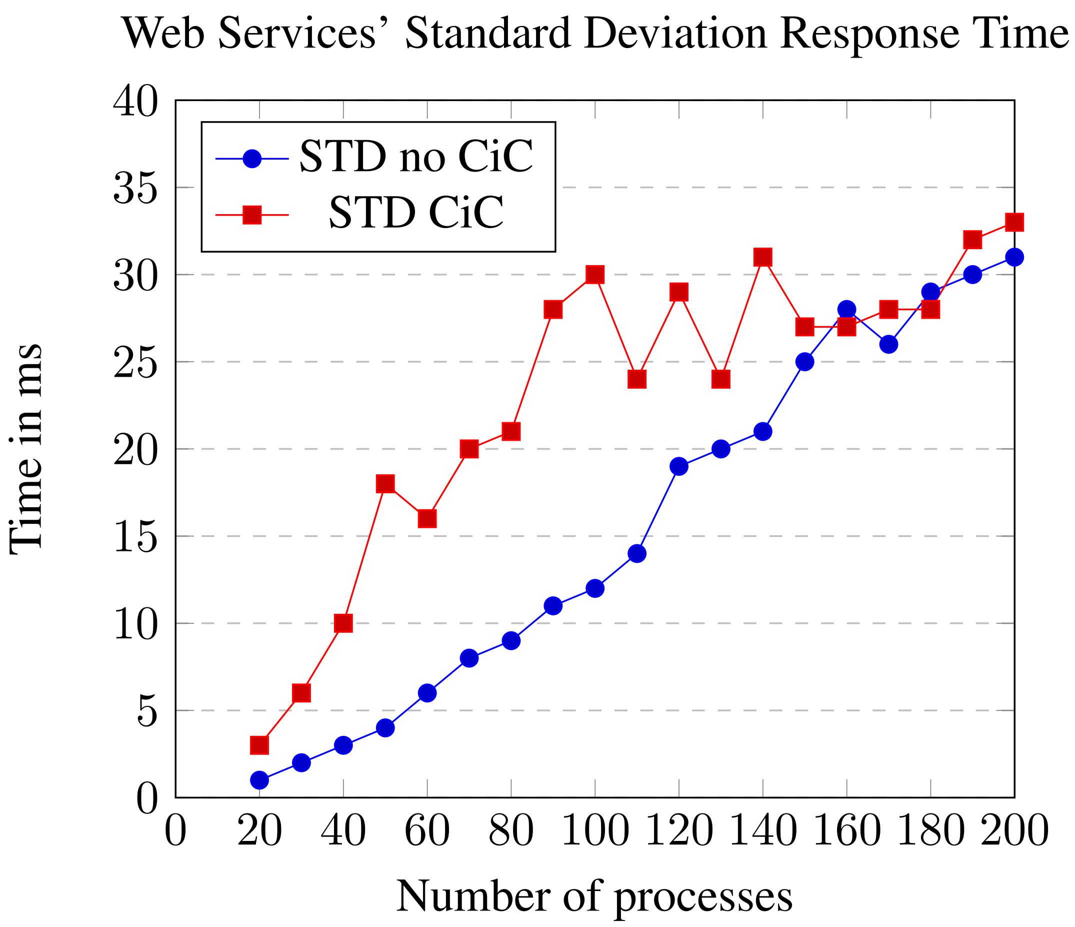


Figure 5.5: Standard Deviation for Response Time using and without using CiC.

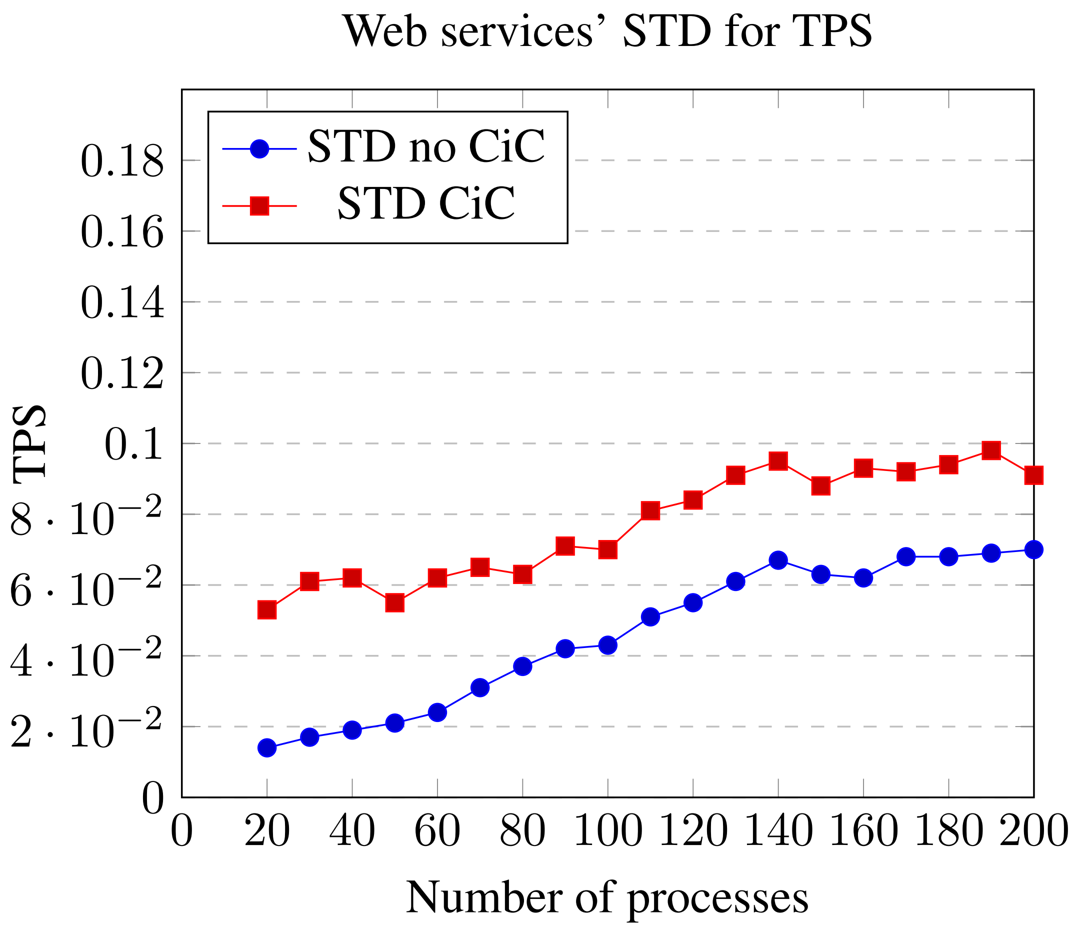


Figure 5.6: Standard Deviation for TPS using and without using CiC.

Table 5.4: Throughput

# of Processes	$TPSnoCiC$	$TPSCiC$	$Inc\%$	$STDnoCiC$	$STDCiC$
20	0.064	0.097	34%	0.014	0.053
30	0.060	0.104	42%	0.017	0.061
40	0.062	0.110	43%	0.019	0.062
50	0.064	0.099	35%	0.021	0.055
60	0.069	0.111	38%	0.024	0.062
70	0.075	0.116	36%	0.031	0.065
80	0.079	0.112	30%	0.037	0.063
90	0.081	0.124	35%	0.042	0.071
100	0.084	0.124	32%	0.043	0.070
110	0.090	0.131	32%	0.051	0.081
120	0.095	0.135	32%	0.055	0.084
130	0.111	0.140	30%	0.061	0.091
140	0.120	0.154	21%	0.067	0.095
150	0.108	0.146	22%	0.063	0.088
160	0.108	0.148	27%	0.062	0.093
170	0.119	0.150	28%	0.068	0.092
180	0.119	0.164	21%	0.068	0.094
190	0.121	0.167	27%	0.069	0.098
200	0.123	0.160	23%	0.071	0.091

mechanism. In this regard, the system can restore its execution from previous executed events, i.e. from a consistent global snapshot.

Table 5.4 shows the systems' performance degradation, here we compare system performance when using the CiC mechanism and when not using it.

Therefore, given evidence from the performance tests we can conclude that system performance is not affected when the mechanism of communication-induced checkpointing (CiC) is implemented, the underlying actions corresponding to the rest of the MAPE loop should be seamlessly executed.

5.3 Conclusions of the autonomic computing and asynchronous checkpointing

We suggested an approach that implements the *Monitoring Analyze Plan and Execute (MAPE)* control loop within Web services based on checkpointing protocols. Afterwards, we presented a Web services compositions to support fault tolerance using an asynchronous communication-induced checkpointing (CiC) which is domino effect free. To prove the feasibility of this using an asynchronous checkpointing mechanism, specifically an algorithm that leverages the communication-induced checkpointing, oriented for Web services compositions interactions. Our algorithm can be applied to Web services since it supports an asynchronous communication and a non-coordinated execution. Our approach reduces forced checkpoints by establishing certain triggering rules that we call safe checkpoint conditions. The results show that the CiC mechanism does not introduce high overhead to current Web services compositions.

Merging these two widely used paradigms, autonomic computing and checkpointing protocols, is a challenge that remains open. However, we showed how a CiC mechanism can help upon autonomic computing. Besides, we consider an adaptive CiC based on the systems performance. Therefore, the implementation of an Autonomic Service Bus (ASB) based on CiC protocols will be taken into account.

Another area of interest concerns optimizing the number of checkpoints, this reduction of checkpoints is considered a good strategy since it reduces a large amount of communication overhead that is generated by the communication-induced checkpointing mechanisms. The aforementioned strategy can be carried out by predicting quality of service variation, which represents how the systems behave during a certain period.

Chapter 6

Dynamic Quasi-Asynchronous Checkpointing for Distributed and Collaborative Environments

Before we define our approach for dynamically checkpointing, we introduce our fuzzy diagnostic model. Then, we define the fuzzy consistency evaluation system on which we rely for the dynamic generation of forced checkpoints.

There is a correlation between the failure free execution time and the overhead introduced by checkpointing mechanisms. As consequence, there is a need to consider it. Organizations that rely on Web Services for their business processes, need not to introduce high overhead. This can be accomplished by making inferences from information gathered autonomously, and with small to no human intervention. At such aim, we propose using fuzzy logic for the diagnostic process brought-up by the MAPE control loop.

6.1 Fuzzy approach towards dependable business processes

Software applications have suffered many changes in the way they are designed; many paradigms arise for integrating diverse technologies and applications in heterogeneous environments [SQV+14]. Currently, most organizations follow and apply the Service-Oriented Architecture (SOA) paradigm, where individual pieces of software and applications are integrated to carry out complex functions; thus SOA standard allows heterogeneous applications to interact. Yet, Internet's unreliable nature not only in its communication channels and protocols increases the need for fault tolerant systems and applications.

In a SOA context large number of concurrent interactions amongst providers and consumers can take place, this competition for resources of shared services can lead to unpredictable conditions or events such as service unavailability, high response time, having as consequence not warranting reliability. To improve both performance and therefore reliability such anomalies need to be addressed by proposing efficient approaches and strategies. Yet, these complex systems can require a lot of human expertise, time and skills for their configuration, for repair and management. Therefore it is mandatory avoiding managing systems manually because doing so becomes more expensive and difficult.

IBM proposed an initiative known as Autonomic Computing which aims at designing and building systems capable of managing themselves, therefore monitoring and evaluating their state periodically for applying changes or taking action to improve its performance, also to recover upon a failure. Other improvements include reliability, security and availability properties [C+06]. Autonomic Computing introduces the Autonomic Elements consisting on Autonomic Manager and components named as Managed Elements. The Autonomic Manager is an entity that manages the Managed Elements. The Autonomic Manager is composed by a MAPE (Monitoring, Analysis, Planning and Execution) control loop with a shared knowledge base, where observations, analysis results and self- managing plans are considered [KDVSD+14].

The goal of this work is to prove the feasibility of using fuzzy logic for ranking and determining QoS behavior for Web services compositions in collaborative environments, and therefore the diagnostic process can formulate corrective strategies following the MAPE loop of autonomic computing. We show how fuzzy logic can be used to implement the diagnostic process for Web services. With different observations made for Web services running under different workload conditions, we built a Fuzzy Inference System (FIS) which is the central part to carry out inferences that allow to discover or predict future performance degradations for Web services.

On the other hand, many different solutions are aimed at rising Web services features, for example, Web services replication [DJ07]. Self-healing Web services [AH11, GZ05, MSSD06] its name is self-explanatory, autonomic Web services [BJPK+05, TZZ+05] aiming for less human intervention among other solutions. In the literature we could also find other works aim for Web services selection problem .

Many works have been issued in order to improve the QoS offered by Web Services by means of Autonomic Computing properties, i.e. implementing the MAPE cycle. Still, there is no unified or standardized form for implementing this cycle for Web Services, and Web Services composition. Some works treat each one of the MAPE loop phases as an individual Web Service [GZ05, KGM11]. Others only tackle a single feature of the autonomic computing paradigm, in particular self-healing [MSSD06, TZZ+05]. Tian et al., suggest to address the entire MAPE loop adding other interfaces to confront functional and non-functional Web Services' requirements [TZZ+05].

6.1.1 Diagnostic model based on fuzzy non-functional dependencies

In a SOA context dynamic and unpredictable conditions must be considered, for example, service consumers and service providers are competing for shared resources as are Web Services. Therefore the Autonomic Manager must deal with such conditions and take corresponding actions. We propose using FIS as knowledge representation of the Managed Element, in order to use this as a model for diagnostic and learning processes. To estimate how Web Services composition's behave one can take into account the fuzzy non-functional dependencies, i.e., quality of service (QoS).

Fig. 6.1 shows the architecture of the Autonomic Web Service. The Web Services instances and the underlying infrastructure are the managed elements. These managed elements are monitored and controlled by Autonomic Mangers via touch-point interfaces.

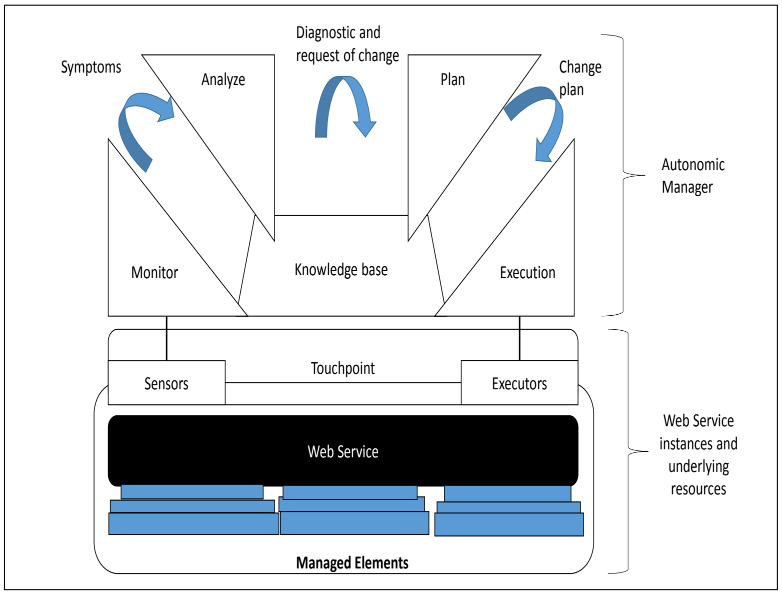


Figure 6.1: Autonomic Web Service Architecture.

The functions carried out by the Autonomic Manager are:

- The monitoring phase takes actions like aggregating data from runtime executions of the managed elements to detect the occurrence of symptoms so they can be analyzed. Unacceptable response times, known as deviations, coming from Service Level Agreements (SLA) established between service consumers and providers are an example of symptoms.
- The analyzing phase uses the diagnosis model based on fuzzy logic. The diagnostic process discovers causes and effects to take corresponding actions, for example, corrective measures. To evaluate Web Services our model has a FIS as a diagnostic model.
- Policies give the runtime configuration for components, and therefore managing of those policies when changes occur. These are defined on a shared knowledge base containing predefined policies, and providing semantics and syntax for the deployed services. To trigger corrective actions information about rules to implement the self-management functions of the autonomic behavior of the Web Services.

Nonetheless, the Autonomic Manager is able to learn about the Web Services behavior through the monitoring. The Autonomic Manager updates the FIS with monitoring data under normal operation condition and also for undesirable situations.

The fuzzy-consistency system

A relation between input and output of non-functional QoS variables can be easily established in the form of if-then rules; the systems' behavior can be described by using simple rules. In this paper, we take advantage of fuzzy logic to measure overall QoS, under different workloads, to manage QoS changes and therefore the diagnosis process.

We measure QoS parameters associated to messages exchanged among different processes, explicitly we propose a QoS temporal association. For this purpose, we define the *fuzzy-consistency system* in such a way that we can know the *degree of system behavior* among messages. Therefore, we compute the overall QoS considering information among exchanged messages and their individual QoS.

The FCSE can be formulated in the form of a *fuzzy-consistency system (FCS)*. Let the FCS relate QoS parameters like: response time, CPU and memory percentage usage in the following way:

"how good the behavior of the systems is at a certain time t regarding QoS parameters".

To achieve this, we define three linguistic variables:

- *response time (RT)* is used to indicate the time it took a process from sending a request to receiving a response.
- *CPU percentage usage* is used to indicate performance metrics during a certain time.
- and the *Memory percentage usage* also used to indicate performance metrics during a certain time.

The state of the FCSE and its degree is determined with the Mamdani fuzzy inference system (FIS) [MA75], a brief definition is given below:

a) *Fuzzification:* For the Mamdani FIS, we fuzzify the inputs and the output using triangular and symmetric fuzzifier (see Definition 2.3). For each linguistic variable, we defined five fuzzy sets related to five linguistic terms as follows:

- for the *Response Time:* $VG(e)$ related to *very good*, $G(e)$ related to *good*, $A(e)$ related to *average*, $B(e)$ related to *bad*, and $VB(e)$ related to *very bad.*
- for the *CPU:* $CPU\_L(e)$ related to *low*, CPU_M related to *medium*, CPU_H related to *high*, and CPU_VH related to *very high.*
- and finally for the *Memory* $MEM\_L(e)$ related to *low*, $MEM\_M(e)$ related to *medium*, MEM_H related to *high*, and MEM_VH related to *very high.*

Table 6.1: Values of variables used in definition of membership functions.

Variable	Values of membership functions			
	Set	L	C	R
Response Time	VG	$\sigma_0 - \sigma_2$	σ_0	σ_2
	G	σ_0	σ_2	σ_4
	A	σ_2	σ_4	σ_6
	B	σ_4	σ_6	σ_8
	VB	σ_6	σ_8	$\sigma_8 + \sigma_2$
CPU	CPU_L	$\epsilon_0 - \epsilon_2$	ϵ_0	ϵ_2
	CPU_M	ϵ_0	ϵ_2	ϵ_4
	CPU_H	ϵ_2	ϵ_4	ϵ_6
	CPU_VH	ϵ_4	ϵ_6	ϵ_8
Memory	MEM_L	$\varphi_0 - \varphi_2$	φ_0	φ_2
	MEM_M	φ_0	φ_2	φ_4
	MEM_H	φ_2	φ_4	φ_6
	MEM_VH	φ_6	φ_8	$\varphi_8 + \varphi_2$

In this way, Table 6.1 resumes the input fuzzy sets for each variable.

The graphical representation of the fuzzy sets are described for each variable, for example, for the response time linguistic variable as shown in Fig. 6.2, and Fig. 6.3 and Fig. 6.4 respectively represent CPU and memory linguistic variables.

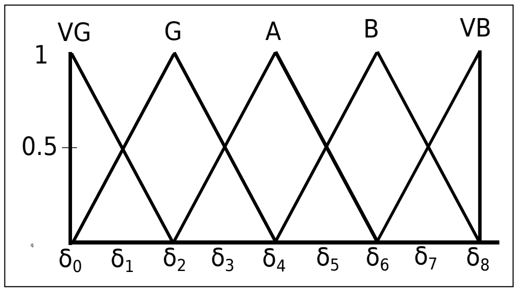


Figure 6.2: Input Fuzzy Set for the Response Time.

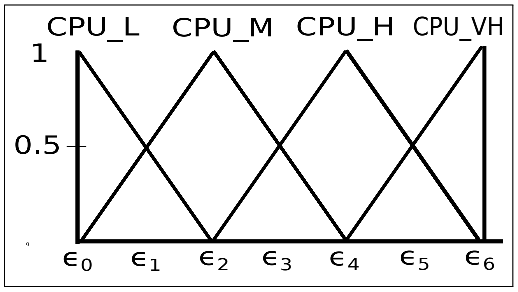


Figure 6.3: Input Fuzzy Set for CPU.

b) *Fuzzy inference:* Once the inputs and outputs of the Mamdani FIS are defined, as shown in Fig. 6.5, the degree of the overall QoS is determined by the inference rules shown in Table 1 and Table 2, presented in Appendix 7.2.

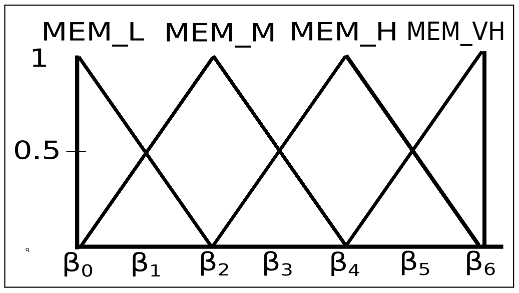


Figure 6.4: Input Fuzzy Set for the memory.

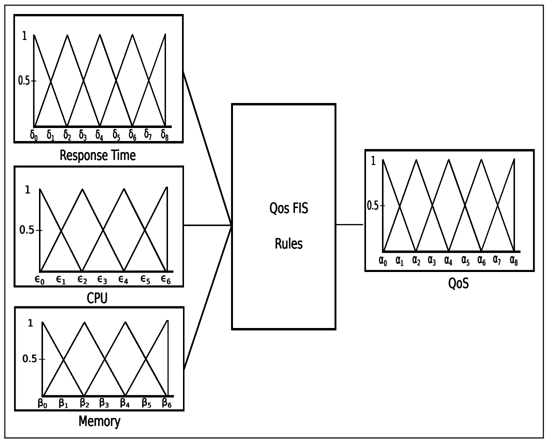


Figure 6.5: Design using Fuzzy inference System.

6.1.2 Experimental results

Experimental setup

In order to show that the diagnostic phase for Web Services based on fuzzy non-functional dependencies do not have a great impact on the overall performance of the system, we performed several performance tests. Specifically, we measured response time as a key performance indicator. For testing we implement our proposed solution within the following hardware: on a workstation with 16 GB RAM with Windows 7 64-bit as operating system. The WSO2 Application Server was used to deploy to Web Services, and diverse concurrent Java clients were emulated, in order to have an approximate real world deployment. We used *Jfuzzylogic* [1] for its compatibility with JAVA.

Fig. 6.6 shows that although the average response time is increased, this increase is maintained for both 10 customers requesting concurrently a Web Service and 200 clients. We observed during the initial emulation with low number of consumers, the average response time for the fuzzy logic diagnostic mechanism is slightly higher than the existing solution for Web services running no diagnostic process. However, with increasing clients' requests, more users use Web Services concurrently, our fuzzy mechanism performs better with reduced average response time. Interactive environments present diverse challenges, one of them has to due with their resource usage, yet our approach remains constant even when the number of consumers increases.

Fig. 6.6 and Fig. 6.7 shows the behavior of Web Services when evaluating performance, showing response time and the standard deviation respectively. For this purpose several iterations of the scenarios were performed (in particular each scenario was executed 100 times). That is, for 20 consumers, 2,000 samples were collected, for 30, 3,000 were collected in increments of 10 to reach 200 where 20,000 samples were collected. Subsequently, average

[1]http://jfuzzylogic.sourceforge.net/html/index.html

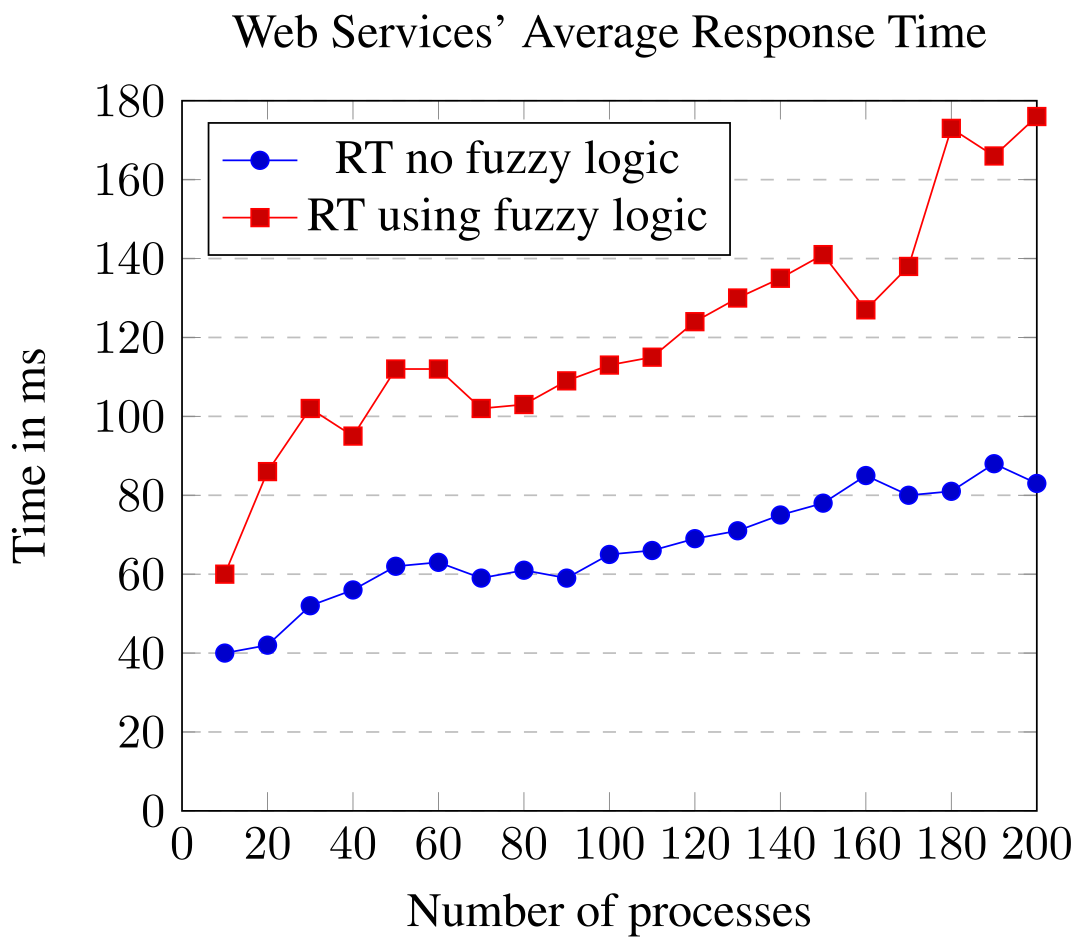


Figure 6.6: Response Time Measurement for Web Services using and without using fuzzy logic.

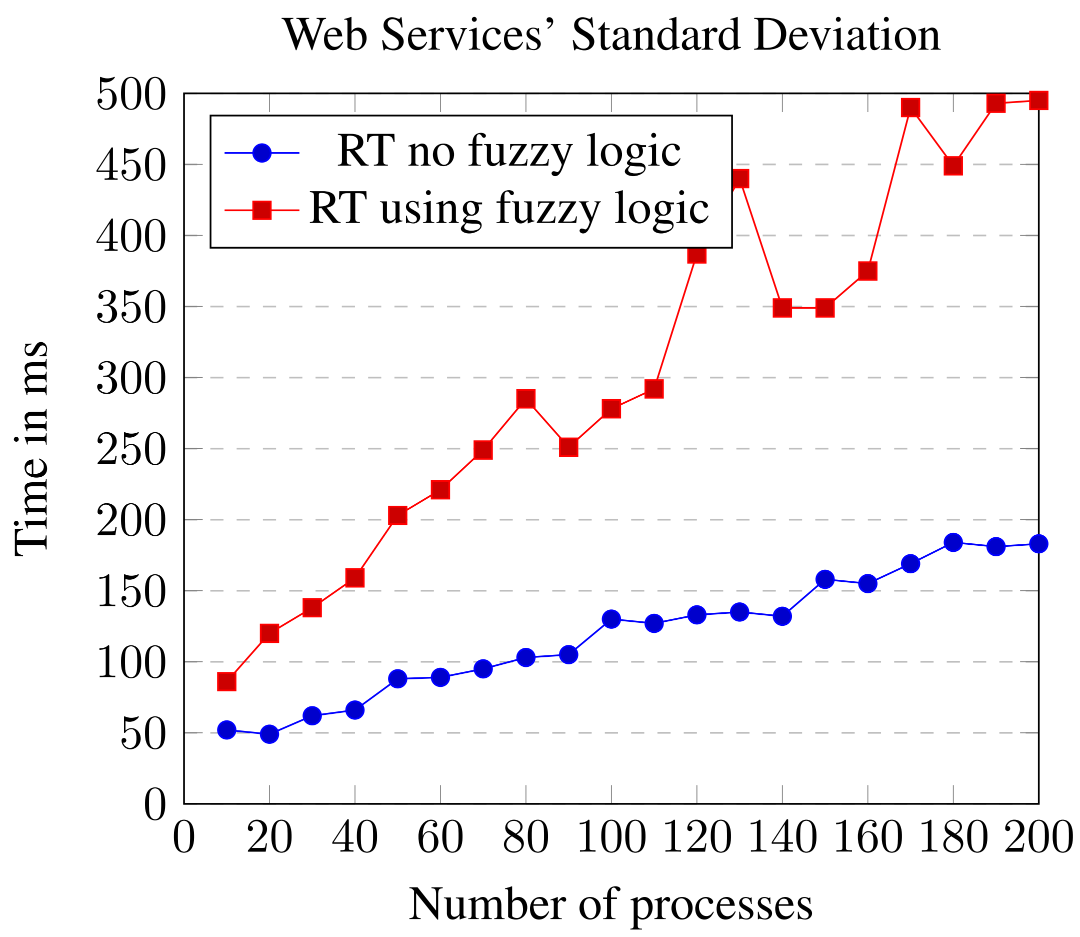


Figure 6.7: Standard Deviation for Web Services using and without using fuzzy logic.

Table 6.2: Values for input/output membership functions.

Variable	Values of membership functions			
	Set	L	C	R
Response Time	VG	−2.5	0	2.5
	G	0	2.5	5
	A	2.5	5	7.5
	B	5	7.5	10
	VB	7.5	10	12.5
CPU	CPU_L	−33.33	0	33.33
	CPU_M	0	33.33	66.67
	CPU_H	33.33	66.67	100
	CPU_VH	66.67	100	133.3
Memory	MEM_L	−33.33	0	33.33
	MEM_M	0	33.33	66.67
	MEM_H	33.33	66.67	100
	MEM_VH	66.67	100	133.3
QoS	QoSVH	−1.25	0	1.25
	QoSH	0	1.25	2.5
	QoSA	1.25	2.5	3.75
	QoSL	2.5	3.75	5
	QoSVL	3.75	5	6.25

of those samples was obtained and the response time was computed. Even though using fuzzy logic increments the response time, we gain the ability to diagnose possible incorrect outcomes in a fast manner.

Membership function tuning

This section gives the considered input and output values for the membership functions. For instance, the response time has as range 0 to 10 seconds. CPU and Memory usage have as range 0 to 100. And QoS range 0 to 5. This is better illustrated in Table 6.2.

We defined the overall QoS set related to five linguistic terms as follows:

- for the *QoS: QoSVH(e)* related to *very high*, *QoSH(e)* related to *high*, *QoSA(e)* related to *average*, *QoSL(e)* related to *low*, and *QoSVL(e)* related to *very low*.

Therefore when the QoS has values near to zero, then QoS would be very high. And when QoS is near to 5 this means the system is having degradation problems, and the QoS is very low.

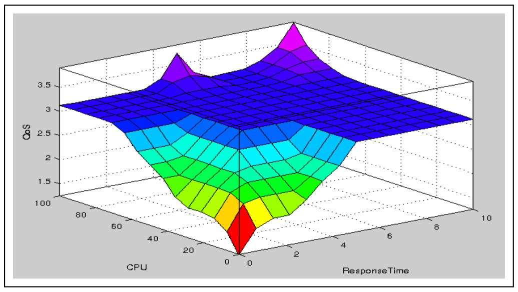


Figure 6.8: Response Time and CPU.

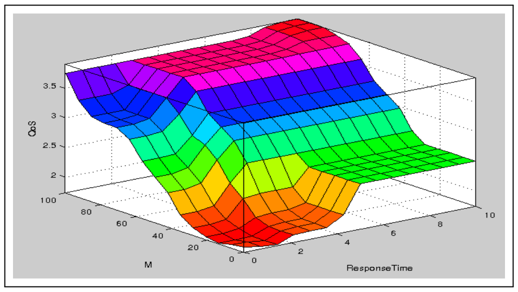


Figure 6.9: Response Time and Memory.

6.1.3 Discussion of the fuzzy diagnosis model

The input values are QoS parameters monitored from the monitoring phase and are injected into the FIS system for their analysis, these values are input data in a specified range according to the given parameter. Each input and output variable value is given a membership function. For example for the overall QoS as output its input QoS parameters are response time, CPU and memory.

We have defined fuzzy rules using the linguistic variables for our fuzzy inference engine. We defined all possible combinations of rules, 80 rules for the overall QoS, this experiment implies the use of low number of inputs, hence, the computational time was not a distinguishing factor. Yet, it is possible to summarize different combinations. Fuzzy conjunction (AND), interpreted as a min function, was applied for all the rules.

Shown in Table 1 are the QoS rules for Web services. As an example, rule number 16 in the table: when the response time is very good in order of ms, CPU usage is very high in order of ($76\% to 100\%$) and the memory is also very high ($76\% to 100\%$) then the overall QoS is inference to be low. Another example, when the overall QoS is very high it can be inference by rule number 1, implying that the response time is very good, the CPU usage is low and the memory used is also low.

The dependency amongst input and output is shown by plotting the surface of the system. Fig. 6.8 displays the dependency of the QoS based on the response time and CPU input variables. It is clear that when CPU is low (in its usage), and response time is very good (fast response) then the QoS is very high (system is behaving optimally). Also when response time is very bad (response takes seconds) and CPU is very high then QoS will be very low (system is suffering bad degradation). Fig. 6.9 depicts the dependency of the QoS based on response time and memory usage. The figure shows that the QoS is very high when the memory is low, and the response time is very good.

6.1.4 Conclusion of the fuzzy diagnostic model

We illustrated a diagnostic model based on fuzzy logic, offering a close representation of the real conditions of a managed element using a FIS as a knowledge base. The FIS permits making inference for the observed evidences, and gives a precise diagnosis. To adjust to the changing conditions and requirements for the managed elements, we proposed an approach for the diagnosis model; still having the capability of learning and updating the knowledge base. New variables of interest can be easily added and it is also easy to make inferences from these variables.

We show results for integrating the diagnosis process for Web Services. First, the variables and their states were defined as linguistic variables for establishing membership functions and then establish inference rules. Subsequently, results for the diagnosis process were exposed.

Unlike related work our proposal integrates both autonomic computing and fuzzy logic to make and ease the diagnosis process for the MAPE cycle. Our approach allows managed elements to discover the degradation of system components and make an inference based on the kind of effect a fault has.

The proposed approach could be extended for tackling Web Services fault tolerance by means of checkpointing, which allows saving states in the managed elements over time. Plus, taking into account non-functional requirements for Web Services can lead to a dynamic checkpointing mechanism.

6.2 Dynamic checkpointing

Distributed systems are ubiquitous and are found practically everywhere, and are used in everyday life without the users knowing they using such infrastructure. Individual users as well as organizations benefit from distributed systems, allowing them to render tasks that could not be accomplished by individual computers or processes. Distributed systems apply the divide and conquer approach, where complicated tasks are separated into smaller parts and distributed among several computers across a networked collaboration infrastructure.

In a nutshell, distributed systems are beneficial to users and organizations as processing cycles are easily executed plus they allow sharing resources, therefore organizations can collaborative on projects efficiently. For example, a vast group of members can be working upon solving challenging problems, while working locally or spanned across the planet using the aggregated computing power of large scale distributed systems [KS08]. However, these systems are prone to errors and may appear to be less dependable than individual systems. As stipulated by Leslie Lamport, you know yourself in a distributed system when a computer you did not know exists fails, and therefore one of your local task cannot be fulfilled. To mitigate the issues mentioned by Leslie Lamport different works have been introduced. Thus, dependability arises to address a broad spectrum of system characteristics, and many dependability techniques have been introduced, addressing systems reliability, availability and fault tolerance to mention some [SS17]. In fact, organizations seek dependability for their computer systems, components and applications, providing fault tolerance, so that with enough information previously saved the system can continue providing services to users even when a set of nodes have failed. Thus, one open challenge for systems' dependability is to offer fault tolerance, especially for low computing power devices in heterogeneous environments. For instance, mobile phones or any other device that support requesting services to a corporate enterprise by means of Web services or any other application. Particularly, solving fault tolerance issues like: monitoring, detecting and recovering from runtime failures.

To address systems' dependability issues based on fault tolerance in an efficient way, and considering that distributed systems susceptibility to failures has hampered their vast computing potential. Many techniques arise, indeed a promising technique for such purpose is rollback recovery. In this regard, communication induced checkpointing (CiC) is a well-known and efficient technique to pursue fault tolerance based on rollback recovery [CSPHPC13, VSPHRHK17]. To achieve fault tolerance processes save their state called checkpoint, this is done during the failure-free execution of the system. When a failure occurs systems have gained a manner to restart from a previously saved state, this reduces the amount of work re-executed. CiC's main goal is to save consistent global snapshots (CGSs) [KS08], one from each process, free of dangerous checkpointing patterns (z-cycles and z-paths) [CSPHPC13, SHC+16]. To build CGSs today's solutions trigger forced checkpoints when detecting dangerous patterns, nevertheless, not all triggered forced checkpoints are necessary as they generate increments in storage space, resource usage and computing processing when saving checkpoints. Therefore, the least number of forced checkpoints taken the better.

The literature presents different communication-induced and index-based checkpointing mechanisms. The HMNR protocol [HMNR00] or Fully Informed (FI) [Tsa05] exchanges rich information about processes causal past. Therefore, new versions of the FI appeared in the literature, proposing a better strategy to reduce the overhead introduced by the FI mechanism. For instance, Tsai introduces the LazyFI approach [Tsa07], applying the lazy strategy for incrementing FI's logical clocks. Another variant but this based on FI, is introduced by Luo and Manivannan [LM08a, LM09], called Fully Informed aNd Efficient (FINE), the authors establish a stronger checkpointing condition using the same control information preserved by FI. Also there exist an optimized version applying the lazy strategy called LazyFINE due to Lou and Manivannan [LM11, LM08b]. Another variant based on FI and addressing system scalability, because of the reduction in information exchanged, delaying the number of forced checkpoints and therefore reducing them, due to Simon et al. [CSPHPC13, SHC+13, SHC+16].

Despite the benefits CiC brings to distributed systems, CiC monitoring is considered to have a high computational cost, whereas all the communication overhead exchanged by each participant or process, because of the amount of storage space needed to save the checkpoints [JF17, VSPHRHK17]. CiC algorithms have overhead but save enough information to have the ability to recover to a consistent state; when processes exchange application messages these piggyback information about a process local checkpoint. CiC, however, considers system consistency as it plays an important role, therefore upon the detection of dangerous checkpointing patterns additional checkpoints are taken [CSPHPC13, SHC+13, SHC+16, GVB17].

Currently, there is not an optimal checkpointing protocol for all checkpoints and communication patterns regardless on how each mechanism takes their forced checkpoints. And which rules they apply to trigger or take checkpoints.

Therefore, we propose an architecture that supports fault tolerance, based on dynamic generation of checkpoints. Even though CiC checkpointing solutions already reduce the number of forced checkpoints taken, we argue that not all are necessary. We consider taking into account the probability of the system to suffer a failure, thus considering system degradation as a key factor when checkpointing. Specifically considering services non-functional requirements such as Quality of Service (QoS) parameters. Degradation problems can be detected while monitoring services' QoS, and lead to dynamic checkpointing in accordance to processes or services requirements. Our approach reduces the number of forced checkpoints by considering QoS requirements, for example, to meet the Service Level Agreements (SLAs) between processes. In this regard, we evaluated the system using fuzzy logic and propose a *fuzzy-consistency system evaluation (FCSE)*, illustrated in previously. We identified when useless checkpoints can be avoided regarding QoS and considering system consistency, thus using fuzzy logic for systems' assessment purposes along with CiC mechanisms. Our approach was simulated using ChkSim [VB05] and *Jfuzzylogic* [2]. We showed that our proposal is more efficient than current solutions. In this interest, we compared three of the most efficient solutions reported in the literature, based on communication-induced checkpointing (CiC) , namely *Delayed Checkpointing Fully Informed (DCFI)* proposed by Simon et. al [CSPHPC13], HMNR also called *Fully Informed (FI)* proposed by Helary et. al [HMNR00], and *Fully Informed aNd Efficient (FINE)* proposed by Luo and Manivannan [LM09].

6.2.1 Dynamic checkpointing for CiC algorithms based on fuzzy non-functional dependencies

General scheme

We propose a general scheme oriented towards heterogeneous and distributed environments, yet dependable enough to guarantee system fault tolerance upon failures, as illustrated in Fig. 6.10. Interactive processes necessitate dependability (explicitly fault tolerance), we consider the order of messages as well as their timing in communication events as these are relevant to system consistency, therefore we extrapolate the principles of CiC to the general architecture namely *Fault Tolerance* layer, illustrated by Fig. 4.1 and used in Fig. 6.10. CiC does not only guarantee consistency in message order but establishes consistency when building consistent global snapshots. Regarding the *Analysis* layer, this is in charge of detecting abnormal situations and to decide based on extracted information from messages and logs if the behavior of the system is normal, suffers any abnormality or if it is in a faulty state. With the aim of predicting the immediate future sate of the system, one can use artificial intelligence and soft computing, for example, based on autonomic computing [VSMRPHD18]. Or any other approach that supports predictions like Hidden Markov Model (HMM) [HGDJ08], Bayesian Networks [KDVSD+14, KDVSD+13] or Fuzzy Logic. We use Fuzzy logic for the Analysis layer, namely, as its foundation or basis, particularly *Fuzzy Consistency System Evaluation (FCSE)*, thus computing the overall QoS crisp value at a given time, as explained further on. Plus the FCSE does not incur in performance degradation. The *Monitoring* phase recollects

[2]http://jfuzzylogic.sourceforge.net/html/index.html

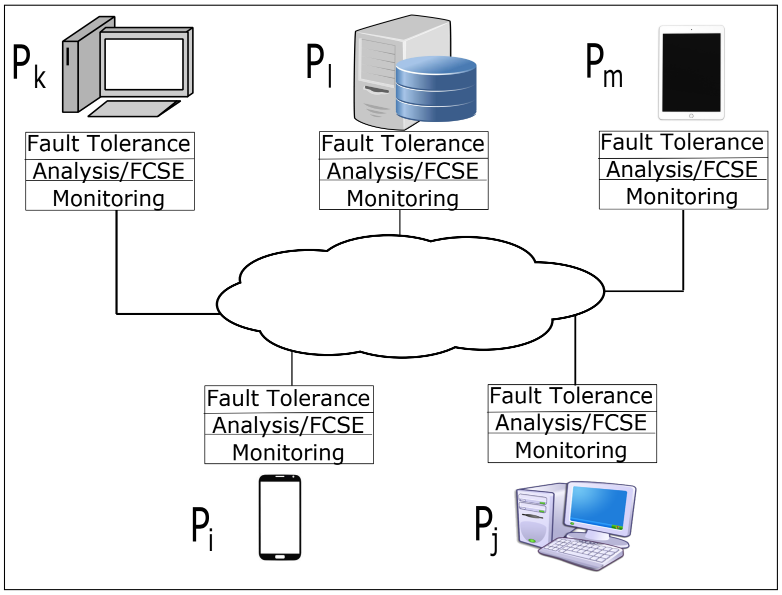


Figure 6.10: General Scheme.

data from each of the processes for the system in play. The monitoring initiates the petitions, sending and receiving processes' requests/responses. Also, computes the QoS parameters as response time, CPU and Memory percentage usage.

6.2.2 Using fuzzy logic for dynamically checkpointing processes

On one hand we have the *Fault Tolerance* layer based on CiC checkpointing, on the other hand we have the dynamic generation of checkpoints based on the FCSE. Thus, we can diminish the common notion that CiC mechanisms have high implementation cost. A relation between input and output for non-functional requirements such as QoS parameters can be easily established in the form of if-then rules; the systems' behavior can be described by using these simple rules. We take advantage of fuzzy logic by measuring FCSE, under different workloads, therefore we are able to manage QoS changes. Implying that different CiC algorithms have the ability of dynamically generating checkpoints.

Case study: Web services

Web services based on communication-induced checkpointing can be better explained through an example. For instance, consider the *Stock Quote* Web service composition. Where many consumers of a service invoke multiple stocks brokers to find out on which to invest, for clients that pay a subscription, premium users, they are able to receive in real-time the stock quote service.

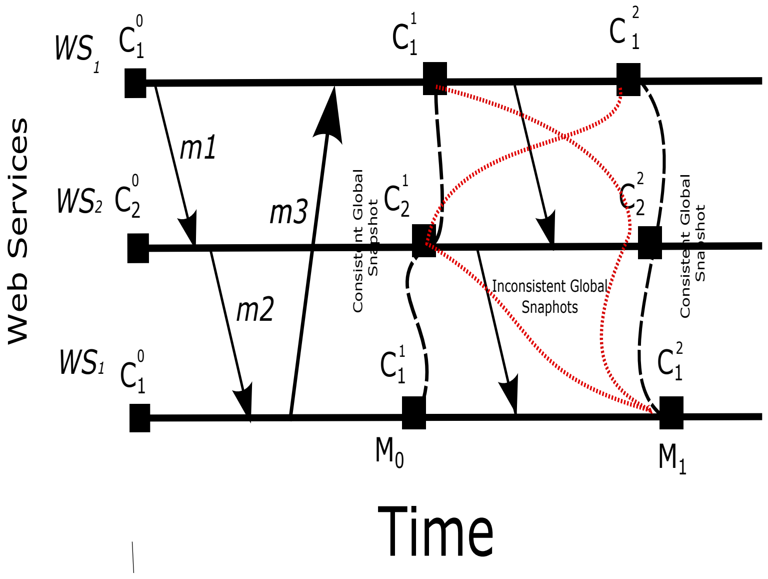

Web Services
WS1
WS2
WS1
C01
C11
C21
C02
C12
C22
C01
C11
C21
m1
m2
m3
Consistent Global Snapshot
Inconsistent Global Snaphots
Consistent Global Snapshot
M0
M1
Time

Figure 6.11: Example Scenario.

The simplest case is shown in Fig. 6.11 where two service consumers make a petition to a stock broker or service provider, however it shows the need for building consistent global snapshots (CGS). As stipulated by Netzer and Xu in presence of zigzag paths and causal paths cannot constitute a CGS [NX95]. Fig. depicts that M_0 and M_1 build CGS while C^1_{c1}, C^2_{c2} and C^2_{p1} cannot be part of a CGS; because of messages $m4$ and $m5$ for instance, although no causal path exists between C^1_{c1} and C^2_{p1} a zigzag path does formed by the aforementioned messages. This means that no CGS can be formed from the checkpoints involved in a zigzag path, in other words no CGS can be built that contains C^1_{c1} and C^2_{p1}.

6.2.3 FCSE checkpointing window

Recall that the membership functions and their values are presented in Table 6.2.

We ran different scenarios with diverse values for QoS parameters, and followed an exponential distribution; simulating real world conditions. Therefore, the crisp values for the FCSE varies from 0 to 5 after evaluated by the FIS system; better illustrated in Fig. 6.12, we named this the FCSE window. For instance, this window can be very rigorous, meaning that forced checkpoints are taken when the output computed crisp value of the FCSE satisfies that it is equal to 0.625 or lower. And we call relax window to that one allowing a more degraded system, having an FCSE output crisp value of 4.375. Also, we consider intermediate crisp values between the rigorous window and the relaxed window and evaluate windows with the following values: $0.625, 1.25, 1.875, 2.5, 3.125, 3.75, 4.375$.

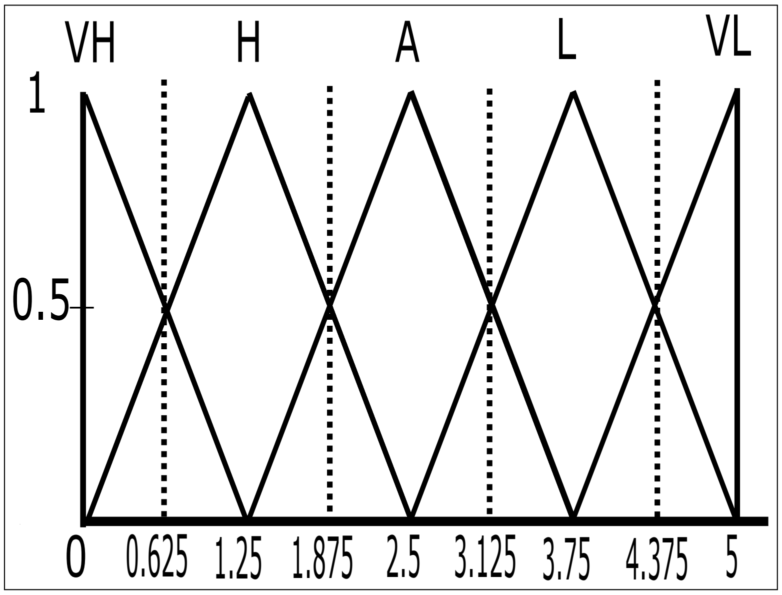


Figure 6.12: FCSE Window.

6.2.4 Performance evaluation

We compare the performance of three different CiC algorithms, namely DCFI, FI and FINE. These were chosen because they are recent algorithms and the most efficient algorithms reported in the literature. We simulated and analyzed these three algorithms using the distributed checkpointing simulator ChkSim [VB05] and JFuzzyLogic. ChkSim simulator, models distributed systems in a deterministic manner, reproducing the same behavior for two or more algorithms, and allows running a simulation as many times as necessary. As metric and as key performance indicator we took the number of forced checkpoints generated by each checkpointing algorithm.

The performance was analyzed for several scenarios. For the sake of space, we only present three scenarios: one with 1000, one with 2500, and one with 5000 messages and varying the QoS window. Each scenario was made with a uniform distribution among events (send, delivery and checkpoints), and by varying the number of processes from 10, 20,. . . , 150. Moreover, for each scenario, 100 iterations were executed with different communication and checkpoint patterns.

Fig. 6.13, Fig. 6.14 and Fig. 6.15 show that the number of forced checkpoints presented by DCFI is the lowest, however it has a strong correlation with the QoS window, for a rigorous QoS the number of forced checkpoints is almost the same as the one observed for DCFI without using our approach. Yet, as the window relaxes the number of forced checkpoints reduces drastically. Furthermore, the number of forced checkpoints of DCFI represents on average a 3% gain with respect to FI, while for FINE, it represents on average a 1.5% gain with respect to FI.

To conclude this section, based on results presented in Fig. 6.13, Fig. 6.14 and Fig.

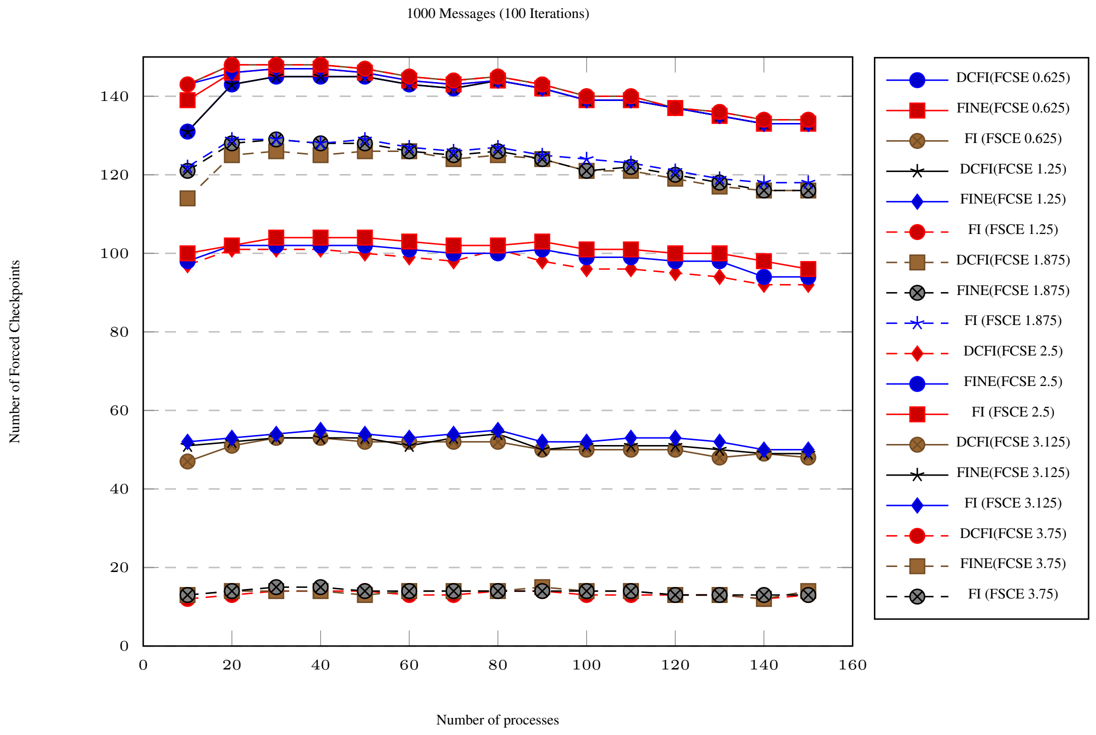


Figure 6.13: Number of forced Checkpoints for 1000 sent messages.

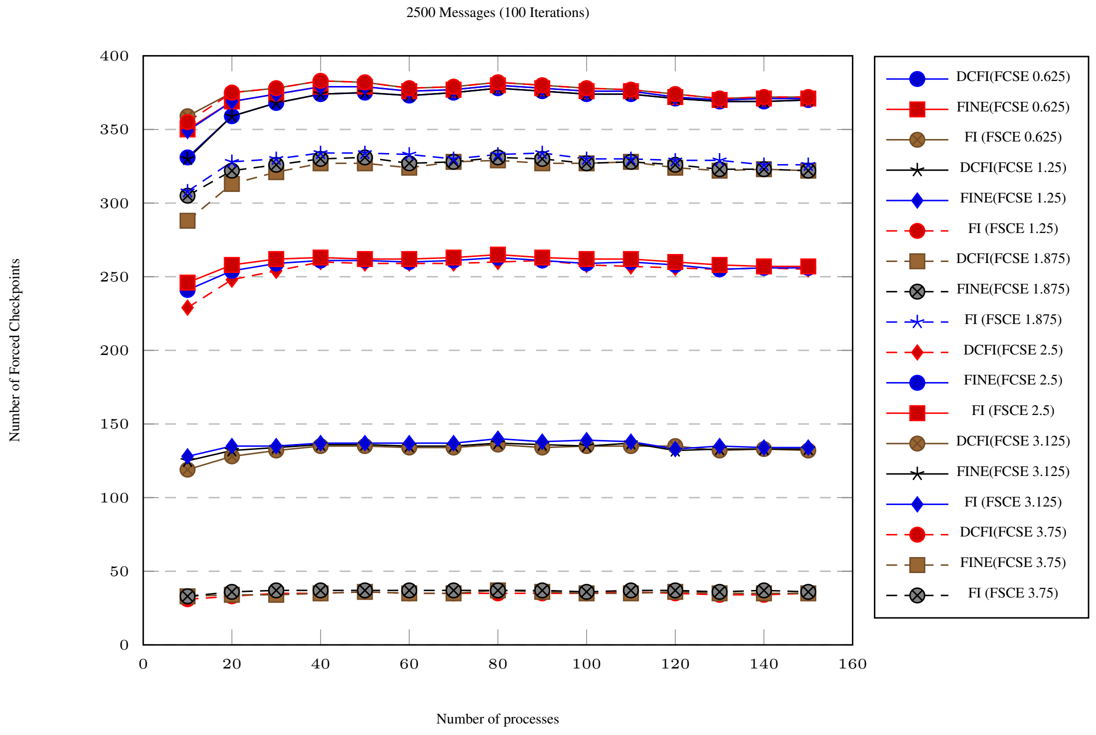


Figure 6.14: Number of forced checkpoints for 2500 sent messages.

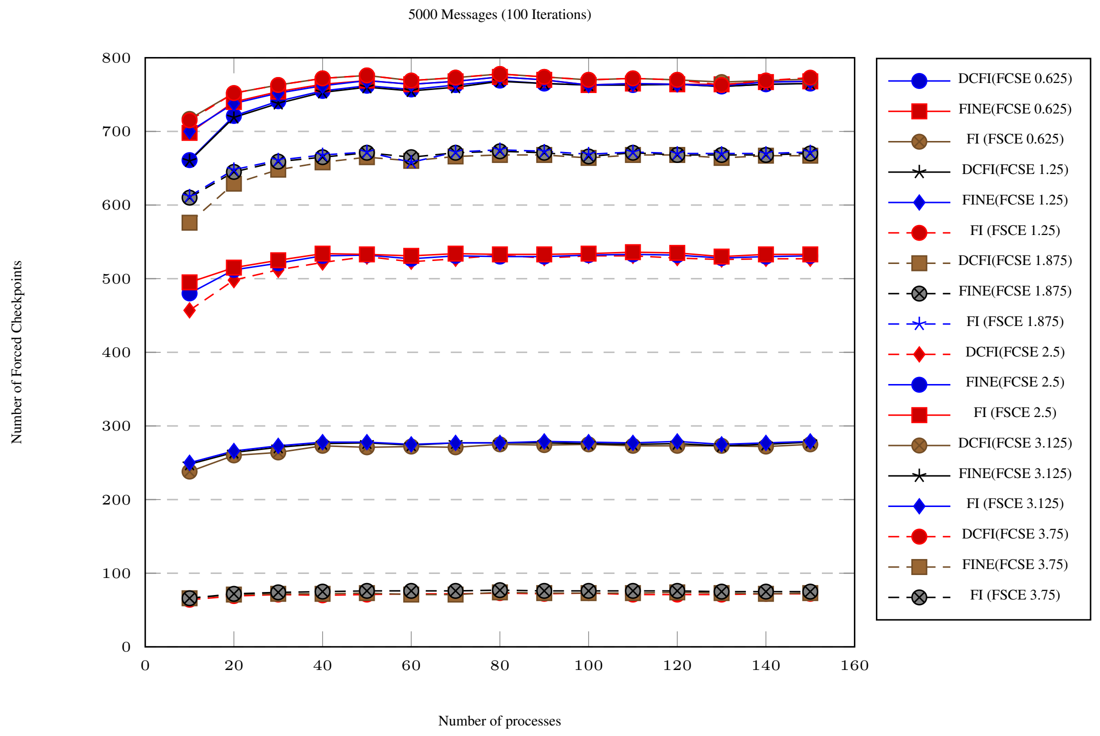


Figure 6.15: Number of forced checkpoints for 5000 sent messages.

6.15 we can argue, that our approach correlates both cost (QoS parameters: response time, CPU, Memory) and benefit (how well the system behaves) in the following way:

- When QoS is very high this implies saving checkpoints. Because, there are enough resources available, and the cost of storing checkpoints is minimum.
- When QoS is very low this implies not saving many checkpoints. Because, there not enough resources available, yet some are saved for preserving consistency upon a failure.

6.2.5 Conclusion of the dynamic checkpointing approach

We presented a fuzzy logic approach for dynamically checkpointing distributed and collaborative environments. In this regard, we consider the system degradation to determine when to take checkpoints, carried out through a fuzzy consistency system evaluation, identifying useless forced checkpoints. Three different approaches that use the communication-induced checkpointing algorithm were compared while implementing our proposal. Experimental results demonstrate our efficiency, as we managed to reduce the number of forced checkpoints, because we took into account the quality of service (QoS) of the system at a given time. The generation of checkpoints is carried out dynamically, since it is considered both a window of rigorous quality of service as well as a relaxed window. For any, the delayed checkpointing algorithm presented the best performance, having the least number of forced checkpoints. In all cases, all algorithms: DCFI, FINE and FI reduced their forced checkpoints generation; the DCFI algorithm is more efficient than the current solutions FI and FINE. The results show that FINE generates on average only a 1.5% gain with respect to FI, while that DCFI generates on average a 3% gain with respect to FI, but the number of forced checkpoints are reduced depending on the QoS window associated to the corresponding case.

Chapter 7

Conclusion and Future Work

7.1 Achievements

During the development of this research, some original and innovative findings were reached. These findings can be useful to solve some open problems related to distributed systems, as we explain below.

- **The usefulness of the IDR**. Causal ordering is an important research subject in distributed systems since allows to provide reliability for some applications, preserving the asynchronous execution of the system. However, for certain applications, for example Web services and Web services composition, where degradation of the system is commonly seen, ensuring the causal order based on the happened-before relation (HBR) is rigid, and negative as using it affects the performance of the system.

  In this dissertation we proposed a message ordering framework (MOF) for collaborative environments, based on JMS and on the IDR for the problem causally preserving the order of messages. Thus, diminishing the common notion that this kind of solutions cause degradation to the system. In addition, our framework is generic and can be exploited by any Web service-based system.

- **The usefulness of CiC mechanisms**. CiC aims to built consistent global snapshots (CGSs), therefore reduce the computational cost upon failures. In such regard, we proposed an algorithm to build CGSs oriented to increase fault tolerance for interactive Web services compositions. This approach is also generic, we added CiC to the fault tolerance layer of the protocol stack, therefore, individual Web services, and/or Web services compositions can exploit it.

- **The usefulness of Autonomic Computing**. Autonomic Web services is another important research area in distributed systems and autonomic computing, since they are self-manageable entities that preserve in most cases important business transactions and functionalities for organizations. On one hand, by implementing the MAPE control loop systems' complexity is mitigated. However, not all the cycle can be focused by this paradigm itself, it can be aided by checkpointing mechanisms for the planning and execution of recovery actions, as shown in this work.

- **The usefulness of the Fuzzy Consistency System Evaluation**. We also proposed a fuzzy consistency system evaluation (FCSE) to infer a degree of system behavior among messages. In this sense, the FCSE can be used to determine if forced checkpoints dealing with certain messages is mandatory or such forced checkpoints can be discarded. Thereby, by applying QoS parameters criteria, based on the fuzzy consistency system evaluation the performance of the system can be improved. Therefore, correlating system degradation with the generation of checkpoints makes our solution feasible for low computer power devices and components.

7.1.1 Dissertation-derived articles

Author's publications

- International journals: 4
- International conferences: 2
- National conference: 1

International journals

- **M. Vargas-Santiago**, Hernandez, S. E. P., Rosales, L. A. M., & Kacem, H. H. (2016). Fault Tolerance Approach Based on Checkpointing towards Dependable Business Processes. IEEE Latin America Transactions, 14(3), 1408-1415.
- **M. Vargas-Santiago**, S.E. Pomares-Hernández, L.A. Morales-Rosales, & H. Hadj-Kacem (2017). Message Ordering Framework for Collaborative Web Services-Based Environments. IEEE Latin America Transactions, (Accepted to be published, 2017-2018).
- **M. Vargas-Santiago** , S.E. Pomares-Hernández, L.A. Morales-Rosales, & H. Hadj-Kacem (2017). Survey on Fault Tolerance Approaches based on checkpointing mechanisms . Journal of Software, 12(7), 2017.
- **M. Vargas-Santiago**, L.A. Morales-Rosales, S.E. Pomares-Hernández, & K. Drira (2017). Autonomic Web services based on Asynchronous checkpointing. IEEE Access, vol. PP, no. 99, pp. 1-1. doi: 10.1109/ACCESS.2017.2756867

International Conferences

- Abdennadher, I., Bouassida Rodriguez, I., Jmaiel, M., **Santiago, M. V.**, & HernÃąndez, S. P. (2015, November). Towards a Decision Approach for Autonomic Systems Adaptation. In Proceedings of the 13th ACM International Symposium on Mobility Management and Wireless Access (pp. 77-80). ACM.
- **M. Vargas-Santiago**, S.E. Pomares-Hernández, L.A. Morales-Rosales, & H. Hadj-Kacem (2017). Towards Dependable Web Services in Collaborative Environments Based on Fuzzy Non-functional Dependencies. In 2017 *International Conference in Software Engineering Research and Innovation*(CONISOFT).

National Conference

- **M. Vargas-Santiago**, Hernandez, S. E. P., Drira, K., Dominguez, E. L., & Clark, R. H. Message Ordering Framework for Collaborative Web Services-Based Environments. Encuentro Nacional de Ciencias de la Computación (ENC 2014)

7.2 Future work

For future work we can consider the following ideas:

- **Packet loss**. We consider reliable communication channels and the development of our mechanism was based on specific system model considerations. We support fault tolerance upon failures of the system, we do not implement techniques for supporting packet loss. It is necessary to develop or implement methods for the detection and the recovery of lost messages (for example forward error correction, however this is a totally different problem or issue). Nevertheless, the packet loss tolerance was not part of the objectives of this dissertation. In the future we can take this as consideration.
- **Smart cities**. We could implement an Autonomic Service Bus for interconnecting all type of smart devices, like: cameras, traffic control devices, cellphones, vehicles and any other component that supports and need to share information.

Acknowledgment. This Thesis was supported by the National Council of Science and Technology of Mexico (CONACyT) through CVU under Grant 417439.

Acronyms

AC	**A**utonomic **C**omputing
ACS	**A**utonomic **C**omputing **S**ystem
ASB	**A**utonomic **S**ervice **B**us
BPEL	**B**usiness **P**rocess **E**xecution **L**anguage
BPM	**B**usiness **P**rocess **M**anagement
B2B	**B**usiness **T**o **B**usiness
CiC	**C**ommunication **i**nduced **C**heckpoint
CGS	**C**onsistent **G**lobal **S**napshot
CCP	**C**heckpoint and **C**ommunication **P**attern
CP	**C**heckpointing **P**rotocol
CWS	**C**omposite **W**eb **S**ervice
DCFI	**D**elayed **C**heckpointing **F**ully **I**nformed
EAI	**E**nterprise **A**pplication **I**ntegration
ESB	**E**nterprise **S**ervice **B**us
FCS	**F**uzzy **C**onsistency **S**ystem
FCSE	**F**uzzy **C**onsistency **S**ystem **E**valuation
FI	**F**ully **I**nformed
FINE	**F**ully **I**nformed a**N**d **E**fficient
FIS	**F**uzzy **I**nference **S**ystem
HBR	**H**appened **B**efore **R**elationship
IDR	**I**mmediate **D**ependency **R**elationship
JNDI	**J**ava **N**aming and **D**irectory **I**nterface
MAPE	**M**onitoring **A**nalyzing **P**lanning **E**xecuting
MOF	**M**essage **O**rdering **F**ramework
P2P	**P**oint-to-**P**oint
QoS	**Q**uality **o**f **S**ervice
REST	**RE**presentational **S**tate **T**ransfer
RT	**R**esponse **T**ime
SLA	**S**ervice **L**evel **A**greement
SOA	**S**ervice **O**riented **A**rchitecture
SOC	**S**ervice **O**riented **C**omputing
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
TCWS	**T**ransactional **C**omposite **W**eb **S**ervice
TPS	**T**ransactions **P**er **S**econd
UDDI	**U**niversal **D**escription **D**iscovery **I**ntegration
WS	**W**eb **S**ervice
WSCI	**W**eb **S**ervice **C**horeography **I**nterface
WSDL	**W**eb **S**ervice **D**escription **L**anguage
WS-RM	**W**eb **S**ervice **R**eliable **M**essaging
XML	**e**X**tensible **M**arkup **L**anguage

Notation

$\emptyset$	Empty set
$\cup$	Union
$\setminus$	Difference
$\exists$	Exists
$\forall$	For all
p_i, p_j	Processes
WS_i	Web service monitored information
$BPEL_i$	Business Process Execution Language
Sx	Set of symptoms associated to a Web service
D	Set of diagnostics associated to a Web service
P	Set of plans associated to a Web service
P_G	Recovery plan for the Web service composition
$\rightarrow$	Happened-Before Relation (HBR)
$\downarrow$	Immediate Dependency Relation (IDR)
$\leftarrow$	Assignation operator
$C(p_i)[i]$	Is a local checkpoint
S	Set of snapshots
m	Is the quadruplet $m = (k, t_k, message, H_m)$
k	Is the local process identifier
t_k	Is the value of the local clock
$message$	Is the structure that carries the data
H_m	Is the immediate history of m.
$VT(p_i)[i]$	Is a vector clock
CI	Is a set of entries $ci_{k,tk} = (k, t_k)$ that represents control information
CI_i	Is a set of entries $ci_{k,tk} = (k, t_k)$ of a specific process.

Bibliography

[AAY11] Mohamed Almulla, Kawthar Almatori, and Hamdi Yahyaoui. A qos-based fuzzy model for ranking real world web services. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 203–210. IEEE, 2011.

[ACR13] Rafael Angarita, Yudith Cardinale, and Marta Rukoz. Dynamic recovery decision during composite web services execution. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, pages 187–194. ACM, 2013.

[AH11] Issam Al Hadid. Airport enterprise service bus with self-healing architecture (aesb-sh). *International Journal of Aviation Technology, Engineering and Management (IJATEM)*, 1(1):1–13, 2011.

[ALR$^+$01] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability.* University of Newcastle upon Tyne, Computing Science, 2001.

[AM15a] Vani Vathsala Atluri and Hrushikesha Mohanty. Time and cost aware check-pointing of choreographed web services. In *Distributed Computing and Internet Technology*, pages 207–219. Springer, 2015.

[AM15b] Vani Vathsala Atluri and Hrushikesha Mohanty. Web service response time prediction using hmm and bayesian network. In *Intelligent Computing, Communication and Devices*, pages 327–335. Springer, 2015.

[AMH10] Ahmed Al-Moayed and Bernhard Hollunder. Quality of service attributes in web services. In *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, pages 367–372. IEEE, 2010.

[AP11] Sanjay P Ahuja and Amit Patel. Enterprise service bus: A performance evaluation. *Communications and Network*, 3(3):133–140, 2011.

[ARC15] Rafael Angarita, Marta Rukoz, and Yudith Cardinale. Modeling dynamic recovery strategy for composite web services execution. *World Wide Web*, pages 1–21, 2015.

[ARM15] Rafael Angarita, Marta Rukoz, and Maude Manouvrier. Dynamic composite web service execution by providing fault-tolerance and qos monitoring. In *Service-Oriented Computing-ICSOC 2014 Workshops*, pages 371–377. Springer, 2015.

[BHH10] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing web services: A survey. *Department of Computer Science, KingŠs College London, Tech. Rep. TR-10-01*, 2010.

[BJPK$^+$05] Alberto Bartoli, Ricardo Jiménez-Peris, Bettina Kemme, Cesare Pautasso, Simon Patarin, Stuart Wheater, and Simon Woodman. Adapt: towards autonomic web services. *Distributed Systems Online*, 2005.

[C$^+$06] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31, 2006.

[CBS$^+$09] K.S.May Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. A fault taxonomy for web service composition. In Elisabetta Di Nitto and Matei Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 363–375. Springer Berlin Heidelberg, 2009.

[CDK$^+$02] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet computing*, 6(2):86–93, 2002.

[CGK$^+$03] Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, S Weerawarana, et al. Business process execution language for web services, version 1.1. 2003.

[Cha04] D. Chappell. *Enterprise service bus.* O'Reilly Media, Inc., 2004.

[CL$^+$99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[CRA13] Yudith Cardinale, Marta Rukoz, and Rafael Angarita. Modeling snapshot of composite ws execution by colored petri nets. In *Resource Discovery*, pages 23–44. Springer, 2013.

[CSPHPC13] Alberto Calixto-Simon, Saúl E. Pomares-Hernandez, and Jose R. Perez-Cruz. A delayed checkpoint approach for communication-induced checkpointing in autonomic computing. In *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 56–61, June 2013.

[Dio15] Code Diop. *An autonomic service bus for service-based distributed systems.* PhD thesis, Toulouse, INSA, 2015.

[DJ07] Schahram Dustdar and Lukasz Juszczyk. Dynamic replication and synchronization of web services for high availability in mobile ad-hoc networks. *Service Oriented Computing and Applications*, 1(1):19–33, 2007.

[DMM+02] Vijay Dialani, Simon Miles, Luc Moreau, David De Roure, and Michael Luck. Transparent fault tolerance for web services based architectures. In *Euro-Par 2002 Parallel Processing*, pages 889–898. Springer, 2002.

[DTTV09] Douglas B Davis, Yih-shin Tan, Brad B Topol, and Vivekanand Vellanki. Checkpointing and restarting long running web services, July 14 2009. US Patent 7,562,254.

[FF12] John Footen and Joey Faust. *The service-oriented media enterprise: SOA, BPM, and web services in professional media systems.* CRC Press, 2012.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[FLLL07] Chen-Liang Fang, Deron Liang, Fengyi Lin, and Chien-Cheng Lin. Fault tolerant web services. *Journal of Systems Architecture*, 53(1):21 – 38, 2007.

[GG11] D Ganesh Gopal. A novel approach for efficient resource utilization and trustworthy web service. *International Journal of Computer Science and Security (IJCSS)*, 5(2):168, 2011.

[GJGT10] Ínigo Goiri, Ferran Julia, Jordi Guitart, and Jordi Torres. Checkpoint-based fault-tolerant infrastructure for virtualized service providers. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 455–462. IEEE, 2010.

[GUR11] Le Gao, Susan D Urban, and Janani Ramachandran. A survey of transactional issues for web service composition and recovery. *International Journal of Web and Grid Services*, 7(4):331–356, 2011.

[GVAGM10] Charles Gouin-Vallerand, Bessam Abdulrazak, Sylvain Giroux, and Mounir Mokhtari. A software self-organizing middleware for smart spaces based on fuzzy logic. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 138–145. IEEE, 2010.

[GVB17] Islene C. Garcia, Gustavo M. D. Vieira, and Luiz Eduardo Buzato. A rollback in the history of communication-induced checkpointing. *CoRR*, abs/1702.06167, 2017.

[GZ05] S. Gurguis and A. Zeid. Towards autonomic web services: Achieving self-healing using web services. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[HBBK07] Judith Hurwitz, Robin Bloor, Carol Baroudi, and Marcia Kaufman. *Service oriented architecture for dummies.* John Wiley & Sons, 2007.

[HCR12] SE Pomares Hernandez, JR Perez Cruz, and M Raynal. From the happened-before relation to the causal ordered set abstraction. *Journal of Parallel and Distributed Computing*, 72(6):791–795, 2012.

[HDFD02] S Pomares Hernandez, Khalil Drira, Jean Fanchon, and Michel Diaz. An efficient multi-channel distributed coordination protocol for collaborative engineering activities. In *2002 IEEE International Conference on Systems Man and Cybernetics (SMCŠ02), Hammamet (Tunisie)*, pages 6–9, 2002.

[Her15] Saúl Eduardo Pomares Hernández. The minimal dependency relation for causal event ordering in distributed computing. *Applied Mathematics & Information Sciences*, 9(1):pp–57, 2015.

[HFD04a] Saul Pomares Hernandez, Jean Fanchon, and Khalil Drira. The immediate dependency relation: an optimal way to ensure causal group communication. In *ANNUAL REVIEW OF SCALABLE COMPUTING, EDITIONS WORLD SCIENTIFIC, SERIES ON SCALABLE COMPUTING*, pages 61–79, 2004.

[HFD04b] SE Pomares Hernandez, Jean Fanchon, and Khalil Drira. The immediate dependency relation: an optimal way to ensure causal group communication. *Annual Review of Scalable Computing*, 3:61–79, 2004.

[HGDJ08] Riadh Ben Halima, Mohammed Karim Guennoun, Khalil Drira, and Mohamed Jmaiel. Providing predictive self-healing for web services: a qos monitoring and analysis-based approach. *Journal of Information Assurance and Security*, 3(3):175–184, 2008.

[HM08] Markus C Huebscher and Julie A McCann. A survey of autonomic computingŮdegrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.

[HMNR00] J-M Hélary, Achour Mostefaoui, Robert HB Netzer, and Michel Raynal. Communication-based prevention of useless checkpoints in distributed computations. *Distributed Computing*, 13(1):29–43, 2000.

[IP14] Anne Immonen and Daniel Pakkala. A survey of methods and approaches for reliable dynamic service compositions. *Service Oriented Computing and Applications*, 8(2):129–158, 2014.

[JF17] Bentolhoda Jafary and Lance Fiondella. Optimal checkpointing of fault tolerant systems subject to correlated failure. In *Reliability and Maintainability Symposium (RAMS), 2017 Annual*, pages 1–6. IEEE, 2017.

[JGH09] Meiko Jensen, Nils Gruschka, and Ralph Herkenhöner. A survey of attacks on web services. *Computer Science - Research and Development*, 24(4):185–197, 2009.

[KC03] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KDVSD$^+$13] Roberto Koh-Dzul, Mariano Vargas-Santiago, Code Diop, Ernesto Exposito, and Francisco Moo-Mena. A smart diagnostic model for an autonomic service bus based on a probabilistic reasoning approach. In *Ubiquitous Intelligence* *and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*, pages 416–421. IEEE, 2013.

[KDVSD$^+$14] Roberto Koh-Dzul, Mariano Vargas-Santiago, Codé Diop, Ernesto Exposito, Francisco Moo-Mena, and Jorge Gómez-Montalvo. Improving esb capabilities through diagnosis based on bayesian networks and machine learning. *Journal of Software*, 9(8), 2014.

[KGI13] Ajay Kattepur, Nikolaos Georgantas, and Valerie Issarny. Qos composition and analysis in reconfigurable web services choreographies. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 235–242. IEEE, 2013.

[KGM11] Mohamed-Hedi Karray, Chirine Ghedira, and Zakaria Maamar. Towards a self-healing approach to sustain web services reliability. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 267–272. IEEE, 2011.

[KHKS09] Amina Khalid, Mouna Abdul Haye, Malik Jahan Khan, and Shafay Shamail. Survey of frameworks, architectures and techniques in autonomic computing. In *Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on*, pages 220–225. IEEE, 2009.

[KKM11] Aarti Karande, Milind Karande, and BB Meshram. Choreography and orchestration using business process execution language for soa with web services. *International Journal of Computer Science Issues IJCSI*, 11:224–232, 2011.

[KS08] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2008.

[KSS15] Raj Kumar, R Sureshkumar, and B Saravanabalaji. Description logic program and fuzzy logic based web service selection. *International Journal of Applied Engineering Research*, 10(9):6662–6667, 2015.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, (1):23–31, 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LDB16] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: a survey of techniques and tools. *ACM Computing Surveys (CSUR)*, 48(3):33, 2016.

[LM08a] Yi Luo and D Manivannan. Fine: A fully informed and efficient communication-induced checkpointing protocol. In *Systems, 2008. ICONS 08. Third International Conference on*, pages 16–22. IEEE, 2008.

[LM08b] Yi Luo and D Manivannan. Theoretical and experimental evaluation of communication-induced checkpointing protocols in f e family. In *Performance, Computing and Communications Conference, 2008. IPCCC 2008. IEEE International*, pages 217–224. IEEE, 2008.

[LM09] Yi Luo and D Manivannan. Fine: A fully informed and efficient communication-induced checkpointing protocol for distributed systems. *Journal of Parallel and Distributed Computing*, 69(2):153–167, 2009.

[LM11] Yi Luo and D Manivannan. Theoretical and experimental evaluation of communication-induced checkpointing protocols in and families. *Performance Evaluation*, 68(5):429–445, 2011.

[LML$^+$11] Jonathan Lee, Shang-Pin Ma, Shin-Jie Lee, Chia-Ling Wu, and Chiung-Hon Leon Lee. Towards a high-availability-driven service composition framework. *Service Life Cycle Tools and Technologies: Methods, Trends and Advances*, pages 221–243, 2011.

[MA75] Ebrahim H Mamdani and Sedrak Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *International journal of man-machine studies*, 7(1):1–13, 1975.

[Man03] Ann Thomas Manes. *Web Services: A Manager's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[MBMS10] A. Moody, G. Bronevetsky, K. Mohror, and B.R. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[MD11] Houwayda Elfawal Mansour and Tharam Dillon. Dependability and rollback recovery for composite web services. *Services Computing, IEEE Transactions on*, 4(4):328–339, 2011.

[Men07] Falko Menge. Enterprise service bus. In *Free and open source software conference*, volume 2, pages 1–6, 2007.

[MJ13] Soumaya Marzouk and Mohamed Jmaiel. A policy-based approach for strong mobility of composed web services. *Service Oriented Computing and Applications*, 7(4):293–315, 2013.

[MMBJ09] S. Marzouk, A. Maalej, I. Bouassida, and M. Jmaiel. Periodic checkpointing for strong mobility of orchestrated web services. In *Services-I, 2009 World Conference on*, pages 203–210. IEEE, 2009.

[MMJ10] S. Marzouk, A. Maalej, and M. Jmaiel. Aspect-oriented checkpointing approach of composed web services. In Florian Daniel and FedericoMichele Facca, editors, *Current Trends in Web Engineering*, volume 6385 of *Lecture Notes in Computer Science*, pages 301–312. Springer Berlin Heidelberg, 2010.

[MMSZ07] Louise E Moser, P Michael Melliar-Smith, and Wenbing Zhao. Building dependable and secure web services. *Journal of Software*, 2(1):14–26, 2007.

[MSSD06] A. Moga, J. Soos, I. Salomie, and M. Dinsoreanu. Adding self-healing behaviour to dynamic web service composition. In *Proceedings of the 5th WSEAS International Conference on Data Networks, Communication and Computers, Bucharest, Romania*, pages 206–211, 2006.

[MVM14] B. Murugananthan, K. Vivekanandan, and D. Mondal. Rollback recovery approach for complex composite web services to enhance reliability of service. *Interantional Journal of Engineering Research and Technology*, 3(2):2289–2292, 2014.

[NX95] R. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and distributed Systems*, 6(2):165–169, 1995.

[OC09] Luis Oliva and Luigi Ceccaroni. Rest web services in collaborative work environments. In *CCIA*, pages 419–427, 2009.

[OFD06] Michael J Oudshoorn, M Muztaba Fuad, and Debzani Deb. Towards autonomic computing: Injecting self-organizing and self-healing properties into java programs. *FRONTIERS IN ARTIFICIAL INTELLIGENCE AND APPLICATIONS*, 147:384, 2006.

[Pas05] James Pasley. How bpel and soa are changing web services development. *IEEE Internet Computing*, 9(3):60–67, 2005.

[Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[Pet16] Charles J Petrie. *Web service composition*. Springer, 2016.

[Ray92] Michel Raynal. About logical clocks for distributed systems, 1992.

[RCA12] Marta Rukoz, Yudith Cardinale, and Rafael Angarita. Faceta*: Checkpointing for transactional composite web service execution based on petri-nets. *Procedia Computer Science*, 10:874 – 879, 2012. ANT 2012 and MobiWIS 2012.

[RFG12] Mohsen Rouached, Walid Fdhila, and Claude Godart. Web services compositions modelling and choreographies analysis. *Web Service Composition and New Frameworks in Designing Semantics: Innovations: Innovations*, page 1, 2012.

[RH08] AM Riad and QF Hassan. Service-oriented architecture–a new alternative to traditional integration methods in b2b applications. *Journal of Convergence Information Technology*, 3(1):41, 2008.

[RS96] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[SHC+13] Alberto Calixto Simon, Saul E Pomares Hernandez, Jose Roberto Perez Cruz, Pilar Gomez-Gil, and Khalil Drira. A scalable communication-induced checkpointing algorithm for distributed systems. *IEICE TRANSACTIONS on Information and Systems*, 96(4):886–896, 2013.

[SHC+16] Alberto Calixto Simón, Saul E Pomares Hernandez, Jose Roberto Perez Cruz, Riadh Ben Halima, and Hatem Hadj Kacem. Self-healing in autonomic distributed systems based on delayed communication-induced checkpointing. *International Journal of Autonomous and Adaptive Communications Systems*, 9(3-4):183–200, 2016.

[SHRK16] Mariano Vargas Santiago, Saul Eduardo Pomares Hernandez, Luis Alberto Morales Rosales, and Hatem Hadj Kacem. Fault tolerance approach based on checkpointing towards dependable business processes. *IEEE Latin America Transactions*, 14(3):1408–1415, 2016.

[ŞLL10] Ioana Şora, Gabriel Lazăr, and Silviu Lung. Mapping a fuzzy logic approach for qos-aware service selection on current web service standards. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 553–558. IEEE, 2010.

[SP13] Gupta Shuchi and Bhanodia Praveen. A fault tolerant mechanism for composition of web services using subset replacement. *Architecture*, 1:2, 2013.

[SQV+14] Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decadeŠs overview. *Information Sciences*, 280:218–238, 2014.

[SS17] Daniel Siewiorek and Robert Swarz. *Reliable Computer Systems: Design and Evaluatuion*. Digital Press, 2017.

[TS85] Tomohiro Takagi and Michio Sugeno. Fuzzy identification of systems and its applications to modeling and control. *IEEE transactions on systems, man, and cybernetics*, (1):116–132, 1985.

[Tsa05] Jichiang Tsai. An efficient index-based checkpointing protocol with constant-size control information on messages. *IEEE Transactions on Dependable and Secure Computing*, 2(4):287–296, 2005.

[Tsa07] Jichiang Tsai. Applying the fully-informed checkpoiniting protocol to the lazy indexing strategy. *Journal of information science and engineering*, 23(5):1611–1621, 2007.

[TT08] Vuong Xuan Tran and Hidekazu Tsuji. Qos based ranking for web services: Fuzzy approaches. In *Next Generation Web Services Practices, 2008. NWESP'08. 4th International Conference on*, pages 77–82. Ieee, 2008.

[TTMR03] S. Tai, Stefan Tai, Thomas A. Mikalsen, and Isabelle Rouvellou. Using message-oriented middleware for reliable web services messaging. In *ISSN 0302-9743*, pages 89–104, 2003.

[TVS07] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[TZZ$^+$05] Wenhu Tian, Farhana Zulkernine, Jared Zebedee, Wendy Powley, and Pat Martin. Architecture for an autonomic web services environment. In *Web Services and Model-Driven Enterprise Information Services, Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Services, WSMDEIS 2005, In conjunction with ICEIS 2005, Miami, USA*, pages 32–44. Citeseer, 2005.

[UT06] Ken Ueno and Michiaki Tatsubori. Early capacity testing of an enterprise service bus. In *Web Services, 2006. ICWS'06. International Conference on*, pages 709–716. IEEE, 2006.

[Vat11] A Vani Vathsala. Optimal call based checkpointing for orchestrated web services. *International Journal of Computer Applications*, 36(8), 2011.

[Vat12] A Vani Vathsala. Global checkpointing of orchestrated web services. In *Recent Advances in Information Technology (RAIT), 2012 1st International Conference on*, pages 461–467. IEEE, 2012.

[VB05] Gustavo MD Vieira and Luiz E Buzato. Chksim: A distributed checkpointing simulator. *Technical Report IC-05-034*, 2005.

[VG10] Angel Jesus Varela Vaca and Rafael Martínez Gasca. Opbus: Fault tolerance against integrity attacks in business processes. In Álvaro Herrero, Emilio Corchado, Carlos Redondo, and Ángel Alonso, editors, *Computational Intelligence in Security for Information Systems 2010*, volume 85 of *Advances in Intelligent and Soft Computing*, pages 213–222. Springer Berlin Heidelberg, 2010.

[VGBH11] Angel Vaca, Rafael M. Gasca, Diana Borrego, and Sergio Pozo Hidalgo. Fault tolerance framework using model-based diagnosis: towards dependable business processes. *International Journal on Advances in Security*, 4(1 and 2):11–22, 2011.

[VM12] A Vani Vathsala and Hrushikesha Mohanty. Using hmm for predicting response time of web services. In *Proceedings of the CUBE International Information Technology Conference*, pages 520–525. ACM, 2012.

[VM14a] A. Vani Vathsala and Hrushikesha Mohanty. Interaction patterns based checkpointing of choreographed web services. In *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, PESOS 2014, pages 28–37, New York, NY, USA, 2014. ACM.

[VM14b] A. Vani Vathsala and Hrushikesha Mohanty. A survey on checkpointing web services. In *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, PESOS 2014, pages 11–17, New York, NY, USA, 2014. ACM.

[VSMRPHD18] Mariano Vargas-Santiago, Luis Morales-Rosales, Saúl Pomares-Hernández, and Khalil Drira. Autonomic web services enhanced by asynchronous checkpointing. *IEEE Access*, 6:5538–5547, 2018.

[VSPHRHK17] Mariano Vargas-Santiago, Saúl Pomares-Hernandez, Luis A. Morales Rosales, and Hatem Hadj-Kacem. Survey on web services fault tolerance approaches based on checkpointing mechanisms. *Journal of Software*, 12(7), 2017.

[WFB⁺04] Wenjun Wu, Geoffrey C Fox, Hasan Bulut, Ahmet Uyar, and Harun Altay. Design and implementation of a collaboration web-services system. 2004.

[YCD⁺09] Jianwei Yin, HHanwei Chen, Shuiguang Deng, Zhaohui Wu, and Calton Pu. A dependable esb framework for service integration. *Internet Computing, IEEE*, 13(2):26–34, 2009.

[Zha07a] Wenbing Zhao. Bft-ws: A byzantine fault tolerance framework for web services. In *EDOC Conference Workshop, 2007. EDOC'07. Eleventh International IEEE*, pages 89–96. IEEE, 2007.

[Zha07b] Wenbing Zhao. A lightweight fault tolerance framework for web services. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 542–548. IEEE Computer Society, 2007.

[Zha09] Wenbing Zhao. Design and implementation of a byzantine fault tolerance framework for web services. *Journal of Systems and Software*, 82(6):1004–1015, 2009.

[ZJ03] Liang-Jie Zhang and Mario Jeckle. The next big thing: Web services collaboration. In *Web Services-ICWS-Europe 2003*, pages 1–10. Springer, 2003.

[ZL10] Zibin Zheng and Michael R Lyu. An adaptive qos-aware fault tolerance strategy for web services. *Empirical Software Engineering*, 15(4):323–345, 2010.

[ZL12] Zibin Zheng and Michael R Lyu. Optimal fault tolerance strategy selection for web services. In *Web Service Composition and New Frameworks in Designing Semantics: Innovations*, pages 218–237. IGI Global, 2012.

[ZL13] Zibin Zheng and Michael R Lyu. Qos-aware fault tolerance for web services. In *QoS management of Web services*, pages 97–118. Springer, 2013.

[ZL15] Zibin Zheng and Michael R Lyu. Selecting an optimal fault tolerance strategy for reliable service-oriented systems with local and global constraints. *IEEE Transactions on Computers*, 64(1):219–232, 2015.

Online References

[Cha13] Chanaka, F. (2013, December 09). Building an In-Order, Guaranteed Delivery Messaging System for Your Enterprise with WSO2 Products. Retrieved March 28, 2016, from goo.gl/pf8hac, Guaranteed Delivery Messasing System

[Ora14] Oracle®. (2014, February 12). Fusion Middleware Programming JMS for Oracle WebLogic Server. Retrieved March 28, 2016, from https://docs.oracle.com/cd/E24329_01/web.1211/e24387/toc.htm, Oracle WebLogic Server

[Sch11] Schumacher, D. (2011, July 15). In-Order Message Delivery. Retrieved March 28, 2016, from http://www.dalnefre.com/wp/2011/07/in-order-message-delivery/, In-Order Message Delivery

APPENDIX A

A.1 Definitions

Throughout this thesis we have mentioned concepts like reliability, domino-effect and dependability. Here we present, as possible, their formal definitions.

Reliability: The IEEE (1990) defined it as: "The ability of a system or component to perform its required functions under stated conditions for a specific period of time".

Domino-effect: The phenomenon of cascaded rollback is called the domino effect. Just to clarify, consider the situation where the sender of a message m rolls back to a state that precedes the sending of m. The receiver of m must also roll back to a state that precedes *m's* receipt; otherwise, the states of the two processes would be inconsistent because they would show that message m was received without being sent, which is impossible in any correct failure-free execution.

Dependability: the classical definition of dependability encompasses the attributes of reliability, availability, safety, integrity and maintainability. Avizienis et al. [ALR+01] gave two definitions for dependability:

- "The ability to deliver service that can be justifiably trusted."
- "The ability to avoid service failures that are more frequent and more severe than is acceptable"

A.2 Fuzzy Rules

In this appendix we show the fuzzy rules used for diagnostic purposes, as presented in Chapter 6. Shown in Table 1 and Table 2 for illustration.

Table 1: Inference Rules

1. If (RT is VG) and (CPU is CPU_L) and (MEM is MEM_L) then (QoS is QoS_VH)
2. If (RT is VG) and (CPU is CPU_L) and (MEM is MEM_M) then (QoS is QoS_H)
3. If (RT is VG) and (CPU is CPU_L) and (MEM is MEM_H) then (QoS is QoS_H)
4. If (RT is VG) and (CPU is CPU_L) and (MEM is MEM_VH) then (QoS is QoS_A)
5. If (RT is VG) and (CPU is CPU_M) and (MEM is MEM_L) then (QoS is QoS_VH)
6. If (RT is VG) and (CPU is CPU_M) and (MEM is MEM_M) then (QoS is QoS_H)
7. If (RT is VG) and (CPU is CPU_M) and (MEM is MEM_H) then (QoS is QoS_A)
8. If (RT is VG) and (CPU is CPU_M) and (MEM is MEM_VH) then (QoS is QoS_L)
9. If (RT is VG) and (CPU is CPU_H) and (MEM is MEM_L) then (QoS is QoS_A)
10. If (RT is VG) and (CPU is CPU_H) and (MEM is MEM_M) then (QoS is QoS_A)
11. If (RT is VG) and (CPU is CPU_H) and (MEM is MEM_H) then (QoS is QoS_L)
12. If (RT is VG) and (CPU is CPU_H) and (MEM is MEM_VH) then (QoS is QoS_L)
13. If (RT is VG) and (CPU is CPU_VH) and (MEM is MEM_L) then (QoS is QoS_A)
14. If (RT is VG) and (CPU is CPU_VH) and (MEM is MEM_M) then (QoS is QoS_A)
15. If (RT is VG) and (CPU is CPU_VH) and (MEM is MEM_H) then (QoS is QoS_L)
16. If (RT is VG) and (CPU is CPU_VH) and (MEM is MEM_VH) then (QoS is QoS_L)
17. If (RT is G) and (CPU is CPU_L) and (MEM is MEM_L) then (QoS is QoS_H)
18. If (RT is G) and (CPU is CPU_L) and (MEM is MEM_M) then (QoS is QoS_H)
19. If (RT is G) and (CPU is CPU_L) and (MEM is MEM_H) then (QoS is QoS_A)
20. If (RT is G) and (CPU is CPU_L) and (MEM is MEM_VH) then (QoS is QoS_L)
21. If (RT is G) and (CPU is CPU_M) and (MEM is MEM_L) then (QoS is QoS_H)
22. If (RT is G) and (CPU is CPU_M) and (MEM is MEM_M) then (QoS is QoS_A)
23. If (RT is G) and (CPU is CPU_M) and (MEM is MEM_H) then (QoS is QoS_L)
24. If (RT is G) and (CPU is CPU_M) and (MEM is MEM_VH) then (QoS is QoS_L)
25. If (RT is G) and (CPU is CPU_H) and (MEM is MEM_L) then (QoS is QoS_A)
26. If (RT is G) and (CPU is CPU_H) and (MEM is MEM_M) then (QoS is QoS_A)
27. If (RT is G) and (CPU is CPU_H) and (MEM is MEM_H) then (QoS is QoS_L)
28. If (RT is G) and (CPU is CPU_H) and (MEM is MEM_VH) then (QoS is QoS_L)
29. If (RT is G) and (CPU is CPU_VG) and (MEM is MEM_L) then (QoS is QoS_A)
30. If (RT is G) and (CPU is CPU_VH) and (MEM is MEM_M) then (QoS is QoS_A)
31. If (RT is G) and (CPU is CPU_VH) and (MEM is MEM_H) then (QoS is QoS_L)
32. If (RT is G) and (CPU is CPU_VH) and (MEM is MEM_VH) then (QoS is QoS_L)
33. If (RT is A) and (CPU is CPU_L) and (MEM is MEM_L) then (QoS is QoS_A)
34. If (RT is A) and (CPU is CPU_L) and (MEM is MEM_M) then (QoS is QoS_A)
35. If (RT is A) and (CPU is CPU_L) and (MEM is MEM_H) then (QoS is QoS_L)
36. If (RT is A) and (CPU is CPU_L) and (MEM is MEM_VH) then (QoS is QoS_L)
37. If (RT is A) and (CPU is CPU_M) and (MEM is MEM_L) then (QoS is QoS_A)
38. If (RT is A) and (CPU is CPU_M) and (MEM is MEM_M) then (QoS is QoS_A)
39. If (RT is A) and (CPU is CPU_M) and (MEM is MEM_H) then (QoS is QoS_L)
40. If (RT is A) and (CPU is CPU_M) and (MEM is MEM_VH) then (QoS is QoS_L)
41. If (RT is A) and (CPU is CPU_H) and (MEM is MEM_L) then (QoS is QoS_A)
42. If (RT is A) and (CPU is CPU_H) and (MEM is MEM_M) then (QoS is QoS_A)
43. If (RT is A) and (CPU is CPU_H) and (MEM is MEM_H) then (QoS is QoS_L)
44. If (RT is A) and (CPU is CPU_H) and (MEM is MEM_VH) then (QoS is QoS_L)
45. If (RT is A) and (CPU is CPU_VH) and (MEM is MEM_L) then (QoS is QoS_A)
46. If (RT is A) and (CPU is CPU_VH) and (MEM is MEM_M) then (QoS is QoS_L)
47. If (RT is A) and (CPU is CPU_VH) and (MEM is MEM_H) then (QoS is QoS_L)
48. If (RT is A) and (CPU is CPU_VH) and (MEM is MEM_VH) then (QoS is QoS_VL)
49. If (RT is B) and (CPU is CPU_L) and (MEM is MEM_L) then (QoS is QoS_A)

Table 2: Inference Rules

50. If (RT is B) and (CPU is CPU_L) and (MEM is MEM_M) then (QoS is QoS_A)
51. If (RT is B) and (CPU is CPU_L) and (MEM is MEM_H) then (QoS is QoS_L)
52. If (RT is B) and (CPU is CPU_L) and (MEM is MEM_VH) then (QoS is QoS_L)
53. If (RT is B) and (CPU is CPU_M) and (MEM is MEM_L) then (QoS is QoS_A)
54. If (RT is B) and (CPU is CPU_M) and (MEM is MEM_M) then (QoS is QoS_A)
55. If (RT is B) and (CPU is CPU_M) and (MEM is MEM_H) then (QoS is QoS_L)
56. If (RT is B) and (CPU is CPU_M) and (MEM is MEM_VH) then (QoS is QoS_L)
57. If (RT is B) and (CPU is CPU_H) and (MEM is MEM_L) then (QoS is QoS_A)
58. If (RT is B) and (CPU is CPU_H) and (MEM is MEM_M) then (QoS is QoS_A)
59. If (RT is B) and (CPU is CPU_H) and (MEM is MEM_H) then (QoS is QoS_L)
60. If (RT is B) and (CPU is CPU_H) and (MEM is MEM_VH) then (QoS is QoS_L)
61. If (RT is B) and (CPU is CPU_VH) and (MEM is MEM_L) then (QoS is QoS_A)
62. If (RT is B) and (CPU is CPU_VH) and (MEM is MEM_M) then (QoS is QoS_A)
63. If (RT is B) and (CPU is CPU_VH) and (MEM is MEM_H) then (QoS is QoS_L)
64. If (RT is B) and (CPU is CPU_VH) and (MEM is MEM_VH) then (QoS is QoS_VL)
65. If (RT is VB) and (CPU is CPU_L) and (MEM is MEM_L) then (QoS is QoS_A)
66. If (RT is VB) and (CPU is CPU_L) and (MEM is MEM_M) then (QoS is QoS_A)
67. If (RT is VB) and (CPU is CPU_L) and (MEM is MEM_H) then (QoS is QoS_L)
68. If (RT is VB) and (CPU is CPU_L) and (MEM is MEM_VH) then (QoS is QoS_L)
69. If (RT is VB) and (CPU is CPU_M) and (MEM is MEM_L) then (QoS is QoS_A)
70. If (RT is VB) and (CPU is CPU_M) and (MEM is MEM_M) then (QoS is QoS_A)
71. If (RT is VB) and (CPU is CPU_M) and (MEM is MEM_H) then (QoS is QoS_L)
72. If (RT is VB) and (CPU is CPU_M) and (MEM is MEM_VH) then (QoS is QoS_L)
73. If (RT is VB) and (CPU is CPU_H) and (MEM is MEM_L) then (QoS is QoS_A)
74. If (RT is VB) and (CPU is CPU_H) and (MEM is MEM_M) then (QoS is QoS_A)
75. If (RT is VB) and (CPU is CPU_H) and (MEM is MEM_H) then (QoS is QoS_L)
76. If (RT is VB) and (CPU is CPU_H) and (MEM is MEM_VH) then (QoS is QoS_VL)
77. If (RT is VB) and (CPU is CPU_VH) and (MEM is MEM_L) then (QoS is QoS_A)
78. If (RT is VB) and (CPU is CPU_VH) and (MEM is MEM_M) then (QoS is QoS_L)
79. If (RT is VB) and (CPU is CPU_VH) and (MEM is MEM_H) then (QoS is QoS_VL)
80. If (RT is VB) and (CPU is CPU_VH) and (MEM is MEM_VH) then (QoS is QoS_VL)