Autonomic Web Services Based on Different Adaptive Quasi-Asynchronous Checkpointing Techniques

Mariano Vargas-Santiago, Luis Morales-Rosales, Raul Monroy, Saúl Eduardo Pomares Hernández, Khalil Drira

To cite this version:

Mariano Vargas-Santiago, Luis Morales-Rosales, Raul Monroy, Saúl Eduardo Pomares Hernández, Khalil Drira. Autonomic Web Services Based on Different Adaptive Quasi-Asynchronous Checkpointing Techniques. Applied Sciences, 2020, 10 (7), pp.2495. 10.3390/app10072495 . hal-02545415

HAL Id: hal-02545415
https://laas.hal.science/hal-02545415

Submitted on 10 Nov 2020

Autonomic Web Services Based on Different Adaptive Quasi-Asynchronous Checkpointing Techniques

Mariano Vargas-Santiago [1,†], **Luis Morales-Rosales** [2,*,†], **Raul Monroy** [1,†], **Saul Pomares-Hernandez** [3,4,5,†] **and Khalil Drira** [4,5,†]

[1] Tecnologico de Monterrey, School of Engineering and Science, Carretera al Lago de Guadalupe Km. 3.5, Atizapán, Estado de México 52926, Mexico; mariano.v.santiago@tec.mx (M.V.-S.); raulm@itesm.mx (R.M.)
[2] Faculty of Civil Engineering, CONACyT-Universidad Michoacana de San Nicolás de Hidalgo, Morelia 58000, Mexico
[3] Department of Computer Science, National Institute for Astrophysics, Optics and Electronic (INAOE) , Tonantzlintla 72840, Mexico; spomares@inaoep.mx
[4] CNRS, LAAS, 7 Avenue du Colonel Roche, F-31400 Toulouse, France; khalil@laas.fr
[5] Université de Toulouse, LAAS, F-31400 Toulouse, France
* Correspondence: lamorales@conacyt.mx
† These authors contributed equally to this work.

Abstract: Companies, organizations and individuals use Web services to build complex business functionalities. Web services must operate properly in the unreliable Internet infrastructure even in the presence of failures. To increase system dependability, organizations, including service providers, adapt their systems to the autonomic computing paradigm. Strategies can vary from having one to all (S-CHOP, self-configuration, self-healing, self-optimization and self-protection) features. Regarding self-healing, an almost identical tool is communication-induced checkpointing (CiC), a checkpoint contains the state (heap, registers, stack, kernel state) for each process in the system. CiC is based on quasi-synchronous checkpointing where processes take checkpoints relying of control information piggybacked inside application messages; however, avoiding dangerous patterns such as Z-paths and Z-cycles; in such a regard the system takes forced checkpoints and avoids inconsistent states. CiC, unlike other tools, does not incur system performance, our proposal does not incur high overhead (as results show), and it has the advantage of being scalable. As we have shown in a previous work, CiC can be used to address dependability problems when dealing with Web services, as CiC mechanism work in a distributed and efficient manner. Therefore, in this work we propose an adaptable and dynamic generation of checkpoints to support fault tolerance. We present an alternative considering Quality of Service (QoS) criteria, and the different impact applications have on it. We propose taking checkpoints dynamically in case of failure or QoS degradation. Experimental results show that our approach has significantly reduced the generation of checkpoints of various well-known tools in the literature.

Keywords: autonomic computing; web services; autonomic systems; internet technologies; checkpointing

1. Introduction

We find distributed systems everywhere (they are ubiquitous) and people use them in their everyday life. Organizations and users benefit from distributed systems, for these kinds of systems make it possible to complete tasks that otherwise could not be carried out by a single computer; people all over the world can collaborate on projects for solving difficult tasks; organizations benefit one from another through sharing resources and building large scale distributed systems; distributed systems exploit the divide and conquer approach, if applicable, decomposing a task into smaller sub-tasks, and sending them along several computers, for their execution [1–3].

Because distributed systems rely on a network for passing information to accomplish a task, system communication mostly takes place over an unreliable Internet infrastructure. Distributed systems are error prone, due to two main factors: first, their design, and, second, the communication channels. Paraphrasing Leslie Lamport's words: "You know you have a distributed system when the crash of a computer you have never heard of stops you from getting any work done" [4]. Dependability is the most promising technique out of many to address the issue pointed out by Lamport. Dependability is the ability of a system to provide services to its users, even under different threats, such as malicious attacks and software/hardware bugs. System dependability addresses a broad spectrum of characteristics, including system reliability, system availability and fault tolerance, to mention a few [5]. In distributed computing, dependability is crucial, as organizations demand more dependable computer systems, expecting components and applications to work even in the occurrence failures.

There are three main strategies to improve the dependability of a distributed system: fault avoidance, fault detection and diagnosis, and fault tolerance. We address fault tolerance, as in our previous work [1], where a system recovers from different faults with minimum service interruption, based on checkpointing. In this vein, it is now common that organizations want to improve the dependability of distributed systems based in fault tolerance techniques. This technique relies on rollback recovery, a technique that guarantees a system can continue providing services to users upon failure occurrence. One open challenge for distributed computing is to offer fault tolerance techniques to all computing devices, even those that are less powerful but still work in heterogeneous environments; including mobile phones, IoT cameras, or any other small device that support service requests from corporate enterprises through web services or any other application. True for hybrid approaches that rely on autonomic computing and fault tolerance techniques addressing issues such as minoring, detecting and system recovery [1,6,7].

Checkpointing provides fault tolerance to increase dependability in distributed systems and a checkpointing implementation incurs low runtime overhead. The Communication induced Checkpointing (CiC) technique achieves the aforementioned checkpointing characteristics relying on rollback recovery [3,8,9]. To implement any checkpointing based technique a process has to save their state, a so-called *checkpoint*, checkpointing regularly. Usually, processes record their checkpoints individually during failure-free systems execution. After a failure the system has then the ability to restart from the last saved checkpoint, reducing the amount of work to be re-executed by the system. CiC's main goal is to save consistent global snapshots (CGSs) [10,11], one from each process, so that all checkpoints are free from dangerous checkpointing patterns (z-cycles and z-paths) [9,12,13]. Current mechanisms force the insertion of a checkpoint when anticipating the appearance of one such dangerous pattern; nevertheless, as we will argue, not all forced checkpoints triggered are necessary because saving useless information can lead to an excess of storage space, resource usage, and computation; the lesser the number of forced checkpoints, the better.

Building upon our previous work [1] where we proposed autonomic Web services enhanced by a quasi-asynchronous checkpointing mechanism, we merge autonomic computing and dependability techniques such as fault tolerance. In this article, we follow the approach as mentioned earlier; however, our main contribution is focused into present the first dynamic and adaptable generation of forced checkpoints based on monitoring the system performance by considering non-functional requirements. The goal is to dynamically adapt the generation of forced checkpoints through a diffuse approach considering Quality of Services parameters associated with system performance measures. Therefore, we only record useful system snapshots reducing the overhead and system overload. We consider the probability of system failure and system degradation as key factors before checkpointing. Specifically, we consider non-functional requirements, such as Quality of Service (QoS) parameters yet implemented within the MAPE control loop of a web service; this enables us to reduce the number of forced checkpoints further while maintaining system consistency. Detecting degradation problems via monitoring QoS is carried out also in accordance with process or service requirements, including Service Level Agreements (SLAs).

A summary of our contributions are as follows:

- A dynamic and adaptable generation of checkpoints, by considering the monitoring of system performance through non-functional requirements and a fuzzy approach, allowing to reduce the overhead and system overload.
- The design of an autonomic web services (MAPE) control loop that allows an easy implementation of diverse checkpointing techniques.
- The adaptation of diverse checkpointing techniques to offer consistency (checkpoints free of dangerous patterns such as zigzag paths and z-cycles) and non-functional requirements (system evaluation for individual web services).
- A comparative study that shows the performance of different fault tolerance techniques that rely on the most commonly used checkpointing techniques. We hypothesize that a fault tolerance technique can be used under diverse applications and that by adopting one technique many more can easily further be adapted for other computer-related areas.

Using web services as a black box containing all relative MAPE features, we implemented three different checkpointing techniques, namely, Delayed Checkpointing Fully Informed (DCFI) credited to Simon et al. [9], HMNR also called Fully Informed (FI) presented by Helary et al. [14], and Fully Informed and Efficient (FINE) of Luo and Manivannan [15]. In order to detect a faulty component of a system, we propose implementing the MAPE control loop individually by each web service; doing so can lead to discarding useless checkpoints or to update the frequency these are taken. We evaluate web services QoS, at runtime, using a fuzzy approach; in particular, we propose a *fuzzy-consistency system evaluation (FCSE) function*, as previously reported in [1], we identify useless checkpoint concerning QoS and system consistency. To the best of our knowledge, this is the first mechanism that combines a fuzzy approach for system assessment purposes with CiC mechanisms. We used ChkSim [16] and *Jfuzzylogic* http://jfuzzylogic.sourceforge.net/html/index.html, for simulation as both are compatible with JAVA programming language. We show that our proposal is more efficient than current solutions. To this end, we have compared our approach against three of the most efficient CiC based solutions, reported in State-of-the-Art.

We organized the rest of the paper as follows. In Section 2, we give an overview of related work. Then, in Section 3, we analyze the mathematical model for checkpointing and briefly introduce fuzzy logic. In Section 4, we provide an analysis of the environment and the assumptions upon which our solution builds, and present a case study. Next, we elaborate on the results that support our approach, showing the feasibility, Section 5. Finally, conclusions drawn from this study appear in Section 7, along with suggestions for further work.

2. Related Work

In this section, we provide some of the most efficient fault tolerance techniques that rely on checkpointing. In our previous work, we list some of the work that merges autonomic computing and checkpointing techniques (work that addresses some know fault tolerance techniques for Web services and business processes) [1]. Besides, in [1], we ensure the dependability by merging CiC mechanisms and autonomous computing. In [17], we implemented a fuzzy evaluation model into the MAPE loop to measure system performance without degrading the system. In contrast, in this paper, we show how to generate dynamically checkpoints with less overhead and system overload.

There exist many methods for checkpointing such as synchronous checkpointing, asynchronous checkpointing and quasi-asynchronous checkpointing; we focus on quasi-asynchronous checkpointing. A complete survey can be found in [3], where we give a new taxonomy for business processes that rely on checkpointing mechanisms for both orchestration and choreography.

State-of-the-Art literature classifies quasi-asynchronous checkpointing further into two different communication induced and index-based protocols, which in turn implement a variant of Lamport's logical clock. For instance, the HMNR protocol [14], also called Fully Informed (FI) [18], called this way because it bears specific information about the causal past of processes. New versions of the FI method appear regularly in State-of-the-Art literature, proposing a more efficient strategy. For example, Tsai introduces the LazyFI approach [19], which applies a lazy strategy to increment FI's logical clocks. Fully Informed and Efficient (FINE) is other FI variant, introduced by Luo and Manivannan [15,20]. In FINE authors establish a stronger checkpointing condition using the same control information preserved by FI. An optimized version of FINE, called LazyFINE, applies a lazy strategy using the work of Lou and Manivannan [21,22]. Finally, Simon et al. [9,12,23] propose another FI variant, which addresses system scalability, aimed for large-scale systems. Simon et al. reduce the number of forced checkpoints by delaying non-forced checkpoints.

Fault tolerance techniques relying on checkpointing protocols can apply a *rollback* recovery when detecting system failure. However, using these techniques is known to have a high computational cost (since more and more constraint resource devices need these techniques) new strategies must evolve to meet these constraints. The comparison between checkpointing protocols is in terms of the number of forced checkpoints; the lesser the number of forced checkpoints the better. Therefore, fault tolerance techniques that rely on checkpointing must reduce the message overhead exchanged by each process because failing to do so can require a large amount of storage space to store checkpoints [3,24]. Currently, there is not an optimal checkpointing protocol for all checkpoints and communication patterns, regardless of how each mechanism triggers forced checkpoints [25].

3. Background and Definitions

3.1. System Model

Distributed systems have specific characteristics such as: communication between processes is done by exchanging messages, time is not global for all processes, each process can have its own time, and processes do not share a common memory. Processes can be one or more nodes, and processes consist of a single thread or any number of threads (a process can be a computer, a service, a cell phone, a video game console, etc.). In a distributed computation $P = \{P_1, P_2, \ldots, P_n\}$ represent a finite set of processes; where communication channels can be unpredictable, presenting finite delays, however, these channels are considered asynchronous and reliable. In a distributed system, there exist two types of events *internal* and *external*.

An *internal* event reflects a change in the state of a process, for example, when checkpointing. A checkpoint contains the state of a process such as heap, registers, stack, and the kernel state. We denote C_i^x as checkpoints that contain the state of a process i and E_i denotes all checkpoints previously taken. We consider *external* events, those that affect the state of the system globally, for instance, *send* and *delivery* of messages, where m is a message; $send(m)$ denotes the sending of m by any process $P_i \in P$; and $delivery(P_j, m)$ denotes the delivery of message m to another process $P_j \in P$, where P_i ƒ= P_j. The set of messages M containing external events is given by the following expression:

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

where E_m is the set of events associated with M. All set of events in the system, E, are:

$$E = E_i \cup E_m$$

Therefore, modeling of a distributed computation is done through a partially-ordered set $\hat{E} = (E, \rightarrow)$ where $\rightarrow$ and denotes Lamport's *happened-before relationship* [26].

3.2. Background

The Happened Before Relation (HBR) determines the causal precedence dependencies for a set of events. HBR has a strict partial order (i.e., it is transitive, irreflexive and antisymmetric). Lamport defined the HBR in the following way [26]:

Definition 1. *HBR, denote by →, is defines the smallest relationship for a set of events E, established by three rules, as follows:*

1. *Let a and b be events in the same process. Thus, if an event a occurs before b, then* $a \rightarrow b$.
2. *If a stands for the event send(m), and b for the event delivery*(P_i, m), *then* $a \rightarrow b$.
3. *If* $a \rightarrow b$ *and* $b \rightarrow c$, *then* $a \rightarrow c$.

In practice, HBR is expensive, since it accounts for the precedence of every pair of events. We mitigate expensiveness by introducing a stronger relationship; identifying and attaching the minimal amount of control information per message to ensure causal ordering.

Immediate Dependency Relation (IDR). The IDR is expressed by ↓, and it is the transitive reduction of the HBR. We define the IDR in the following way [27]:

Definition 2. *Two events,* $a, b \in E$, *are related by the IDR,* $a \downarrow b$, *if:*

$$a \downarrow b \Leftrightarrow [a \rightarrow b \wedge \forall c \in E, \neg(a \rightarrow c \rightarrow b)]$$

Where E stands for a set of events.

Checkpoint and communication pattern (CCP): A CCP contains all information about processes local checkpoints as well as their transitive messages, i.e., incoming and outgoing messages towards other processes; the CCP has the following definition [28]:

Definition 3. *A CCP is a pair* $(\hat{E}, E_i)$

where $\hat{E}$ is a partially-ordered set modelling a distributed computation and E_i is a set of local checkpoints defined on $\hat{E}$.

Figure 1 illustrates a CCP and all associated checkpoints, intervals, and exchanged messages. For example, *checkpoint interval* is illustrated as $I^x,_k(x \neq 0)$, representing that a checkpoint belongs to process P_k. A checkpointing interval represents events that happened before another checkpoint in P_k, from C_k^{x-1} to C_k^x, $(x \neq 0)$.

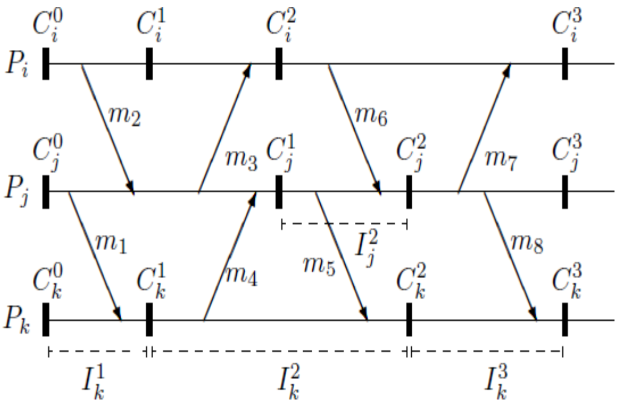

Figure 1. An example of a CCP, where $m_i \in M$ $(i \in \{1, \ldots, 8\})$ represent messages; being M all messages, and each C_k^x represents a process local checkpoint.

3.3. *How Consistent Global Snapshots (CGS) Are Build*

There exist diverse ways to take checkpoints for a distributed system such as synchronous checkpoint, asynchronous checkpointing and quasi-asynchronous checkpointing, where process take checkpoints independently, and systems gain the ability to restart their computation after those checkpoints. However, not all saved checkpoints guarantee to return to a consistent global snapshot (CGS). Netzer and Xu [13] gave the following definition:

Definition 4. *A CGS cannot contain any causally related checkpoints (see below); for any pair of checkpoints C_i and C_j we have:*

$$\neg(C_i \rightarrow C_j) \wedge \neg(C_j \rightarrow C_i)$$

where → is now extended over checkpoints in a conventional manner.

CGS must avoid dangerous patterns as stipulated by Netzer and Xu a *zigzag path* or *z-path*) generalizes the Happened-Before Relation (HBR) in the following way:

Definition 5. *Let P_i, $P_j \in P$. There exists a z-path from $C^x{}_{P_i}$ to $C^y_{P_j}$ if there are messages $m_1, m_2, ..., m_l \in M$*

such that:

1. *m_1 is sent by process P_i after $C^x_{P_i}$*
2. *if some process P_r receives $m_k(1 \leq k < l)$, then m_{k+1} is sent by P_r in the same or at a later checkpoint interval (although m_{k+1} can be send before or after m_k is received), and*
3. *Process P_j receives m_l before $C^y{}_{P_j}$*

Definition 6. *A checkpoint C^x_i is in a zigzag cycle or* z-cycle, *if there is a z-path from C^x_i to itself.*

A z-path was defined by Helary et al. [14] in the following way:

Definition 7. *A z-path $[m_1, ..., m_q \in M]$ is a* causal z-path *if for each pair of consecutive messages m_k and m_{k+1}, delivery$(P_i, m_k) \rightarrow$ send(m_{k+1}), with $P_i \in P$. Otherwise, it is a* non-causal z-path.

Definition 8. *A local checkpoint C^y_j* z-depends *on a local checkpoint C^x_i, denoted $C^x_i \xrightarrow{z} C^y_j$, if:*

1. *$j = i$ and $y > x$, or*
2. *there is a* z-path *from C^x_i to C^y_j*

3.4. *Fuzzy Logic*

Definition 9. *Zadeh [29] gives the following definition: a* set *A, is defined as a membership function $f_A(x)$ that maps the elements of a universe X to elements in the interval* $[0, 1]$: $f_A(x) : X \rightarrowtail [0, 1]$ *and represent a degree of membership of x in A.*

Meaning that the close $f_A(x)$ values to 1 implies a higher the degree of membership of x in A.

To represent the fuzzy set A consider a pair of values: each element $x \in X$ having a degree of membership to A.

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

Definition 10. *Fuzzification converts a value or quantity to fuzzy quantity.*

One of the most common fuzzifiers is the triangular function, next described:

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

L,C and R values delimit the fuzzy set A, where C is the most substantial input value to A.

Definition 11. *Defuzzification converts a fuzzy quantity into a value or quantity.*

Defuzzification is done by applying the *weighted average* method, using the next algebraic equation:

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

where $\sum$ denotes algebraic summation and a_c is the centroid of each symmetric membership function.

3.4.1. A Brief Description of FIS

A fuzzy inference system uses fuzzy logic to model almost anything into an input space to an output space. Using FIS, one can formalize human language into a simple mathematical model. FIS consists of four modules, next described:

- *Fuzzification module:* This module transforms anything that is modelled by crisp numbers into a membership function and transforms them into fuzzy sets using a fuzzification function.
- *Knowledgebase:* stores the rules that are given by experts in the form of if-then rules.
- *Inference engine:* makes inferences from the inputs and tries to simulate human reasoning by using if-then rules
- *Defuzzification module:* this module gives as output a crisp value that can be easily interpreted by humans or computers; it transforms membership functions into fuzzy sets.

The most known type of FIS are the Mamdani and the Sugeno [30].

- A Mamdani system gives as a result fuzzy outputs and inputs.
  - *If x is A and y is B then z is V*
- A Sugeno system gives as outputs "crisp" values and takes fuzzy inputs from the inference engine.
  - *If x is A and y is B then z = f (x, y)*

4. Dynamic Checkpointing for CiC Algorithms Based on Fuzzy Non-Functional Dependencies

4.1. Architecture

We propose a general architecture for distributed and heterogeneous environments, dependable enough to warranty system fault tolerance upon failures, as illustrated in Figure 2. In a previous work, we merged checkpointing mechanisms (CiC) with autonomic computing (MAPE control loop) [1]. In this work, we add an evaluation mechanism for each transaction that takes place in the system, we evaluate all Web services in play, allowing us to evaluate the overall QoS, as shown in Figure 2. By evaluating individual QoS values for Web services we can decide to checkpoint or to discard checkpoints depending on systems current behaviour. Our approach decreases the number of forced checkpoints when the system is maintained in good or fair condition, thereby reducing the overhead of the system.

Next, we give a small description for the MAPE control loop and we establish how to complement it with this work. The *Monitoring* phase re-collects data from each of the processes for the system in play. Monitoring initiates the petitions, sending and receiving process's requests/responses. This stage also computes QoS parameters such as response time, CPU and memory percentage usage. The *Analysis* layer, in charge of detecting abnormal situations and decision making based on extracted information from messages and logs, i.e., checks whether the behaviour of the system is normal, suffers any abnormality or if it is in a faulty state. To predict the immediate future state of the system, artificial intelligence and soft computing may be used, for example, based on autonomic computing [1] or any other approach that supports predictions like Hidden Markov Model (HMM) [31], Bayesian Networks [32,33] or Fuzzy Logic. The Analysis layer computes the overall QoS value at a given time, using soft computing, during any Web service transaction. We found that using fuzzy logic for QoS evaluation does not incur system degradation, as shown in the results section, i.e., it does not affect its performance.

We consider the order of messages as well as their timing in communication events as these are relevant for system consistency. Therefore, we extrapolate the principles of the communication-induced checkpointing (CiC) mechanism to the *Fault Tolerance* layer. CiC does not only guarantee the order of messages and consistency, but it also establishes uniformity when building consistent global snapshots (CGSs).

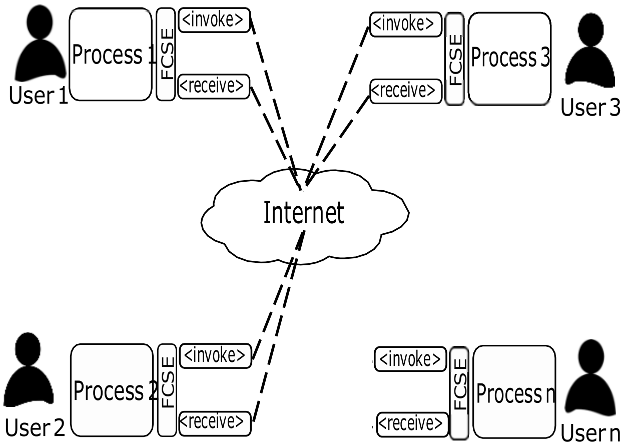

Figure 2. General Architecture.

Distributed systems are dynamic; there exists competition for shared resources between diverse parties, for instance, many clients requesting to buy a product from a web which in turn request availability of the product to several service providers. Autonomic computing address such issues by introducing the notion of an autonomic manager; the autonomic manager is in charge of taking corresponding actions upon detecting anomalies in the system. The autonomic computing is proactive, in it, there exists a knowledgebase that generates information about previous experiences from monitored and managed elements. A fuzzy inference system can be used along with *Managed Elements* for diagnosis and learning purposes; in autonomic computing monitored elements are known as managed element. We represent non-functional requirements (QoS) as fuzzy variables. By monitoring QoS, at runtime, we easily know the behaviour of individual Web services and also for Web services compositions. Take for instance, Figure 3, for autonomic Web services, where *Autonomic Managers* control and manage Web services.

Figure 3 shows the architecture for Autonomic Web Services. Web Services instances and the underlying infrastructure are the managed elements. These managed elements are monitored and controlled by Autonomic Mangers via touchpoint interfaces.

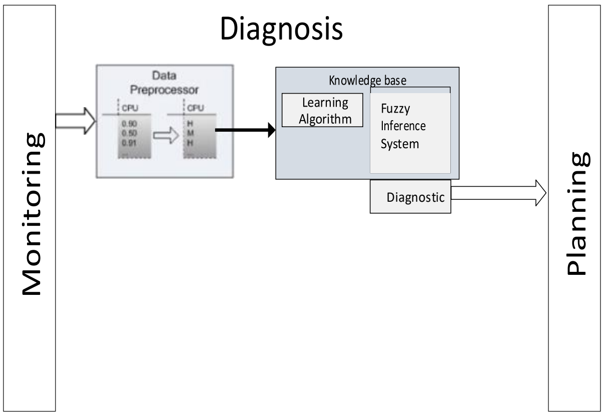

Figure 3. Autonomic Web Service architecture.

The *Autonomic Manager* has the following functions:

- *Monitoring*: the monitoring module aims at detecting, at runtime, changes known as symptoms for managed elements, for instance, when monitoring the Service Level Agreements (SLA) and detecting deviations in response times.
- *Analysis*: to analyze module gives a diagnosis about the current state of the system, based on monitored data, for instance, when QoS parameters overpass a certain threshold. We use FIS to evaluate and give a diagnosis for Web services.
- The systems common *Knowledgebase* stores policies; these policies can change over time. The policies define configurations for monitored components and there exists predefined policies. New policies can appear when changes occur in the system; policies trigger corrective actions. Web services base their autonomic behaviour in these rules to achieve self-management functions.

The autonomic computing establishes that systems have to learn and adapt to changes, we achieve such behavior by relying on FIS. FIS mimics human reasoning by adapting to abrupt changes in the system environment (changes that are usually not considered in the predefined policies). Therefore, FIS complements the Autonomic Manager by achieving an efficient diagnosis for systems behavior as it contemplates all its possibilities.

4.2. Dynamically Checkpointing Autonomic Web Services

On the one hand, the *Fault Tolerance* layer relies on the communication-induced checkpointing (CiC) mechanism [1]. On the other hand, we propose QoS assessment and systems performance evaluation for generating checkpoints dynamically, at any given time, thus obliterating the notion that CiC mechanisms have high implementation cost. Web services performance can be known using simple if-then rules, rules brought up by fuzzy logic. It is easy to evaluate a system and for it to adapt by establishing a set of rules from relationships between inputs (monitored data) and outputs (systems actual behaviour). We exploit QoS criteria by measuring systems non-functional requirements and simulate under different workloads. Therefore, managing the QoS threshold implies the ability to checkpoint, different CiC algorithms, in a dynamic manner.

Measuring QoS parameters associated with messages exchanged between different processes; we propose a QoS temporal association. We define a fuzzy-consistency system between messages, we know the actual systems performance; we compute the overall QoS by retrieving information among exchanged messages and compute their individual QoS parameters.

We formulate the FCSE based on the *fuzzy-consistency system (FCS)*. Let the FCS relate monitored QoS parameters such as CPU, memory percentage usage and response time (RT) as follows:

“The quality of system behaviour at a certain time *t* relative to the QoS parameters."

We consider three linguistic variables:

- *RT* indicates the time it took a process to send a request up until it received its response back.
- *CPU %* indicates the CPU usage; it is used as behavioural metric to know the performance at a certain time.
- and the *Memory %* indicates the memory percentage usage; it is used as a behavioural metric to know the performance at a particular time.

We use FIS to determine the state of the FCSE; next we give a brief definition:

(a) Fuzzification: To fuzzify (as crisp values) inputs and outputs we use a triangular symmetric fuzzifier, for instance, Mamdani FIS (refer to Definition 10). Each variable was defined as follows:

- *RT: VG(e)* associated to *very good*, *G(e)* associated to *good*, *A(e)* associated to *average*, *B(e)* associated to *bad*, and *VB(e)* associated to *very bad*.
- for the *CPU: CPU_L(e)* associated to *low*, *CPU_M* associated to *medium*, *CPU_H* associated to *high*, and *CPU_VH* associated to *very high*.
- and finally for the *Memory MEM_L(e)* associated to *low*, *MEM_M(e)* associated to *medium*, *MEM_H* associated to *high*, and *MEM_VH* associated to *very high*.

Table 1 lists each linguistic variable and gives its corresponding fuzzy set. Where σ represents all the values for the response time, varying from the smallest value to the biggest value; based on *ITU G.1010* standard values for Web services, preferred $\in [0-2)$ seconds, acceptable $\in (2-4)$ seconds and unacceptable $\in$ (4 to infinity). Where ϭ represents all the values for CPU $\in [0,100]$. Finally, ϕ represents all the values for the memory used at a specific time $\in [0,100]$.

Table 1. Membership functions and their corresponding variables.

Variable Values of Membership Functions				
	Set	L	C	R
Response Time	VG	$\sigma_0 \quad \sigma_2$	σ_0	σ_2
	G	$\overline{\sigma}_0$	σ_2	σ_4
	A	σ_2	σ_4	σ_6
	B	σ_4	σ_6	σ_8
	VB	σ_6	σ_8	$\sigma_8 + \sigma_2$
CPU	CPU_L	ϭ$_0$ − ϭ$_2$	ϭ$_0$	ϭ$_2$
	CPU_M	ϭ$_0$	ϭ$_2$	ϭ$_4$
	CPU_H	ϭ$_2$	ϭ$_4$	ϭ$_6$
	CPU_VH	ϭ$_4$	ϭ$_6$	ϭ$_8$
Memory	M_L	$\phi_0 - \phi_2$	ϕ_0	ϕ_2
	M_M	ϕ_0	ϕ_2	ϕ_4
	M_H	ϕ_2	ϕ_4	ϕ_6
	M_VH	ϕ_6	ϕ_8	$\phi_8 + \phi_2$

(b) Fuzzy inference (FI):

We determine the FCSE by using Mamdani inference rules; thus, we compute the performance of the system (based on QoS parameters) at any particular time.

We defined 80 combinations for inference rules [17], therefore computational is not considered. We argue that our evaluation does not incur in high system overhead, as experimental results show.

(c) Defuzzification: The output (FCSE) of the Mamdani FIS is a fuzzy variable, we convert it to a scalar value easily interpreted by a human. We use the defuzzication for FIS's output using the weighted average method, please refer to Definition 11.

5. Results

The results are twofold; first, we show the performance of our approach, and results show it does not incur in system degradation, i.e., as first step we only measure the fuzzy implementation. Second, we evaluate QoS non-functional requirements between all messages exchanged in the system, reducing dynamically the number of forced checkpoints diverse algorithms generate depending on actual systems behavior.

5.1. Experimental Setup

The first aim of the experiments is to show that using our solution does not incur in high system overhead; we established a simulation environment with the following characteristics; we deployed Web services to carry out dummy services, yet interactive enough to intercommunicate with other Web services and service providers; we simulated different workload conditions for diverse system performance; we measure our model of fuzzy non-functional parameters.

We experimented in a Dell Precision Workstation with the following characteristics: 16 GB RAM, Windows 7 64-bit operating system. We host the WS02 Application Server for Web services deployment, and we stress-test those Web services with concurrent requests incoming from diverse JAVA based clients, to have a more or less real-world situation, for instance, when diverse clients are trying to buy something online from different service providers. We used the *Jfuzzylogic* http://jfuzzylogic.sourceforge.net/html/index.html framework for the FCSE and fuzzy related programming compatible with JAVA.

5.2. Fuzzy Setup

We performed two experiments. The first experiment, consisted of measuring systems performance using and not using the proposed evaluation method. The second experiment, consisted of stress-tests for diverse system performance and the overall QoS evaluation to dynamically checkpoint using three different communication-induced tools such as DCFI, FI and FINE.

First, we described the corresponding membership functions for inputs and outputs. The RT (response time) was $\in (0, 100)$ seconds. *ITU G.1010* tries to standardize Web services performance for real-life scenarios and recommends the following values: preferred $\in (0, 2)$ seconds, acceptable $\in (2, 4)$ seconds and unacceptable $\in (4, infinity)$. CPU % and Memory % usage $\in$ (0, 100), depending on its actual usage. The FCSE $\in$ (0, 5), in charge of the overall QoS; we associated a membership function with all the variables above as shown in Table 2.

Just to clarify, the FCSE set is associated to the following linguistic variables:

- *FCSE: FCSE_VH(e)* associated to *very high*, *FCSE_H(e)* associated to *high*, *FCSE_A(e)* associated to *average*, *FCSE_L(e)* associated to *low*, and *FCSE_VL(e)* associated to *very low*.

FCSE evaluation gave crisp output values $\in$ (0, 5); if the FCSE output value was close to zero, meaning no degradation of the system was found, the FCSE was very high, therefore the RT crisp value corresponded to very high; the CPU % and Memory % usage crisp values were very low. Contrarily, if the FCSE crisp value was close to five, it means the system suffered degradation; if the FCSE was very low it resembled performance problems, therefore, Web services had very high RT and very high CPU % and Memory % usage.

Table 2. Membership functions values for inputs and outputs.

Variable	Values of Membership Functions			
	Input			
	Set	L	C	R
Response Time	VG	−2.5	0	2.5
	G	0	2.5	5
	A	2.5	5	7.5
	B	5	7.5	10
	VB	7.5	10	12.5
CPU	CPU_L	– 33.33	0	33.33
	CPU_M	0	33.33	66.67
	CPU_H	33.33	66.67	100
	CPU_VH	66.67	100	133.3
Memory	M_L	– 33.33	0	33.33
	M_M	0	33.33	66.67
	M_H	33.33	66.67	100
	M_VH	66.67	100	133.3
	Output			
FCSE	FCSE_VH	–1.25	0	1.25
	FCSE_H	0	1.25	2.5
	FCSE_A	1.25	2.5	3.75
	FCSE_L	2.5	3.75	5
	FCSE_VL	3.75	5	6.25

5.3. Performance Evaluation

5.3.1. Using QoS Criteria for Diagnostics

From the first experiment, recall that the purpose was to stress-test Web services with a lot of concurrent requests and to measure our proposed approach and the impact it has in systems performance. We obtained the following results: Figures 4 and 5 show the same number of messages sent from 10 clients up to 5000 clients. In both Figs. we report the average response time; the response time measured the time it takes a client from sending their request (a message) until they get their response back (from a service provider). From Figures 4 and 5 we observed that the average RT and its corresponding standard deviation were quite similar in both (approaches without QoS evaluation and after using the QoS evaluation based on FIS) graphs for all cases even after highly increasing the number of clients; the QoS evaluation performed better by reducing abrupt increments in the average response time as the number of clients grows. In collaborative environments where intercommunication take place an important issue is resource usage and storage. Therefore, proposed solutions must remain constant and scalable even in the presence of large-scale environments, our solution aims to support these issues, and as illustrated by Figure 5. Besides, our proposal remained constant for a high number of interactions; therefore, it is scalable.

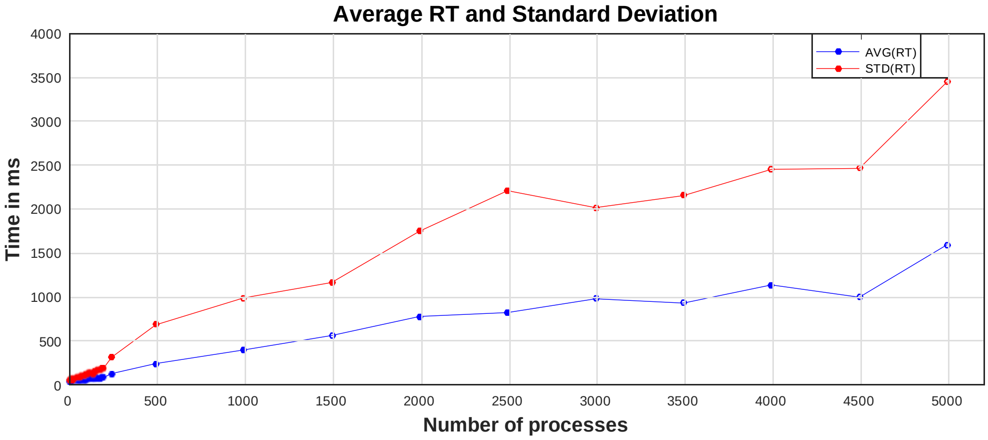

Figure 4. No QoS evaluation computing average response time and its standard deviation.

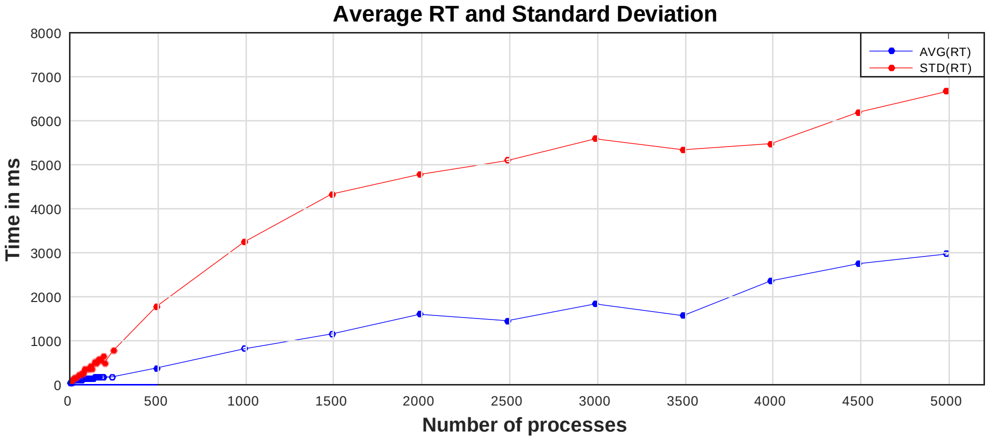

Figure 5. Using QoS evaluation, computing average response time and its standard deviation.

To have a significant statistical sample, we executed each run for 100 times. For instance, when 20 concurrent clients requested a Web service we collected 2000 samples, and computed their average RT and standard deviation, as shown in Figure 5. For 50 concurrent clients, we collected 5000 samples, so on. In addition, each scenario constituted a dedicated server that hosted Web services and each Web service instance computed the overall QoS for each request/response. Specifically, computing at runtime, at any point in time, the fuzzy-consistency system evaluation (FCSE). The statistical sample size followed the Cochran's sample size formula [34]:

$$n = \frac{t^2 pq}{d^2}$$

$$n = \frac{(1.65)^2(.5)(.5)}{(0.0825)^2} = 100$$

where t = value for the selected alpha level of 0.05 in each tail = 1.65, $(p)(q)$ = estimate of variance = 0.25, d = acceptable margin of error for the portion being estimated = 0.0825 (the error researcher was willing to accept 8.25% for our case).

From Figures 4 and 5 we can conclude that the proposed QoS evaluation approach slightly incremented the average RT, however, it remained constant, and it helped having a fast evaluation of the system at any time. The evaluation helped us to diagnose the system and take proactive actions, in our case decide if a checkpoint was essential or not. Hence, we could compute the evaluation of the system; at runtime our next goal was to leverage this calculation. Adapting to changes in the environment one can take advantage by establishing policies for QoS non-functional requirements and reduce the number forced checkpoints taken. In the next section, we show how by evaluating QoS non-functional parameters we can discard useless checkpoints regarding Web services' behavior at any given time.

5.3.2. Dynamic Checkpointing for CiC Algorithms

We ran different scenarios with diverse values for QoS parameters, and followed an exponential distribution, simulating real-world conditions, see Figures 6–8. Therefore, the crisp values for the FCSE varied from 0 to 5 after evaluated by the FIS system; this was named the FCSE window. This window can be very rigorous; for instance, we took forced checkpoints when the output computed crisp value of the FCSE satisfied that it was equal to 0.625 or lower. We referred to a relax window to that one allowing a more degraded system, having an FCSE crisp output value of 3.75. Also, we considered crisp intermediate values between the rigorous window and the relaxed window and evaluate windows with the following values: 0.625, 1.25, 1.875, 2.5,3.125, y 3.75.

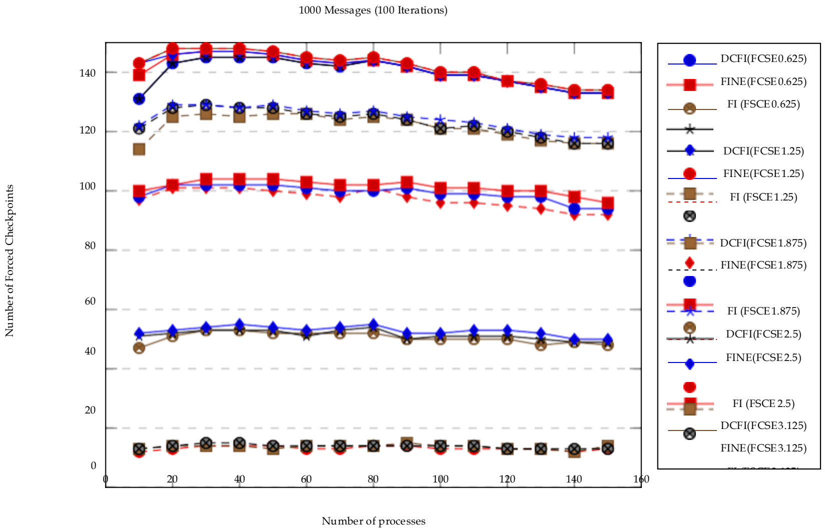

Figure 6. The number of forced checkpoints for 1000 messages sent. Evaluating three Quality of Service (QoS) non-functional parameters.

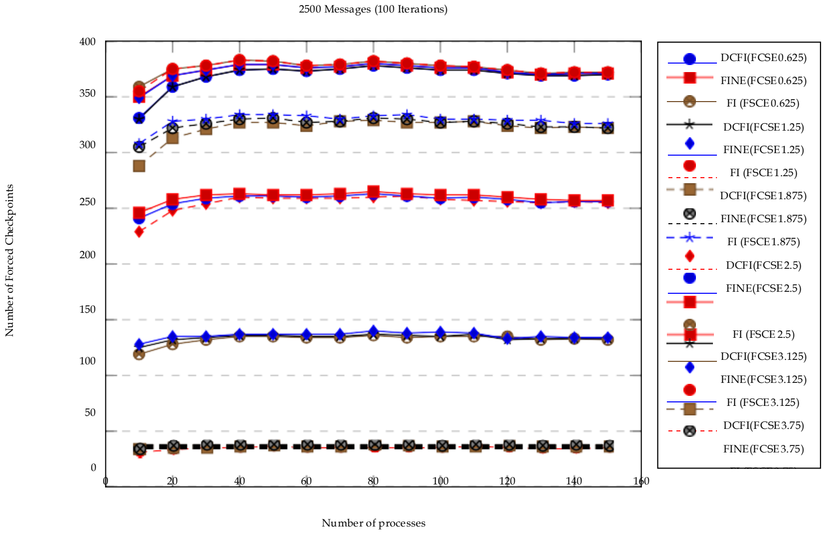

Figure 7. The number of forced checkpoints for 2500 messages sent. Evaluating three Quality of Service (QoS) non-functional parameters.

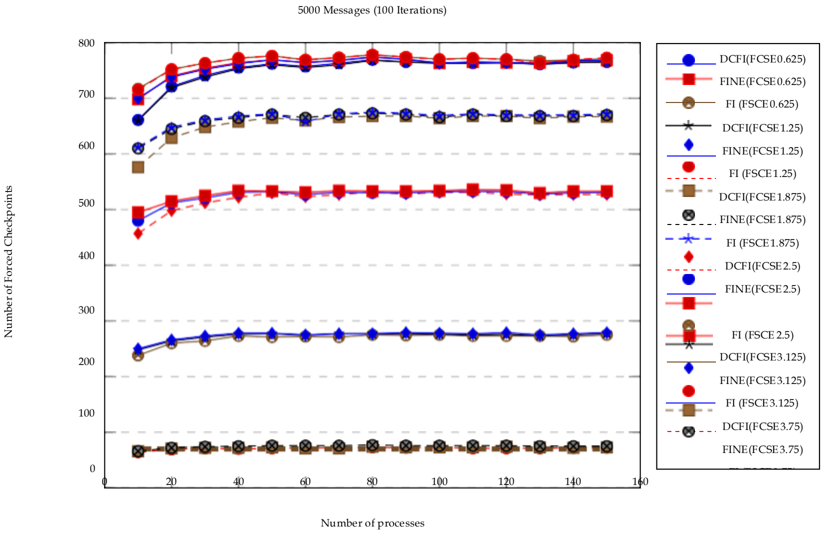

Figure 8. The number of forced checkpoints for 5000 messages sent. Evaluating three Quality of Service (QoS) non-functional parameters

In order to show that our proposal can be adapted, to simulate several system requirements, we also ran different scenarios where we include four QoS non-functional parameters, such as CPU (Good Medium, Bad), Memory (Good, Medium, Bad), Response Time (Good, Average, Bad) and Reliability(Good, Average, Bad), see Figures 9–11.

We compared the performance of three different CiC algorithms, namely DCFI, FI and FINE. These were chosen because they are recent algorithms and the most efficient reported in the literature. We simulated and analyzed these three algorithms using the distributed checkpointing simulator ChkSim [16] and JFuzzyLogic. ChkSim simulator, models distributed systems in a deterministic manner reproducing the same behavior for two or more algorithms, and allows running simulation as often as necessary. We took the number of forced checkpoints as key performance indicator to compare each checkpointing algorithm regarding the overhead each one of them produce.

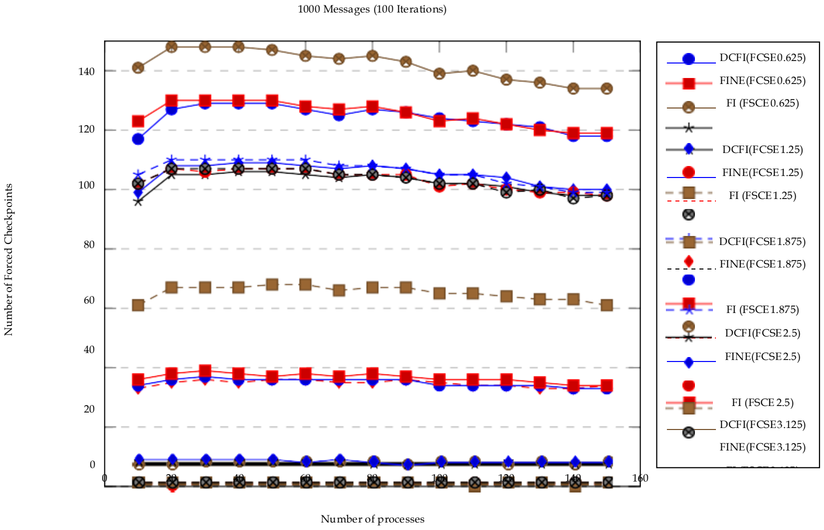

Figure 9. The number of forced checkpoints for 1000 messages sent. Evaluating four QoS non-functional parameters.

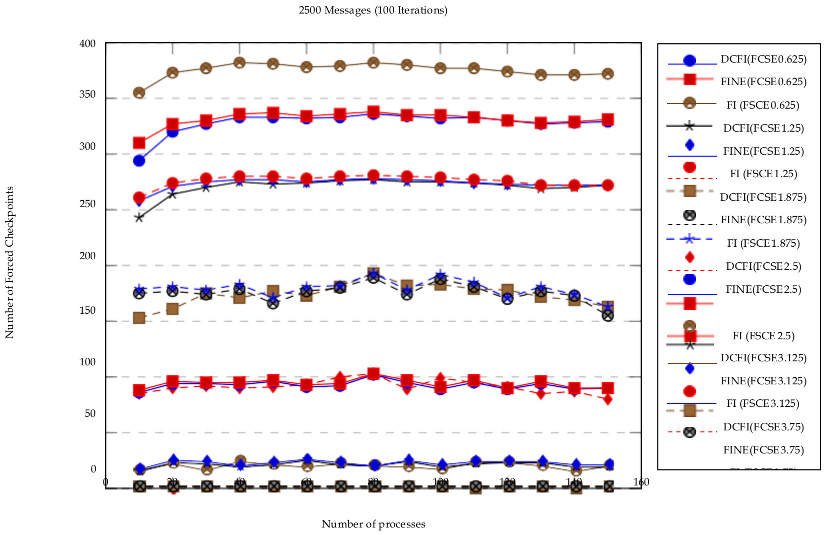

Figure 10. The number of forced checkpoints for 2500 messages sent. Evaluating four QoS non-functional parameters.

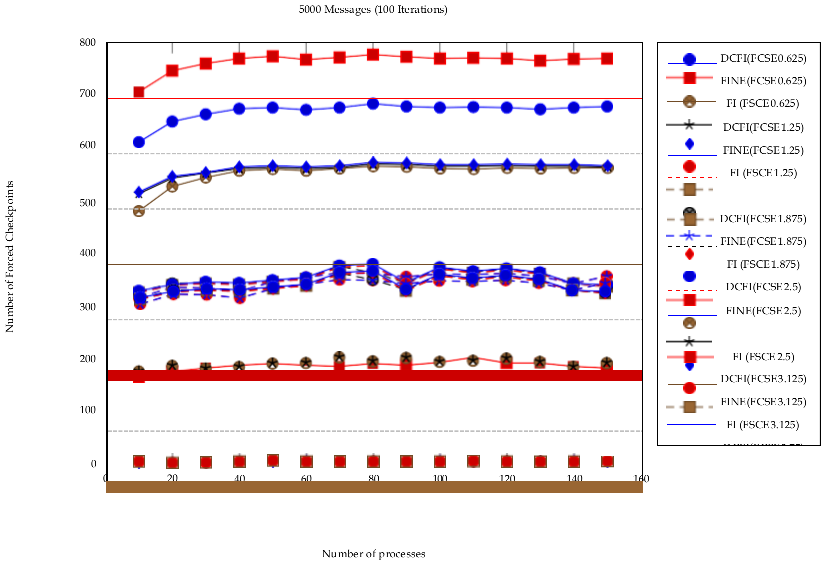

Figure 11. The number of forced checkpoints for 5000 messages sent. Evaluating four QoS non-functional parameters.

In Figures 6–8, we analyzed the performance of three different communication-induced checkpointing mechanisms (DFCI, FINE, and FI) in terms of forced checkpoints. We used the next configuration; scenario one consisted of 0 to 160 concurrent clients each sending 1000 messages; scenario two consisted of 0 to 160 concurrent clients each sending of 2500; scenario three consisted 0 to 160 concurrent clients each sending 5000 messages; for each scenario, we varied the QoS window. Again to have a significant statistical sample we executed each point of each scenario for 100 iterations with different checkpointing patterns as well as their communication or messages exchanged.

Figures 6–8 shows that the DCFI, for all checkpointing mechanisms, generated the lowest number of forced checkpoints, this has to do with the QoS window. For example, for a rigorous QoS the number of forced checkpoints was similar to the one presented observed for DCFI without using our approach. Yet, as the window relaxed the number of forced checkpoints reduced drastically. For all cases the best performing checkpointing mechanism was the DCFI, because it generated fewer checkpoints compared to FI and FINE. FI was the second-best checkpointing mechanism, and FINE also reduced its checkpointing generation but compared to the other two mechanisms it presented the worst cases.

Experiments presented in Figures 9–11 shows that if the number of non-functional parameters increased to evaluate the system performance, then this allowed relaxing and detecting smaller changes in the system behavior; as a consequence, fewer forced checkpoints were generated. For all the cases presented in these Figs. the number of checkpoints generated was lower by improving the granularity that monitors the system behavior.

To conclude this section, based on results presented in Figures 6–8, we can argue that our approach correlated both cost (QoS parameters: response time, CPU, Memory, reliability) and benefit (how well the system behaves) in the following way:

- When QoS was very high it implied saving checkpoints, because there were enough resources available, and the cost of storing checkpoints was minimal.
- When QoS was very low it implied not saving many checkpoints, because there were not enough resources available, yet our approach saved some checkpoints for preserving consistency upon a failure.
- The granularity to monitor the system performance through non-functional parameters allowed us to detect smalls changes, and the system evaluation was enriched; hence fewer forced checkpoints could be generated when the system behavior was good.

5.4. Comparison between Static Approaches and Our Proposed Dynamic Approaches

We designed a case of study to observe the differences between static CiC approaches, that did not adapt to the degradation of the system, and our proposal. The case of study considered that the QoS conditions to satisfy were the most demanding, so it was necessary to guarantee the highest QoS for the attention of users. Tables 3–5 show how as the number of messages and processes increased, considering the idealized conditions with the highest rigor in the quality of service parameters, using the most rigorous QoS window of 0.6225; there was a small decrease in the number of forced checkpoints and even in some cases the number of forced checkpoints taken were the same. We reduced the number of forced checkpoints by 0.2% to 4% with respect to CiC static methods, however Tables 3–5 only show the QoS rigorous case; the reduction in the number of forced checkpoints is clearly expressed in Figures 6–8, which would be more significant for environments where the quality of service could be relaxed without compromising the system efficiency if applications allowed.

Table 3. Comparison between static approach and dynamic approach for 1000 messages.

*N*º of Processes	*DCFI* No Fuzzy	*DCFI* Fuzzy	Reduction %	*FINE* No Fuzzy	*FINE* Fuzzy	Reduction %	*DCFI* No Fuzzy	*DCFI* Fuzzy	Reduction %
10	131	131	0	139	138	0.7194	143	142	0.6993
20	143	142	0.6993	146	145	0.6849	148	147	0.6757
30	145	144	0.6897	147	146	0.6803	148	147	0.6757
40	145	144	0.6897	147	146	0.6803	148	146	1.3514
50	145	144	0.6897	146	145	0.6849	147	146	0.6803
60	143	142	0.6993	144	143	0.6944	145	144	0.6897
70	142	141	0.7042	143	142	0.6993	144	143	0.6944
80	144	142	1.3889	144	143	0.6944	145	144	0.6897
90	142	141	0.7042	142	141	0.7042	143	142	0.6993
100	139	137	1.4398	139	138	0.7194	140	139	0.7143
110	139	138	0.7194	139	138	0.7194	140	139	0.7143
120	137	136	0.7299	137	136	0.7299	137	136	0.7299
130	135	134	0.7407	135	134	0.7407	136	135	0.7353
140	133	132	0.7519	133	132	0.7519	134	133	0.7463
150	133	132	0.7519	133	132	0.7519	134	133	0.7463

Hence, it is essential to remark that when we established a very high quality of service, our model behaved almost similar to static CIC approaches. This is possibly because the fuzzy evaluation approach can adapt its behavior and maintain the system consistency and efficiency regarding to established QoS parameters by the user. To conclude, adaptability goes hand on hand with the allowed system degradation, since the number of forced checkpoints generation is adjusted for situations where the quality of service can be relaxed.

Table 4. Comparison between static approach and dynamic approach for 2500 messages.

Nº of Processes	***DCFI* No Fuzzy**	***DCFI* Fuzzy**	**Reduction %**	***FINE* No Fuzzy**	***FINE* Fuzzy**	**Reduction %**	***DCFI* No Fuzzy**	***DCFI* Fuzzy**	**Reduction %**
10	331	328	0.9063	367	350	4.6322	375	359	4.2667
20	359	357	0.5571	371	369	0.5391	378	375	0.7937
30	368	366	0.5435	377	374	0.79576	383	378	1.3055
40	374	372	0.5347	379	376	0.79156	383	382	0.2611
50	375	373	0.5333	379	377	0.5277	382	378	1.0471
60	373	370	0.8043	376	374	0.5319	379	378	0.2639
70	375	372	0.8	377	377	0	382	379	0.7853
80	378	376	0.5291	380	377	0.7894	382	380	0.5236
90	376	374	0.5319	378	374	1.0582	380	378	0.5263
100	374	372	0.5347	376	373	0.7979	378	377	0.2646
110	374	371	0.8021	376	372	1.0638	377	374	0.7958
120	371	368	0.8086	372	368	1.0753	374	371	0.8021
130	369	366	0.8130	370	368	0.5405	372	371	0.2688
140	369	367	0.5420	371	369	0.5391	372	372	0
150	370	367	0.8108	371	369	0.5391	372	371	0.26882

Table 5. Comparison between static approach and dynamic approach for 5000 messages.

Nº of Processes	***DCFI* No Fuzzy**	***DCFI* Fuzzy**	**Reduction %**	***FINE* No Fuzzy**	***FINE* Fuzzy**	**Reduction %**	***DCFI* No Fuzzy**	***DCFI* Fuzzy**	**Reduction %**
10	661	656	0.7564	698	693	0.7163	752	717	4.6543
20	721	717	0.5548	740	734	0.8108	763	752	1.4417
30	741	737	0.5398	754	749	0.6631	772	763	1.1658
40	755	749	0.7947	764	759	0.6545	776	772	0.5155
50	762	757	0.6562	769	764	0.6510	776	769	0.9021
60	757	752	0.6605	764	759	0.6545	773	769	0.5175
70	763	758	0.6553	768	763	0.6510	778	773	0.6427
80	769	764	0.6510	774	768	0.7752	778	774	0.5141
90	765	760	0.6536	770	765	0.6494	774	770	0.5170
100	763	758	0.6553	763	761	0.2621	772	770	0.2591
110	763	760	0.3932	765	763	0.2614	772	770	0.2591
120	764	759	0.6545	764	762	0.2618	770	767	0.3896
130	761	756	0.6570	764	759	0.6545	769	767	0.2601
140	764	758	0.7853	767	761	0.7823	771	769	0.2594
150	765	760	0.6536	768	761	0.9115	771	769	0.2594

6. Discussion

Implementing the communication-induced checkpointing (CiC) mechanism to whichever device or system, brings benefits such as supporting fault tolerance, and increments regarding dependability. We leverage the fact that there is a trade-off between systems performance under different circumstances, i.e., when system performance is correct and when system performance suffers degradation or deviations. Therefore, we minimize performance issues due to storage space and propose a solution that checkpoints efficiently.

Various approaches for distributed checkpointing algorithms are available (see Section 2), but most of them rely their efforts on strategies on how to checkpoint and avoid dangerous patters: zigzag paths and z-cycles. Most do not consider discarding forced checkpoints and therefore, are not appropriate for less powerful devices, cannot consider non-functional requirements and cannot discard unnecessary the frequency for checkpointing in their solutions.

Considering QoS while the checkpointing mechanism is run, systems can leverage opportunities to discard forced triggered checkpoints. Thus the frequency for generating checkpoints is diminished.

7. Conclusions

We show the first solution that allows a dynamic and adaptable generation of forced checkpoints. Carried out through a fuzzy consistency system evaluation, identifying useless forced checkpoints. We compared three different communication-induced checkpointing-based mechanisms; experimental results demonstrate our efficiency, as we managed to reduce the number of forced checkpoints, because we took into account the quality of service (QoS) of the system at a given time. We consider systems non-functional requirements for the generation of checkpoints, we dynamically adapt to changes in the system since we consider good and bad system performance for our measurements; we consider a window of rigorous quality of service as well as a relaxed window. For these windows the delayed checkpointing algorithm (DCFI) outperforms other algorithms and has the least number of forced checkpoints. We reduced the number of forced checkpoints by using our approach; experimental results show that the DCFI mechanism has better performance over FI and FINE. For instance, DCFI performs better than the other two mechanisms but only 3% better. All compared algorithms reduce their generation of forced checkpoints, however, as shown in the results the reduction depends on the actual systems behavior. We can also argue that CiC based solutions are scalable because they continue to work correctly even in the presence of a high number of interactions. Plus these kinds of solutions increase systems dependability without harming systems performance.

As future work, we can assign a weight to all inputs and check if a gain or reduction, in terms of the number of forced checkpoints, is present for diverse communication-induced checkpointing mechanisms. In addition, we will extend the proposed solution by including a self-healing mechanism in order to detect malicious attacks and carry out system recovery by fuzzy evaluating the consistency of the system.

Author Contributions: Conceptualization, M.V.-S., L.M.-R. and S.P.-H.; formal analysis, M.V.-S., L.M.-R. and R.M.; investigation, M.V.-S.; methodology, M.V.-S.; software, M.V.-S.; supervision, L.M.-R., R.M., S.P.-H. and K.D.; validation, L.M.-R., R.M., S.P.-H. and K.D.; writing—original draft, M.V.-S.; writing—review and editing, L.M.-R. and R.M. All authors have read and agreed to the published version of the manuscript.

Funding: Mariano Vargas-Santiago specially thanks El Instituto Nacional de Astrofísica Óptica y Electrónica for the grant granted for the fulfillment of this research. Besides, Luis Morales-Rosales thanks to CONACYT for the research project 613 named, Cyber-physical Systems for the Development of Intelligent Transport Systems to carry out this contribution.

Acknowledgments: The authors want to thank Rebekah Hosse Sullivan for taking her time and reviewing this work.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

CCP	Checkpointing an Communication Patterns
CGS	Consistent Global Snapshot
CiC	Communication-induced Checkpointing
DCFI	Delayed Checkpointing Fully Informed
FCS	Fuzzy Consistency System
FCSE	Fuzzy Consistency System Evaluation
FI	Fully Informed
FINE	Fully Informed aNd Efficient
HMM	Hidden Markov Model
QoS	Quality of Service
WS	Web Services

References

1. Vargas-Santiago, M.; Morales-Rosales, L.; Pomares-Hernandez, S.; Drira, K. Autonomic Web Services Enhanced by Asynchronous Checkpointing. *IEEE Access* **2018**, *6*,5538–5547.
2. Zhao, W. *Building Dependable Distributed Systems*; John Wiley & Sons: Hoboken, NJ, USA, 2014.
3. Vargas-Santiago, M.; Pomares-Hernandez, S.; Rosales, L.M.; Hadj-Kacem, H. Survey on Web Services Fault Tolerance Approaches Based on Checkpointing Mechanisms. *J. Softw.* **2017**, *12*,507–525.
4. Tanenbaum, A.S.; Van Steen, M. *Distributed Systems: Principles and Paradigms*; Prentice-Hall: Upper Saddle River, NJ, USA, 2007.
5. Siewiorek, D.; Swarz, R. *Reliable Computer Systems: Design and Evaluation*; Digital Press: Washington, MA, USA, 2017.
6. Kephart, J.O.; Chess, D.M. The vision of autonomic computing. *Computer* **2003**, *36*, 41–50.
7. Huebscher, M.C.; McCann, J.A. A survey of autonomic computing—Degrees, models, and applications. *ACM Comput. Surv. (CSUR)* **2008**, *40*, 1–28.
8. Elnozahy, E.N.; Alvisi, L.; Wang, Y.M.; Johnson, D.B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surveys (CSUR)* **2002**, *34*, 375–408.
9. Simon, A.C.; Hernandez, S.E.P.; Cruz, J.R.P. A Delayed Checkpoint Approach for Communication-Induced Checkpointing in Autonomic Computing. In Proceedings of the 2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Hammamet, Tunisia, 17–20 June 2013; pp. 56–61.
10. Chandy, K.M.; Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. (TOCS)* **1985**, *3*, 63–75.
11. Kshemkalyani, A.D.; Singhal, M. *Distributed Computing: Principles, Algorithms, and Systems*; Cambridge University Press: Cambridge, UK, 2008.
12. Simón, A.C.; Hernandez, S.E.P.; Cruz, J.R.P.; Halima, R.B.; Kacem, H.H. Self-healing in autonomic distributed systems based on delayed communication-induced checkpointing. *Int. J. Auton. Adapt. Commun. Syst.* **2016**, *9*, 183–200.
13. Netzer, R.; Xu, J. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.* **1995**, *6*, 165–169.
14. Hélary, J.M.; Mostefaoui, A.; Netzer, R.H.; Raynal, M. Communication-based prevention of useless checkpoints in distributed computations. *Distrib. Comput.* **2000**, *13*, 29–43.
15. Luo, Y.; Manivannan, D. Fine: A fully informed and efficient communication-induced checkpointing protocol for distributed systems. *J. Parallel Distrib. Comput.* **2009**, *69*, 153–167.
16. Vieira, G.M.; Buzato, L.E. Chksim: A Distributed Checkpointing Simulator; Technical Report IC-05-034; 2005. Available online: https://dcomp.sor.ufscar.br/gdvieira/chksim/ (accessed on 3 March 2020).
17. Vargas-Santiago, M.; Pomares-Hernandez, S.; Morales-Rosales, L.; Khlif, H.; Hadj-Kacem, H. Towards Dependable Web Services in Collaborative Environments Based on Fuzzy Non-functional Dependencies In Proceedings of the 5th International Conference in Software Engineering Research and Innovation (CONISOFT), Mérida, Yucatán, Mexico, 25–27 October 2017.
18. Tsai, J. An efficient index-based checkpointing protocol with constant-size control information on messages. *IEEE Trans. Depend. Secure Comput.* **2005**, *2*, 287–296.
19. Tsai, J. Applying the Fully-Informed Checkpointing Protocol to the Lazy Indexing Strategy. *J. Inf. Sci. Eng.* **2007**, *23*, 1611–1621.
20. Luo, Y.; Manivannan, D. FINE: A Fully Informed aNd Efficient communication-induced checkpointing protocol. In Proceedings of the Third International Conference on Systems, IEEE Computer Society: Washington, MA, USA, 13–18 April 2008; pp. 16–22.
21. Luo, Y.; Manivannan, D. Theoretical and experimental evaluation of communication-induced checkpointing protocols in and families. *Perform. Eval.* **2011**, *68*, 429–445.
22. Luo, Y.; Manivannan, D. Theoretical and Experimental Evaluation of Communication-Induced Checkpointing Protocols in F E Family. In Proceedings of the 2008 IEEE International Performance, Computing and Communications Conference, Austin, TX, USA, 7–9 December 2008; pp. 217–224.
23. Simon, A.C.; Hernandez, S.E.P.; Cruz, J.R.P.; Gomez-Gil, P.; Drira, K. A scalable communication-induced checkpointing algorithm for distributed systems. *IEICE Trans. Inf. Syst.* **2013**, *96*, 886–896.

24. Jafary, B.; Fiondella, L. Optimal checkpointing of fault tolerant systems subject to correlated failure. In Proceedings of the 2017 Annual Reliability and Maintainability Symposium, Orlando, FL, USA, 23–26 January 2017; pp. 1–6.
25. Garcia, I.C.; Vieira, G.; Buzato, L.E. A Rollback in the History of Communication-Induced Checkpointing. *arXiv* **2017**, arXiv:170206167.
26. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **1978**, *21*, 558–565.
27. Hernandez, S.P.; Fanchon, J.; Drira, K. The immediate dependency relation: An optimal way to ensure causal group communication. *Annu. Rev. Scalable Comput.* **2004**, *3*, 61–79.
28. Wang, Y.M.; Fuchs, W.K. Lazy checkpoint coordination for bounding rollback propagation. In Proceedings of the IEEE 12th Symposium on Reliable Distributed Systems, Princeton, NJ, USA, 6–8 October 1993; pp. 78–85.
29. Zadeh, L.A. Fuzzy sets. *Inf. Control.* **1965**, *8*, 338–353.
30. Takagi, T.; Sugeno, M. Fuzzy identification of systems and its applications to modeling and control. *IEEE Trans. Syst. Man Cybern.* **1985**, *1*, 116–132.
31. Halima, R.B.; Guennoun, M.K.; Drira, K.; Jmaiel, M. Providing predictive self-healing for web services: A qos monitoring and analysis-based approach. *J. Inf. Assur. Secur.* **2008**, *3*, 175–184.
32. Koh-Dzul, R.; Vargas-Santiago, M.; Diop, C.; Exposito, E.; Moo-Mena, F.; Gómez-Montalvo, J. Improving ESB capabilities through diagnosis based on Bayesian networks and machine learning. *J. Softw.* **2014**, *9*, 2206–2211.
33. Koh-Dzul, R.; Vargas-Santiago, M.; Diop, C.; Exposito, E.; Moo-Mena, F. A smart diagnostic model for an autonomic service bus based on a probabilistic reasoning approach. In Proceedings of the 2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing, Vietri sul Mere, Italy, 18–21 December 2013; pp. 416–421.
34. Kotrlik, J.; Higgins, C. Organizational research: Determining appropriate sample size in survey research appropriate sample size in survey research. *Inf. Technol. Learn. Perfor. J.* **2001**, *19*, 43.