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

Mariano Vargas Santiago, Saúl E. Pomares Hernández, Hatem Hadj Kacem, Luis A. Morales Rosales

Technical Report No. CCC-15-005
December 2015

Computational Sciences Coordination
Instituto Nacional de Astrofísica, Óptica y Electrónica
Tonantzintla, Puebla, Mexico

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

Mariano Vargas-Santiago[a] Saúl E. Pomares-Hernández[a, b, c] Hatem Hadj-Kacem[d]
Luis A. Morales Rosales[e]

[a]Computer Science Department at Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE)
Luis Enrique Erro # 1, Santa María Tonantzintla, Puebla, CP. 72840, Mexico.
[b]LAAS-CNRS, Tolouse, France
[c] Univ. de Toulouse, Tolouse, France
[d] ReDCAD Laboratory, Sfax, Tunisia
[e] Instituto Tecnológico Superior Misantla, Veracruz, México.
E-mail: mariano.v.santiago@ccc.inaoep.mx

Abstract

IBM created a new paradigm called Autonomic Computing (AC); where systems are seen as self-manageable. To attack the problem of intercommunication and heterogeneity the Enterprise Services Bus (ESB) technology arises. Big enterprises need their business information systems to communicate with each other where millions of online commercial interactions per second can take place. Enterprises are concern about their successful business processes termination; in this sense the ESBs systems in charge of the integration must be dependable. Therefore, the design of more dependable ESBs is a necessity. Nevertheless, while adopting dependable solutions some of the existing approaches affect the systems' performance and in few other cases even the scalability of the system is jeopardized. Such issues can be diminished by injecting self-healing within the ESBs systems. A useful tool for this purpose can be the technique known as Communication-induced Checkpointing (CiC).

Keywords: *Distributed systems, autonomic computing, self-healing, Enterprise Service Bus, checkpointing, Web services.*

Contents

1 Introduction

The Autonomic computing (AC) paradigm was introduced by IBM dozen years ago [15], due to todays systems' complexity. Computing systems are constantly increasing their complexity which brings up many challenges for software developers and system managers who design, integrate, evaluate and manage these systems [10]. Even the most skilled system managers will have trouble in the near future while trying integrating, tuning, and detecting issues. On one hand, applications are constituted of millions of lines of code. On the other hand trying to extend them beyond company boundaries into the Internet increases the complexity. The AC vision considers the computing systems as self-manageable, given high-level objectives from administrators. To achieve this four major aspects must be considered: self-configuration, self-healing, self-optimization and self-protection, also known as S-CHOP.

Additionally computing systems are being increasingly distributed and operate over heterogeneous environments. To attack the problem of design more distributed and reusable software the Service-Oriented Architecture (SOA) was created. SOA is based on Web-Services (WS) and it is standard-based; distributed, loosely coupled, reusable software components that encapsulate a discrete functionality, and it can be accessed using standard Internet and XML-based protocols. Nowadays, several network applications and distributed systems are designed as a composition of distributed services following the SOA paradigm.

In order to attack the problem of heterogeneity the Enterprise Service Bus (ESB) system arises. An ESB is a middleware based on SOA that allows the integration of heterogeneous applications. It provides foundational services for more complex SOA systems via an XML-based messaging engine (the bus), and thus provides an abstraction layer on top of an enterprise messaging system that allows integration developers to exploit the messaging without writing code [41].

Together SOA and ESB provide an infrastructure, that enable communications between different applications, running on different platforms and written in diverse languages. An ESB allows faster and cheaper integration of systems and it permits to have a better scalability solution for enterprise distributed deployments. The ESB systems provides flexibility in updating systems as the requirements change. In summary, the ESB plays the critical role of connecting heterogeneous applications and services in an SOA. The ESB standard is currently the most used for system integration in business information systems, that includes business to business (B2B) collaboration, service-oriented computing (SOC) and standard based communication infrastructure [37].

Big enterprises need their business information systems to intercommunicate among each other and/or with third party systems in an environment where millions of on-line commercial interactions per second can take place. Enterprises are concern about their successful business processes termination; in this sense the ESB system in charge of the integration must be scalable and fault-tolerant; therefore dependable. Hence, the ESB must be available 24/7 and guarantee a certain Quality of Service (QoS). To support this kind of work of being dependable ensuring proper functioning over time, there is ongoing research [47], [2], [16], [17] these works try to inject the self-healing property into the ESB system to make it more dependable; but unfortunately they jeopardize the systems performance, have a high cost of implementation, are not scalable and are merely for diagnosis purposes. Dependability has been a wide area of research, a whole taxonomy about it can be found in [3]. Many different aspects of the system affects its dependability: reliability, availability, security, maintainability and consider systems to be self-manageable [7]. In our research dependability shall consider two aspects: reliability and availability which we propose to be achieved by self-healing and checkpointing protocols.

Self-healing is oriented towards the recollection of data, from which the system can detect, diagnose, plan and repair potential problems that arise from software/hardware components. An useful tool for this purpose can be the technique known as Communication-induced Checkpointing (CiC). In general, Checkpointing Protocols (CPs) have been used to manage a wide range of problems in distributed systems area like: rollback recovery, software debugging and software validation among others. As the complexity of todays systems grows, where pervasive computing is not a dream anymore, the costs in maintenance are cutting significantly the enterprises budgets; there exists a reciprocal relation between this complexity and maintaining the systems. In this context, a need for practical solutions like evolving to the self-healing capability of autonomic computing is feasible. Moreover, while evolving systems to self-healing the systems in play shall not be altered or affected by the way self-healing is adapted, to our knowledge, taking checkpoints and recovering from such does not affect. However, some considerations must be taken into account when proposing solutions for distributed systems; these communicate solely by message passing, there is no physical reference or time and processes do not share common memory. There have been many attempts to solve the time issue in distributed systems starting with the work carried out by Lesli Lamport [21] and this issue continues to be a topic of interest up until today [39].

Therefore the synthesis of these two widely used paradigm is a challenge that to our knowledge remains open. For that reason, we propose to develop more efficient dependable ESB systems through checkpointing protocols towards autonomic computing; particularly oriented to offer self-healing services.

The research proposal is structured as follows: Section 2 gives a brief background on two major sections: software paradigms and distributed computing. In Section 3 the problem description of this work is given. Section 4 discusses the related works relevant to the statement of the problem. A detailed description of the research proposal is given in Section 5: research aim, objectives, contributions and questions along with the methodology. Section 6 presents a tentative work plan. Finally, Section 7 illustrates the advances till the time of writing this work. Finally, preliminary conclusion and future work directions are discussed in Section 8.

2 Background

2.1 Software Paradigms

Definitions

Dependability: the classical definition of dependability encompasses the attributes of reliability, availability, safety, integrity and maintainability [40]. Avizienis et al. [3] gave two definitions for dependability:

1. “The ability to deliver service that can be justifiably trusted”.
2. “The ability to avoid service failures that are more frequent and more severe than is acceptable”.

Mission Critical Systems: It is a system on which depends the correct execution of the operations of a business or organization. When it fails, business operations are significantly impacted [1]

[1]http://www.techopedia.com/definition/23583/mission-critical-system

Table 1. Four aspects of self-management as they are now and would be with autonomic computing reproduced from [15].

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.

2.1.1 Autonomic Computing

The notion of ACSs was proposed by IBM in 2001 [15], as a system with monitoring and analysis capabilities of the states of its components. An ACS can detect problems that arise from failure and continue operation performing maintenance and/or adjustment parameters in QoS degradation without human intervention. In general ACSs 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 1, showing four aspects of self-management as they are now and how should be under the AC paradigm.

Many of the big enterprises like HP, Sun, IBM are trying to evolve their system to the AC paradigm [36], having as main goal the self-managing of systems. Often ACSs are being tackled globally; for example Java programs with self-healing and self-optimizing properties studied in [30]. 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 [16]; the authors argue they transformed such ESB to an Autonomic Service Bus (ASB). But all the computing is done offline, 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 [2], where the authors propose a self-healing architecture for Airports systems integration. We will focus on the self-healing property; which in [36] remains still as an open challenge, and it is currently under study by academic researchers and by the industry.

As stipulated by the AC paradigm, systems must evolve to self-management. To achieve such 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)* 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 [17]. However, experts in the field must choose 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 in 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's 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, our in their time, were good integration technologies like: point-to-point and Enterprise Application Integration (EAI) [6]. 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 [35]. To mitigate such complexity and heterogeneity imposed by the traditional point-to-point integration technology the EAI arises, where all the communication pases 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 [41].

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 WS. SOA is an architecture that can build business applications from a set of loosely coupled black box components [4]. 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 [28]. 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 [28].

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 [6].

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 [1].

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 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 characteristic are that 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 $P = \{P_1, P_2, \dots, P_n\}$. We assume that channels have an unpredictable yet finite transmission delay and that these 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 Happens-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 [21], 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 [20]."

Leslie Lamport defined the HBR trying to totally order events (sending or reception of messages) in a distributed system [21]. 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. *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 e3 such that e1 $\rightarrow$ e2 and e2 $\rightarrow$ e3, then e1 $\rightarrow$ e3 (transitive property).*

- *Given an event e1, e1 $\nrightarrow$ e1 (irreflexive property)* [2]

- *Given two events e1 and e2, if e1 $\rightarrow$ e2 then e2 $\nrightarrow$ e1 (antisymmetric property).*

Definition 2. *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. [32] 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 *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, $\neg(e1 \rightarrow e3 \rightarrow e2)$.*

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

Definition 2.1. 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 $\widehat{E}$.

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

An example of a CCP is exhibited in Figure 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 $(x > 0)$

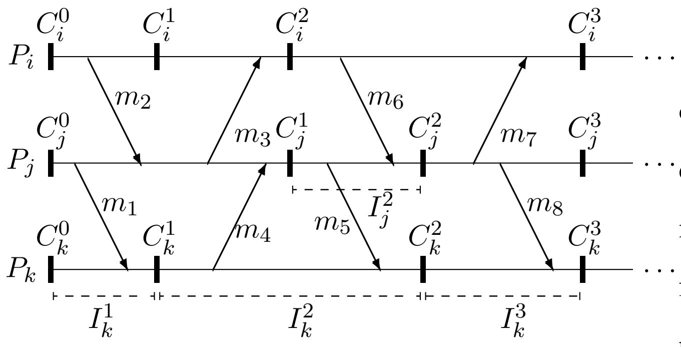


Figure 1. CCP [38].

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 [22]. Simon et al., introduce *save* checkpoint conditions, with those they identified when a forced checkpoint can be removed and/or delayed, details of which are in [38].

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 [29]. Now that we have defined the fundamentals needed to take a consistent global snapshot or simply checkpoint, we give a brief introduction to it.

2.2.5 Checkpointing Protocols

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 consist on 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 systems computation, so in case of failure the system can recover partially or totally.

Distributed systems are ubiquitous but are not fault-tolerant but 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 to 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 checkpointed state is called *rollback recovery*” [20].

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.

2.2.6 Asynchronous or uncoordinated checkpointing

Uncoordinated checkpointing in a distributed system consists of each participant taking its own checkpoint, the main flaw for this approach is that it is susceptible to the domino effect; this effect is also know in the literature as rollback propagation where processes that should not rollback do so. There advantages are that they eliminate the synchronization overhead and have lower overhead during normal execution [20]. Juang and Venkatesan [14] proposed an uncoordinated checkpointing algorithm.

2.2.7 Synchronous or Coordinated checkpointing

Coordinated checkpointing solves the domino effect flaw of uncoordinated checkpointing since process always restart 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 by two principal manners: either the sending of messages is blocked for the duration of the protocol or, checkpointing indexes are piggybacked to avoid locking. Koo and Toueg’s [18] coordinated checkpointing takes two kind of checkpoints on the stable storage: permanent and tentative, however this algorithm assumes that a single process invokes the checkpointing algorithm when taking a permanent checkpoint.

2.2.8 Quasi-synchronous or communication-induced checkpointing

In the communication-induced checkpointing processes take checkpoints based on the control information piggybacked on the application messages it receives 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 [20]. Many studies have been carried out for CiC [23] [11] [38].

There exist three divers ways for checkpointing: virtual machine snapshotting, application level checkpointing and transparent checkpointing.

Virtual Machine Snapshotting

This kind of checkpointing is basically for checkpointing virtual machines; saving the state of the entire virtual machine commonly used for virtual machine migration and it is a slow solution since it generates larger checkpoint images, a extensive discussion can be found in a survey made by Medina et al. [27].

Application Level Checkpointing

In this checkpointing technique programmers or developers inserts code to the applications, this requires knowledge of the application; making this solution application-specific, in order for such to be able to take checkpoints and save the process state. With such saved state in case of failure the system can roll back. There are several techniques [46] and frameworks that provide tools to assist in application-level checkpointing. Examples include Pickle for Python and Boost serialization for C++.

Transparent Checkpointing

In this checkpointing technique neither the source or binary code suffer changes; it is also know as system level checkpointing. There are two main categories: user-level MPI libraries and kernel-level checkpointing.

3 Problem Description

Computing systems complexity is overpassing the most experienced systems managers or administrators, needing teams of experts for problems resolution and being a time consuming task for a human team. To mitigate this situation IBM's future vision of autonomic computing paradigm arises [15] as a promising solution; specifically tackling such issues with 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 [17]. These two contexts: autonomic computing, specifically self-healing, and systems integration, ESB systems, can be better tackled by dependability. Since dependability for systems can be seen as the ability of such to avoid failures that are severe and frequent and affect the quality perceived by the end users. Notwithstanding that there exist sophisticated solutions that suggest dependability, they have drawbacks: in systems performance, implementation costs [47], others jeopardize scalability [2] and a few are for systems' diagnosis only [16] [17]. Definitely, there is a demand for more efficient dependable solutions and self-healing can be achieved through checkpointing protocols which have gain a great level of maturity. For a clear understanding of the challenges involved in developing dependability for ESBs systems having as foundation self-healing and checkpointing protocols, we have divided this section into three parts. Firstly, we explain why messages ordering within ESB systems is important. 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.

3.1 Partial ordering algorithms for ESBs

Before we start talking about checkpointing algorithms and protocols some considerations must be contemplated. For example, in distributed and heterogeneous networked environments intercommunications are merely accomplished by business processes exchanging messages and the order of how such events happened is well-known. Nonetheless, distributed systems are characterized for not having a common or reliable notion of time. As illustrated by Lamport in [21], 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, with the causality concept or an event *happening before* another, we could have an approximation of it. The 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 [9], [33], [34] 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 [20].

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 [12] 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, this eases the task of designing domino effect free checkpointing protocols.

3.2 Characteristics of Checkpointing Protocols and Self-Healing for ESBs systems

The checkpointing algorithms have been used purely for distributed and parallel systems, and have proved to be useful as a fault-tolerant mechanism among other issues like: software debugging and validation [20]. 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 a small degradation, if the checkpointing protocol (CP) is efficient, and no benefit was obtain from it. 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 [30] [38] and suggested it is possible to integrate self-healing and checkpointing protocols. Here we will identify which autonomic computing aspects can be solved making use of checkpointing protocols. For example, we initially identified that the rollback recovery mechanism can be useful to tackle the problem of self-healing. On the other hand it is almost certain that rollback recovery solutions based on checkpointing protocols can not be directly applied towards autonomic computing, since it presents the important characteristic of self-management in face of: adding/erasing services, participants, hardware/software components, and the availability of resources. Finally, we need to verify certain properties such as the absence of z-paths and z-cycles are met as they may result or lead to inconsistencies; this aided by the *partial ordering algorithms for ESBs* previously mentioned.

Hence, we want to develop a new checkpointing protocol aided with self-healing that can dynamically adapt to the systems needs. For example, when there is a low level of QoS degradation then the system must not take many checkpoints, and vice versa when the QoS degradation is high the system should take many checkpoints. Because of this dynamism, the developed checkpointing protocol must take into account situations faced on previous statements.

3.3 Self-healing and Checkpointing Protocols synthesis

In this section we will talk about the identified problems while trying to synthesize self-healing and checkpointing protocols. For instance, self-managed systems implement the MAPE (Monitoring, Analysis, Planning and Execution) loop as consequence self-healing systems shall be able to: diagnose, monitor, plan and execute recovery actions. At an early stage, we have identified that to give more dependability to the ESBs systems based on self-healing and checkpointing protocols the MAPE loop can be implemented. Taking as managed element the entire ESB system; the *Monitoring* phase will monitor the QoS: response time, transactions per second, percentage usage of the CPU and RAM memory. One challenge presented here is the frequency rate with which we shall *Monitor* the ESB system; if is done too frequently its consequence may be sever degradation to the ESBs systems performance, if is done in a slow rate then when rollback recovery takes action it could reverse to a point in time where everything was working properly and must re-do the same actions. Therefore, it is indispensable to optimize this *Monitor* frequency rate and if possible link it with the checkpointing generation rate, another open challenge identified; since, the QoS and the checkpointing generation are correlated . . . The *Analysis* phase can be accomplished through the use of Markov Chains or using Bayesian Networks; here the challenge is to gather enough information so these approaches can make QoS predictions at runtime, which leads to dynamically generate checkpoints. We believe that optimizing the number of checkpoints is a good strategy because it reduces a lot of communication overhead that is generated by the communication-induced checkpoint algorithms. We think that the above can be done predicting the variation of QoS, which represents a good indicator of correctness or malfunction of the system. For the *Planning* phase we believe that with the retrieved predictions from the analysis phase we could schedule the next forced checkpoint(s). Finally, the *Execution* phase will be in charge of applying rollback recovery mechanism when systems failures exists or when there is a violation on the QoS policies that we will establish.

Summary

In the most general view our research problem is to gain a more dependable ESB system by developing self-healing services and dynamic checkpointing protocols; following that whenever a there is systems failure/degradation or QoS violation detected the system can implement rollback recovery mechanisms. The Fig. 2 illustrates our scope/domain.

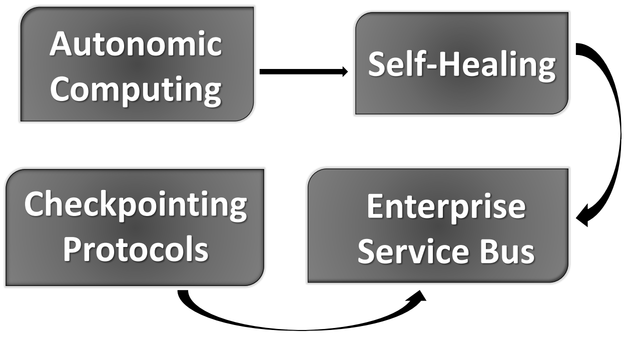


Figure 2. Problem Description.

Since our proposal will be able to adapt to the dynamism of the environment and to evaluate its state; supporting fault tolerance, we could conclude we will be developing an Autonomic Service Bus (ASB) as shown in Fig. 3.

As illustrated by Fig. 3 the managed elements will be components which can be monitored: CPU and RAM. A common knowledge base is needed in order to take decisions based on previous experiences, in other words based on the probability of systems degradations. These decisions will be taken by adopting the MAPE loop cycle.

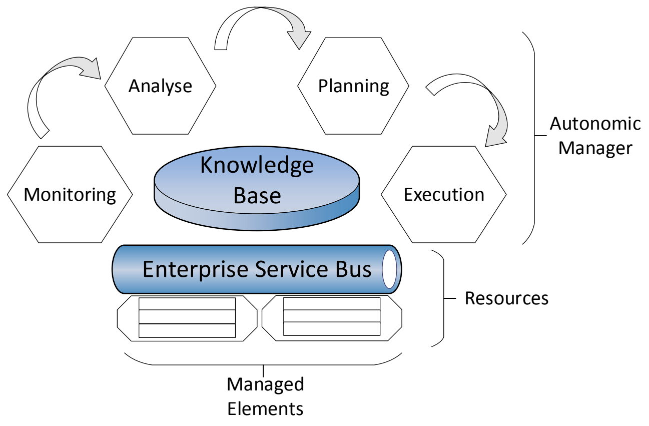


Figure 3. Autonomic Service Bus.

4 Related Works

In this section we present some recommended solutions for dependable systems that suggest to use the autonomic computing (AC) and checkpointing protocols (CP) paradigms under distributed systems, specifically those for business processes, as are: Service-Oriented Architecture (SOA), Web Services (WS), Enterprise Service Bus (ESB). In the first part we present some proposed solutions that combine both SOA and checkpointing protocols paradigms.

In the second part we present some fo the emerging solutions in the WS field that suggest the use of CP, and also we present a comparative analysis and a discussion of the limitations of the emerging solutions.

In the last part we discuss, dependability under general environments such as Service-Oriented Computing (SOC), autonomic dependable systems and then we narrow down our research to a more punctual topic; ESB with self-healing capabilities which implies fault tolerance techniques. Also, we discuss the limitations and drawbacks of the current solutions.

4.1 SOA and Checkpointing

Hiltunen and Schlichting in [13] present that currently most of the organizations are following the SOA paradigm where services and applications are constructed as collections of interacting services. SOA follows the Service-Oriented Computing (SOC) approach, where software systems built across machines in a network environment discover their functionality through well-defined interfaces. One of the main attractions of SOA is that services are independent, however the authors stipulate that dependability is achieved in a cooperative manner by: services, service providers and resource providers; warranting a certain Quality of Service (QoS). This work's central goal is to highlight the architectures dependability implications based on decoupled and independent services such as SOAs and to offer possible enhancements for dependability.

The focus of this paper is on SOAs' *dependability*. Dependability is a concept that includes a number of QoS aspects of the service in the architecture: *service availability, service reliability, service timeliness and security*.

- Service availability, measures the probability that a service instance is reachable and operational when the service is requested.
- Service reliability, measures the probability that a service provides correct response when invoked.
- Service timeliness, measures the response time of the service.
- Security, addresses the privacy and integrity issues of the services and the overall services.

The authors state that the dependability issues in SOAs are caused by the very same features that make them attractive as a software structuring paradigm, they call it the *dependability dilemma*. Paraphrasing Leslie Lamport, defining SOA as "an architecture in which the failure of a service you didn't even know exited can render your application unusable". To solve this dilemma any successful SOA requires collaborative QoS.

Dependability which implies fault tolerance in distributed systems has advantages over centralized architectures, because machine faults are often independent from one another. But, other dependability challenges are introduced like: handling the partial failure of an executing program when a machine crashes and conserving data coherency across machines in such situations. Many proposals for increasing SOAs dependability involve: external monitoring of services and if a failure occurs then rebinding to an alternative implementation. Yet, have numerous limitations: *Lack of alternatives, application state rollback, monitoring delay and other monitoring limitations*. We are interested on the application rollback; "A service may maintain state on behalf of the application client. In such situations, rebinding to a different service instance may require a rollback at the client to the point where the state was stored or created. If the client provides interactive service to a human user, such a rollback can cause unintended consequences and possible discontinuities in the user experience. If the service's client is actually another service, a cascading rollback may occur affecting multiple applications and numerous users. Even if the service state is checkpoint periodically, alternative service instances may not be able to interpret the checkpoint and even if they do, a small rollback may still be required."

In addition Hiltunen and Schlichting, suggest that to mitigate the *dependability dilemma* services need to collaborate with services they use and services that use them, and consider two types of collaboration: between service consumer-provider and among all of the consumers of a service provider. For the producer-consumer collaborative QoS to implement collaboration they do it by means of *translucent QoS interfaces* exposed in addition to their normal service interfaces. The authors consider three specific variants of this type of QoS interface: (1) an interface that reports current QoS values, (2) an interface that exposes QoS mechanisms, and (3) an interface that can be used to negotiate QoS requirements. We are interested in the second interface for exposing QoS mechanism, exposing details of the dependability mechanism used by the service enables more powerful forms of collaborative QoS. In collaborative checkpointing, the service provider exposes the checkpoint of the consumer's session state to the consumer to be stored and used later if necessary. For example, the response messages for service requests that modify the session state are enhanced to include a checkpoint of the service session state. Then, an additional operation is provided for the service consumer to restore the service provider with the stored checkpoint.

We are interested on this kind of solutions, to our believe enhancing or adding a QoS API is a good and sophisticated strategy, however, a communication-induced checkpointing (CiC) protocol may come in handy and suitable in this context, considering that their is a necessity of efficiently take checkpoints (small overhead and domino effect-free).

4.1.1 Summary

Although the Service-Oriented Architecture (SOA) is an attractive software structuring solution, for interacting services, dependability depends on all the participants of the system as declared by the Hiltunen and Schlichting. Adding additional interfaces, as the QoS collaborative interface, can come in handy, nonetheless may also jeopardize the systems performance. Plus this solution considers taking checkpoints by the provider of the service and saving the consumers state without considering consistency, this do not contemplate consistent global snapshots.

4.2 Web Services and Checkpointing

Marzouk et al. in [24] illustrate that web services (WS) are implemented to follow a business logic, they could be deployed either locally inside an enterprise or they could cooperate in a distributed environment, either way WS are crucial and must implement fault tolerance mechanisms. WS orchestrations are a composition of several web services that allow fast and modular creation of complex applications. The execution of services in a workflow form is usually done using the Business Process Execution Language (BPEL) engine. The authors principal concern is about ensuring self-adaptivity of composite services, they provide an approach for adapting composite WS based on *strong mobility* code; achieved with generic source code transformation. The authors identified that the strong mobility approach for WS-BPEL processes requires periodic checkpointing. When self-adaptivity is a necessity one or several instances of the orchestration process will be migrated to the previous checkpoint and resumed starting from the interruption point; beginning from their last captured checkpoint. Plus, they also argue that their approach guarantees checkpoint consistency, of the orchestration process at migration time.

Strong mobility of orchestrated processes consists on stopping all or a subset of executing instances and to migrate them to another node resuming them from their last checkpoint. Marzouk et al. determined two cases: firstly, mobility could be launched because of node failure or unavailability; all orchestration processes instances are interrupted, migrated, rollback and resumed starting from the last checkpoint. Secondly, when within the orchestration process QoS might of been violated; a forced mobility is employed by migrating a subset of the running instances to an orchestration process replica to decrease the load of the initial one. For evaluating purposes the authors used the travel agency application and measured the BPEL performance with and without their proposal.

However the authors do not make mention of which checkpointing protocol they used, nor if this solution has low overhead and computational cost, or even if it is domino effect-free. Using strong mobility requires replicating services, if possible, but in other cases finding similar services becomes a complicated task. This due to services format, for instance, one service uses an integer instead of a float.

In [26] Marzouk et al. stipulate that self-adaptivity is needed for applications under highly dynamic environments where applications components fail or sometimes when there exists performance degradations causing QoS degradation. There is a need for mechanisms that enable the adjustment of the application execution at runtime; again they used the strong mobility mechanism. 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." They identified, also, that most of the related works focus their efforts on the unavailability or performance degradation of composite WS and often use substitution for recovering web services causing high overhead. It is because, other proposed works do not use checkpointing techniques and have to start the whole WS composition or orchestration. Marzouk et al. discuss that their approach pursue the self-healing mechanism whereas in case of failure the orchestration process is migrated to a different server and in case of QoS violation a subset of running instances may be migrated to a new server in order to decrease the initial host load.

For long running applications, in particular, adjusting to the appropriate checkpoint policy at design time may be very difficult. To overcome this issue, Marzouk et al. offer a flexible solution allowing the checkpointing policy dynamically to change at runtime. Whenever the execution context changes an execution manager decides when to change the checkpointing policy. The authors argue that, their solution has no risk of non deterministic execution when recovering flow activities since they always save a unique state for each instance. But, a recovery state is built after synchronizing all flow branches which permit saving a consistent checkpoint. Yet, to our believe using synchronization for constructing a consistent checkpoint makes this approach expensive because of the barrier imposed from synchronizing; hence this solution is slow and with no concurrency.

Finally the most complete work from Marzouk et al. can be found in [25], it has both transformation rules and the rules followed by aspects for strong mobility. They also illustrate that the checkpoints can be forced based on policy-oriented techniques. Since checkpointing techniques allow you to save the state of a process orchestration and revert to the last checkpoint taken, if any process fails it can make use of aspects, of the transformation rules for source code transformation and strong mobility; when the orchestration resumes it runs only the code still has not yet been executed. The authors also consider the quality of service (QoS) for orchestrating Web services taking into account the checkpointing interval based on Markov chains. Although this is a sophisticated approach, synchronizing of the executed parallel branches within a BPEL process results in an expensive solution, their work is not intended to build consistent global state of business processes requiring interaction but they are able to build this type of consistent states of a single BPEL instance. Further, the authors do not specify whether the checkpointing mechanism used has low overhead of the control information or whether this approach is inexpensive computing.

Varela and Martínez in [42] identified that while executing business processes they are susceptible to intrusion attacks, which can be the cause of severe faults. Fault tolerance techniques tackle such issues, decreasing risk of faults and therefore being more dependable, with the aim of achieving dependability before business processes automation. The authors claim that, to resists faults related to integrity attacks fault tolerance techniques can be applied and that their approach, at the time of writing, was the first solution to achieve fault tolerance over business processes based on checkpointing and rollback using *constraint programming*. So, 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*. More details of each layer can be found in [42], we are only interested on the *Fault Tolerance* layer. Since, it is based on checkpointing and rollback recovery, the idea behind checkpointing is to save the state of the system periodically and in case of a fault, recover the system from the last saved state. The authors illustrated the simulation of a checkpoint approach for determining faults of integrity in services and only in case of faults launched a recovery mechanism.

- Integrity sensors (Checkpoint). These sensors take data information about data inputs and outputs from the services.

- Compensation handlers (Rollback) allow to rollback the process execution from a specific point, executing a set of tasks to undo the transaction already initiated.

To show the feasibility of their approach, the authors ran a performance evaluation, simulating the injection of integrity attack events triggering the corresponding fault tolerance function of the services and measured the recovery time when using their framework. However, the authors do not mentioned which CP they use; often, new and improved checkpointing protocols are proposed in the literature, we believe that the recovery overhead time can be reduced making use of such improved protocols.

Varela et al. in [43] argue that companies need to intercommunicate exchanging information between business logics, thus deciding to deploy what is called Business Process Management System (BPSM). BPSM aids to automate business processes, but in this context systems are error prone and can not guarantee a perfect execution over time. Therefore a new paradigm called Business Process Management (BPM) arises, 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 mean 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, checkpointing among other, focusing on the service-oriented business processes context.

Varela et al. proposed a framework that uses the fault diagnosis method, which identifies the failing parts in a business process. Their framework architecture consists in four layers: *Modeling layer, Application layer, Fault Tolerance layer and Service layer*. We are interested on the *Fault Tolerance layer*, since it implements checkpointing. The *Fault Tolerance* layer developed, has the purpose of controlling possible faults and to alleviate them with corrective actions. The authors claim that for fault tolerance replication and recovery are the most used techniques, yet do not consider the state of the business process. So, one fault tolerance mechanism is focused on the simulation of checkpoint and recovery oriented towards business processes.

The authors propose four different approaches for the *Fault Tolerant* layer, they evaluated the following cases: without fault tolerance, primary/backup approaches, multi-version (N-Version Programming) approach and finally the checkpointing approach. The checkpointing adds two components: *Sensor (Checkpoints)* and *Compensation handlers (Rollback)*. However, such approach requires the introduction of extra components (sensors) into the business process design, extra time to check each sensor and recovery of business process service in rollback.

In [44] the authors propose a way of building global checkpointing of orchestrated Web services to achieve such 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. They introduce the notion of Call-based checkpointing of web services, thus they employ a set of checkpointing policies. This policies identify the calls within web services, for instance a *one way* request will checkpoint its state for further use latter. Notwithstanding they tackle only the orchestration process and do not take into account interaction among multi party orchestration processes.

In [45] Vathsala and Mohanty present a survey and categorize existing approaches techniques based on fault tolerance into forward and backward techniques. When Web services recovery is said to be backwards these are restored to previous checkpoint. When recovery is forward, the technique that is generally used for orchestration, is to try to make services continue their implementation, usually by replacing Web services. Vathsala and Mohanty, discuss the need for Web services checkpointing and identify cases in which the checkpointing mechanisms may be applied effectively, and the considerations that should be taken. For example, generating checkpoints periodically is not a viable option for Web services. Another example is when the orchestration service coordinator fails in this case using replacement is not admissible, here is the justification to use checkpointing mechanisms and considerably improve the reliability of the Web services orchestrations. Similarly, the backward recovery has advantages over forward recovery as it is platform independent, one can retrieve unpredictable errors, rollback ensures that transient faults not resurface. Finally, they present several existing approaches based on backward recovery (based on checkpointing) and forward (based on substitution). Until the writing of this work none of the works referred are intended to create CGSs from the interaction of asynchronous business processes and the combinations of choreography and orchestration of Web services compositions of these techniques has not been addressed.

Table 2 summarizes the related works combining web services and checkpointing, it illustrates the goal, environment, evaluation, used checkpointing protocol (CP) and finally what each work tries to recover.

4.2.1 Summary

Complex business processes are usually WS compositions or orchestrations using the Business Process Execution Language (BPEL) engine. In this context, WS are essential and must be dependable. Ongoing research, often use substitution for recovering WS causing high overhead in their solutions, other techniques opt for migrating to alternative WS when they fail or QoS violation occurs. Other suggestions propose the use of synchronization mechanisms and other components such as sensors, before taking a checkpoint they synchronize all processes and in case of failure the system can latter recover to such state. Although, this are practical proposals, we believe that CiC protocols can alleviate much of the overhead imposed by just taking checkpoints at any given time.

Table 2. Web Services and Checkpointing.

Parameter / Article	[13]	[24]	[42]	[43]	[26]
Goal	Highlight the dependability implication when using SOA and propose enhancements for dependability.	Ensure self-adaptivity of composite services in particular for BPEL.	Restore attacked business processes with the aid of fault-tolerance based on checkpoint and rollback.	Increase dependability of business processes.	Propose a solution for strong mobility of composed WS.
Environment	Distributed systems.	Distributed enterprises business logic.	Distributed systems (Busineess).	Distributed systems.	Web-services orchestration.
Evaluation	Collaborative QoS.	The travel agency (BPEL orchestration).	Recovery Time.	Recovery Time.	The travel agency (BPEL).
Used CP	?	?	?	?	Checkpointing policies.
Recovers	———	BPEL process.	Business processes.	Performance time in terms of delivery of messages.	Orchestration process using self-healing.

4.3 Dependability

Dependability has been widely studied by researches. In the past, CORBA, for example, uses entity redundancy to achieve object fault tolerance (http://www.omg.org/spec/FT/1.0). In the present, studies of dependability have been carried out, for Service-Oriented Computing (SOC) [7], for autonomic dependable services [31], for ESB dependability [47], to name a few research areas . . .

Although, Eila Ovaska et al. present their work [31] for smart cities, the very same concepts of dependability are applicable to our research. The definition of dependability truly depends on the context and it is considered from three different points: as a system property, as a service capability and a failure free operation [31]. Dependability as a system property is defined as the systems ability to deliver a service that can justifiably be trusted. Dependability as a service is the behavior perceived by the service user. Dependability of a failure free operation refers to the systems ability to avoid frequent and severe failures. This last definition correlates to two core features: self-healing and self-protection of autonomic systems.

Another current trend is explored by Asit Dan and Priya Narasimhan in [7] for dependable Service-Oriented Computing. SOC is based on SOA, where software peaces are assembled for building complex business processes. In SOC an overall business solution comprises interacting, interoperable and reusable services: designed, developed and deployed independently from one another which bear some business functionality through a well-defined interface. As identified by Asit Dan and Priya Narasimhan a niche for research and that is currently investigated is on how to test dependability for an aggregate set of services; as well as, automated self-managed SOAs. What information should be monitored, and how often, to accurately and rapidly diagnose the source of failures, attacks or problems? Although, existing solutions try to answer these kind of questions, we believe more efficient dependable solutions are still a niche for research.

For more punctual studies on dependability we could find [47] that Jianwei Yin et al. highlight a real life large-scale implementation of a dependable ESB deployed for the Chinese Healthcare Service Integration (CHSI) ensuring that a service is secure and available. However, to achieve fault tolerance the authors make use of backup servers, backing up the entire ESB systems having a primary, standard and backup servers rising the cost of deployment while adopting such solution; and this work does not consider services' dynamic behavior.

Clearly there is a correlation between dependability and self-healing both are concern on improving the systems fault tolerances and although sophisticated mechanisms have been suggested few have been deployed within the Enterprise Service Bus. Open source ESBs such as Mule, ServiceMix and OpenESB do not provide a fully dependable mechanism.

4.4 Self-healing capabilities for ESBs systems

Currently there is ongoing research trying to integrate self-healing capabilities into the ESBs [2], [16], [17], however, some of these solutions are implemented in off-line mode.

In [2] the authors deployed self-healing capabilities for the ESB under a mission-critical environment. In particular, the authors used the ESB as the integration backbone for the Airports systems. Because the ESB can adapt to the growing requirements of the autonomic information exchange between different departments; these systems were developed by diverse vendors and were not design for interoperability, making it a difficult task to integrate them. Issam Al Hadid, considered within the self-healing architecture: reliability, maintainability, evolvability, extensibility, scalability and interoperability. Thereby, the Airport ESB (AESB) with self-healing (AESB-SH) provides an implementation backbone for the Airport and Aviation systems to control flow and translation of the exchanged messages between systems. The self-healing *Agent* presented, maximizes the reliability, adaptability, robustness and availability by monitoring the service layer task; done by extracting all information about the service's tasks, diagnosis errors by examining and analyzing extracted information and repairs the fault by executing an action according to the faulty type. However, the authors achieve self-healing by adding a self-healing *Agent* which can cause QoS degradation to the deployed Web Services for this reason solution is not scalable; in each WS the self-healing *Agent* must be implemented.

In [16] [17] the authors propose a way to diagnosis the ESB and try to integrate the self-healing capability into it, transforming it into an Autonomic Service Bus (ASB). The ESB system is viewed as a managed element from which a common knowledge base is generated. Afterwards, they built an Bayesian Network (BN) and show how it can be used to implement the diagnosis process for an ASB. However, the authors gain only a way to know the relationship between variables since all the computation is done off-line; the

BN allows discovering performance degradations for the ESB platform.

Table 3 summarizes characteristics taken into account by the related works for self-healing within the ESBs systems, where the uppercase X means it was taken into account and the slash (-) means it was not. This table illustrates, some of the weakness of the existing approaches.

Table 3. Comparative Analysis.

Characteristic	[2]	[16]	[17]
Self-healing	X	X	X
Recovers	Web Services	System	System
Dynamic Environment	-	-	-
Recovery time	-	-	-
Real situations	X	-	-

4.5 Discussion about related works

Many sophisticated works were analyzed that try to achieve dependability and consider self-healing to do so; many lack on all the features that define dependability entirely. We could conclude that the price for achieving fault tolerance, one core feature dependability considers, in some cases affects the scalability of the system. Or has negative impact on the systems performance and many of the times dynamical environments are not taken into account; where the participants are interacting, entering and leaving the systems constantly and in the presence of errors. Thus, more work has to be conducted to tackle the dependability as a service; in this particular case for Web Services and for the ESB system as both affect the quality perceived by the end user. More work also needs to be carried out, for dependability of failure free operations. We propose that dependability, many of its core features (reliability, availability, fault tolerance), for ESBs systems can be achieved by adopting self-healing and checkpointing protocols under two circumstances: for WS or business processes and for the entire ESB system. Putting special care no to outage recovery time and respecting stipulated standards as the one recommended by the *UTI G.1010 Recommendation* for WS.

5 Research Proposal

5.1 Research Aim, Objectives and Questions

The main contribution of this research is the synthesis of two widely used paradigms: autonomic computing and checkpointing protocols which imply the development of self-healing capabilities based on checkpointing protocols for ESBs under mission-critical environments; setting up a cloud-based testbed environment. Which brings up other contributions:

- Partial ordering algorithms for ESBs; solving inconsistency problems due to information exchange in WS-based collaborative environments.
- An efficient, scalable, low overhead and domino effect-free, dynamic checkpointing protocol based on the monitoring of QoS degradation.
- Finally, test such synthesis on a more general environment such as SOA.

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 protocols.

5.1.1 Research Questions

For the synthesis of the two aforementioned paradigms, in particular designing and developing self-healing capabilities based on CPs proposed in this research many questions arise.

I. How to design efficient dependable services for critical ESB environments which satisfy time constraints for system recovery?

   This general question motivates further questions which help us deal with this research. In ESBs systems it is desirable not to stop its execution; there can be millions or billions of requests interacting with it every second, causing a severe degradation of the system for this reason some questions arise.

II. How to develop and implement efficient adaptive CP within an ESB?

III. How to monitor, detect, the QoS degradation at runtime for an ESB?

IV. How to use the information retrieved from question III to reduce or adapt the generation of checkpoints?

V. How to design self-healing capabilities for an ESB at runtime?

5.2 Research Objectives

5.2.1 General Objective

- To develop dependability capabilities for ESBs systems through merging checkpointing protocols and self-healing.

5.2.2 Specific Objectives

Although, many studies have been carried out by many researchers in academia and industry, in general there is not an unified vision on how to handle the self-management property.

I. Identify gaps in research relating to self-healing and checkpointing protocols for ESBs; finding gaps on current dependable ESB systems.

   The aim of this objective is to carry out a detail literature review to establish the state of the art of the self-healing feature based on checkpointing techniques for ESBs and to critique existing methods and gaps in knowledge.

II. To implement a cloud-based testbed environment and deploy the self-healing MAPE cycle along with CP for ESBs.

   In cloud-based testbed environment first we need to identify issues when using an ESB, then the self-healing MAPE cycle could be deployed and after a CP tested.

III. To develop partial ordering algorithms for ESB systems.

   Before talking about CPs some restrictions need to be satisfied to have a coherent or consistent state to where the system will recover. The order of message is important in this context, since CP need to avoid causal paths and zigzag paths among saved checkpoints.

IV. To monitor and detect services' QoS degradation at runtime for an ESB. Using this information reduce or adapt checkpoints generation.

Monitoring the QoS can lead to adapt the rhythm on which the CP takes checkpoints, plus if a QoS policy is violated the system can opt for rollback recovery to a point in time where the system was functioning properly.

V. To design and test the adaptive CP created towards the recovery of services for ESBs in critical environments which satisfy time constraints for system recovery.

Besides being a scalable CP, our solution needs to guarantee time constraints when recovering the system.

5.3 Methodology and Experimentation

The research methodology is a way to systematically solve the research problem. In this case we are in charge of exposing the research decisions to evaluation before they are implemented; we need to specify clearly and precisely what decisions we select and why so that they can be evaluated by others also [19].

For this research according to the stated objectives, four general phases are identified: *Causal Ordering Algorithms for ESBs, QoS degradation Monitoring, Adaptive Checkpointing Protocol and Dependable ESB* as depicted in the Fig 4. A detailed description of each phase is next given along with their own sub-phases.

I. *Causal Ordering Algorithms for ESBs.*

• Analysis and characterization of mission critical environments for ESB systems and identification of critical communication services that requires message ordering of events.

• Study and analysis of technologies that transports Web Services (WS) and identify if such technology suffers gaps.

• Design and implementation of a message ordering framework, from the phase the Java Message Service (JMS) was choose over the analyzed transport technologies. Such framework is based on the Immediate Dependency Relationship (IDR).

• Design and integration of a *Membership* service for ESBs systems to avoid possible inconsistencies due to dynamism of the system.

• The protocol primitives where developed for the Message Ordering Framework (MOF).

II. *QoS degradation Monitoring*

• Implement testbed cloud-based environment like Proxmox[3].

• Design diverse stress test towards ESBs systems to test different workloads that implies different QoS behavior.

• Deploy dummy web services that are able to monitor their own performance and QoS aspects: response time, transactions per second (TPS), etc...

• Design monitoring services within the ESB system; these shall not impact the system performance. For these, we will test two different approaches: monitoring done via the Java Virtual Machine (JVM) and monitoring done via Linux shell scripts.

• Write QoS policies for web services (WS) and for the overall ESB system. If these policies are violated then rollback mechanism shall come in handy.

• Test different monitoring frequencies to find out which rate does not have great impact performance, test for WS and for the ESB system.

• Test if the same monitoring frequency is optimal for checkpointing rate generation. To our knowledge the *transparent checkpointing* technique will fulfill our requirements for rollback recovery mechanism.

III. *Adaptive Checkpointing Protocol*

• Design of an efficient communication-induced checkpointing (CiC) for business processes in web services.

• Built a common knowledge base from monitored data.

• With the aid of phase I, the designed Message Ordering Framework, design domino effect free communication induced checkpointing (CiC) protocols, that is messages that do not have HBR and zigzag paths.

• With the information retrieved by phase II, design adaptive checkpointing protocols based on QoS

[3]A server virtualization solution based on open source systems. Allows creating KVM and containers and manages virtual machines, http://www.proxmox.com/es

policies.

• Test if the dynamic checkpointing generation rate, considerably reduces the number of checkpoints and increases the systems performance.

IV. *Dependable ESB*

• Analyze which components will be the managed elements and which will be the autonomic managers; this from the autonomic computing paradigm.

• So far from the MAPE loop, only the *Monitoring* has been tackled. For the *Analysis*, we need a means of analyzing such data for prediction, we will explore the following techniques: Markov Chains, Bayesian Networks and fuzzy logic might come in handy.

• Design *Planning* actions depending on the analyzed data, where checkpointing protocols also depend on and will generate forced checkpoints.

• Design *Execution* actions for system recovery or web services recovery, when policies are violated.

• The ESBs systems dependability is gained when reliability is increased by the self-healing property of AC and availability by rollback recovery mechanisms. Finally, we propose comparing our work with others that achieve dependability.

Fig. 4 summarizes in a general manner all the phases described above and as summary; first we need to implement *Causal Ordering Algorithms for ESBs*; with the aim of causally ordering events and afterwards design a CP domino effect-free. But, before we need to implement a testbed, to test the effectiveness of our solution for this we propose a cloud-based environment based on open source systems. Such allows the creation of several Virtual Machines (VM). After implementing the testbed, we will need a manner of *QoS degradation Monitoring*, for this we propose our own demons which shall not degrade systems performance. The next step is to design of an *Adaptive Checkpointing Protocol* which dynamically shapes its checkpoint generation rate based on the QoS previously monitored. Before we implement the MAPE loop, we can check if we are going in the right direction by implementing our CiC algorithm for fault tolerance in WS. Now, we could MAPE (monitor, analyze, plan and execute) loop the ESB, we have to verify our solution can be deployed during runtime and that it does not take more time to recover than that stipulated for web services.

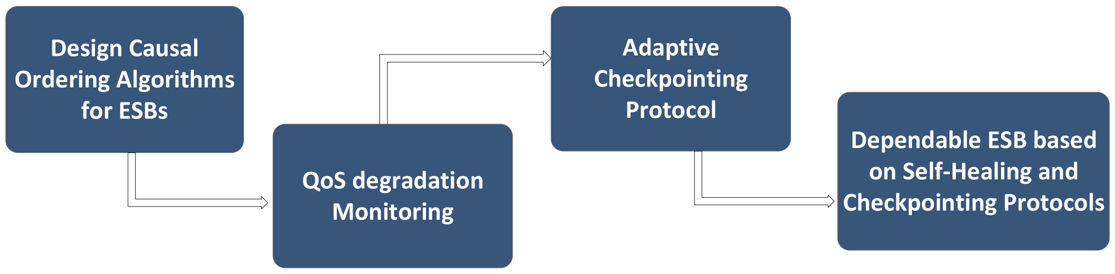

Design Causal Ordering Algorithms for ESBs
QoS degradation Monitoring
Adaptive Checkpointing Protocol
Dependable ESB based on Self-Healing and Checkpointing Protocols

Figure 4. Methodology.

5.4 Contributions

The main contributions of this research are listed according to their relevance as follows:

- More dependable ESBs systems under mission-critical environments based on the synthesis of self-healing and dynamic checkpointing protocols.
- The deployment of partial message ordering for ESBs; solving inconsistency problems due to information exchange.
- The development of a dynamic checkpointing protocol based on monitoring the services' and systems' QoS degradation.
- Finally, implementing self-healing based on the same synthesis principles but in a more general environment like SOAs, under critical environments.
- An efficient approach that implements the Monitoring, Analyze, Plan and Execute (MAPE) control loop within Web services based on checkpointing protocols.
- We propose a distributed fault tolerance approach for dependable interactive BPEL processes based on communication-induced checkpointing (CiC).

6 Work Plan

This section presents the schedule that we are going to follow to lead our research problem.

Table 4. Schedule of activities.

PhD Propasal	Year / Cuatrimestre / Month											
	2014			2015			2016			2017		
PhD Research	Q 1	Q 2	Q 3	Q 1	Q 2	Q 3	Q 1	Q 2	Q 3	Q 1	Q 2	Q 3
State of the art and thesis writing	X	X	X	X	X	X	X	X	X	X	X	X
Implement Partial Order Algorithm	-	X	X	-	-	-	-	-	-	-	-	-
Characterization and identification of message order	X	-	-	-	-	-	-	-	-	-	-	-
Design and implementation of MOF	-	X	-	-	-	-	-	-	-	-	-	-
Membership and Protocol primitives design	-	-	X	-	-	-	-	-	-	-	-	-
Prepare Research Proposal	-	-	X	-	-	-	-	-	-	-	-	-
First Conference Article	-	-	X	-	-	-	-	-	-	-	-	-
Research Proposal defense	-	-	-	X	-	-	-	-	-	-	-	-
Stay at LAAS	-	-	-	-	X	X	-	-	-	-	-	-
Monitoring QoS Degradation	-	-	-	X	X	-	-	-	-	-	-	-
Testbed implementation	-	-	-	X	-	-	-	-	-	-	-	-
Develop diverse performance tests	-	-	-	X	-	-	-	-	-	-	-	-
Optimal frequencies	-	-	-	-	X	-	-	-	-	-	-	-
Design and Implement Adaptive Checkpointing Protocol	-	-	-	-	X	X	-	-	-	-	-	-
Fault tolerance for WS	-	-	-	-	X	X	-	-	-	-	-	-
First Journal Article	-	-	-	-	-	X	-	-	-	-	-	-
Stay at LAAS	-	-	-	-	-	-	X	X	X	-	-	-
Prove Dependability of the ESB	-	-	-	-	-	-	X	X	X	-	-	-
Analysis of AC	-	-	-	-	-	-	X	-	-	-	-	-
Tackle MAPE loop	-	-	-	-	-	-	-	X	-	-	-	-
Test ESB dependability	-	-	-	-	-	-	-	-	X	-	-	-
Second Journal Article	-	-	-	-	-	-	-	-	X	X	-	-
Thesis correction and revision	-	-	-	-	-	-	-	-	-	-	X	-
PhD Defense	-	-	-	-	-	-	-	-	-	-	-	X

Q 1 = Quarter 1, Q 2 = Quarter 2, Q3 = Quarter 3

7 Advances

7.1 Design Causal Message Ordering for ESB systems

7.1.1 Analysis

A practical way to understand web services (WS) and how they imply the use of an Enterprise Service Bus (ESB) is discussed in this section. WS usually carry out complex business processes, occasionally using the BPEL engine, these can be locally deployed or they could cooperate in distributed environments, most common situation in real world deployments; in particular for mission critical environments, where collaborations between many companies exists. In the literature we could find many examples of mission critical environments, to mention some: [5], [47], [8].

7.1.2 Characterization

In many cases Web Services (WS) are used to discover a business functionality, presented as services, that are available through the network, are taken advantage of and invoked by corporative partners through a well-known interface. WS are inter-operable and suitable for distributed environments since they follow and apply defined standards. In this context, corporate partners share knowledge, ideas, and modify information; by sharing information in a collaborative manner can minimize time spent in problem resolution. As shown in Fig. 5 business partners need to share and modify information remotely; for instance, consider 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 WS and expose their Computer-aided Design (CAD) systems as such.

Respecting the order on how events occurred is necessary in the context of mission critical cooperative and distributed systems; information must be coherently presented to each user to preserve data integrity. Many proposals can be found in the literature and in the Internet. The next subsection identifies some of these proposals and clarifies why be opt on using the Immediate Dependency Relationship (IDR).

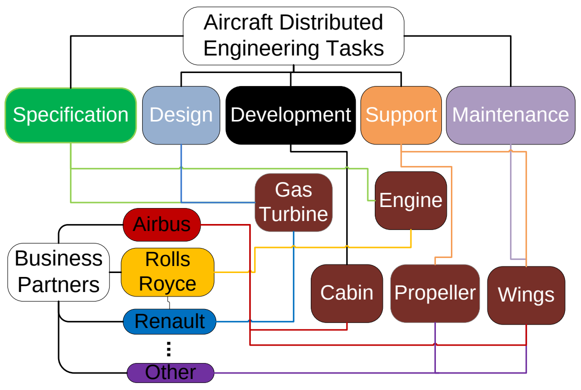


Figure 5. Aircraft Distributed Collaboration.

7.1.3 Identifying the need for message ordering

Message ordering can also be achieved by diverse means. Message order can be guaranteed by configuring proxies [4] where the author explicitly specifies the *happened-before* relationship stating that a message can not be received if it has not been sent. But packages can arrive in different order as they were sent due to network delay and sometimes even the size can lead to give priority to smaller packages. Other alternatives for message ordering, for example consider adding a *label* and modifying the *proxy*. When a label is added both ends of the channel must agree on the initial label and sequence. When modifying the proxy, the destination end of the channel expects sequence-labeled messages and will re-order them if they arrive out-of-order.

Other times, each infrastructure has its own way of implementing message ordering. For example, WSO2 [5] products rely on several components.

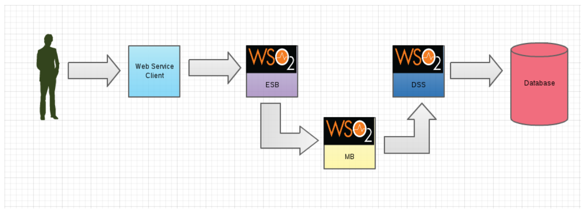


Figure 6. In-order Messaging [2]

[4]http://www.dalnefre.com/wp/2011/07/in-order-message-delivery/

[5]http://wso2.com/library/articles/2013/12/building-an-in-order-guaranteed-delivery-messaging-system-for-your-enterprise-with-wso2-products/

Figure 6 shows how the users (Web Service Clients) must invoke the service, usually using a web browser, then the ESB system will be in charge of routing it to its corresponding provider. Passing through a Message Broker in charge of temporally storage of messages, they will be processed in-order. And finally the Data Services Server responsible for writing into the database.

Although many message ordering techniques are recommended they make use of proprietary software or are too specific: like modifying proxies and adding labels. Others introduce a *broker* a that can become well known point of failure in the network. A more generic way of ordering messages is a necessity. Implementing the HBR can be feasible, piggybacking control information inside each message, however their implementation is expensive to set up in distributed systems [20]. 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 [12].

7.1.4 Study and analysis of transport technologies for WS

WS client applications usually use the HTTP as connection protocol. Nevertheless, 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, WS 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; yet, JMS alone can not grant order in such environments. Therefore, we suggest an extension framework to the JMS API.

7.1.5 Design and implementation of the Message Ordering Framework

We propose a Message Ordering Framework (MOF) for collaborative environments; this ensures causal ordering according to the causal view of the systems involved while maintaining a low overhead and computational costs it is based on the IDR. The aim of the MOF is to propose an extensible framework for the JMS API which can be exploited by WS, particularly in collaborative work environments over heterogeneous networks; 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. 7a 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.

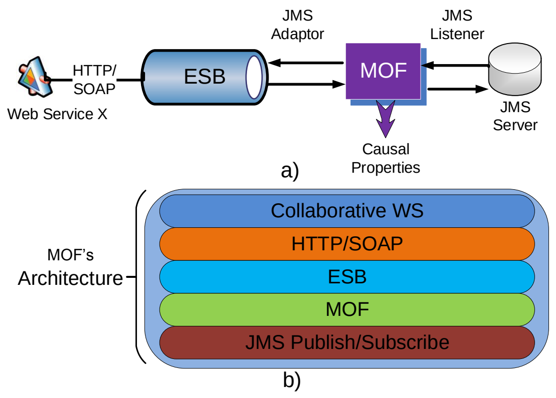


Figure 7. MOF's Architecture.

Fig. 7b shows the architecture of our suggested solution; such must be read alike a network protocol stack. Showing that *Collaborative Web Services* are usually deployed by remote servers hosted somewhere on the Internet and need to collaborate in order to support complex business processes, and since they are remotely invoked and distributed in different parts of the world, they use the *HTTP/SOAP* as transport.

The main actions of the rest of the components are:

- The *ESB* provides the WS callback mechanism as a proxy that can be easily accessed. It also adapts the message in format so that it is compatible with the MOF, which uses JMS.
- The *MOF* 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 Publish/Subscribe* provides or consumes messages, depending on its invocation.

7.1.6 Design and implementation of the *Membership* service for ESB systems

Mission critical collaborative environments need a more reliable transport mechanism particularly where the order of the message matters, but JMS alone cannot grant such. The MOF is built over JMS; extending its properties. Furthermore, all communications passing through it are enriched with a small overhead, piggybacked control information, keeping track of the order of messages. MOF uses the ESB messaging middleware over distributed heterogeneous networks to support the publish/subscribe communication model linking autonomously various publishers and subscribers. MOF does not depend on the WS in place; to achieve such we present the scheme shown in Fig. 8, 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. 9a for subscription and 9b for departure.

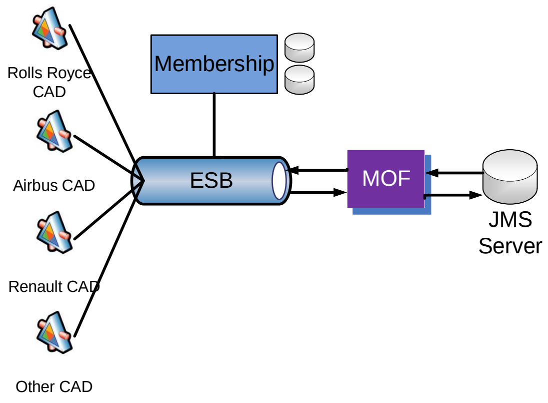


Figure 8. MOF Scheme.

7.1.7 Membership subscription

Fig. 9a 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.

7.1.8 Membership departure

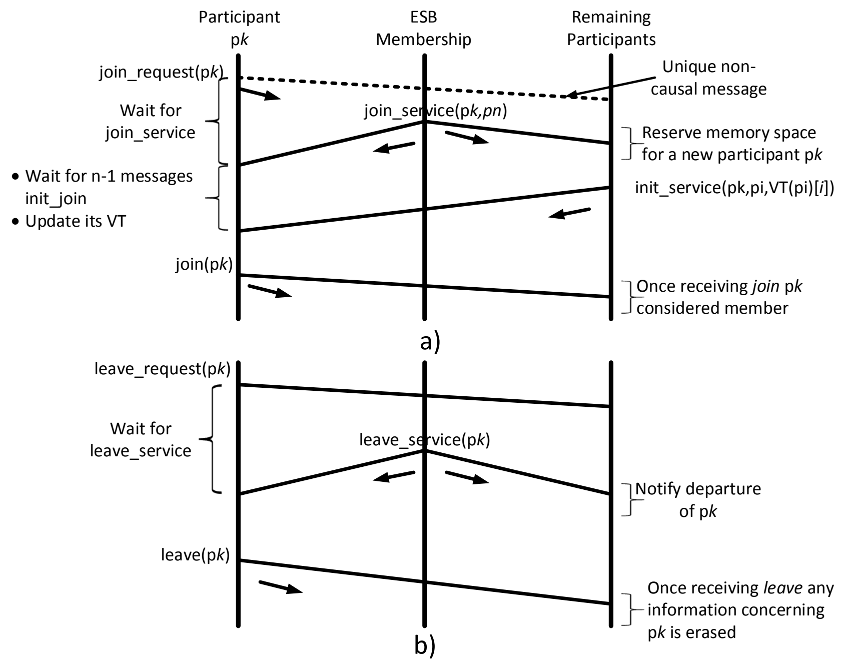
Fig. 9b shows the departure of a user from the collaborative work environment. When a user wants to leave, first he must send a petition request $leave\_request(p_k)$, which is sent by the ESB to all other users announcing that a user will leave the environment. Finally, the user leaving, sends the $leave(p_k)$ 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.

7.1.9 Protocol Primitives

In this subsection we illustrate the MOF's protocol primitives, these where built on top of the JMS API; making use of the all ready implemented API, but enhancing it for collaborative environments.1

The JMS API for publishing consists of 6 steps: (1) Perform a 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 7 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*.



Participant pk
ESB Membership
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 9. Membership.

Table 5. Original JMS interfaces and equivalent.

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

Table 5, shows the original interfaces of JMS and the other columns show the equivalent interfaces provided by our MOF. Table 6, shows the original classes and the other columns show 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 a threading restriction; thus a session may not be operated on by more than one thread at a

Table 6. Original JMS classes and equivalent.

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

time. Hence in the JMS publish/subscribe style two sessions are mandatory.

Then we exhibit 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 are used by the *writeMessage()* method. In our case we invoke the *writeMessageCausal* before publishing a message to a topic.

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*

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

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

To conclude this section, adapting the IDR protocols one can transmit the minimal and sufficient amount of control information to preserve the causal order of messages. We adapted the minimal broadcast causal protocol (MBCP) presented in [12] to work with JMS. Thus, the MOF extends JMS making it more reliable, and also scalable. As suggested in Algorithm 1, before using MOF all participants are assigned an unique id (UUID), their logical clock VT are set to 0 as well as their control information (CI), in steps 7 to 11. Steps 1 through 6 implement the restrictions imposed by JMS, creating two sessions one for the publisher and another one for the subscriber, creating a *Connection Factory* and a *Topic name*. For publishing or sending events, steps 12 to 19, the MOF uses the *WriteMessageCausal* method where VT and CI are updated to keep track of possible IDR messages and publishing to a *Topic* is done in step 17.

When a WS need to consume messages, MOF implements steps 20 to 40. The first step is to check if the queue of messages not empty, steps 22 to 24, if not then this messages are processed first, step 23. In other case, the incoming text message has to be slit into a readable format, steps 25 to 29. If the delivery condition is satisfied, step 30, the message is delivered; otherwise messages are queue in step 31. Steps 34 to 39 update possible IDR messages, step 34, and removes unnecessary control information in steps 35 to 37. Lastly, the control information that having possible IDR is update in step 38.

Algorithm 1: The Minimal Broadcast Causal Protocol (MBCP)

```
1: procedure CREATE CAUSAL SESSION
2:     $pubSession = C_TopicSeesion$
3:     $subSession = C_TopicSeesion$
4:     $Destination = MyTopic$
5:     $ConnFactory = ConnFactoryName$
6: end procedure
7: procedure INITIALIZATION
8: Initially
9:     $VT(id_i)[j] = 0 \forall j : 1, 2, ..., n$
10:     $CI_i \leftarrow \emptyset$
11: end procedure
12: procedure WRITE MESSAGE CAUSAL(m)
13:     $VT(p_{id})[i] = VT(p_{id})[i] + 1$
14:     for $k = 0$, $k{+}{+}$, while $k < CI.size$
15:     $H_m[k] = CI_i[k]$
16:     22 $m = (i, t_i = VT(p_i)[i], message, H_m)$
17:     Publish($m$)
18:     $CI_i \leftarrow \emptyset$
19: end procedure
20: procedure ON MESSAGE(m) for $p_{idj}, i \neq j$
21:     $m = (k, t_k, message, H_m)$
22:     if $lifo \neq \emptyset$ then
23:       m = lifo.pop()
24:     end if
25:     $temp[] = split.m$
26:     $k = temp[0]$
27:     $t_k = temp[1]$
28:     $message = temp[2]$
29:     $H_m = temp[3]$
30:     if not($t_k = VT(p_{idj})[k] + 1$ && $t_l \leq VT(p_{idj})[l] \forall (l, t_l) \in H_m$) then
31:       lifo.push(m)
32:     else
33:       $delivery(m)$
34:       $VT(p_{idj})[k] = VT(p_{idj})[k] + 1$
35:       if $\exists ci_{s,t'} \in CI_j | k = s$ then
36:         $CI_j \leftarrow CI_j \backslash \{ci_{s,t'}\}$
37:       end if
38:       $CI_j \leftarrow CI_j \cup \{(k, t_k)\}$
39:       $CI_j \leftarrow CI_j \backslash H_m$
40:     end if
41: end procedure
```

7.2 QoS degradation Monitoring

7.2.1 Testbed

Based on previous experiences the design of a virtual environment has many benefits, since it gives information on how systems and components behave; yet, having a cheaper implementation costs compared to real life deployments. Virtualization allows the deployment of complex large scale controlled environments with few physical resources. Currently we have implemented a virtual environment in Proxmox [6]. As depicted in Fig 10, one could create many Virtual Machines (VM). For our testbed we created a VM specifically hosting the ESB system, other VMs contain the services' consumers and providers. These VMs can be customized and several performance test can be carried out, obtaining QoS metrics for the ESB system with several values for the VM's resources. With these QoS metrics: response time, transactions per second (TPS), CPU and RAM percentage usage; we will find an optimal frequency for monitoring the ESB system and also see if we could find the optimal checkpointing generation rate.

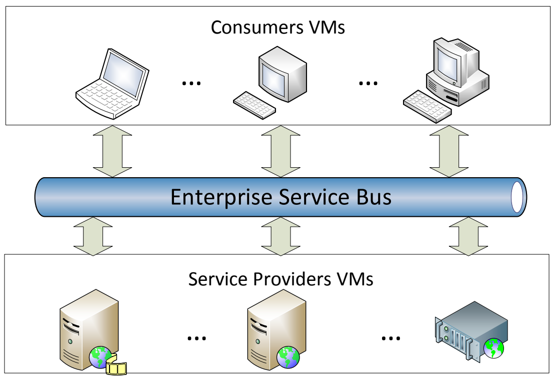


Figure 10. Testbed Virtual Environment.

We have developed dummy web services that emulate the behavior of real world services. We have also developed our on stress test tool in order to obtain a way of retrieving diverse quality of service (QoS) metrics. One open challenge here, from the methodology of this phase, is the development of monitoring services and to find the optimal frequencies for monitoring and checkpointing generation rate.

[6]http://www.proxmox.com/es

Table 7. Towards an Adaptive CiC.

Concept	First Version	Second Version
Fault tolerance	A first step to obtain dependability can be to adapt CiC protocols within Web Services.	Once, WS are dependable, the very same concepts can be transferred to ESBs.
Adaptive Checkpointing Protocol	After proving dependability can be achieved through CiC protocols, for WS; we could adapt such solution for dynamic environments.	Again, the very same concepts can be transferred to ESBs, even when implementing the MAPE loop of autonomic computing.

7.3 Towards an Adaptive Checkpointing Protocol

Before the development of an adaptive checkpointing protocol, we found out which is the most efficient up until now asynchronous communication-induced checkpointing protocol [38], and to find out if we are following the right path for the synthesis of autonomic computing and checkpointing protocols; we suggest to adapt the algorithm found in [38] to obtain dependable business processes. The Table 7 discusses the importance for our research to obtain dependability for web services.

Currently, we are developing a framework for the first version of fault tolerance for business processes which implies web services as depicted in Table 7.

8 Conclusion

In this doctoral proposal, we have described till the moment of writing the design and implementation of dependable ESB systems based on self-healing and checkpointing principles. The ESB systems role is to *“glue”* applications and services in most cases complex business processes. Hence an ESB plays the aforementioned role and it is currently the uttermost used integration technology, there is a lot of studies around it, trying to make it dependable, tackling areas like self-healing and fault tolerance. In this research we aim at integrating two widely used paradigms whose integration will result on efficient and self-manageable ESB systems.

On one hand, rollback based on checkpointing protocols guarantees in case of failure that the system will rollback to a consistent point in time where the ESB system was behaving properly. On the other hand, implementing the MAPE loop aims towards autonomic systems where systems managers implication is minimum and in best cases null.

8.1 Future Work

In future, we will develop dependable capabilities, such as reliability and availability, for ESB systems based on self-healing and an adaptive checkpointing protocol. To achieve such as first step, we will address the fault tolerance for web services, in this stage are going to implement a communication-induced checkpointing protocol within web services. Later as second step, we will implement this CiC protocol on an adaptive manner based on quality of service monitoring and policies violations. As third step, we are going to tackle the self-healing capability for the ESB system following the above principles, first implementing the most efficient CiC found in the literature to later on making it adaptive based on quality of service and policies violations monitoring.

9 Conclusions

Frequent approximate subgraph (FAS) mining is a widely used technique in Data Mining applications where there is some distortion into the data. However, FAS mining in multi-graph collections, remains a challenge. Moreover, usually a large number of frequent patterns is computed by the FAS mining algorithms. Using only representative patterns instead of using all the patterns is a technique that can be used to reduce the dimensionality of the set of patterns computed by a FAS mining algorithm. Therefore, the aim of this PhD research is to propose algorithms for mining FASs and representative FASs from multi-graph collections.

As preliminary results, we introduce a method for computing FASs in multi-graph collections. This method includes a way for transforming a multi-graph collection into a simple-graph collection, then the mining is performed over this collection and later, an inverse transforming process is applied to return the mining results (simple-graphs) to multi-graphs, without losing any topological or semantic information. The performance of this proposed method is evaluated over several synthetic graph databases. Based on these experiments, our proposal for transforming multi-graphs to simple-graphs and vice versa is able to process multi-graph collections in a reasonable time. Furthermore, the usefulness of our transformation method for image classification has been shown. It is important to highlight that before our proposed method the application of FAS miners in multi-graphs would be infeasible. And, as it was commented in the experimental section, in some cases the results obtained by our proposal outperform those results obtained by state-of-the-art classifiers non-based on FAS.

In addition, an extension of the canonical form based on canonical adjacency matrix (CAM) representation is proposed. Using the proposed CAM we were able to process multi-graph collections extending an state-of-the-art algorithm for mining FASs. Our proposal has as an important advantage, that the proposed extended CAM can be used for extending other FAS miner to allow them to process multi-graph collections without requiring critical changes into the mining algorithms. Finally, based on our experiments, the usefulness of the use of multi-graph representation has been shown in an image classification task. Furthermore, as it was commented in the experimental section, in most of the cases the results achieved by our proposal, which identifies FASs over multi-graph collections, outperform those results obtained by a method that computes FASs on simple-graph collections.

Based on the preliminary results we can conclude that our goals are achievable following the proposed methodology, within the time defined by the Computational Sciences Coordination for a PhD research.

References

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

[2] 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.

[3] Algirdas Avizienis, J Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *NATO SECURITY THROUGH SCIENCE SERIES E HUMAN AND SOCIETAL DYNAMICS*, 23:10, 2007.

[4] Robin Bloor, Marcia Kaufman, Fern Halper, et al. *Service Oriented Architecture (SOA) For Dummies*. John Wiley & Sons, 2013.

[5] Paul Brebner. Service-oriented performance modeling the mule enterprise service bus (esb) loan broker application. In *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, pages 404–411. IEEE, 2009.

[6] David Chappell. *Enterprise service bus*. " O'Reilly Media, Inc.", 2004.

[7] Asit Dan and Priya Narasimhan. Dependable service-oriented computing. *Internet Computing, IEEE*, 13(2):11–15, 2009.

[8] Code Diop, Ernesto Exposito, and Christophe Chassot. Qos and scalability management in an autonomic cloud-based networked service bus. In *Telecommunications (ICT), 2013 20th International Conference on*, pages 1–5. IEEE, 2013.

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

[10] Alan G Ganek and Thomas A Corbi. The dawning of the autonomic computing era. *IBM systems Journal*, 42(1):5–18, 2003.

[11] 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.

[12] SP Hernndez, J Fanchon, and K Drira. The immediate dependency relation: an optimal way to ensure causal group communication [j]. *Annual Review of Scalable Computing*, 6(3):61–79, 2004.

[13] MattiA. Hiltunen and RichardD. Schlichting. Is collaborative qos the solution to the soa dependability dilemma? In Antonio Casimiro, RogÃ©rio de Lemos, and Cristina Gacek, editors, *Architecting Dependable Systems VII*, volume 6420 of *Lecture Notes in Computer Science*, pages 227–248. Springer Berlin Heidelberg, 2010.

[14] TT-Y Juang and S Venkatesan. Crash recovery with little overhead. In *Distributed Computing Systems, 1991., 11th International Conference on*, pages 454–461. IEEE, 1991.

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

[16] 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.

[17] 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):2206–2211, 2014.

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

[19] Chakravanti Rajagopalachari Kothari. *Research methodology: methods and techniques*. New Age International, 2004.

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

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

[22] 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.

[23] D Manivannan and Mukesh Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 100–107. IEEE, 1996.

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

[25] 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.

[26] Soumaya Marzouk, AfefJmal Maalej, and Mohamed 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.

[27] Violeta Medina and Juan Manuel García. A survey of migration mechanisms of virtual machines. *ACM Computing Surveys (CSUR)*, 46(3):30, 2014.

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

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

[30] 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.

[31] Eila Ovaska, Liliana Dobrica, Anu Purhonen, and Marko Jaakola. Technologies for autonomic dependable services platform: Achievements and future challenges. In MarÃaJosÃ© Escalona, JosÃ© Cordeiro, and Boris Shishkov, editors, *Software and Data Technologies*, volume 303 of *Communications in Computer and Information Science*, pages 199–214. Springer Berlin Heidelberg, 2013.

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

[33] Michel Raynal. A distributed algorithm to prevent mutual drift between n logical clocks. *Information Processing Letters*, 24(3):199–202, 1987.

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

[35] 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.

[36] Mazeiar Salehie and Ladan Tahvildari. Autonomic computing: emerging trends and open problems. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[37] Themba Shezi, Edgar Jembere, and Mathew Adigun. Performance evaluation of enterprise service buses towards support of service orchestration.

[38] Alberto Calixto Simon, Saul E Pomares Hernandez, and Jose Roberto Perez Cruz. A delayed checkpoint approach for communication-induced checkpointing in autonomic computing. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2013 IEEE 22nd International Workshop on*, pages 56–61. IEEE, 2013.

[39] 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.

[40] Kishor S Trivedi, Dong Seong Kim, Arpan Roy, and Deep Medhi. Dependability and security models. In *Design of Reliable Communication Networks, 2009. DRCN 2009. 7th International Workshop on*, pages 11–20. IEEE, 2009.

[41] 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.

[42] Angel Jesus Varela Vaca and Rafael Martínez Gasca. Opbus: Fault tolerance against integrity attacks in business processes. In *Computational Intelligence in Security for Information Systems 2010*, pages 213–222. Springer, 2010.

[43] Angel Jesus Varela Vaca, Rafael Martinez Gasca, Diana Borrego Nuñez, 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.

[44] 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.

[45] 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.

[46] John Paul Walters and Vipin Chaudhary. Application-level checkpointing techniques for parallel programs. In *Distributed Computing and Internet Technology*, pages 221–234. Springer, 2006.

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