People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

Ferhat Abbas University–Setif-1

Faculty of Sciences

Department of Computer sciences

Fault Tolerance in Embedded Systems

Cloud Computing Systems

A thesis submitted in partial fulfillment of the requirements for the Doctorate degree

By: Mounya Smara

Committee composition:

Prof. Kamel Nadjet	Ferhat Abbas University -Setif 1	President
Prof. Aliouat Makhlouf	Ferhat Abbas University –Setif 1	Director
Prof. Bensalem Saddek	Grenoble University-France	Co-Director
Prof. Boukarram abdallah	Abderrahmane Mira University - Bejaia	Examiner
Prof. Benmohamed Mohamed	Constantine - 2- University.	Examiner

July 2017

Abstract

Cloud computing has become a popular computational technology across all industries, by which desired services can be accessed from any place and at any time. Cloud environments are characterized by the big data, non centralization, distribution and non-heterogeneity that bring some challenges such as: reliability which is still a major issue for cloud service providers. Fault tolerance is an active line of research in design and implementation of dependable systems. It means to handle unexpected defects, so that the system meets its specification in the presence of faults. Specification guarantees can be broadly characterized by safety and liveness properties. Reliability in cloud environment is handled by a set of fault detection and fault tolerance techniques. The fault detection is configured by monitoring and heartbeat strategies whereas the fault tolerance is performed by using techniques based on time and space redundancy such as checkpointing, retry, SGuard….etc. The main aim of this thesis is the incorporation of recovery blocks scheme to enhance reliability of cloud computing systems by providing Fail-Silent and Fault-Masking nodes. A Fail-Silent cloud node is a safe component that uses the acceptance test for self-fault detection whereas a Fault-Masking node is a safe and live component that can detect and recover from failures using the acceptance test and try blocks. The proposed strategies are proved and time and space complexities are estimated. Furthermore, a case study and a verification using the model-checker are provided for the proposed schemes to prove their efficiency and their applicability.

Keywords: Reliability, Cloud computing, Recovery Blocks, Fault Detection, Fault Tolerance, Fault Masking, Acceptance test, Component-based approach.

Résumé

Le cloud computing ou l'informatique en nuage est devenu une technologie de calcul populaire dans toutes les industries, par laquelle les services souhaités peuvent êtres consultés à partir de n'importe quel endroit et à tout moment. Les environnements cloud sont caractérisés par la grande masse de données, la non-centralisation, la distribution et le manque d'hétérogénéité. Ces caractéristiques apportent quelques défis tels que l'assurance de fiabilité ; qui reste un problème majeur pour les fournisseurs des services cloud. La Tolérance aux fautes est une ligne de recherche active dans la conception et la mise en œuvre des systèmes fiables. Cela signifie de gérer les pannes inattendues de sorte que le système réponde à ses spécifications en présence de fautes. Les garanties de spécification peuvent être largement caractérisées par des propriétés de sécurité-innocuité et de vivacité. La fiabilité dans l'environnement cloud est gérée par un ensemble de techniques de détection et de tolérance aux fautes. La détection des fautes est opérée par les stratégies de surveillance et de battement de cœur alors que la tolérance aux fautes est réalisée en utilisant des techniques basées sur la redondance spatiale et temporelle telles que le checkpointing, le ré-essai, le Sguard, …etc. Le but principal de cette thèse c'est l'incorporation du schéma des blocs de reprise pour améliorer la fiabilité des systèmes cloud computing en fournissant des nœuds défaillants silencieusement et des nœuds masquants des fautes. Un nœud défaillant silencieusement est un composant sécurisé qui utilise le test d'acceptation pour la détection automatique des fautes alors qu'un nœud masquant des fautes est un composant sûr et actif qui peut détecter les fautes et faire une reprise vers l'avant en utilisant un test d'acceptation et un ensemble de blocks d'essai. Les stratégies proposées ont été prouvées dans un contexte de support de modélisation et vérification formelle BIP et la complexité temporelle et spatiale a été estimée. De plus, une étude de cas et sa vérification à l'aide d'un model-checker ont été réalisées sur des schémas proposés afin de prouver leur efficacité et leur applicabilité.

Mots-clés : Fiabilité, Blocs de Reprises, Détection des Fautes, Tolérance aux fautes, Cloud computing, Test d'acceptation, Reprise vers l'avant, Approche de Conception à base de composants.

To my husband

To my parents

To my brothers and sisters

and

To my son Mouataz Billah.

Acknowledgement

In the name of ALLAH, The most gracious and the most merciful, Alhamdulillah, all praises to ALLAH for his blessing in completing this thesis.

This thesis would have never seen the daylight without the help of some very important people to whom I would like to express my deepest gratitude.

My most sincere gratitude goes to my supervisor, Prof. Makhlouf Aliouat and to Dr. Zibouda Aliouat for their support, advice and contribution to my studies.

Further, I would like to express my deepest gratitude to Prof. Al-Sakib Khan Pathan for providing indispensible advices, information and support on different aspects of my research.

I thank the members of my committee for their comments for my dissertation.

Last and not least, I would like to thanks all my friends and colleagues for their encouragement.

Table of Contents

List of Tables

List of Figures

List of Appendices

Introduction

Embedded Computing systems could be seen now almost everywhere in our daily life. They are found in household items, multimedia equipment, in mobile phones as well as in cars, smart munitions, satellites and so on. However, despite increasing hardware capabilities, these mobile devices will always be resource-constrained compared to fixed hardware. In order to mitigate the hardware limitations on mobile and wearable devices, cloud computing [1], [2], [3], [4], [5], [72] allows users to use remote infrastructure in an on-demand fashion. Over the past years, cloud computing has become a popular computational technology across all industries. It brings many vast advantages such as the reduction of costs, development of efficiency, central promotion of software, compatibility of various formats, unlimited storage capacity, easy access to services at any time and from any location and most importantly, the independence of these services from the hardware [94]. Cloud computing is a type of parallel and distributed computing system which consists of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) [8], [9], [10].

We could fairly state that applications developed on cloud systems are often critical in terms of human lives. For instance, many such applications could be practically employed in healthcare, military, or disaster management scenarios. Furthermore, desired services in cloud computing can be accessed from any place and at any time. These cause removing the restrictions using in systems and traditional networks in providing service to users. But that can bring some new problems, restrictions, and challenges for users and applications. The reliability of cloud application is still a major issue for providers and users. Failures of cloud apps generally result in big economic losses as core business activities now rely on them [145]. This was the case in 2011, there was a Microsoft cloud service outage which lasted for approximately 2,5 hours [149]. In December 24, 2012 a failure of Amazon web services caused an outage of Netflix cloud services for 19 hours. In October 2013, Facebook reported an unavailable service for photos and “Likes”. In January 2014, one of Google services (Gmail) was down for about 25-50 min [150].

The demand for highly dependable cloud apps has reached high levels [147]. However, there is still no clear methodology in industry today for developing highly dependable cloud applications [145]. A research presented in [146] has revealed that infrastructure and platform services offered by big players like Amazon, Google and Microsoft suffer from regular performance and availability issues due to service overload, hardware failures, software errors and operator errors. Moreover, because of the constantly increasing complexity of cloud apps and because developers have little control over the execution environment of these applications, it is exceedingly difficult to develop fault-free cloud apps. Therefore, cloud apps should be robust to failures if they are to be highly dependable [148].

Fault tolerance has always been an active line of research in design and implementation of dependable systems. It involves providing a system with the means to handle unexpected defects, so that the system meets its specification in the presence of faults. Fault tolerance is carried out via *fault detection* and *recovery* [130]. In this context, the notion of specification may vary depending on the guarantees that the system must deliver in the presence of faults [45]. Such guarantees can be broadly characterized by *safety* and *liveness* [20] properties. In fact, *Safety* properties can be ensured by fault detection techniques whereas recovery mechanisms are used to meet *liveness* properties.

In cloud computing systems, failure detection is processed by using two main strategies: Intrusion detection systems (IDS) for network or hosts attacks detection [32], [33] and Heartbeat/Pinging strategy [43] for hardware fault detection. In the other side, fault tolerance capability is configured in cloud systems via proactive and reactive fault tolerance techniques [22-27][94][98][101][102][103]. However, fault tolerance strategies used in clouds [22-27] are based on time or spatial redundancy which can tolerate only hardware faults without dealing with software bugs. According to our thorough investigation of the area, there is clearly a lack of formal approach that rigorously relates the cloud computing with software fault tolerance concerns.

Recovery blocks scheme [29],[30] is a variant of design diversity for software fault tolerance [28]. It is based on the selection of a set of operations on which recovery operations are based. Recovery blocks are composed of a set of try blocks and an acceptance Test. This earlier is an internal audit that can configure the fault detection

process. While the forward recovery can be present by the set of try blocks. For constructing highly hardware and software fault tolerance in real-time distributed computer systems, Distributed Recovery Blocks (DRB) is formulated by Kim Kan in 1983 [109][110][132]. It is a scheme that can handle the software and hardware faults in the same manner in distributed real-time environment.

In this thesis, we propose a novel formal framework for constructing reliable cloud modules using the recovery blocks scheme. The aim is to provide strategy that can enhance cloud reliability by uniform treatments of software and hardware faults by constructing Fail-Silent and Fault-Masking nodes. A Fail-Silent node is able of self-fault detection by using the acceptance test. This earlier can guarantee initial safety requirement in spite of faults. In the other hand, a Fault-Masking node is apt to handle (i.e., detect and tolerate) software, hardware and response time faults by using both the acceptance test and try blocks to ensure safety and liveness properties in the same time. In order to well explain the proposed schemes, Fire Control System is used as a case study. Time & space complexity for such schemes is estimated. Also, safety and liveness verification using the model-checker is applied on the deduced models to prove the efficiency and the applicability of the proposed schemes. BIP (Behavior, Interaction, Priority) [14], [15], [16] is used as a Component-based framework with multi-party interactions for system modelization and UPPAAL model-checker is used as a tool for simulation and verification.

The thesis is divided into five chapters. First, we introduce the background to and the motivation for the research and identify key research problems and contributions. After, the chapter 1 explains the background of fault tolerance including definitions and basic concepts then it presents the cloud computing systems, fault detection and fault tolerance techniques in the cloud environment. In Chapter 2, a survey of some current related works on fault detection and fault tolerance in cloud computing are cited. Chapter 3, introduces the component-based cloud computing approach and BIP as a Component-based framework. The Chapter 4 presents a fault detection scheme for constructing Fail-Silent cloud nodes that ensures safety properties in the presence of faults. In Chapter 5, Fault-Masking scheme is described for fault detection and recovery in cloud modules that can ensure both *safety* and *liveness*

properties in the same time. Finally, conclusion and future perspectives are cited in the conclusion section.

Chapter One
Background

Summary

1.1 Introduction

Reliability is the ability of a system or component to perform its required functions under stated conditions and for a specified period of time. One way to increasing the reliability is by employing fault tolerance strategies. Fault tolerance is defined as the ability of a system to deliver desired results even in the presence of faults. A system is considered as fault tolerant if the behavior of the system, despite the failure of some of its components, is consistent with its specifications [106].

1.2 Fault Tolerance

Fault Tolerance is carried out via fault detection and recovery [130]**.** The fault detection is the phase in which the presence of a fault is deduced by detecting an error in the state of some subsystem. After the fault detection phase, the error in the system has to be corrected this is what we call recovery. With a system recovery task, the system will reach an error-free state

1.2.1 Faults model

Three terms are crucial and related to system failure and thus need to be clearly defined, which are named failure, error and fault. Failure, error and fault [104], have technical meaning in the fault tolerance literature. A failure occurs when *"a system is unable to provide its required functions"*. An error is *"that part of the system state which is liable to lead to subsequent failure"*, while a fault is *"the adjudged or hypothesized cause of an error"*. For example, a sensor may break due to a fault introduced by overheating. The sensor reading error may then lead to a system failure. A fault can be of hardware origin, which is caused by physical malfunctions or can be a software fault which is caused by software bugs in system development.

A fault can be classified into three main groups, namely permanent, intermittent and transient faults [133], according to their stability and occurrence:

Permanent faults, are caused by irreversible physical changes. The most common sources for this kind of faults are the manufacturing processes.

Intermittent faults, are occasional error bursts that usually repeat themselves. But they are not continuous as permanent faults. These faults are caused by unstable hardware and are activated by an environmental change such as a temperature or voltage change.

Transient faults, are temporal single malfunctions caused by some temporary environmental conditions which can be an external phenomenon such as radiation or noise originating from other parts of the system.

In this thesis, the terms fault, error and failure refers to the same meaning which is the deviation from the regular behavior of the system.

1.2.2 Safety and liveness properties

Tolerating faults involves providing a system with the means to handle unexpected defects, so that the system meets its specification even in the presence of faults. In this context, the notion of specification may vary depending upon the

guarantees that the system must deliver in the presence of faults. Such guarantees can be broadly characterized by safety and liveness properties [112]. Every possible property can be expressed a by a conjunction of safety and liveness properties [113].

Safety property can be described over the state that must hold for all executions of the system. It rules that *"bad things never happen"*. As an example, the requirement for a system controlling the traffic lights of a street intersection that the lights for two crossing streets may never be green at the same time.

Liveness property, this property can be expressed via a predicate that must be eventually satisfied, guaranteeing that *"a good thing will finally happen"*. As an example of liveness violation is deadlock involving two or more processes, which cyclically block each other indefinitely in an attempt to access common resources.

1.2.3 Fault Tolerance techniques

Fault tolerance is based on redundancy. It can be: time, hardware or software redundancy [105].

Time redundancy, is based on the execution of some instructions many times (e.g., Checkpointing and rollback recovery).

Hardware redundancy is based on the idea to overcome hardware faults by using additional physical components (e.g., TMR, Coding…).

Software redundancy or design redundancy, is based on all programs and instructions that are employed for supporting fault tolerance (e.g., N version programming, Recovery blocks).

1.2.4 Recovery Blocks technique

Recovery Blocks technique [107], [108] is a variant of N Versions Software (NVS). It is based on the notion of try blocks. The try blocks are a set of operations (of a program) that can be considered as a unit of detection and recovery. Each try block contains a primary block, zero or more alternate blocks and an acceptance Test (see Figure 1.1). The possible syntax of a recovery block is the following:

ensure<Acceptance Test> by <B_1*> else by <*B_2*> ….else by <*B_n*> else error*. Where B_1 is the primary try block and B_K $(1 \leq k \leq n)$, is the alternate try block.

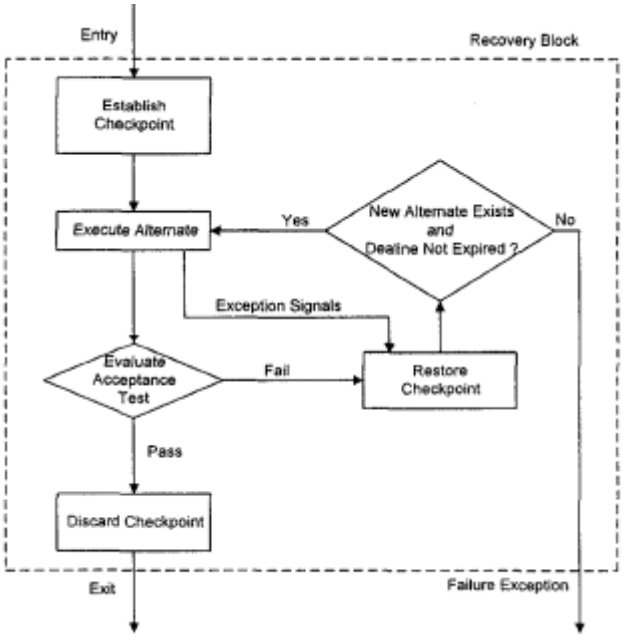


Figure 1.1. Recovery blocks architecture

The primary try block is the first block entered. It performs conventionally the desired operation. The alternate try block, is entered when the primary block fails to pass the acceptance Test. It is required to perform the desired operation in a different way or to perform some alternative action acceptable to the program as a whole. All, primary or alternates blocks must pass on exit on the acceptance test to judge their outputs. The acceptance test is a section of program which is invoked in order to ensure that the operation performed by the recovery block is to the satisfaction of the problem. The acceptance test is an internal audit logic by which the component can possesses the capability of judging the reasonableness of its computation results.

The forward recovery mechanism used in recovery blocks can enhance the efficiency in terms of the overhead (time and memory) it requires. This can be crucial in real-time applications where the time overhead of backward recovery can exceed stringent time constraints [109] [110].

1.2.5 Distributed Recovery Blocks (DRB)

Since its first formulation in 1983 by Kim Kan, [109][132] distributed recovery blocks (DRB) has been a technology for constructing highly hardware and software fault-tolerance in real-time distributed computer systems.

DRB uses a pair of self-checking processing (PSP) nodes structure together with both software internal audit and watchdog timer to facilitate real-time hardware fault

tolerance. For facilitating real-time software fault tolerance, the software implemented internal audit function and multiple versions of real-time task software which are structured via the recovery block scheme [107], [108] and executed concurrently on multiple nodes within a PSP structure. The DRB is a based on forward recovery which is primarily used when there is no time for backward recovery.

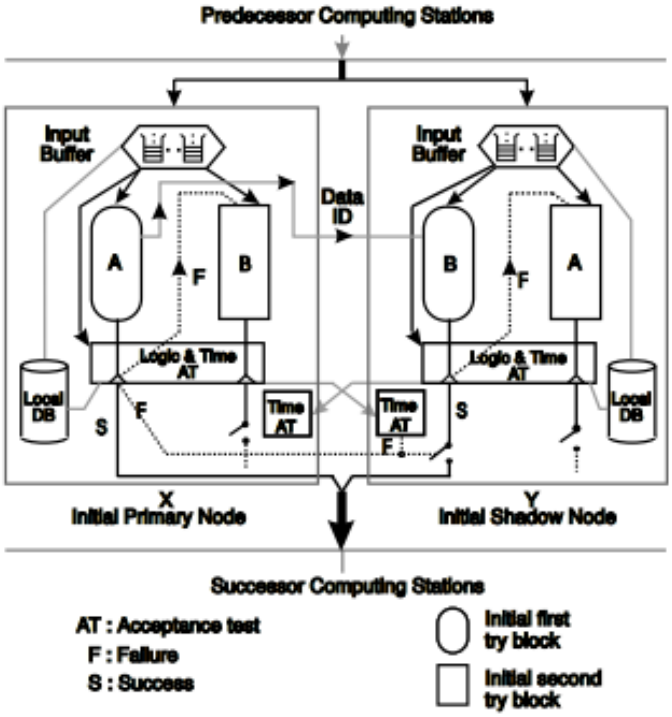


Figure 1.2. Basic structure of Distributed Recovery Blocks[109]

The Figure 1.2 presents the DRB scheme structure. *X* is the primary node which executes the primary try block *A* and *B* is the alternate try block. In the other hand, the backup node *Y* executes *B* as the primary try block and *A* as the alternate try block. We can see that the nodes use the try blocks in reverse order, this aims to avoid the failure coincidence between the nodes. In other meaning, if both nodes use the same order of try blocks, the same faults in the try block that causes a node to fail in processing a certain data set will cause the other node to fail too.

Both nodes will receive the same input data and process them concurrently by the use of two different try blocks (i.e., the try block *A* on *X* and the try block *B* on *Y*). After the execution of the try blocks, the results judgment is performed by using the common

acceptance test. As soon as each node passes the acceptance test, it updates its local database. If we assume that *X* and *Y* never fail in the same time, three cases are possible [109]:

Fault free situation, both nodes will pass the acceptance test with the results computed with their primary try blocks. In such a case, the primary node *X* notifies *Y* of its success of the acceptance test. Therefore, only the primary node sends its output to the successor node.

Failure of the primary node X, and the backup node *Y* pass the acceptance test. In this case, the node *X* attempts to inform the backup node upon its failure. At just reception of the notice, the backup node *Y* will send its output to the successor and then the role of the primary and backup nodes are reversed (see Figure 1.3). For the new primary node *Y,* the try block *A* must become the primary try block. In this time, the new backup node X (i.e., the failed primary node) will use the try block *B* for recovery in order to bring the database in the node up to date without disturbing the new primary node *Y*. After the successful retry, the try block *B* remains as the primary in the new backup node *Y*. In the case when the primary crash completely, the backup node will recognize the failure of the primary upon expiration of the preset time limit.

Failure of the backup node Y, in this case, the primary node X needs not be disturbed. The backup node will just make a retry with try block A to achieve localized recovery.

DRB is an attractive strategy for two raisons: First, the two nodes always execute two different try blocks. An advantage here is that if a data set causes one of the try blocks to fail but not both of them, then one acceptable result can be sent to the successor with little delay. Second, the current primary node always uses *A* as the primary try block and try block A is generally designed to produce better quality outputs than try block B.

A primary node can have one or more backup nodes. In other words, the primary try block can have more than one alternate try block. As long as there are more backup nodes with more alternate try blocks, the system will be more reliable.

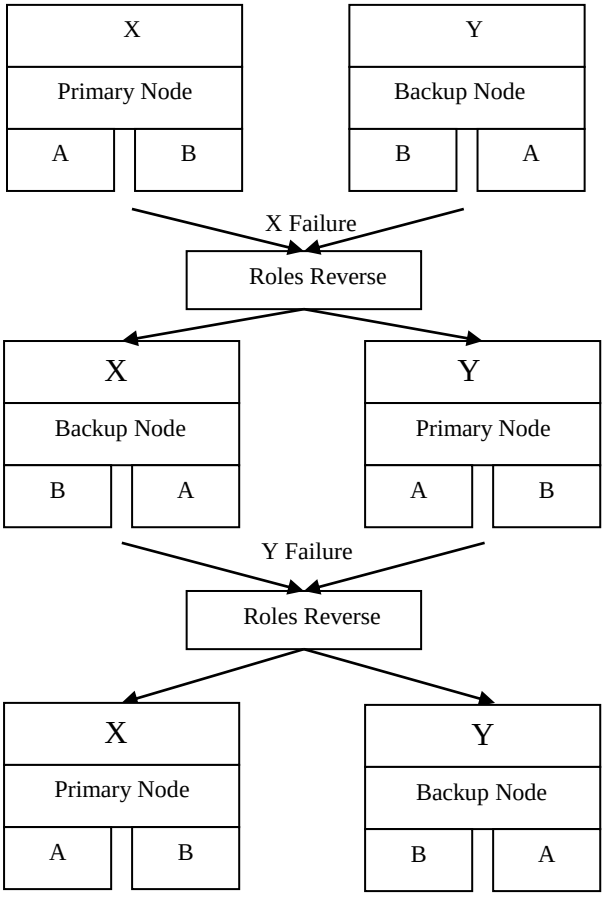


Figure 1.3. Roles reverse in DRB scheme

1.3 Cloud computing systems

1.3.1 Definition

Cloud computing [8], [9] is a type of parallel and distributed computing system which consists of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements (SLAs) established through negotiation between the service provider and the consumers [8], [10] (Figure 1.4).

The Figure 1.5 shows some examples of various cloud service providers

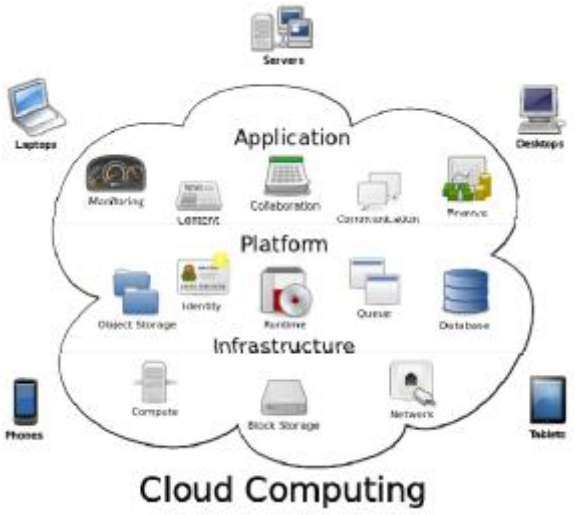


Figure 1.4. Overview of cloud computing [98]

Figure 1.5. Top cloud computing services providers

1.3.2 Architecture

The architecture of the cloud computing [96] can be divided into 4 layers: the hardware /datacenter layer, the infrastructure layer, the platform layer and the application layer, as shown in the Figure 1.6.

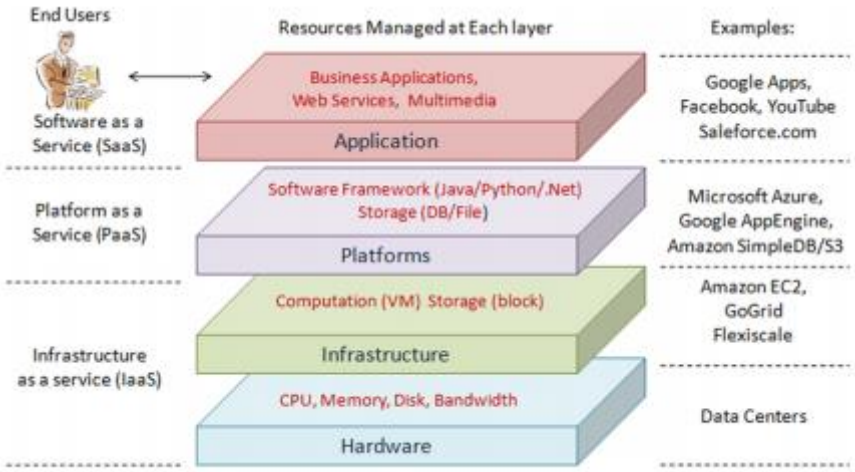


Figure 1.6. Cloud computing architecture [96]

Hardware Layer, This layer is responsible for managing the physical resources of the cloud, including physical servers, routers, switchers…etc. in practice, the hardware layer is typically implemented in data centers.

Infrastructure layer, this layer creates a pool of storage and computing resources by partitioning the physical resources using virtualization technologies.

Platform layer, consists of operating systems and application frameworks. The purpose of this layer is to minimize the burden of deploying applications directly into virtual machine containers.

Application layer, is the highest level of the architecture. The application layer consists of the actual cloud applications. Different from traditional applications, cloud applications can leverage the automatic –scaling feature to achieve better performance, availability and lower operating cost.

1.3.3 Reliability in cloud computing

The emergence of cloud computing has brought new dimension to the world of information technology. Even though cloud computing provides many benefits, one key challenge in it is to ensure continuous reliability and guaranteed availability of resources provided by it. Therefore, there is a serious need for fault tolerant mechanisms in cloud environments. Before dealing with the fault tolerance techniques in cloud systems, it should first explore the different faults model that may occur in such system. The failures in cloud computing are categorized in four classes [99]:

Hardware faults: mainly occur in processors, hard disk drive, integrated circuits sockets and memory.

Software faults: provided as a result of software bugs.

Network faults: this type of failures inhibits the communication between the cloud and the end users. It is caused by server overload and network congestion.

Timeout failure [100]: can be considered as a result of failures (e.g., hardware, software, and network). It occurs when the time needed for executing a task exceeds the delay set by the service monitor.

In our thesis, we focalize on tolerating hardware faults, software faults and Timeout failures.

1.3.3.1 Fault Detection in Cloud computing

Failures in cloud computing systems are processed by using two main strategies: Intrusion detection and Heartbeat/Pinging.

a. Intrusion and Anomaly Detection Systems (IDSs)

IDSs [32], [33], [34], [35], [36] are strongly adopted in clouds. Generally, IDSs are used for detection of network or hosts attacks (e.g., Denial of service, Buffer overflow, Sniffer attacks). They are based on *behavior observation* of the component and an alarm is raised if an abnormal behavior is detected. They can be grouped into two detection principles, namely misuse-based (or Signature-based) and anomaly-based IDS.

Signature-based IDS

This kind of IDS recognizes intrusions and anomalies by matching observed data with pre-defined descriptions of intrusive behavior. Therefore, a signature database corresponding to known attacks is specified a priori.

Anomaly-based IDS

The strategy of anomaly detection is based on the assumption that abnormal behavior is rare and different from normal behavior, and thus it tries to model what is normal rather than what is anomalous. Anomaly detectors generate an anomaly alarm whenever the deviation between a given observation at an instant and the normal behavior exceeds a predefined threshold (see Figure 1.7). Anomaly detection refers to the important problem of finding non-conforming patterns or behaviors in live traffic data. These non-conforming patterns are often known as anomalies. Three types of

anomaly-based IDS techniques are available for cloud Computing: statistical, data mining, and machine learning techniques [32],[33], [34], [35], [44].

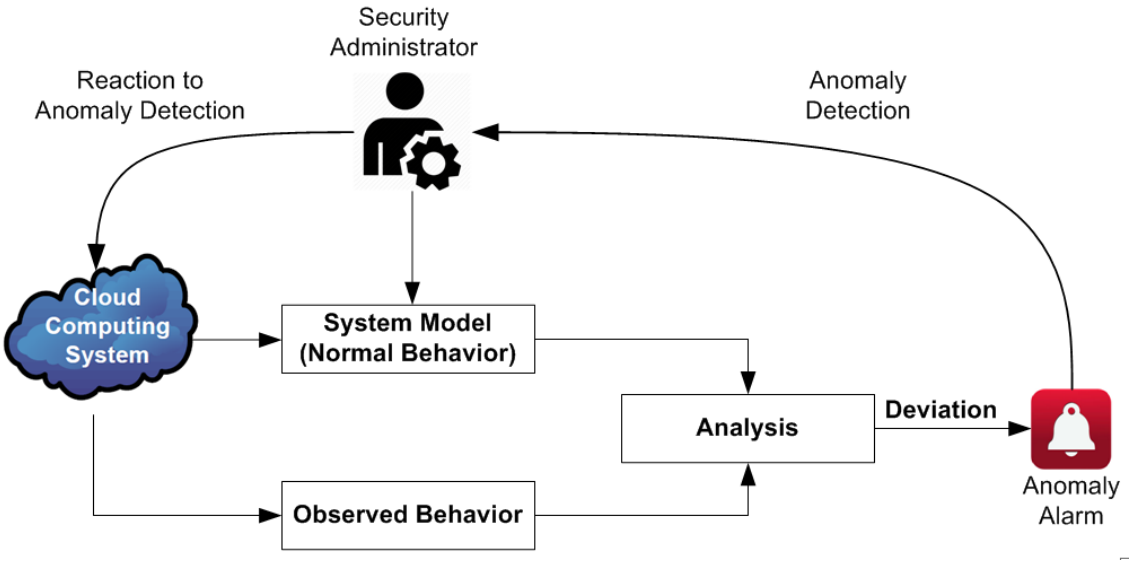


Figure 1.7. Anomaly Detection System.

Statistical based anomaly detection- In this technique, anomaly detection is realized by observing computations in the cloud and it creates a profile which stores a value to represent their behavior. In order to detect failures using these techniques, two profiles must be used. The first one stores the ideal profile while the second one stores the current profile which is updated periodically (this one calculates anomaly score). If anomaly score of current profile is higher than the threshold value of stored profile, then it is considered as anomaly and it can be detected. A survey of statistical based anomaly detection is presented in [37]. Statistical anomaly detection systems can detect unpredictable anomalies. They can monitor activities such as CPU (Central Processing Unit) usage, number of TCP (Transmission Control Protocol) connectors in term of statistical distribution but more time is required to identify attacks and detection accuracy is mainly based on the amount of collected behaviors.

Data mining based anomaly detection- Data mining techniques such as: classification, clustering and association rule mining can be used for failure detection. Data mining techniques use an analyzer which can differentiate normal and abnormal activity within clouds by defining some boundaries for valid activities in the cloud. A good number of approaches are proposed for this issue in [38]. Data mining anomaly detection techniques are largely used because they do not need any prior knowledge of the system

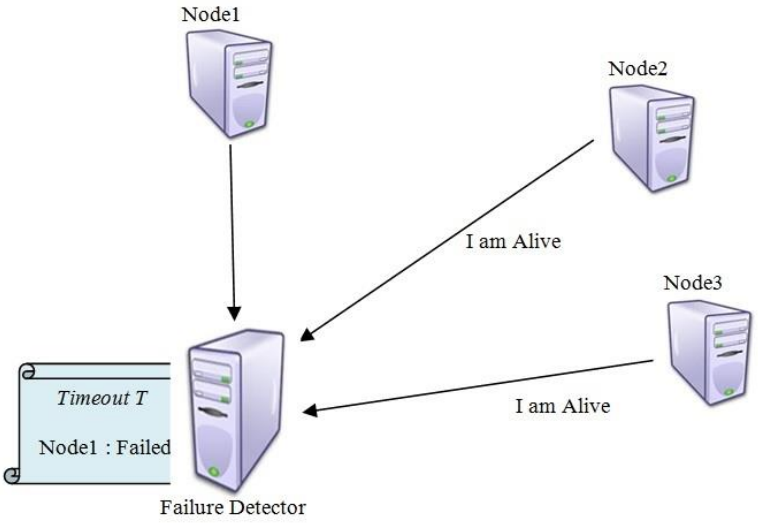
but their algorithms are generally computation-intensive. Moreover, data mining techniques can produce high false alarm rate (FAR) and they require more time and more sample training.

Machine learning based anomaly detection - The ability for programs or software to improve performance over time by learning is an important technique for the detection of anomaly. Verified values or normal behaviors of data are stored; when anomaly occurs or is being detected, the machine learns its behavior, stores the new sequence or rules. This technique creates a system that can improve performance of the program by learning from the prior results [39],[40], [41]. A survey on existing techniques based on machine learning is presented in [42]. Machine learning techniques alone can just detect known attacks. Therefore, they must be accompanied with statistical or data mining techniques in order to ensure detection of suspected unknown anomalies. We can see that each of the previous techniques has its strengths and weaknesses; the recent works for anomaly detection in cloud computing are focusing on development of more efficient hybrid techniques from the existing IDSs. Hybrid techniques are efficient for anomaly detection but they often come with high computational cost.

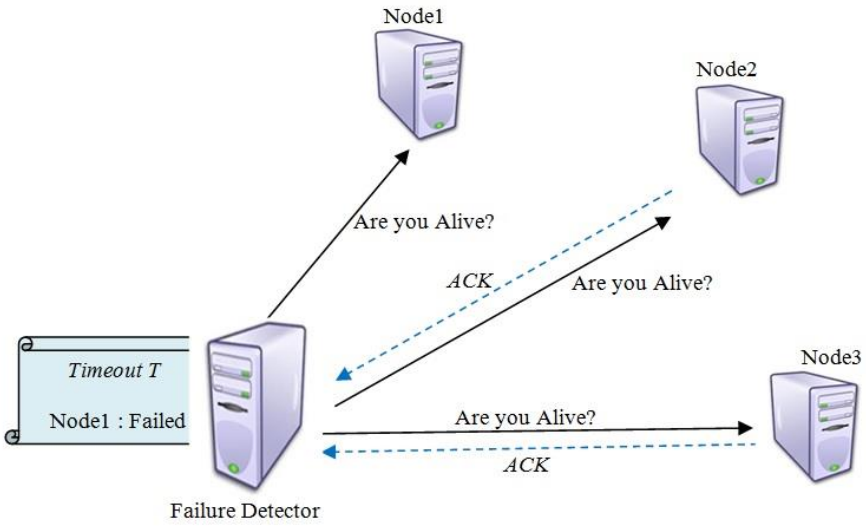
b. Heartbeat and Pinging Strategies

The most common implementation for fault detection in cloud computing systems is based on two *keep-alive massage* strategies: heartbeat and pinging [43]. In Heartbeat strategy, a message is periodically sent from a monitored node to the failure detector to inform that it is still alive. If the heartbeat does not arrive before a timeout, the failure detector suspects the node is faulty (see Figure 1.8 (a)).

In pinging strategy, a message is continuously sent from a failure detector to a monitored node. The failure detector expects to receive as answer an *ACK*. If a *keep-alive message* fails, a probe (i.e., a series of messages separated by a time interval) can be used to verify whether a node is really faulty (Figure 1.8 (b)).



(a)



(b)

Figure 1.8. a) Heartbeat strategy; b) Pinging strategy.

Heartbeat or pinging strategies are used for permanent hardware fault detection where the detection is focused on finding the crashed nodes. Furthermore, they are based on message passing which can produce an overflow in network connections.

In cloud computing systems, failure detection is done with the aid of intrusion detection and heartbeat/pinging strategies. Intrusion detection systems are dedicated to ensuring *safety* requirements by preventing any malicious attacks against the cloud connections or nodes. This strategy is based on monitoring the system behavior to detect any abnormal behavior produced by malicious attacks. The failure detection in this case is effected by an external monitor component which manipulates a set of data and applies

a sequence of computations to decide whether there is an anomaly or not. This type of process requires more time, and more computations. That is why, it cannot offer high accuracy for failure detection and this justifies the high false alarm rate (FAR) in IDSs. The second strategy used in cloud networks is heartbeat/pinging. It is useful for detecting the crashed nodes. Heartbeat strategy is based on message-passing between the failure detector and the set of monitored nodes. As noted earlier, this can lead to an overflow of the network connections. In both IDS and Heartbeat, fault confinement in the cloud network is not processed. This means that if one node fails, all of its neighbors can simply get infected and the failure would be transferred over the network. By this effect, the *safety* of the cloud network becomes a great concern.

1.3.3.2 Fault tolerance in cloud computing

The techniques that are used to create the fault Tolerance capability in cloud computing can be divided into two main categories: proactive fault tolerance and reactive fault tolerance [98][101][102][103] (see Figure 1.9).

a. Proactive Fault Tolerance

It is based on avoid failures by proactively taking preventative measures. It makes sure that the job gets done completely without any reconfiguration. Two techniques are based on proactive fault tolerance which are: Preemptive migration and software rejuvenation.

Software Rejuvenation, it designs the system for periodic reboots and it restarts the system with clean state with a fresh start.

Pre-emptive Migration, in this technique, the applications are constantly monitored, analyzed and depend on a feedback-loop control mechanism.

Self-Healing, for better performance, a big task can divided into parts. Running various instances of an application on various virtual machines can automatically handle failures of application instances.

b. Reactive Fault Tolerance

It aims to reduce the effect of the faults already occurred in cloud. Some of the fault tolerance policies are:

Checkpointing and rollback recovery, is useful for the long running and the big applications. It is doing after every change in the system. When the task fails, the job will be restarted from the recently checkpoint rather than restarting from the beginning.

Job Migration, in which the task can be migrated to another machine after failure detection. HAProxy can be used for migration of the jobs to another machine.

Replication, in order to make the execution succeed, various replicas of task are run on different resources. HAProxy, Hadoop and AmazonEc2 are used for implementing replication.

SGuard, is based on rollback recovery. It can be implemented in Hadoop and AmazonEc2.

Retry, *is* the simplest among all. In which the failed task is implemented again and again on the same resource.

Rescue Workflow, it allows the workflow to resist after failure of any task until it will not able to proceed without rectifying the fault.

Task Resubmission, at runtime, the failed task is resubmitted either to the same or to a different resource for execution.

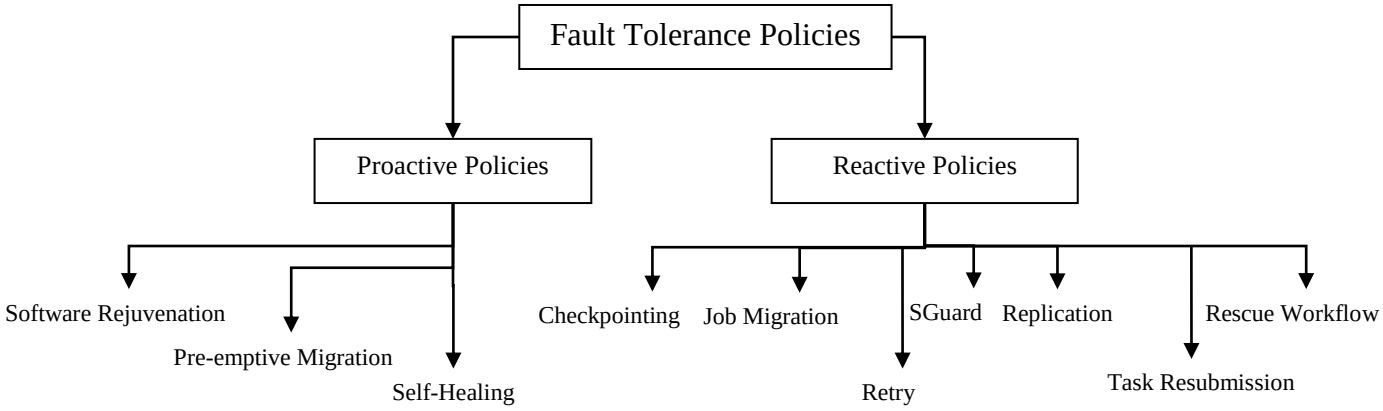


Figure 1.9. Fault tolerance techniques in Cloud computing

We can observe that fault tolerance techniques in cloud systems can be categorized under two main categories: Rollback recovery (or time redundancy) and physical redundancy (or space redundancy). The rollback recovery mechanism consists of the re-execution of the system from the last correct state (e.g., Checkpointing and rollback recovery) or even the restart of the system from the begin (e.g., SGuard, Retry, Software Rejuvenation). Space redundancy consists of the concurrent execution of many versions of the same program or the division of one program to many parts

executed concurrently on different machines (e.g., Replication, Self-Healing) or to migrate a process from a failed machine to an operational one (.g., Job Migration, Task Resubmission). Rollback recovery is very convenient for transient hardware fault tolerance in long applications. But it is not supportable by Real-time cloud applications because it needs more time for recovery. Furthermore, a consistent state must be calculated for each recovery and this is not easy to get especially in high scalable distributed cloud systems. Space redundancy can tolerate only permanent hardware crashes. It is very convenient for Real-time applications but it requires the implementation of complicated communication policies between the collaborative machines.

We can say that the existent strategies used for the fault tolerance in cloud computing have an observable missing in software fault tolerance. This latest can be ensured via software redundancy.

1.4 Conclusion

In this chapter, some basic concepts of fault tolerance are introduced such as: faults model, safety and liveness properties and fault tolerance techniques. Then, recovery blocks is presented as a forward recover fault tolerance scheme. After, the DRB scheme is described as a parallel execution of recovery blocks for software and hardware fault tolerance in real-time distributed systems. Then, Cloud computing systems are introduced in the next section. Its architectures and characteristics are highlighted. After that, reliability in cloud environment is discussed and the main fault detection and fault tolerance techniques are detailed.

Chapter Two

Related Works

Summary

2.1 Introduction

There are quite a good number of works on fault detection and fault tolerance in cloud computing systems either in component-based systems. Before wrapping up this thesis, we would like to mention a few of them. The mentioned researches are classified into three main classes: Fault detection in cloud computing, Fault tolerance in cloud computing and fault tolerance in component-based systems.

2.2 Fault detection in cloud computing systems

Many researches have been provided for fault detection in cloud Computing. Fan et al. in [46] use Petri Nets model to propose a fault detection strategy for cloud module by providing a cloud computing fault Net (CFN). The CFN aims to model different basic components of the cloud application as either the detection or failure process. By the CFN, byzantine fault detection can be done dynamically in the execution process. Wang et al. in [47] propose an online incremental clustering approach to recognize access behavior patterns and use CCA (Canonical-Correlation Analysis) to model the correlation between workloads and the metrics of application performance/resource utilization in a specific access behavior pattern. In [48], Barhuiya et al. introduce a lightweight anomaly detection tool (LADT) which monitors system-level and virtual machine level metrics in cloud data to detect node level anomalies using simple metrics and correlation analysis. In this work, LADT addresses the complexity of implementing efficient monitoring and analysis tools in large-scale cloud data centers by collecting and storing the metrics generated by node and virtual

machines using Apache Chukwa. T. Wang et al. present in [49] a correlation analysis based approach to detecting the performance anomaly for internet ware using kernel canonical correlation analysis (KCCA) to model the correlation between workloads and performance based on monitoring data. Furthermore, XmR control charts are used to detect anomalous correlation coefficient and trend without a prior knowledge. In [50], C. Wang et al. propose an algorithm that computes statistics on data based on multiple time dimensions using statistical methods. The proposed algorithms have low complexity and are scalable to process large amounts of data. The works in [47], [48], [49],[50] are based on statistical monitoring techniques which are based on observing the system behavior to detect any abnormal behavior. This process requires a prior knowledge which is extremely difficult in large scale systems.

Kumar et al. in [51] present a fault detection algorithm for faulty services using data mining's outlier detection method that can help to detect accurate and novel faulty services without any prior knowledge. In [52], Prasad and Krishna present statistical chart approach which is the standard algorithm applied to outlier detection for anomaly detection in continuous datasets. In [53], Ranjan and Sahoo present a new clustering approach based on K-medoids method for intrusion detection. The works in [51], [52], [53] are based on data mining system monitoring. These techniques present some hard computations and generate a high false alarm rate. In [54], Singh et al. propose a collaborative IDS framework in which known stealthy attacks are detected using signature matching and unknown attacks are detected using decision tree classifier and support vector machine (SMV)**.** In [55], Pandeeswari and Kumar introduce an hybrid algorithm which is a mixture of Fuzzy C-Means Clustering algorithm and Artificial Neural Network (FCM-ANN). In [56] Sha et al. propose a statistical learning framework by adopting both the high-order markov chain and multivariate time series. Ghanem et al. propose in [57] a hybrid approach for anomaly detection in large scale datasets using detectors generated based on multi-start meta heuristic method and genetic algorithm. The works in [54], [55], [56], [57] are hybrid system monitoring techniques which require high computational costs.

In [58], Arockiam and Francis present fault detection technique based on two strategies: *push* model and *pull* model. In push model, fault detector sends signals to various nodes in the cloud to check their health status. On the other hand, in pull model,

each component in the system sends signals to fault detector telling its health status. Some techniques based on heartbeat strategy are presented in [59], [60]. In [61], Hayashibara et al. presents the φ Accrual failure detector. It is based on heartbeat strategy but instead of providing information of boolean nature (Trust or Suspect); it produces a suspicious level on a continuous scale. By this, the applications can directly use the value output by the accrual failure detector as a parameter to their actions. These approaches are designed to adapt dynamically to their environment and in particular, adapt their behavior to changing network conditions. In [62], Lavinia et al. present a failure detection system that combines the power of existing approaches such as gossip protocol with the decoupling of monitoring and interpretation as offered by the accrual failure detection solutions. This combination gives a better estimation of the inter-arrival times of heartbeat and an increase level of confidence in the suspicion of process being lost. The works in [58], [59], [60], [61] and [62] are focalized only on hardware fault detection in the cloud computing nodes without detecting software faults. In the works [46-62] fault detection strategies in cloud computing are presented without considering the component-based architecture, unlike our proposition which is dedicated to fault detection in component-based cloud computing architecture.

2.3 Fault Tolerance in Cloud computing systems

In this section, some current researches of fault tolerance are presented. Ganesh et al. in [22] emphasizes fault tolerance by considering reactive and proactive fault tolerance policies. In proactive fault tolerance policy, preemptive migration and software rejuvenation techniques were discussed. Then, Checkpointing/Restart, replication and task resubmission were discussed in reactive fault tolerance. Zhang et al. in [114] proposed a novel approach called byzantine fault tolerant cloud (BFT-Cloud) for tolerating different types of failures in voluntary-resource clouds. BFT (Byzantine Fault Tolerant Cloud) can tolerate different types of failures including the malicious behaviors of nodes by making up a BFT group of one primary and 3f replicas. BFT clouds are used for building robust systems in voluntary-resource cloud environments. In [115], Jia et al. focus on the principle of fault correction by replacing the failed component by a functionally equivalent one. The authors proposed the fault correction by providing a light-weight fault handling for migration long-running application services into shared open cloud infrastructures. To minimize failure impact

on services and application executions, they presented a diagnosis architecture and a diagnosis method based on the service dependence graph (SDG) model and the service execution log for handling service faults. Therefore, by analyzing the dependence relations of activities in SDG model, the diagnosis method identifies the incorrect activities and explains the root causes for the web service composition faults, based on the differences between successful and failed executions of composite service. Choi et al. in [116], proposed a fault tolerance and a QoS (Quality of Service) scheduling using CAN (Content Adressable Network) in mobile social cloud computing by which, members of a social network share cloud service or data with other members without further authentication by using their mobile device. Fault tolerance and QoS scheduling consists of four sub-scheduling algorithms: malicious user filtering, cloud service delivery, QoS, provisioning replication and load balancing. Under the proposed scheduling, a mobile device is used as a resource for providing cloud services, faults caused from user mobility or other reasons are tolerated and user requirements for QoS are considered. By using fault tolerance and QoS scheduling, faults arising from mobile device are tolerated such as: network disconnection, battery drain. In [117], Jing et al. proposed matrix multiplication as a cloud selection strategy and technique to improve fault tolerance and reliability and prevent faulty and malicious clouds in cloud computing environment. Sun et al. in [118] presented a dynamic adaptive fault tolerance strategy DAFT. It is based on the idea of combining two fault tolerance models: a dynamic adaptive checkpointing fault tolerance model and a dynamic adaptive replication fault tolerance model in order to maximize the serviceability. In [119], Yi et al. proposed a fault tolerance job scheduling strategy for grid computing. The scheduling strategy includes JRT (Job Retry), JMG(Job Migration without Checkpointing) and JCP(Job Migration with Checkpointing). The authors concluded that JRT strategy has the most optimal system performance improvement for small jobs and JCP strategy leads to the lowest performance improvement. An adaptive fault tolerance of real-time applications (AFTRC) running on virtual machines in cloud environment is proposed by Malik and Huet in [120]. The AFTRC scheme tolerates the faults on the basis of reliability of each computing node. It is based on such modules like: Acceptance Test (AT), Time Checker (TC), Reliability assessor (RA), and Decision Mechanism (DM). Unfortunately, the acceptance test of the virtual machines

is not discussed. In [121], Wu et al. puts forward that resource consumption is also an important evaluation metric for any fault tolerant approach. The corresponding evaluation models based on mean execution time and resource consumption are constructed to evaluate any fault tolerant approach. In [121], an approach that aims to handling quite a complete set of failures arising in grid environment by integrating basic fault tolerant approaches is proposed. It is based on the four basic approaches: retry, alternate resource, checkpoint/ restart, and replication and it can dynamically and automatically decide which one is used by analyzing the current state of the running task. An evaluation model for mean execution time is constructed and used to evaluate fault tolerant approaches. A membership management solution over social graphs in the presence of byzantine nodes is proposed by Lim et al. in [122]. A novel software rejuvenation based fault tolerance scheme is proposed by Liu et al. in [135]. This scheme comes from two inherently related aspects. First, adaptive failure detection is proposed to predict which service components deserve foremost to be rejuvenated. Second, a component rejuvenation approach based on checkpoints with trace replay is proposed to guarantee the continuous running of cloud application systems. Gang et al. in [136] proposed a framework to provide load balancing and fault prevention in web servers in proactive manner to ensure scalability, reliability and availability. This framework is based on autonomic mirroring and load balancing of data in database servers using MySQL and master-master replication. Garraghan et al in [137] introduced a byzantine fault tolerance framework that leverages federated cloud infrastructure. An implementation of the proposed framework is discussed and detailed experiments are provided. Alannsary et al in [138], proposed a reliability analysis model that enables SaaS providers to measure, analyze and predict its reliability. Reliability prediction is provided by analyzing failures in conjunction with the workload. Mohammed et al. in [140] propose an infrastructure for IaaS cloud platforms by optimizing the success rate of virtual computing node or virtual machines. The main contribution is to develop an optimized fault tolerance approach where a model is designed to tolerate faults based on the reliability of each compute node and can be replaced if the performance is not optimal. Reddy et al. in [141], proposed an FT2R2Cloud as a fault tolerant solution using time-out and retransmission of requests for cloud applications. FT2R2Cloud measures the reliability of the software components

in terms of the number of responses and the throughput. The authors proposed an algorithm to rank software components based on their reliability calculated using a number of service outages and service invocation. Zheng et al. in [142], identified major problems when developing fault tolerance strategies and introduced the design of static and dynamic fault tolerance strategies. The authors identify significant components of complex service-oriented systems, and investigate algorithms for optimal fault tolerance strategy selection. An heuristic algorithm is proposed to efficiently solve the problem of selection of a fault tolerance strategy. Chen et al. in [143] presented a lightweight software fault tolerance system called SHelp, which can effectively recover programs from different types of software faults. As final work, Moghtadaeipour and Tavoli in [144] proposed a new approach to improve load balancing and fault tolerance using work-load distribution and virtual priority. We can see clearly that the current researches focus on improving the fault tolerance in cloud environments by improving the existent strategies or by collaboration of such strategies to develop one more efficient. Thus, the proposed works are restricted on hardware faults tolerance without dealing with software fault tolerance.

2.4 Fault tolerance in component-based systems

In this section, some researches dealing with fault tolerance in component-based systems are presented. The component-based analysis of fault tolerance was first studied by Arora and Kulkarni in [63], [65]. They proved that a fault tolerant program is a decomposition of a fault intolerant program and a set of fault tolerance components. A fault tolerant program satisfies safety and liveness properties. In [65], the authors proved that fault tolerance components are: Detectors and Correctors, where Detectors ensure safety property and Correctors ensure liveness property. The work in [65] was extended to the context of real-time systems in [66]. In [63], [64], [65], [66], a program is presented as a set of guarded commands in the shared memory model. Moreover, the Detector (resp. Corrector) component which ensures *safety* (resp. *liveness*) property is defined based on state predicate. State predicate means that properties or requirements verification is done on the state level. Unlike those works, in this thesis, an actual system is designed incrementally by composing smaller components. Each component has its own state space, behavior, interface, and each component is responsible for delivering a certain set of tasks. Roohitavaf and Kulkarni in [67] presented algorithms

for adding stabilization and fault tolerance in the presence of unchangeable environment actions. Bensalem et al. presented in [68] an heuristic method for compositional deadlock and verification of Component-based systems using the Invariant. This work focuses on just Deadlock detection. In [45], Bonakdarpour et al. introduced a theory of fault recovery for component-based models. A non-masking model was constructed from BIP models in order to ensure *liveness* property using *Corrector* component. But, the authors in [45] have not dealt with fault detection concerns. Wu et al. in [76] present a model-driven approach to describe specification and semi-automatic configuration of fault tolerance solutions for component-based systems on the software architecture level. In this work, the fault tolerance mechanisms are implemented by the system in the form of a specific kind of component named tolerance facilities. In [77], Tambe et al. present a model driven technique used to specify the special fault tolerance requirement for component-based systems. In [78], Jung and Kazanzides presented a run-time software environment for safety research on component-based medical robot systems. In both [77], [78], the mechanisms and services are designed to be middleware. Liu and Joseph in [79], [80] introduced a uniform framework for specifying, refining and transforming programs that provides fault tolerance and schedulability using the temporal logic of actions. In [81], a formal framework for the design of fault detection and identification components has been proposed where the framework is based on formal semantics provided by temporal epistemic logic. Temporal logic is a logical language for formal specification of requirements. Generally, temporal logics are used with model checkers for model verification (e.g., UPPAAL, KRONOS). Finally, Alko and Mattila in [64] have evaluated effectiveness of service oriented architecture approach to fault tolerance in mission critical real-time systems without dealing with component-based approach.

2.5 Conclusion

In this chapter, some current researches are highlighted. In the first part, some researches of fault detection in cloud systems are cited. They can be categorized under two main categories: fault detection using systems monitoring and fault tolerance using heartbeat/pinging strategies. In the second part, some current fault tolerance researches in cloud computing are mentioned. We can observe that the researches aimed to enhance the existent fault tolerance techniques by collaboration between more than

one technique either by reinforcing the existing techniques by novel opportunities. Finally some research on fault tolerance and fault recovery in component-based systems are cited.

Chapter Three
Component-based Cloud computing

Summary

3.1 Introduction

A cloud application is composed of a number of cloud modules [10]. Each cloud module has a virtual machine used to realize its function and each function is composed of a set of *tasks*. It is evident that the cloud computing architecture, its layers and its composition of components and services need to be designed as web service components [11] based on well proven component-based software engineering. component-based approach is a popular divide-and-conquer technique for designing and implementing large systems as well as for reasoning about their correctness. It stipule that a system is designed incrementally by composing smaller components, each responsible for delivering a certain set of tasks to separate different concerns. Thus, component-based design and analysis of fault tolerant systems is highly desirable in order to achieve systematic modularization of such system [45]. Here, a component represents an entity that provides a specific functionality. The components are expected to be scalable, fault tolerant, manageable, and autonomous [13]. Several tools are available for modeling heterogeneous embedded systems founded on component-based models. One of them is BIP (Behavior, Interaction, Priority) tool [14], [15], [16]. In this

thesis, we will use BIP as a Component-based framework. It has been used successfully in the field of robotics [17], [18], [19]. In BIP framework, a process is represented as a Transition Labeled System (TLS), where the principal components are: the atomic component and the composite component.

3.2 BIP framework for component-based design

BIP framework [7] is used for modeling heterogeneous real-time components which integrates results developed at *Verimag* Laboratory. It supports a component construction methodology based on the idea that components are obtained as superposition of three layers: the first one is the behavior layer which presents the internal behavior by a set of transitions and states. The intermediate layer includes a set of connectors describing the interactions between transitions of the behavior. The upper layer is a set of priority rules describing scheduling policies for interactions. Layering implies a clear separation between behavior and structure (i.e., connectors and priority rules). The principle components in BIP framework are: atomic component and composite component.

3.2.1 Atomic component

We define an atomic component as a Labeled Transition System (LTS) with a set of ports labeling individual transitions. These ports are used for communication between different components.

Definition1. An atomic component B is a labeled transition system represented by a tuple $(Q,\ P,\ \rightarrow,\ X,\ q^0)$ where:

Q**:** is a set of states $\{q^0,\ q^1, \dots,\ q^n\}$;

P**:** is a set of communication ports $\{p_0,\ p_1, \dots,\ p_n\}$; we can distinguish two types of ports: Complete or Incomplete.

Complete Port (Black Triangle): An interaction that contains a complete port is a complete interaction in the sense that complete port does not need to be synchronized with other ports to accomplish an interaction.

Incomplete Port (Black Circle): An incomplete port needs to be synchronized with other ports in order to achieve an interaction. Therefore, an interaction that contains an incomplete port is incomplete.

$\rightarrow : Q \times P \times G \times F \times Q$ is a set of transitions, each transition is a tuple of the form $(q^s,\ p,\ g,\ f,\ q^t)$ where:

q^s: is the state which is the transition source;

p: is the transition label and is the port associated to the transition;

g: is the transition guard which is a boolean condition on the set of variables X; the transition can be executed *iff* its guard g is true and some interactions including the port p are offered;

f: is an internal action on the set of variables X, the function f is executed when the transition t is enabled and we write $t(f)$. In an atomic component, variables are treated and modified by component internal functions;

q^t: is the state which is the transition target;

X**:** is a set of variables $\{x_i\}$which are manipulated by the internal functions, f_i;

q^0**:** is the initial state of the atomic component.

If we have a variable x which has an initial value v and we write $: x(v) - t(f) \rightarrow x_i(v')$, by which we mean that: there exists a transition t which contains an internal function f such that after the achievement of the transition t, the function f will modify the variable x from the value v to the new value v'.

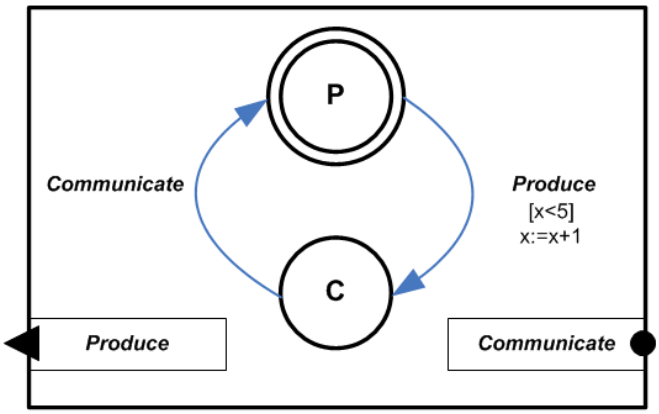


Figure 3.1. A BIP atomic component (Producer).

The Figure 3.1. shows an atomic component *(Producer)*, where:

$$Q = \{P,\ C\}$$

$$P = \{\ Produce,\ Communicate\};$$

$$\rightarrow = \{< P,\ Produce,\ [x < 5],\ x := x + 1,\ C >, < C,\ Communicate,\ [True],\ P >\};$$

$$X = \{x\},$$

$q^0 = P$

An execution cycle of an atomic component $B = (Q, P, \rightarrow, X, q^0)$ is $Cy_B = t_1 t_2 \ldots t_n$ such that t_1 is the first transition in *B* and t_n is the last one. An *execution cycle* of an atomic component is the execution of all of its inner transitions for *one* time.

The behavior of a system as defined in [134] is what the system does to implement its function and is described by a sequence of states that can be: Computation, Communication or stored information.

A *Behavior of an atomic component* $B = (Q, P, \rightarrow, X, q^0)$ is $Beh(B) = f_1 f_2 \ldots f_n$ and for all *i*:

- $f_i \in F$(i.e.,*F* is the set of internal functions in *B*) and

- There exists a transition sequence $t_1 t_2 \ldots t_n$ and a state sequence $q_1 q_2$ ... such that:

$q_0 - t_1(f_1) \rightarrow q_1 - t_2(f_2) \rightarrow q_2 - \ldots\ldots\ldots t_n(f_n) \rightarrow q_n$

The atomic component behavior has a direct effect on the set of variables *X*. If the initial value of the set X is v_0, it will be v_n after the achievement of the atomic component behavior: $X(v_0) - Beh(B) \rightarrow X(v_n) = X(v_0) - f_1 \rightarrow X(v_1) - f_2 \rightarrow X(v_2) - f_3 \rightarrow \cdots X(v_n)$. *X* is the set of variables and $v_1, v_2, \ldots v_n$ are the values of the set *X*.

Hence, the behavior of B in one execution cycle is:

$Beh_{Cy}(B) = q_0 - t_1(f_1) \rightarrow q_1 - t_2(f_2) \rightarrow q_2 - \ldots\ldots\ldots \rightarrow t_n(f_n) \rightarrow q_0$.

This means that $Beh_{Cy}(B)$ produces final results after achievement of one execution of the entire internal functions of the atomic component B.

3.2.2 Composite component

The composite component is constructed from a set of interacted atomic components. It represents the cloud computing system which is composed of interacted cloud nodes.

Definition 2. A composite component $B = \gamma(B1 \ldots Bn)$ is defined by a composition operator parameterized by a set of interactions. It is a transition system$(Q, \gamma, \rightarrow, X, q^0)$, where different mathematical notations carry the meanings as shown in Table 3.1.

Table 3.1.Some mathematical notations and their meanings.

$Q = \otimes_{i=1}^{n} Qi$	The set of global sates is obtained by the cartesian product of all the atomic components' states in the composite component.
$q^0 = (q_1^0, \dots q_n^0)$;	The set of all the atomic components' initial states.
$X = \bigcup_{i=1}^{n} X_i$;	The union of the atomic components' variables sets.
$\rightarrow$: is the least set of transitions satisfying the rule [45]: $\frac{a = \{p_i\}_{i\in 1..m} \forall i \in I: q_i \rightarrow q_i' \forall}{\rightarrow (q_1' \dots q_n')}$	- As mentioned in [45], a composite component $B = \gamma(B_1 \dots B_m)$ can execute an interaction$a \in \gamma$,*iff* for each port $p_i \in a$, the corresponding atomic component $B_i$can execute a transition labeled with p_i - the states of the components that do not participate in the interaction stay unchanged.
$\gamma = \cup_{i=1}^{l} \beta_i$	The set of connectors which rely on the atomic components.

3.2.3 Connector

A connector $\beta_i = \{p_i\}_{i\in 1..m}$is a set of ports of the atomic components involved in β_i. It represents the network connection in the cloud system. We assume that a connector contains at most one port from each atomic component. The Interaction of a connector is any non-empty subset of this set. As defined in [45], for a given system built from a set of n atomic components $B_i = \{(Q_i, P_i, \rightarrow, X, q^0)\}_{i=1}^{n}$, we assume that their respective sets of ports are pairwise disjoint, (i.e., for any two $i \neq j$ from $\{1..n\}$, we have $P_i \cap P_j = \emptyset$). We can therefore define the set $P = \cup_{i=1}^{n} P_i$ of all ports in the system. An interaction is a set $a \subseteq P$ of ports. When we write $a = \{p_i\}_{i\in I}$, where $I \subseteq \{1..m\}$. An interaction can be a *rendezvous or* a *broadcast* interaction.

3.2.3.1 Rendezvous Connector

Or strong synchronization enables an exchange of information between the nodes. In this type of interaction, all the ports are synchronous (see Figure 3.2). The initiative meaning of the *synchronous* is that it has to wait for other ports in order to execute the interaction. The connector $\beta_1 = \{p_{i=1..3}, p_i \subset B_{i=1..3}\}$ defines only one

interaction: $a = B_1B_2B_3$ in which, all the atomic components must synchronize at the same time in order to achieve the interaction a.

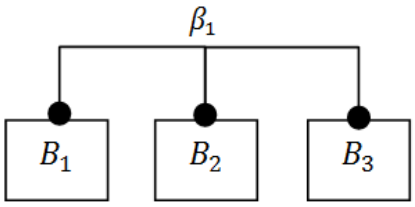


Figure 3.2.Rendezvous interaction.

3.2.3.2 Broadcast Connector

Or weak synchronization is used to update information stored at the nodes. It includes one trigger (i.e., initiator) port in B_1 and two synchronous ports. The intuitive meaning of trigger is that it can initiate the interaction, even if all other ports are not enabled.

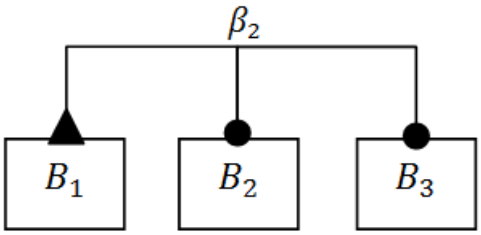


Figure 3.3. Broadcast interaction.

The connector (in Figure3.3) $\beta_2 = \{p_{i=1..3},\ p_i \subset B_{i=1..3}\ , B_1\ is\ the\ Broadcat\ initiator\ \}$ describes the set of all interactions that contains at least B_1, which are:$a_1 = \{B_1\}$, $a_2 = \{B_1B_2\}$, $a_3 = \{B_1B_3\}$,$a_4 = \{B_1B_2B_3\}$. We can see that all the possible interactions contain the initiator $B_1$and the maximum one contains all the atomic components: B_1,B_2 and B_3.

3.3 Recapitulations

In the next chapters, the cloud system will be considered as a complex system which is composed of a set of atomic components (i.e., nodes) supported by network connections. The atomic component is the simpler component; it reflects the cloud module and the atomic component transitions reflect the cloud module tasks, where the composite component represents the cloud computing system that is composed of a set

of interacted cloud modules. The set of component-based concepts used in this thesis and their equivalents in cloud computing system are presented in the Table 3.2.

Table 3.2. Component-based concepts and their equivalents in Cloud system.

Cloud system concepts	**Component-based approach concepts**
Cloud module / node	Atomic component
Module task	Atomic transition
Cloud system	Composite component
Network connections	Connectors
Primary block	Primary behavior
Alternate block	Alternate behavior

3.4 Conclusion

In this chapter, the component-based approach for cloud systems is introduced. Then, the main concepts of BIP as a framework for component-based design such as: the atomic component, the composite component and connectors are described. Finally, a recapitulation of the used terms in this thesis is given to facilitate the comprehension of the rest of chapters.

Chapter Four

Fault Detection in Component-based Cloud computing

Summary

4.1 Introduction

Fault Detection is considered as one of the main challenges in large-scale dynamic environments and thus, for maintaining the reliability requirements of cloud systems. Most of the popular existing techniques for fault detection applied on the cloud computing environment in general, are based on system-monitoring despite the extreme

difficulty of keeping track of all machines with their huge number in cloud systems. In this chapter, we propose a fault detection framework for the component-based cloud computing by using *Recovery Blocks*' acceptance test. This framework aims to construct *Fail-Silent* cloud modules which have the ability of self-fault detection. In this, the detection process of transient hardware faults, software faults, and response-time failures is performed locally on each computing machine in the cloud system. We assume that there is no permanent crash in the cloud nodes and the acceptance test is reliable and cannot be altered. Each cloud node has one predefined function and the software developer can set the acceptance test of each cloud node on the system.

4.2 Acceptance test for fault detection

Critical systems are usually related to human life, thus ensuring safety property is very important in order to avoid catastrophic consequences caused by failures. Final results of a critical system must be validated in order to judge their correctness. This validation can be offered by the acceptance test. An acceptance test AT of a component B is a boolean expression on the set of variables, X. It is used to validate final results' correctness. The acceptance test ensures that the final results are acceptable but not always they may be the desired results (i.e., some results may not be desired).Thus, it ensures the continuity of service offered in spite of degradation in the system quality, just to be safe from any disaster.

4.2.1 Fault detection in atomic component

An atomic component $B = (Q, P, \rightarrow, X, q^0)$ produces results after each execution cycle. The results could be correct or not correct. Without a mechanism of fault detection, we cannot judge the correctness of final results. Therefore, an atomic component must have an acceptance test which is a boolean expression on the set of variables X of the atomic component.

Definition 1. An acceptance test $AT_B(X)$ of the atomic component $B = (Q, P, \rightarrow, X, q^0)$ is a boolean expression on the set of variables, X. The acceptance test validates the correctness of B's final results and ensures that they do not lead to disastrous consequence even if they are not the expected results.

After one execution cycle, $X(v) - Beh_{Cy}(B) \rightarrow X(v')$, the set of variables X will be modified by $Beh_{Cy}(B)$ from the value v to the new value v'. Final results v'will be checked by $AT_B\ (X)$ and three cases are possible here. Final results may be:

- The Correct results c, which satisfy the acceptance test and which are considered as the desired results.
- The Acceptable results *a,* which satisfy the acceptance test but they are not the desired results and they do not lead to disaster for the system.
- The Faulty results f, that do not satisfy the acceptance test. These kinds of results can incur huge damages to the system.

Basing on these latest cases, the AT judges the behavior of the atomic component B and decides its correctness.Now, again two cases are possible for B:

- *If* the final results validate the acceptance test (i.e.,$AT_B\ (v')$ = *True*) then, *B has a correct or acceptable behavior (Fault-Free Behavior)*and it earns execution.
- *If* the final results do not validate the acceptance test (i.e., $AT_B\ (v')$ = *False*) then, *B has a failed behavior* and it must be stopped immediately to go through recovery and fault correction.

We mean by these two cases that: *if* $Beh_{Cy}(B) \vDash \mathrm{AT_B} \Rightarrow$ *B is correct else B is failed.*

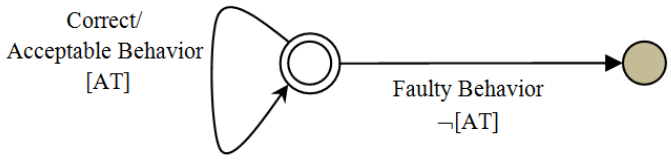


Figure 4.1. Fault detection in atomic component using the acceptance test.

Figure 4.1 shows the fault detection using the acceptance test. The atomic component *B* operates and validates its final results after each execution cycle. If *B*'s final results satisfy the AT, *B* has a correct or acceptable behavior (*i.e., in left state*). *B* stands at that state till detection of failure by $AT_B\ (X)$. At that moment, the atomic component *B* will be considered as failed and *B* will pass to an unstable state (i.e., the *right state*). At that state, the component will be blocked till recovery. An atomic component that has the ability of self-fault detection using an acceptance test is *Fail-Silent atomic component (FS$_B$).* A *Fail-Silent* atomic component satisfies the ω-regular expression:$(c/a\ )^*f$. The atomic component has a correct behavior (c) or an acceptable behavior (a). At just

detection of a failure by the AT, the atomic component will be considered as failed *(f)* and it will be *blocked* immediately attending the correction. The last correct state of the atomic component B will be saved in the history state $Hs(B)$. A *Fail-Silent* atomic component operates without failures and returns a correct or an accepted result; otherwise, it will be blocked immediately.

Proposition: A history state Hs is used in the atomic component in order to save the last correct variable values of the atomic component and the last received messages from the other atomic components. Hs=<$X(v)$, $(msg_1, msg_2, \dots)$ >. The history state is indispensable for recovery phase.

Definition 2. A *Fail-Silent* atomic component, $FS_B = (Q,\ P,\ \rightarrow,\ X,\ q^0,\ Hs,\ AT_B)$is a component which can validate its final results and judge its correctness by the acceptance test AT_B. The ω-regular expression of a Fail-Silent atomic component is $[(c/a)^* f]$.

Algorithm of Fail-Silent atomic component:

Fail-SilentB: Execute$(Beh_{Cy}(B))$

If $(AT_B(X))$ *then*

Update $Hs(B)$

Go to Fail-SilentB

Else

Deadlock

EndIf

End Fail-SilentB

Theorem1: A Fail-Silent atomic component, $FS_B = (Q,\ P,\ \rightarrow,\ X,\ q^0,\ Hs,\ AT_B)$can insure safety property using the cceptance Test AT_B. The AT can validate final results and decide their correctness. In the case of fault detection, the atomic component FS_B will be passed to a Deadlock state till failure correction.

4.2.2 Fault detection in composite component

A composite component $B = \gamma(B_1 \dots B_n)$ is a set of atomic components $B_{i=1..n}$ glued by the set of connectors $\gamma = \{\beta_{i=1..l}\}$. As seen in chapter 3 - section 3.2.3, a connector in a composite component can be rendezvous or broadcast connector.

4.2.2.1 Rendezvous connection

If we have the Rendezvous connector β such that $\beta = \{p_{i=1..m}, p_i \subset B_i\}$, the only possible interaction is $a_i = B_1 B_2 B_3 \dots B_m$ which contains all the atomic components involved in the connector β. Therefore, the failure of one atomic component will directly infect the others atomic components in the same rendezvous interaction. This means that $\forall B_{1 \le i \le m} \in \beta;\ if\ (B_i)$ is failed, then $\forall B_{j \neq i} and B_j \in \beta$, B_j will fail too. Thus, to construct a Fail-Silent rendezvous connector, all its inner atomic components must be Fail-Silent as well. Therefore, $FS_{Rendezvous(\beta)} = \{FS_{B1}, FS_{B2}, \dots, FS_{Bm}\}$.

Lemma1. A rendezvous connector, $\beta = \{p_{i=1..m}, p_i \subset B_i\}$ which involves a set of Fail-Silent atomic component is Fail-Silent rendezvous connector: $FS_{Rendezvous(\beta)} = \{p_{i=1..m}, p_i \subset FS_{Bi=1..m}\}$.

4.2.2.2 Broadcast connection

If we have the broadcast connector :

$$\beta = \{p_{i=1..m},\ p_i \subset B_{i \neq k}\ \ and\ B_k\ is\ Broadcast\ initiator\}.$$

The possible set of interactions in this case are those containing at least one instance of B_k. The minimum interaction is $a_1 = \{B_k\}$ which contains only the broadcast initiator and the maximum interaction is $a_n = \{B_k B_2 B_3 \dots B_m\}$ which contains all the atomic components involved in the connector β. We can see that if the atomic component B_k fails and enters in a deadlock state, the others atomic component involved in the same broadcast connector will be blocked too. But, if $B_2\ or\ B_3$ fails and blocked, it does not affect the broadcast initiator B_k . Thus, to construct a Fail-Silent connector γ, at least the broadcast initiator B_k must be Fail-Silent. Therefore, $FS_{Broadcast(\beta)} = \{FS_{Bk}, B2, \dots, B_m\}$.

Lemma2.A broadcast connector $\beta = \{p_{i=1..m},\ p_i \subset B_i\ and\ B_k\ is\ \ Broadcast\ initiator\}$ which involve at least a Fail-Silent broadcast initiator is a Fail-Silent broadcast connector: $FS_{Broadcast(\beta)} = \{FS_{Bk}, B2, \dots, B_m\}$..

Lemma 3. A composite component which contains Fail-Silent connectors (rendezvous and/or broadcast) is Fail-Silent composite component. The $\omega - regular$ expression of a *Fail-Silent* composite component is: $[(C\ /\ A/\ F)^* F]$.

Theorem 2. A composite component which is composed of a set of Fail-Silent atomic component is Fail-Silent composite component: $FS_B = \gamma(FS_{B1},\ FS_{B2}, \dots,\ FS_{Bn})$.

4.3 Construction of Fail-Silent models

4.3.1 Construction of Fail-Silent atomic Component

Now, let us see how we could construct a Fail-Silent atomic component from an initial model that is not Fail-Silent. Let $B_1 = (Q,\ P,\ \rightarrow,\ X,\ q^0)$ be an atomic component. In order to construct a Fail-Silent atomic component, we must add the acceptance test. This test validates B_1's final results. The Fail-Silent B_1 is $FS_{B1} = (Q',\ P', \rightarrow', X,\ I,\ Hs(B_1),\ AT_{B1})$ such that:

$Q' = Q \cup \{Q^I,\ Q^T\}$;$Q^I$and $Q^T$are two new states. Q^I is the initial state.

$P' = P \cup \{Start_{B1}\ ,\ Test_{B1}\}$, $Start_{B1}\ and\ Test_{B1}$ are two new ports where, $Start_{B1}$ is the first port in the Fail-Silent atomic component and $Test_{B1}$ is the last one.

The first transition $Start_{B1}$ leaves the initial state Q^I to the state q^0, where, the transition $Test_{B1}$ achieves from the state Q^T to the state Q^I.

$\rightarrow' = \rightarrow \cup \{< Q^I,\ Start_{B1},\ Update\ (Hs(B_1)),\ q^0 >,\ < Q^T,\ Test_{B1},\ [AT_{B1}], Q^I >$.

The set of transitions will be enriched by two transitions associated with the ports $Start_{B1}\ and\ Test_{B1}$. The transition$< Q^I,\ Start_{B1},\ Update\ \big(Hs(B_1)\big),\ q^0 >$is the first transition which leaves the initial state Q^I to the state q^0,its internal function is $Update\ (Hs)$ which updates the history state $Hs(B_1)$ by the last correct variable values and the last received messages. The second new transition is$< Q^T,\ Test_{B1}, [AT_{B1}],\ Q^I >$. It is the test transition in the component FS_B– it aims to test and validate the final results of *X* by the guard $[AT_B]$ which is the expression of the acceptance test. The transition $Test_B$ leaves from the state Q^T to the initial state Q^I. This transition is triggered *iff* the acceptance test is satisfied *(*$AT_B(X) = True$*)*; else, the *Fail-Silent* atomic component will be blocked on the state Q^T.

4.3.2 Construction of Fail-Silent composite component

As seen in the section 4.2.2, in order to construct a Fail-Silent composite component for $B=\gamma(B_1, B_2, \dots, B_n)$which contains a set of atomic components glued by a set of connectors γ. All its inner connectors, rendezvous and/or broadcast must be Fail-Silent and so, all its inner atomic components must be Fail-Silent as well.

Therefore, if we have a composite component, $B=\gamma(B_1, B_2, \dots, B_n)$, the Fail-Silent composite component is $FS_B = \gamma(FS_{B1}, FS_{B2}, \dots, FS_{Bn})$.

In the next section, we will apply our approach on the Producer-FIFO-Consumer model.

The Figure 4.2 presents a Producer-FIFO-Consumer (PFC) model. This model is composed of three atomic components: Producer, FIFO, and Consumer. $PFC = \gamma(Producer,\ FIFO,\ Consumer)$. Here, γ is the set of connectors:$\gamma = \{\beta_1,\ \beta_2\}$. β_1 and β_2 are rendezvous connectors.

$\beta_1 = \{a_1 = \ (Producer.Communicate;\ FIFO.Write)\}$.

$\beta_2 = \{a_1 = (FIFO.Read;\ Consumer.Communicate)\}$.

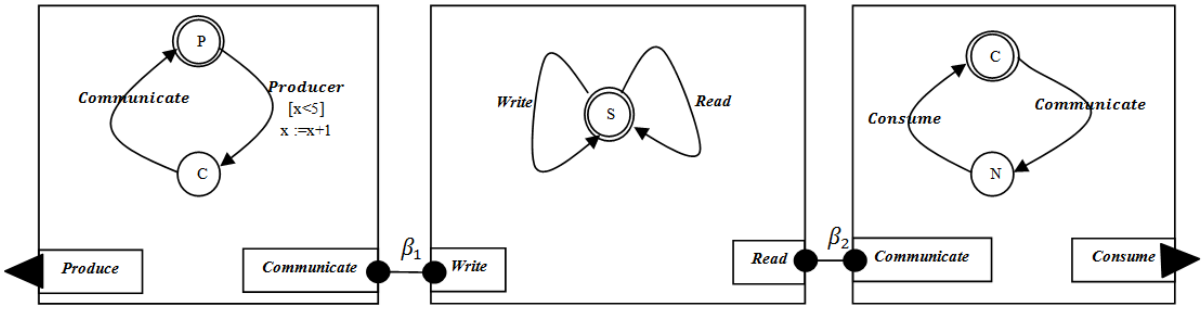


Figure 4.2. PFC composite component model.

In order to construct a Fail-Silent PFC model, we must first construct its inner Fail-Silent atomic component. Therefore, we should construct the Fail-Silent Producer, the Fail-Silent FIFO and Fail-Silent Consumer.

Construction of Fail-Silent producer:

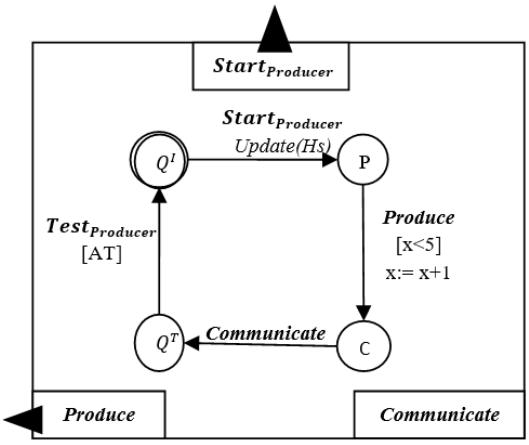


Figure 4.3.Fail-Silent producer**.**

Figure 4.3 shows the Fail-Silent, Producer. It is defined as:

$FS_{Producer} = (Q', P', \rightarrow', X,\ I,\ Hs,\ AT_{Producer})$where:$Q' = \ Q \cup \{Q^I,\ Q^T\} = \{Q^I,\ P,\ C,\ Q^T\}$

$$P' = P \cup \{Start_{Producer}, Test_{Producer}\}$$
$$= \{Start_{Producer}, Produce, Communicate, Test_{Producer}\}$$
$$\rightarrow' = \rightarrow \cup \{< Q^I, Start_{Producer}, [True], Update(Hs), P >,$$
$$< Q^T, Test_{Producer}, [AT_{Producer}], Q^I >\} =$$
$$\{< Q^I, Start_{Producer}, [True], Update(Hs), P >, < P, Produce, [x < 5], x = x + 1, C$$
$$>, < C, Communicate, [True], Q^T >,$$
$$< Q^T, Test_{Producer}, [AT_{Producer}], Q^I >\}$$

At fault free execution, $FS_{Producer}$ performs one execution cycle and before updating the history state Hs with the new values of X, $FS_{Producer}$ first validates the acceptance test on the transition labeled $Test_{Producer}$. If the guard $[AT_{Producer}]$ is true, the results are acceptable and the next execution cycle begins with the transition $Start_{Producer}$. On which, the internal function ($Update(Hs)$) will ensure saving of the last correct variable values on the history state *(Hs)*. If the component $FS_{Producer}$reaches the state Q^T and the variable values do not satisfy the guard $[AT_{Producer}]$, at that moment, the Fail-Silent atomic component $[AT_{Producer}]$ will be blocked on the state Q^T attending the recovery phase. Finally, we can see that we have constructed a Fail-Silent atomic component $FS_{Producer}$ which can insure the safety property using the acceptance test and which respect the $\omega - regular$ expression$[(c/a)^* f]$. In the same manner, we will construct the *Fail-Silent* FIFO (see Figure 4.4) and the *Fail-Silent* Consumer (see Figure 4.5).

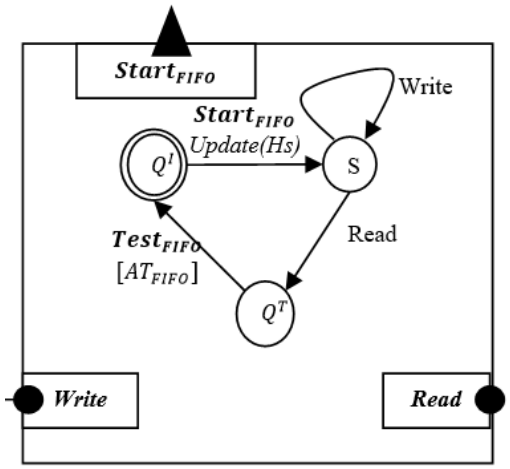


Figure 4.4.Fail-Silent FIFO

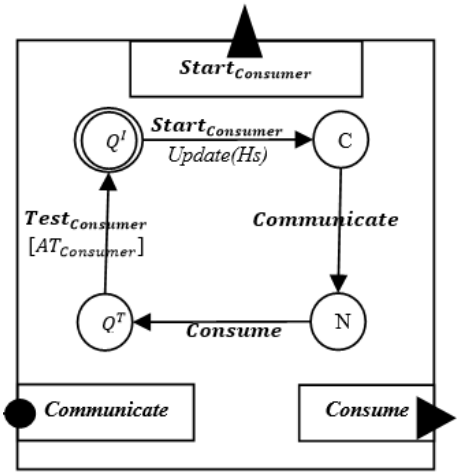


Figure 4.5. Fail-Silent Consumer

After constructing the Fail-Silent atomic components, we will have the Fail-Silent composite component PFC (see Figure 4.5).

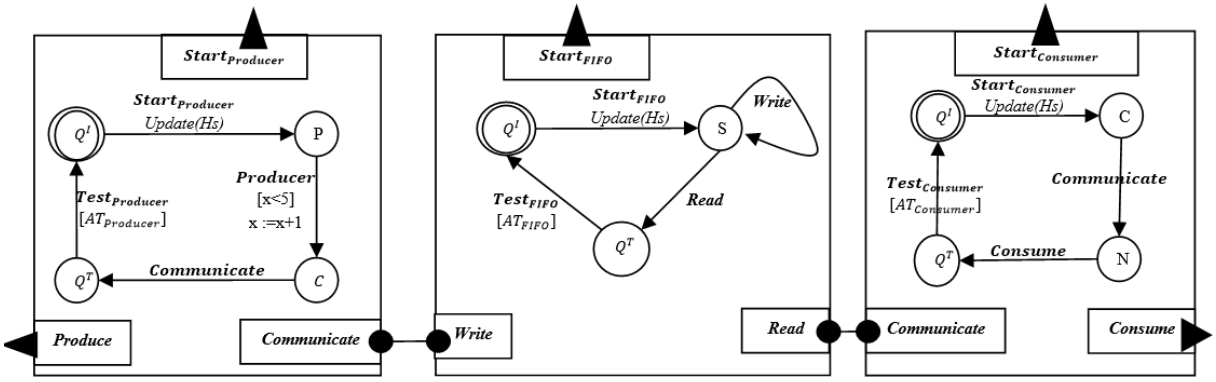


Figure 4.6.Fail-Silent composite component PFC.

The Fail-Silent composite component PFC in the Figure 4.6 is composed of a set of the Fail-Silent atomic components. If we suppose that a failure occurs in the Fail-Silent Producer, then it will be blocked on the sate Q^T because its results do not satisfy the $Test_{Producer}$ guard. At the same time, both FIFO and Consumer are in correct operation. But, in a future moment, the transition "*Write*" of FIFO component will need to synchronize with the component, Producer. This latter is in deadlock state and therefore, the components, FIFO and Consumer will be blocked too. We can see that the failure of one component in the composite component PFC brings the deadlock of all the components which are involved in direct or indirect interaction with the failed component. By this way, we have not only stopped the failed component but also we

have stopped the fault confinement in the composite component. After this fault detection phase, recovery and fault tolerance must be set.

4.4 A CASE STUDY

4.4.1 Fire Control System

Let us explain our approach with a mobile cloud system. Let us consider a fire control system which monitors the temperatures in the forest in order to prevent fires. In our system, we have three main components*:* *Sensor node*, *Cloud node1 and Cloud node2*. In this system, the mobile sensor frequently takes measures of the forest *temperatures* and sends those data to the Cloud node1 (which receives the temperature measures and calculates their *average*).The average temperature would be sent to the Cloud node2 which produces a *status report* which would be transferred to the system control (see Figure 4.7). We have used BIP model to design the fire control system as shown in Figure 4.8.

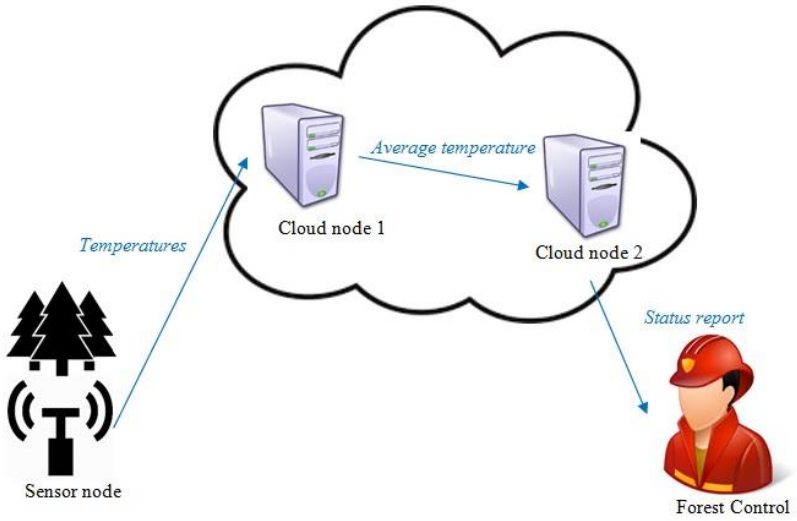


Figure 4.7.Fire control system.

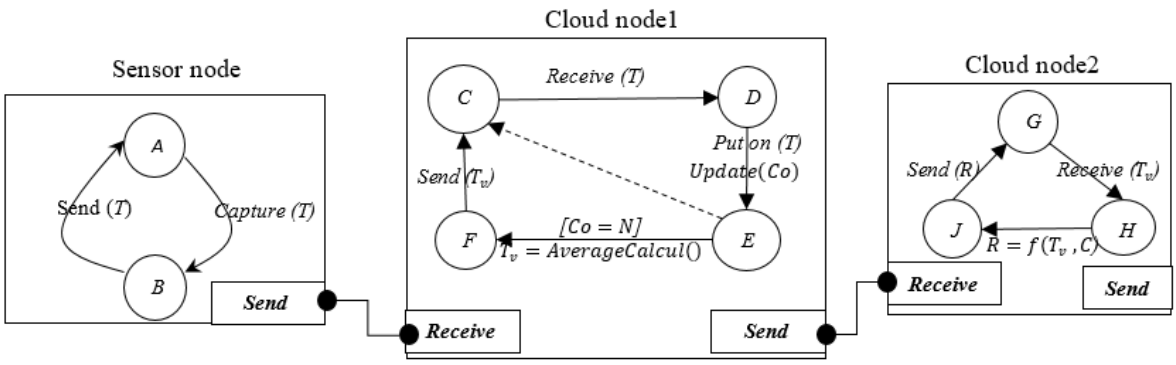


Figure 4.8.Fire Control system BIP model.

The sensor node periodically takes temperature measures $T \in [T_{Min},\ T_{Max}]$such that T_{Min} and T_{Max} are defined according to the area climate conditions and to the sensor node capacities. The difference between two successive temperatures does not exceed α: $|T - T_{Prev}| \leq \alpha$.

The Cloud node1 receives the temperatures T from the sensor node in the system and calculates the average $T_v$of *n* different temperatures. Then, it sends the average $\boldsymbol{T_v}$ to the Cloud node 2. The average temperature $\boldsymbol{T_v}$ must be between the highest received temperature $\boldsymbol{T_h}$ and the lowest one $\boldsymbol{T_l}$*(i.e.,* $\boldsymbol{T_l \leq T_v \leq T_h}$).

The Cloud node 2 receives the average temperature T_v from the Cloud node1. According to the set of conditions $\boldsymbol{C}$ and the average temperature $\boldsymbol{T_v}$**,** the Cloud node2 produces a status report about the forest $R = f(T_v, C)$. The values that must be defined by the software developer are summarized in the Table 4.1.

Table 4.1. The values defined by the system developer.

Notation	**Meaning**
T_{Max}	The highest temperature that can be detected
T_{Min}	The lowest temperature that can be detected
$\alpha\ (alfa)$	The difference between two successive temperatures
N	Required number of temperatures for average calculation
$sensor_{TimeOut}$	Sensor Time-Out
$Node1_{TimeOut}$	Cloud node1 Time-Out
$Node2_{TimeOut}$	Cloud node2 Time-Out
C	The predefined conditions for the Cloud node2 decision

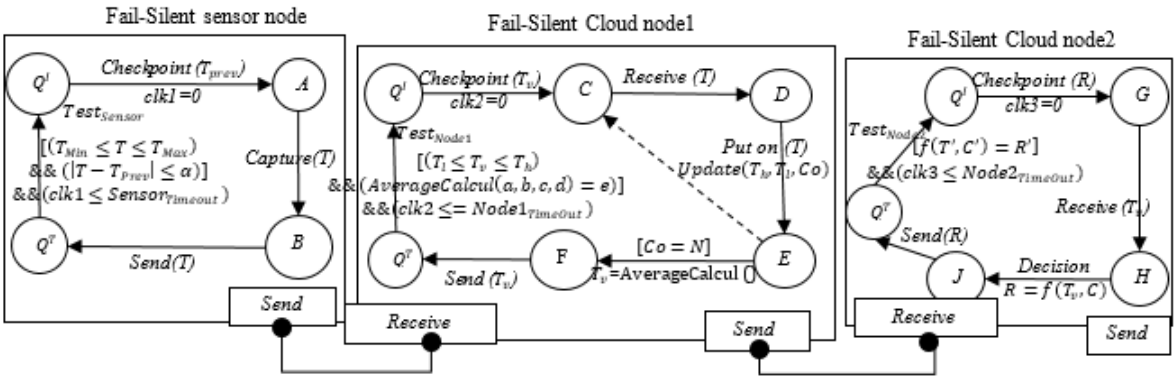


Figure 4.9. Fail-Silent Fire Control system

4.4.2 Construction of the Fail-Silent Fire Control System

In order to construct Fail-Silent system, we will use the acceptance test approach. First, we have to construct the Fail-Silent components. In the next section, we describe each component:

The Sensor node: its main function is measuring the temperature. Therefore, to ensure that the component is correct, we have to validate its behavior using an acceptance test. The system developer has previous knowledge about the sensor characteristics and the area climate where the sensor is deployed. Therefore, according to this information, he can define an adequate acceptance test. In our example, we have supposed that one of the main characteristics of the sensor is that it can detect only temperatures between T_{Min} and $T_{Max}(i.e.,\ T_{Min} \leq T \leq T_{Max})$. Besides, the difference between two successive temperatures would not exceed α (i.e., $|T - T_{Prev}| \leq \alpha$) which is a threshold used to detect whether the sensor gives a random temperature reading. Furthermore, the sensor has to send the temperature to the Cloud node1 before expiration of its Time-Out. The sensor Time-Out is defined by the system developer. A Clock *clk1* is used for calculating the passage of time in the sensor. Finally, we can have the sensor node acceptance test : $AT_{Sensor} = [(T_{Min} \leq T \leq T_{Max})\&\&(|T - T_{Prev}| \leq \alpha)]\&\&\ (clk1 \leq sensor_{TimeOut})$. To say that the sensor operates correctly, it must validate the logical expression of the acceptance test. Therefore, AT_{Sensor} will take place as $Test_{Sensor}$ transition guard (Figure 4.9). If the component validates the AT_{Sensor}, then the temperature will be saved in T_{prev}on the next checkpoint and the Clock *clk1* will be initialized for the next execution cycle. Else, the sensor will be considered as failed and

will be blocked on the state Q^T. The failed sensor can be replaced by an operational one which can get temperature measure from the last correct temperature T_{prev}.

The Cloud node 1: The main function of this component is to calculate the average of received *n* temperatures. Therefore, the average temperature will be included between the highest temperatures T_h and the lowest one $T_l (i.e., \quad T_l \leq T \leq T_h)$. Also, we have to ensure that the Cloud node1 has the ability of correct average calculation. For that, we can test the component by calculating: $AverageCalcul(a, b, c, d) = e$ where $a,\ b,\ c,\ d$ are predefined random values. Furthermore, the Cloud node1 have to calculate and send the average without exceeding its Time-Out (i.e., defined by the system developer). A Clock *clk2* is used to calculate time. Therefore, the acceptance test for the Cloud node1 is: $AT_{Node1} = [(T_l \leq T_v \leq T_h) \&\& (AverageCalcul(a, b, c, d) = e) \&\& (clk2 \leq Node1_{TimeOut})]$.The AT_{Node1} is the guard of the transition $Test_{\text{Node1}}$ (Figure 4.9). If the node satisfies its acceptance test, T_v will be saved on the next checkpoint and the Clock *clk2* will be initialized. If the acceptance test is not satisfied, then the component is failed and it will be blocked on the state Q^T.

The Cloud node 2: The main function of this component is to produce a forest report state *R* according to the temperature average T_v received from the node1 and according to predefined conditions $C{:}R = f(T_v, C)$. Therefore, we have to ensure that the component is able to produce the correct report. For this aim, we can test the component using the same function f but with different data to see whether the component produces the predicted report or not. Furthermore, taking and sending of decision must be before the expiration of the Cloud node2 Time-Out. A Clock *clk3* is used to calculate the time in the Cloud node2. Finally, the acceptance test is: $AT_{Node2} = [(f(T', C') = R') \&\& (clk3 \leq Node2_{TimeOut}]$. The acceptance test AT_{Node2} is the guard of the transition $Test_{Node2}$. If the AT_{Node2} is satisfied, a checkpoint will be taken at the beginning of the next execution. Else, the Cloud node2 will be blocked on $Q^T$and the last correct report can be restored from the checkpoint.

Table 4.2.Key notations and meanings.

Symbol	**Description**
clk2	Cloud node1 Clock
C_o	Counter of received temperature
S	Temperatures Sum

T	Received Temperature
Temp	Table for saving the received temperatures
N	Number of temperatures needed for average calculation
T_v	Temperature Average
T_l	Lowest received temperature
T_h	Highest received temperature
Checkpoint()	Procedure of Checkpoint
AverageCalcul()	Procedure of average temperature calculation
TestNode1	Procedure of the acceptance test

Algorithm 1: Cloud node1

Input: temperatures T;

Output: temperature Average T_v;

```
[1]Co = 0;
[2]S = 0;
[3]AverageCalcul():
[4]Receive(T, Sensor);
[5]Co = Co + 1;
[6]Temp [Co] = T;
[7]If Co < N Then
[8]GoToAverageCalcul();
[9]Else
[10]For i=0 to N-1 do
[11]S=S+Temp[i];
[12]Temp[i] = 0;
[13]End for
[14]  Tv= S / N;
[15]Send (Tv, Node2);        // Send of  " Tv" to the Cloud node2
[16]Co = 0;                  // re-initialization of Co
[17]S = 0;                  // re-initialization of S
[18]Go to AverageCalcul()
[19]End If
[20]End AverageCalcul()
```

The Cloud node1 has a main function which consists of "*Calculating the average T_v of N temperatures received from the Sensor node*". In the Algorithm 1, in order to count the number of received temperatures, a counter Co is used (line 1). First, the node1 receives the temperature T from the Sensor node (line 4), the counter Co is then incremented (line 5) and the temperature T will be saved in the table $Temp$ (line 6). The

Cloud node1 enters in a loop till reception of N temperatures($i.e., Co = N$). At that moment, the temperature average can be calculated (line 9) by first calculus of the sum of N temperature (lines [10-13]). After calculation of the average T_v(line 14), it will be sent to the Cloud node2 ([15]). A re-initialization for the next execution cycle will be done(line 16-17).The notations used in the algorithms and their meanings are presented in Table 9for a quick look-up.

Algorithm 2: Fail-Silent Cloud node1

Input: Temperatures T;

Output: temperature Average T_v;

```
[1]clk2=0;                    // Clock Initialization
[2]Co = 0;
[3]S = 0;
[4]Checkpoint():
[5]Save (Tv);// save of the last correct Tv
[6]clk2=0;// re-initializations for the next execution cycle
[7] Co=0;
[8] S=0;
[9]GoToAverageCalcul();
[10]End Checkpoint
[11]AverageCalcul():
[12]Receive(T, Sensor);
[13]Co = Co + 1;
[14]Temp [Co] = T;
[15]If Co < N Then
[16]GoToAverageCalcul();
[17]Else
[18]Tl=Temp[1];
[19]Th =Temp[1];
[20]For i=0 to N-1 do
[21]S=S+Temp[i];
[22]If Temp[i]>Ththen// Calculation of the highest temperature
[23]  Th= Temp[i];
[24]End If
[25]If Temp[i]<Tl  then//Calculation of the lowest temperature
[26]Tl= Temp[i];
[27]End If
[28]Temp [i] = 0;
[29]End for
[30]Tv= S / N;
[31]Send (Tv, Node2);
[32]GoTo TestNode1();
[33]End If
[34]End AverageCalcul()
[35]TestNode1():
[36] If [(Tl ≤ Tv ≤ Th) &&(AverageCalcul(a, b, c, d) = e) && (clk2 ≤ Node1TimeOut]
then
[37] Go to Checkpoint();
[38] Else
```

```
[39] Deadlock();//Deadlock in the case of non validation of the
Acceptance Test
[40] End If
[41] End TestNode1
```

In the Algorithm 2, The Fail-Silent Cloud node1 executes its main function of calculating the average temperature but with consideration of fault detection and time flow. In order to calculate time, a clock *clk2* is used. It is initialized at the beginning of each execution cycle to calculate time needed by the Cloud node1 to achieve its function.

After the reception of N temperatures by the Cloud node1, the highest temperature is calculated and saved in $\boldsymbol{T_h}$(lines[22-24]) and the lowest temperature $\boldsymbol{T_h}$ also will be calculated and saved (lines [25-27]). After calculating the average temperature $\boldsymbol{T_v}$ **(**line 30), it will be sent to the Cloud node2 (line 31). Before starting a next execution cycle, the output (i.e., T_v)must first pass the acceptance test within the procedure *TestNode1(*line 35). The main role of this procedure is to judge the correctness of the Cloud node1 outputs. The expression of the acceptance test (line 36) is composed of three parts; the first one is:$(T_l \leq T_v \leq T_h)$**.** This part of test is to ensure thatT_v is comprised between the lowest temperature and the highest one. The second part of the acceptance test is $(AverageCalcul(a, b, c, d) = e)$; it tests the calculus rigor of the Cloud node1 by calculating a similar simplified operation such that the input (a, b, c, d) and the output (e) are pre-known. The third part of the acceptance test is$(clk2 \leq Node1_{TimeOut})$. It aims to test whether the outputs are produced after the Time-Out expiration, which means that a response-time failure is occurred. In the case where the acceptance test is passed, the next execution cycle of node1 will start by a checkpoint (line 4) in order to save the last correct T_v (line 5) and to initialize variables and clock (lines [6-8]). In the worst case, when the acceptance test is not validated (i.e., at least one part of the Acceptance Test expression is not satisfied) the Cloud node1 will be considered as *failed* and it will remain in a deadlock state (line 39).

4.4.3 Time and Space complexity

In order to analyze time and space complexity of the previous algorithms, Big Omega asymptotic notation will be used. This notation allows calculating both time and space complexity of an algorithm. We have calculated the running expressions of the

precedent algorithms: Cloud node1algorithm and Fail-Silent Cloud node1algorithm. We have:

$f_{Node1}(n) = 13n + 2 \leq g(n)\ \ for\ all\ \ n \geq 0$, and

$f_{FSNode1}(n) = 19n + 17 \leq g(n)\ \ for\ all\ \ n \geq 0$

Where: $g(n) = 36n\ \ for\ \ n \geq 0$.

We can say that:

$$\left.\begin{matrix} f_{Node1}(n) \\ f_{FSNode1}(n) \end{matrix}\right\} = O(n)\,, n \geq 0.$$

The time and space complexity of the functions $f_{Node1}(n)$ and $f_{FSNode1}(n)$ are calculated according to the *n* values. If we assume that *n* represents the *time unit*, then the graph plot in Figure 4.10 represents the time complexity of the Cloud node1 program. In this case, we can see that the functions $f_{Node1}(n)$ and $f_{FSNode1}(n)$have the same time growth rate. That means that the incorporation of the acceptance test in the Cloud node1 program does not produce any big overhead. If we assume that *n* represents the *space unit*, then the Figure 4.10 is a space complexity graph of the functions $f_{Node1}(n)$ and $f_{FSNode1}(n)$ which are similar in space growth rate. It means that the Fail-Silent Cloud node1 does not need a big storage space compared to the primary Cloud node1 program.

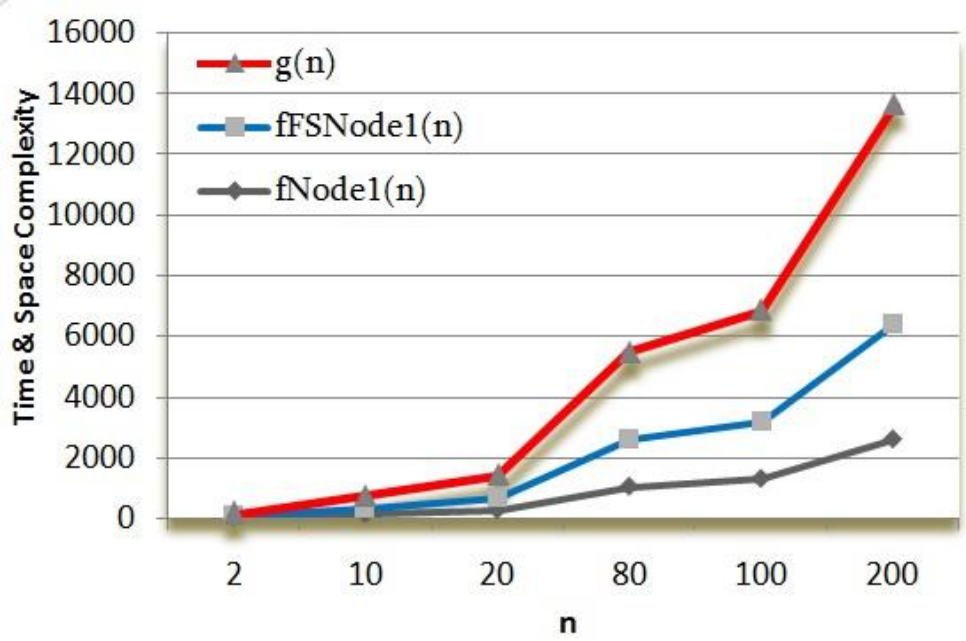


Figure 4.10.Time and space complexity of Cloud node1. fNode1 (resp. fFSNode1) represents the Cloud node 1 complexity before (resp. After) the acceptance test integration.

We can deduce that the time and space complexity of the failure detection process using the acceptance test does not lead to any unreasonable overhead or calculus complexities in the Cloud node1 (Figure 4.10). Finally, our proposed framework allows integration of the failure detection over the cloud nodes without large costs. Hence, this is a fair and practical solution to the issue.

4.4.4 Safety verification using model-checker

As noted previously, the acceptance test strategy aims to ensure safety in cloud systems in spite of failures. In order to prove the efficiency of our framework, uppaal 4.0.14 model-checker for safety verification is used. First, a simulation of the Fail-Silent Fire Control model is done to ensure the practicability of the model (See Figure 4.11). After that, a set of safety properties that must be insured by the Fail-Silent model are specified.

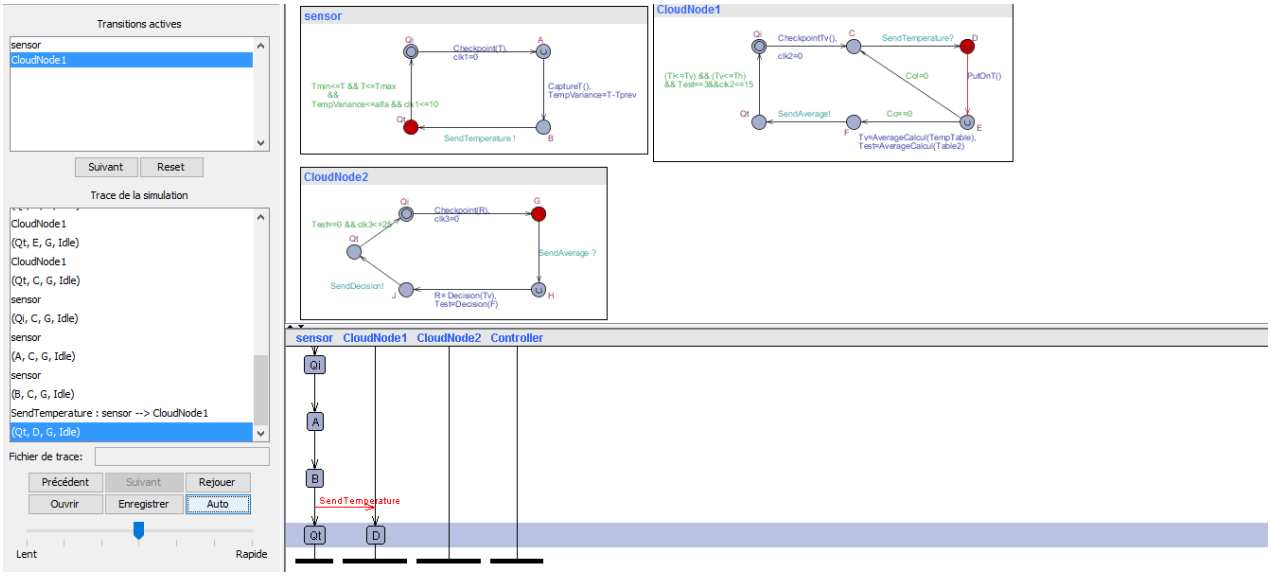


Figure 4.11. Simulation of Fail-Silent Fire Control System model

The safety properties must be satisfied by the Fail-Silent model in order to say that it is Safe. Safety properties are summarized in the Table 4.3.

Table 4.3. Safety property of Fire Control System model.

Safety Properties	**Safety Request**
1. The sensor node never produce random temperature values.	A[]not (sensor.Qi and (T-Tprev)>sensor.alfa) (n.b.,alfa value must be defined).
	A[] not (sensor.Qi and (T>sensor.Tmax or T<sensor.Tmin)) (n.b., sensor.Tmin, sensor.Tmax values must be defined)
2. The Cloud node1 never earns execution if it does not produce the correct temperature average.	A[] not (CloudNode1.Qi and CloudNode1.Test!=<value>) (n.b., <value> must be defined)
3. The Cloud node1 never reaches the procedure AverageCalcul with temperature sum different of 0.	A[] not(CloudNode1.E and CoudNode1.S!=0)
4. The Cloud node2 never earns execution if it does not produce the correct decision	A[] not (CloudNode2. Qi and CloudNode2.Test!=<value>) (n.b., <value> must be defined)
5. The sensor node never earns execution if it sends a	E[] not (sensor.Qi and sensor.clk1>sensor.TimeOut)

temperature after the expiration of its Time-Out.	
6. The Cloud node1 never earns execution if it sends temperature average after expiration of its Time-Out.	E[] not (ClouNode1.Qi and CloudNode1.clk2>CloudNode1.TimeOut)
7. The Cloud node2 never earns execution if it sends decision after expiration of its Time-Out.	E[]not (CloudNode2.Qi and CoudNode2.clk3>CloudNode2. TimeOut)
8. The entire system never earns execution if it produces decisions after the expiration of its Time-Out. (n.b.,The clock of the last component can be considered as the global clock of the system).	E[] not (CloudNode2.Qi and CloudNode2.clk3><SystemTimeOut>)

4.4.4.1 Safety Verification of fault-free model

First, the properties are verified on the fault free Fire Control model using the variable values defined in Table 4.4 and the verification results are presented in Figure 4.12.

Table 4.4.Variable initialization used for the fault free verification.

Variable	Value
T_{Max}	120
T_{Min}	-20
$\alpha\ (alfa)$	40
N	5
$sensor_{TimeOut}$	10
$Node1_{TimeOut}$	15
$Node2_{TimeOut}$	25
C	If temperature average>60 then, Fire Alarm End If

Figure 4.12.safety properties verification on fault-free Fail-Silent Fire Control model.

We can see in the Figure 4.12 that all the safety properties are verified on the fault free Fail-Silent Control model which means that the model is safe.

Table 4.5. Faults injected in the Fail-Silent Fire control model.

Fault	**Fault Type**	**Component**	**Injection**	**Safety property in Table 10**
Production of random values.	Transient hardware	Sensor	High temperature value + high temperature variance	1
Incorrect calculation	Software	Cloud node1	Algorithm 2-line20, i:=1 instead of i:=0	2
Incorrect calculation	Software	Cloud node1	Algorithm 2-line 8, the instruction S=0 is deleted	2-3
Production of random decision	Transient hardware	Cloud node2	Incorrect result	4
Component Time-out	Response-Time	Sensor	Add a loop on the state sensor.Qt (see Figure 4.13)to produce a response-time failure.	5
Component Time-out	Response-Time	Cloud node1	Add a loop on the state CloudNode1.Qt (see Figure 4.13) that produces +10 of execution time	6
Component Time-out	Response Time	Cloud node2	Add a loop on the state CloudNode2.Qt (see Figure 4.13) that produces +10 of execution time	7-8

4.4.4.2 Safety verification of failed model

After safety verification on the fault free model, the safety verification is done on the failed model. The safety properties of the Table 4.3 will be verified on the same Fail-Silent Fire Control model with the same variables but this time with injected faults. A set of faults are injected in the model, in order to make the model failed and to test whether the acceptance test strategy can preserve safety in spite of faults. The set of injected faults are summarized in Table 4.5. For each injected fault, some details are given such as: the type of fault, the component, how the injection is applied and the safety property violated.

After injection of the faults, the produced model is presented in Figure 4.13. After that, the safety properties (Table 4.3) are verified on the failed model and the verification results are presented in the Figure 4.14.

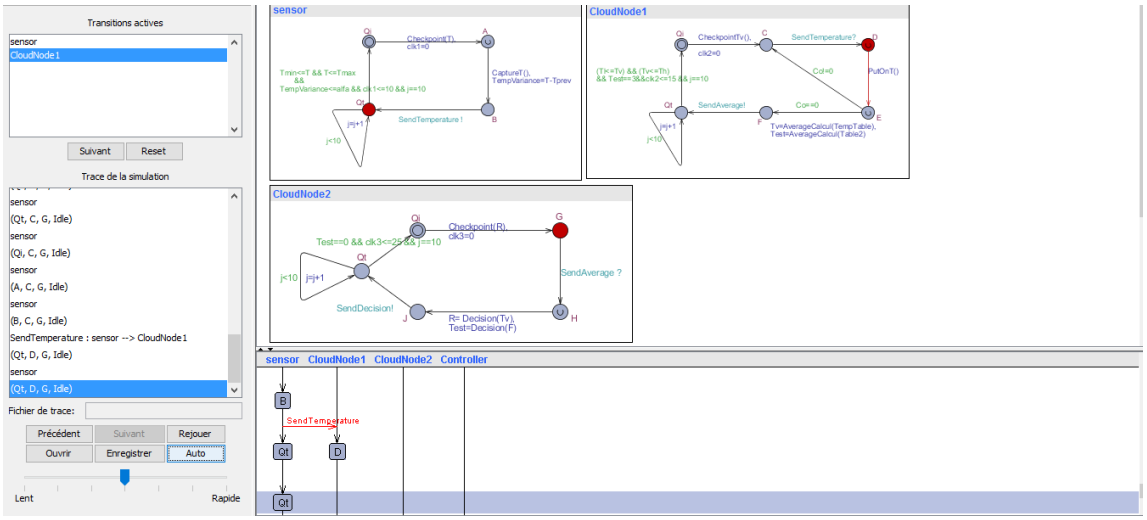


Figure 4.13.Fail-Silent Fire Control model after faults injection.

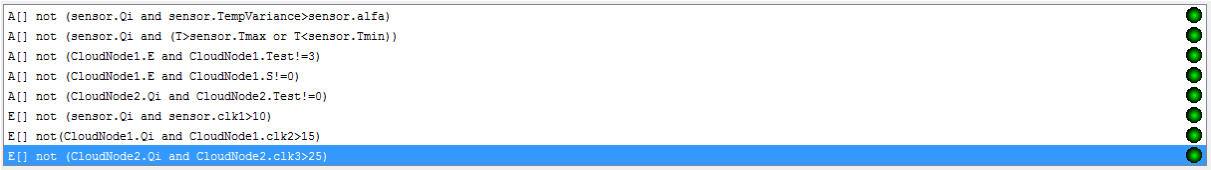


Figure 4.14.Safety verification of the failed Fail-Silent Control model.

As can be seen in Figure 4.14, all the properties are satisfied by the failed Fire Control model. Hence, all safety properties that are satisfied on the correct model are also satisfied on the failed model. This means that the Fail-Silent behavior of the acceptance test strategy preserves safety in spite of presence of failures.

Finally, we can say that the acceptance test strategy is efficient enough for safety insurance in the cloud Systems.

4.5 Comparative Analysis

The comparisons between IDS, Heartbeat/Pinging and acceptance test strategies are summarized in Table 4.6, where the main differences between the strategies are mentioned:

Strategy based on: The strategy of IDS is based on the cloud monitoring in which the Cloud behavior is compared to a previous database which is different than the heartbeat strategy that is based on *keep-alive* message transmission. In the acceptance test strategy, the failure detection is distributed on the cloud nodes where each node has its own acceptance test that can validate its behavior.

Monitoring process centralized or distributed: It is centralized in IDS. In heartbeat, each failure detector node is responsible for a set of cloud nodes; therefore, we can say that it is partially-distributed. In the acceptance test strategy, each node has its own acceptance test. Hence, the failure detection process is distributed over all the cloud nodes.

Detected failure origin: IDS can detect any malicious attack over the cloud nodes or network where the heartbeat strategy can detect only the hardware crashes. The acceptance test strategy can detect any abnormal behavior caused by software faults or transient hardware faults.

Alarm causes: The key question is: *In which cases the alarm announces that there is a failure?* In the IDS strategy, the failure alarm is raised whenever a deviation from the normal behavior is monitored on the cloud system. In the heartbeat strategy, if the cloud node does not send any alive-message to the detector node before the timeout expiration, the failure alarm is raised. In the acceptance test strategy, if any abnormal behavior is detected by the acceptance test over the cloud node, the failure alarm is raised.

Property insurance: *Which non-functional property is ensured by the strategy?* IDS can insure the safety property by protecting the cloud system from malicious attacks. The heartbeat strategy can ensure only liveness of the cloud nodes whereas the acceptance test strategy ensures the safety property by protecting cloud nodes from software faults and hardware transient failures.

Monitored components: In IDS, monitored components are the cloud nodes and the network connections. In the heartbeat and the acceptance test strategies, monitored components are only cloud nodes.

Failure detector component: In IDS, it is the system monitor. In heartbeat, the detector nodes are charged by the crash detection. In the acceptance test strategy, each cloud node is responsible for its failure detection process.

Failure detection accuracy: When we talk about the accuracy of the failure detection strategy, we respond to the question*: Is there really a failure when an alarm is raised?* The failure detection accuracy strongly relies on the monitoring process architecture (i.e., Centralized, partially-distributed, or distributed) which means that the distance between the failure detector and cloud nodes is very important in the cloud network. We have used the scale shown in Table 4.7:

As noted before, in IDS strategy, the most known problem is the False Alarm Rate. This is because of the difficulty of monitoring a huge number of cloud nodes by a central monitoring approach which would produce high distance between the monitor and the monitored components. Therefore, we can say that the accuracy of IDS is low. However in the heartbeat strategy, monitoring is partially-distributed where each crash detector is responsible for a set of nodes. The accuracy of crash alarm here is related to the network conditions and timeout but the distance between the monitor and the monitored component is medium and therefore, the accuracy is medium level compared to that of IDS. In the distributed monitoring such as the acceptance test strategy, the failure detector is the Cloud node; there is no distance between the monitor and the monitored component, thus the failure alarm is raised only in the case of failure. Hence, the accuracy of the failure alarm is high compared to that of IDS and heartbeat.

Component-based Approach: IDS and heartbeat strategies do not deal with component-based architecture of the cloud systems but the acceptance test strategy is based on this approach.

Scalability: The IDS does not support the scalability because it is difficult to provide frequent database knowledge for scalable cloud systems. The heartbeat strategy is known as large-scale crash detection strategy because it supports the scalability. The acceptance test strategy is based on the component-based approach, where atomic cloud nodes are coordinated to construct the global cloud system. The component-based

approach supports the scalability. Furthermore, the fault detection strategy using the acceptance test is independent from the architecture of the cloud system because it depends only on the cloud node behavior. Therefore, the acceptance test strategy is scalable.

Costs: For the IDS, the monitoring algorithms need complicated algorithms, large data and long time. The heartbeat strategy needs large bandwidth for network connections. The acceptance test strategy does not need large costs because Cloud nodes will carry on the monitoring process in addition to their main functions.

4.6 Conclusion

In this chapter, a fault detection framework is proposed for cloud computing systems by using Recovery Blocks' acceptance test. The proposed framework aims to construct Fail-Silent cloud modules which have the ability of self-fault detection. In this, the detection process of transient hardware faults, software faults, and response-time failures is performed locally on each computing machine in the cloud system. The proposed strategy is performed on a case study, time and space complexities are estimated and efficiency is proved using verification by model-checker.

Table 4.6. Comparison of various aspects of IDS, Heartbeat/Pinging, and acceptance test strategies.

Features / **Approaches**	**Strategy based on**	**Centralized/ Distributed Monitoring Process**	**Detected Failure Origin**	**Alarm Causes**	**Property insurance**	**Monitored Component**	**Failure Detector Component**	**Accuracy**	**Component-based Approach**	**Scalability**	**Costs**
Intrusion/Anomaly Detection Systems (IDS)	Cloud System Monitoring	Centralized	Malicious attacks	Behavior Deviation	Safety	Nodes and network Connections	System Monitor	Low	No	No	Complicate algorithms, Time&Data
Heartbeat and Pinging	Keep-Alive Messages	partially-Distributed	Hardware crash Failures	Timeout Expiration	Liveness	Nodes	Node Detector	Medium	No	Yes	Large network connections bandwidth
Acceptance Test Strategy	Acceptance Test	Distributed	Software Faults &Transient Hardware faults	Acceptance Test no validation	Safety	Nodes	Nodes	High	Yes	Yes	Reliable Acceptance Test

Table 4.7. Accuracy scale.

Accuracy / **Distance**	**High**	**Medium**	**Low**
Big	-	-	X
Medium	-	X	-
Small	X	-	-

Chapter Five

Fault Masking in Component-based Cloud Computing

Summary

5.1 Introduction

Fault tolerance has always been an active line of research in design and implementation of dependable systems. It involves providing a system with the means to handle unexpected defects, so that the system meets its specification in the presence of faults. Many Techniques are used to create the fault tolerance capability in cloud systems. They can be divided into two main categories: Proactive Fault Tolerance (i.e., Software Rejuvenation, Pre-emptive migration and Self-healing) and Reactive Fault Tolerance (i.e., Checkpointing, Job Migration, Replication, SGuard, Retry …etc) [22-27][94][98][101][102][103]. Fault tolerance techniques used in cloud computing are

based on time and space redundancy which can tolerate only hardware faults without dealing with software faults. According to our thorough investigation of the area, there is clearly a lack of formal approach that rigorously relates cloud computing with software fault tolerance concerns. In this chapter, a strategy of Fault-Masking in component-based cloud computing based on Recovery Blocks is presented. The aim is to construct reliable and available cloud nodes using the acceptance test and forward recovery.

5.2 Recovery Blocks for Fault-Masking

In order to construct the Fault-Masking component, Recovery Blocks scheme is used. A Fault-Masking node is able to satisfy safety and liveness [16][17] specification properties in spite of faults. That's means that it can detect and tolerate failures at just appearance and continue to offer its main service without any perturbation. The Fault-Masking node is a self-fault detector and a self-stabilizer in the same time. A node that ensures safety property means that it never reaches a non-desirable state whereas a node with liveness property insurance means that it always reaches a stable state after any fault detection. In other meaning, the Fault-Masking node offers secure and continued service in spite of failures.

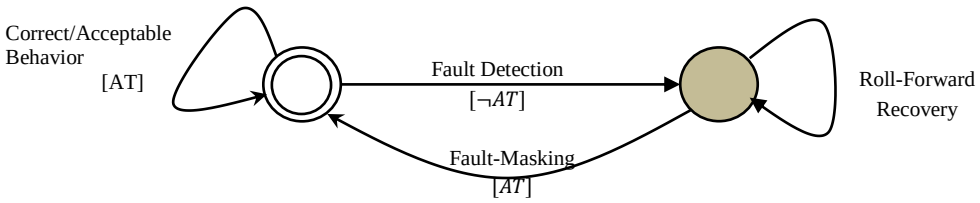


Figure 5.1. Fault-Masking node behavior

The Figure 5.1 shows the Fault-Masking node behavior. In which the cloud node earns execution (i.e., left state) since the behavior is correct or acceptable. At the moment of fault detection, the cloud node will stop operating and then will enter in a forward recovery phase (Figure 5.1. the right state). In the forward recovery an alternate try block will be used to recover from the failure. After the recovery phase, the cloud node behavior will reach a stable state with an acceptable behavior.

Algorithm of Fault-Masking based on Recovery Blocks

Ensure Acceptance Test By
Primary Try block
Else By
Alternate Try Block
End Recovery Blocks

5.2.1 Fault-Masking atomic component

Definition1*:* an atomic component is defined as a tuple $B = (Q, P, Beh, X)$ such that:

Q: is a set of states $\{q^0, q^1, \dots, q^n\}$;

P**:** is a set of communication ports $\{p_0, p_1, \dots, p_n\}$;

$Beh = \{\rightarrow / \ \tau \in \rightarrow = Q \times G \times F \times Q \}$ is the behavior of the atomic component B. It is composed of a set of transitions. Each transition contains one guard and a set of internal functions. The main behavior of an atomic component is considered as its Primary behavior.

X**:** is a set of variables $\{x_i\}$ which are manipulated by the internal functions, f_i;

Definition 2*:* An acceptance test $AT_B(X)$ of an atomic component $B = (Q, P, Beh, X)$ is a boolean expression on the set of variables, X. The acceptance test validates the correctness of B's final results and ensures that they do not lead to disastrous consequence even if they are not the expected results. The acceptance test ensures *Safety* properties in the atomic component. An atomic component that has an acceptance test is a *Fail-Silent* atomic component.

Definition 3*:* A *Fail-Silent* atomic component, $FS_B = (Q, P, Beh, X, \ AT_B)$ is a self-fault detector, it can ensure *Safety* properties using the acceptance test AT_B. In the case of fault detection, the atomic component FS_Bwill pass to a deadlock state till recovery achievement. The ω-regular expression of the *Fail-Silent* atomic component is $[(c/a)^* f]$.

A *Fail-Silent* atomic component has a correct behavior (c) or an acceptable behavior (a). At just fault detection by the AT, the atomic component will be considered as failed *(f)* and it will be blocked immediately attending the recovery phase.

Definition 4*:* a Primary behavior of a *Fail-Silent* atomic component is the main behavior that offers desired results. We write: $FS_B = (Q,\ P,\ Beh_{Primary},\ X,\ AT_B)$.

$Beh_{Primary}$: is the Primary behavior. It performs the desired operation.

Definition 5*:* An Alternate behavior $Beh_{Alternate}$of the *Fail-Silent* atomic component performs the operation in different manner. It aims to replace the Primary behavior in the case of fault detection.

By using the Alternate behavior, the component can performs a roll-forward recovery phase till the reach of a stable state (i.e., correct (c) or acceptable (a) state). The ω-regular expression which design the main role of the Alternate behavior in an atomic component is $r^*(c/a)$ such that:

r : Recovery;

c : Correct behavior;

a : Acceptable behavior.

Lemma 1*:* A Fail-Silent atomic component that uses an Alternate behavior $(i.e., Beh_{Alternate})$ can ensure *Liveness* property even in the presence of faults.

Theorem 1: The use of an Alternate behavior in the *Fail-Silent* atomic component can produce a *Fault-Masking* component that ensures both *Safety* and *Liveness* properties in the same time. The ω-regular expression of the Fault-Masking atomic component is $[(c/a)^* f\ r^*\ (c/a)]^{\omega}$.

Definition 6*:* a Fault-Masking atomic component is a component that can preserve and *Liveness* specification properties in presence of faults. We write:

$FM_B = (Q,\ P,\ Beh_{Primary},\ Beh_{Alternate},X,\ AT_B)$ Such that:

$Beh_{Primary}$: is the Primary behavior and

$Beh_{Alternate}$: is the Alternate behavior. It is required to perform the desired operation in a different way.

5.2.2 Fault-Masking composite component

A composite component $B = \gamma(B_1 \dots B_n)$ is a set of atomic components $B_{i=1..n}$ glued by the set of connectors $\gamma = \{\beta_{i=1..l}\}$. As seen in the chapter 3 - Section 3.2.3, a connector in a composite component can be a rendezvous or broadcast connector.

5.2.2.1 Rendezvous connector

If we have the rendezvous connector γ such that $\gamma = \{p_{i=1..n}, p_i \subset B_i\}$, the only possible interaction is $a_i = B_1\, B_2\, B_3 \dots B_n$ which contains all the atomic components

involved in the connector γ. Therefore, the failure of one atomic component will directly infect the others atomic components. This means that $\forall\, B_{1\leq i\leq n} \in \gamma;\ if\ (B_i)$ is failed, then $\forall B_{j\neq i} and B_j \in \gamma$, B_j will be failed too.

Lemma 2: Let $B = \gamma(B_1 \dots B_n)$ a composite component. In order to construct a Fault-Masking rendezvous connector γ, all the atomic components $B_1 \dots B_n$ involved in it must be Fault- Masking as well. $FM_{Rendezvous(\gamma)} = \{FM_{B1}, FM_{B2}, \dots, FM_{Bn}\}$.

5.2.2.2 Broadcast connector

If we have the broadcast connector $\gamma = \{p_{i=2..n}, p_i \subset B_{i\neq k}\ and\ B_k\ is\ Broadcast\ initiator\}$. The possible set of interactions in this case are those containing at least one instance of B_k. The minimum interaction is $a_1 = \{B_k\}$ which contains only the broadcast initiator where the maximum interaction is $a_n = \{B_k B_2 B_3 \dots B_n\}$ which contains all the atomic components involved in the connector γ.

Lemma 3: Let $B = \gamma(B_k \dots B_n)$ a composite component where γ is a broadcast connector. To construct a Fault-Masking *broadcast* connector γ, at least the broadcast initiator B_k must be Fault-Masking: $FM_{Broadcast(\gamma)} = \{FM_{Bk}, B2, \dots, B_n\}$.

Theorem 2. A composite component that is composed of a set of Fault-Masking atomic component is Fault-Masking composite component: $FM_B = \gamma(FM_{B1}, FM_{B2}, \dots, FM_{Bn})$.

5.3 A Case Study

5.3.1 Construction of Fault-Masking models

In order to describe the construction of *Fault-Masking* models, we will present a case study of Fire Control System (seen in chapter 4-Section 4.4.1). The Figure 5.2 presents the Cloud node 1 model.

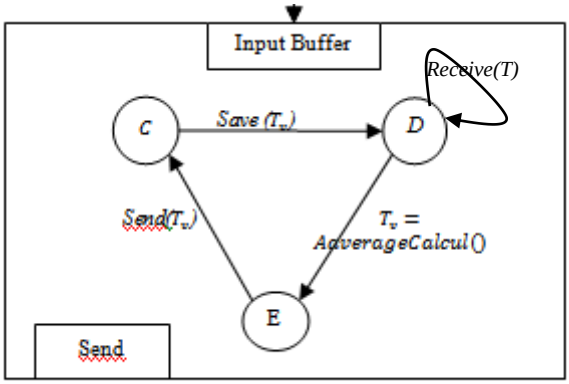


Figure 5.2 Cloud node 1

$CloudNode1 = (Q, P, Beh, X)$ where:

$Q = \{C, D, E\}$

$P = \{Save, AverageCalcul, Send\}$

$Beh = \{< C, save(T_v), D >, < D, Receive(T), D >, < D, T_v = AverageCalcul, E, >, < E, Send\ (T_v), C >\}.$

$X = \{T_v\}.$

Algorithm 1: Cloud node1

Input: temperatures T;

Output: temperature AverageT_v;

```
[1]Co = 0;
[2]S = 0;
[3]ReceiveTemperatures()
[4] Receive(T, Sensor);
[5] Co = Co + 1;
[6] Temp [Co] = T;
[7] If Co < N Then
[8] GoToReceiveTemperatures();
[9] Else
[10]  AverageCalcul():
[11]   For i=0 to N-1 do
[12]    S=S+Temp[i];
[13]    Temp[i] = 0;
[14]  End for
[15]   Tv= S / N;
[16]  EndAverageCalcul()
[17] Send (Tv, Node2);          // Send of  " Tv" to the Cloud node2
[18] Co = 0;                  // re-initialization of Co
```

```
[19] S = 0;                    // re-initialization of S
[20] Go to ReceiveTemperatures()
[21]End If
```

For constructing the Fault-Masking model of the Cloud node1. An acceptance test and an Alternate behavior must be incorporated in the model. The acceptance test can be inserted using the procedure described in chapter 4. By this way, we will have a Fail-Silent component. Then, the Alternate behavior must be inserted in order to construct the Fault-Masking component.

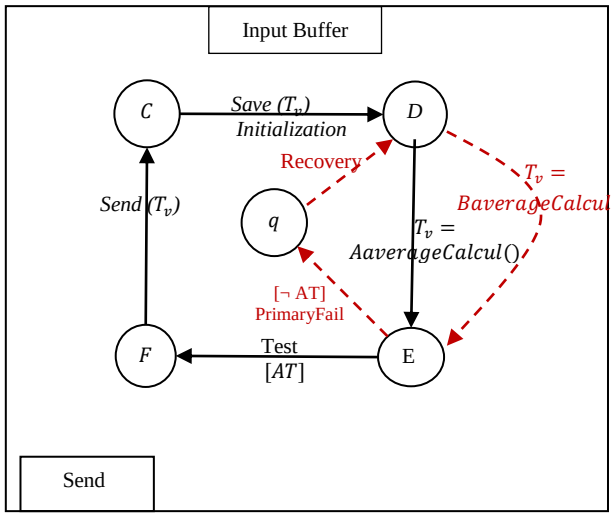


Figure 5.3 Fault-Masking Cloud node 1

The Fault-Masking model of the Cloud node 1 in the Figure 5.3 is composed of:

$FM_{CloudNode1} = (Q', P', Beh_{Primary}, Beh_{Alternate}, X, \ AT_{Producer})$ such that :

$Q' = \{C, D, E, F, q\}$.

P'

$= \{Save, AaverageCalcul, BaverageCalcul, PrimaryFail, Recovery, Test, Send\}$

$Beh_{Primary} = \{< C, save(T_v), D >, < D, T_v = AverageCalcul, E, >, < E, Test, [AT], F >, < F, Send(T_v), C >\}$.

$Beh_{Alternate} = \{< E, [\neg AT], \ PrimaryFail = True, q >, < q, Recovery, D >, < D, T_v = BaverageCalcul, E >, < E, Test, [AT], F >, < F, Send(T_v), C >\}$.

The Fault-Masking Cloud node1 algorithm is the following:

Algorithm 2: Fault-Masking Cloud node1

Input: Temperatures T;

Output: temperature Average T_v;

```
[1]clk2=0;                    // Clock Initialization
[2]Co = 0;
[3]S = 0;
[4] Checkpoint():
[5]  Save (Tv);// save of the last correct Tv
[6]  clk2=0;// re-initializations for the next execution cycle
[7]  Co=0;
[8]  S=0;
[9]  GoToReceiveTemperatures();
[10] End Checkpoint
[11]ReceiveTemperatures():
[12] Receive(T, Sensor);
[13] Co = Co + 1;
[14] Temp [Co] = T;
[15] If Co < N Then
[16] GoToReceiveTemperatures();
[17] Else
[18]  Tl=Temp[1];
[19]  Th =Temp[1];
[20]  For i=0 to N-1 do
[21]  If Temp[i]>Ththen// Calculation of the highest temperature
[22]  Th= Temp[i];
[23]  End If
[24]  If Temp[i]<Tl  then//Calculation of the lowest temperature
[25]  Tl= Temp[i];
[26]  End If
[27] EndFor
[28] GoToAaverageCalcul()
[29]EndReceiveTemperatures()
[30]AaverageCalcul()
[31]  For i=0 to N-1 do
[32]  S=S+Temp[i];
[33]  Temp [i] = 0;
[34] End for
[35]  Tv= S / N;
[36]  If [(Tl ≤ Tv ≤ Th) && (clk2 ≤ Node1TimeOut] then
[37]   Send (Tv, Node2);
[38]    Go to Checkpoint();
[39]  Else
[40]   GoToBaverageCalcul();//non validation of the Acceptance
Test
[41]  End If
[42] End AverageCalcul()
[43] BaverageCalcul():
[44] Tv = (Temp[0] + Temp[n − 1])/2
[45] If [(Tl ≤ Tv ≤ Th) && (clk2 ≤ Node1TimeOut] then
[46] Send (Tv, Node2);
[47] Go to Checkpoint();
[48] Else
```

```
[49] FailedRecoveryBlocks();//non validation of the Acceptance
Test
[50] End If
[51]End BaverageCalcul()
```

As said before, the construction of a Fault-Masking model must follow two phase: the construction of the Fail-Silent model then the incorporation of the forward recovery to reach the Fault-Masking model. The Algorithm 2 describes the fault – Masking model of the Cloud node 1. The main behavior of the Cloud node1 is the calcul of the average temperature T_v of N received temperatures. Then, it sends the output to Cloud node2. The Primary behavior of Cloud node1 is designed by the procedure $AaverageCalcul$ [line 30]. After the calcul of T_v , it must pass the acceptance test [line 36]. If T_v satisfies the test, it will be sent to Cloud node 2, else a forward recovery will be provided by invoking the Alternate behavior which is designed by the procedure $BaverageCalcul$ [line 40]. By using the alternate procedure [line 43-44], the average temperature T_v will be calculated by using only the first received temperature and the last one. At the end, the result must pass the acceptance test to validate its correctness. If T_v is accepted then it will be sent to the successor else the recovery blocks will be considered as failed.

5.3.2 Time and space complexity

In order to analyze time and space complexity of previous algorithms, Big Omega asymptotic notation is used. We have calculated the running expressions of the Cloud node1 algorithm and for the Fault-Masking Cloud node1algorithm. We have:

$f_{Node1}(n) = 15n + 9 \leq g(n) \ for\ all \ \ n \geq 0$, and

$f_{FMNode1}(n) = 25n + 34 \leq g(n) \ for\ all \ n \geq 0$

Where: $g(n) = 59n \ for \ n \geq 0$.

We can say that:

$$\left.\begin{matrix} f_{Node1}(n) \\ f_{FMNode1}(n) \end{matrix}\right\} \mathrm{O}(n), n \geq 0.$$

The time and space complexity of the functions $f_{Node1}(n)$ and $f_{FMNode1}(n)$ are calculated according to the *n* values. If we assume that *n* represents the *time unit*, then the graph plot in Figure 5.4 represents the time complexity of the Cloud node1

algorithm. In this case, we can see that the functions $f_{Node1}(n)$ and $f_{FMNode1}(n)$have the same time growth rate. That means that the incorporation of the fault masking strategy in the Cloud node1 program does not produce any big overhead. If we assume that *n* represents the *space unit*, then the Figure 5.4 is a space complexity graph of the functions $f_{Node1}(n)$ and $f_{FMNode1}(n)$which are similar in space growth rate. It means that the Fault-Masking Cloud node1 does not need a big storage space compared to the primary Cloud node1 program.

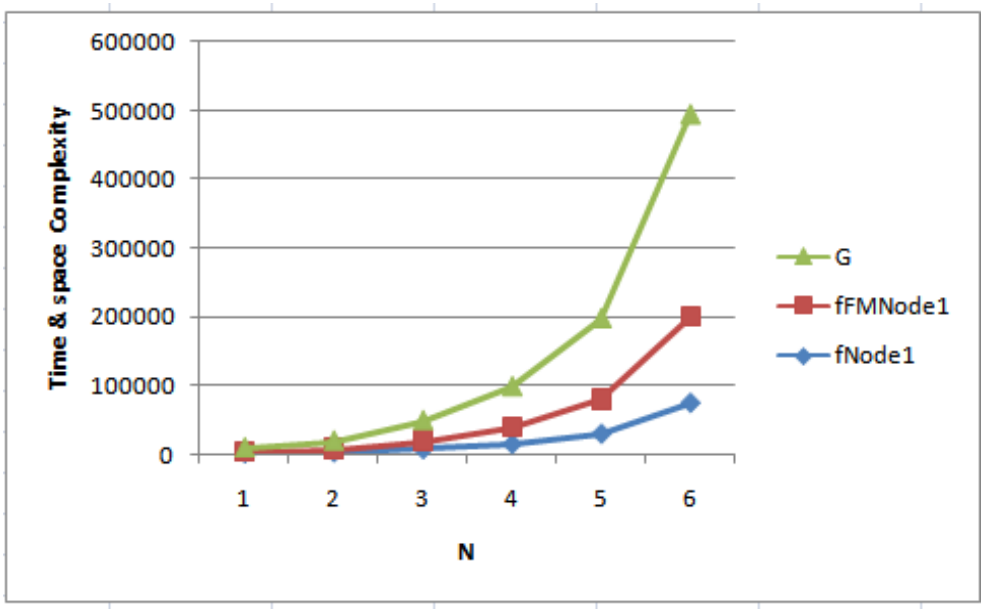


Figure 5.4 Time and space complexity of Cloud node1. fNode1 (resp. fFMNode1) represents the Cloud node 1 complexity before (resp. After) the Fault Masking Integration.

We can deduce that the time and space complexity of the Fault-Masking process using the acceptance test and the try blocks does not lead to any unreasonable overhead or calculus complexities in the Cloud node1 (Figure 5.4).

5.3.3 Distributed Recovery Blocks scheme

Recovery Blocks is an efficient mechanism for Fault-Masking, but it is based on the sequential execution (i.e., if the primary block fail then the alternate block will take place) which provide a latency in response time delays. This last is an important key in real-time applications especially in Cloud applications. In order to adapt Recovery

Blocks scheme for distributed real time constraints, Kim Kan proposed many architectures for Distributed Recovery Blocks. In this section, we will apply DRB scheme on the Cloud node1 model. It is composed of two nodes X and Y. Each node is considered as Fault Masking atomic component. X and Y are performed in distributed and parallel execution where the Primary node is the responsible for response delivery. Each Fault-Masking node has two try blocks A and B. A returns the desired output whereas B returns an acceptable one. The primary node X performs A as the primary block and B as the Alternate block. The backup node Y performs the try blocks in inverse way, by executing B as the Primary block and A as the Alternate one.

We assume that only one node fails at a moment. This assumption aims to ensure that at least one node is operator and hence it can send an output to the successor.

5.3.3.1 Construction of Fault-Masking model using DRB scheme

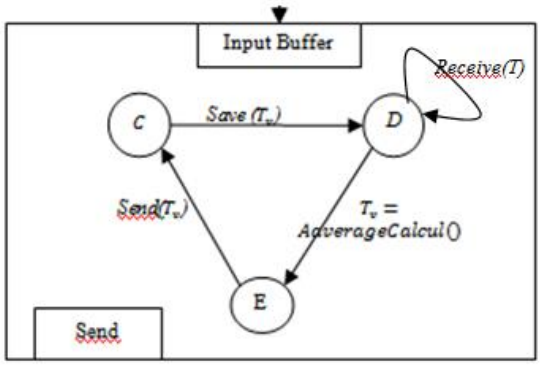


Figure 5.5: Cloud node1 BIP model - Fire Control system.

The Figure 5.5 is the BIP model of the Cloud node1. It is only Fail-Silent but not Fault-Masking model. The Figure 5.6 presents the Fault-Masking model of Cloud node1 using the DRB scheme.

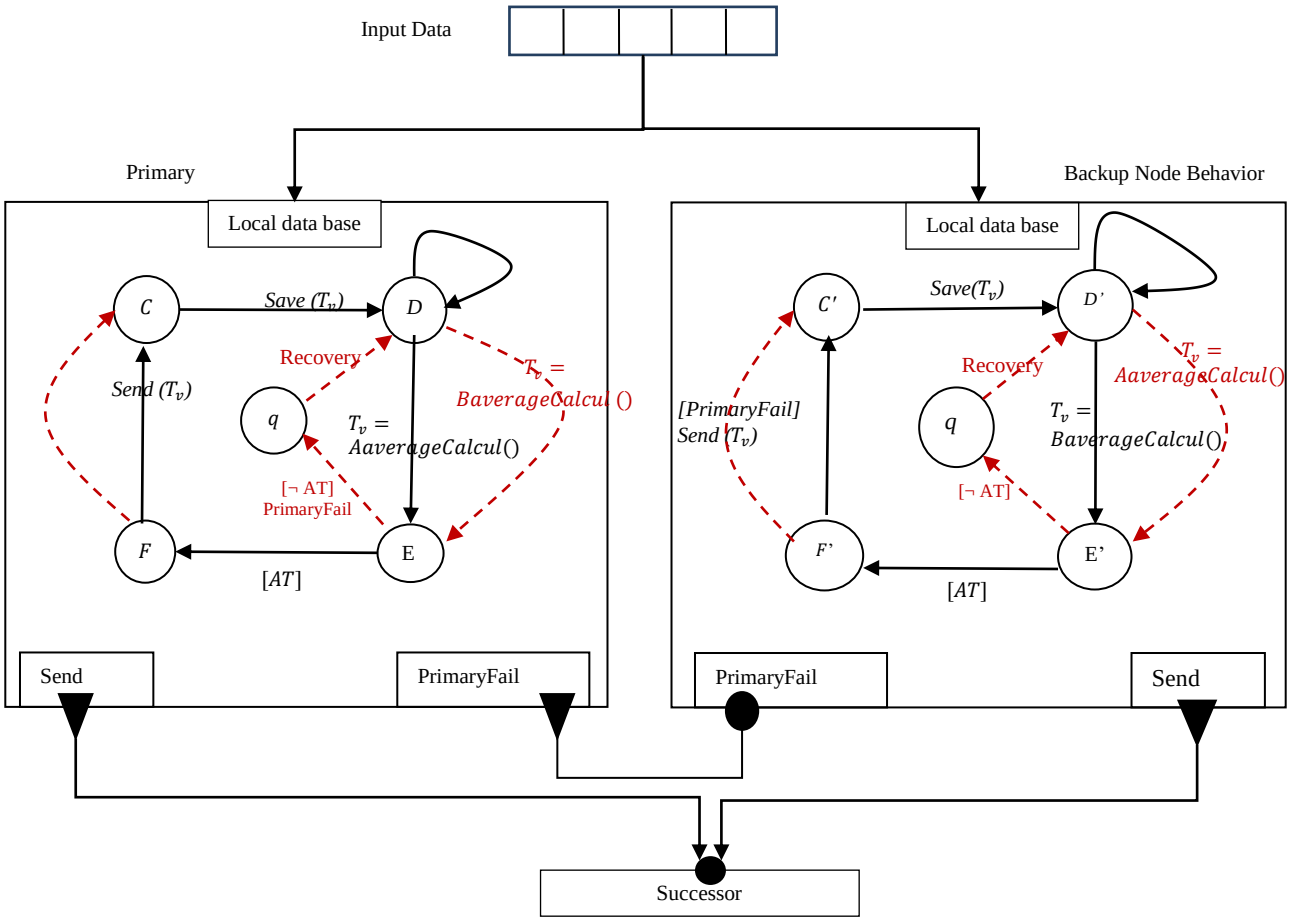


Figure 5.6. Fault-Masking Cloud node1 based on DRB scheme

The Figure 5.6 presents the Fault-Masking model using the DRB scheme of the Cloud node1. It is composed of two nodes a Primary node and a Backup node, each node is a Fault-Masking that uses an acceptance test for fault detection and two try blocks for forward recovery. Therefore, software and hardware faults can be handled in the same manner. At the beginning, each node performs its Primary block. The Primary node execute the primary block that calculate the temperature average using *Aaveragecalcul* which produces the desired result while the Backup node performs the Alternate block that uses the procedure *Baveragecalcul* to produce the correct but not the desired average. If the Primary succeeds to validate the acceptance test, the temperature $T_v$will be sent to the successor and will be saved on the local database. On the Backup node, if T_v validates the acceptance test it will be saved on the local database. Only the result produced by the Primary will be sent to the successor. In the case where the Primary node fails, it will send a message to the Backup node to inform it about the Primary

failure. At that moment, the Backup node sends its result to the successor and saves it on its own database. By that, the failure of the Primary will not affect the response-time delay of the component. In the same time the failed Primary node will perform a forward recovery using its Alternate block that performs the procedure *BaverageCalcul.* The Primary node failure provides a reverse roles on the nodes by which the Backup node will take the role of the Primary node while the failed Primary will be the Backup node. For the next execution cycle, the new Primary performs the Primary try block while the new Backup use the Alternate try block (i.e., the Primary, always, performs the Primary try block first while the Backup, always, performs the alternate try block). The roles reverse of the nodes will be applied with each Primary node failure. By that, the recovery time of the Primary does not affect the response time and the Primary failure has no effect on the service quality of the component. In the case where the Backup node fails, it performs a forward recovery silently using its alternate try block.

The DRB scheme allows masking hardware faults by the use of physical redundancy(i.e., Primary node and Backup node), masking software faults by using design diversity (i.e., Primary try block and Alternate try block) and masking response-time faults by the parallelism execution of same input data on two different nodes.

Table 5.1: key notations and meanings

Symbol	**Description**
X	The Primary node
Y	The Backup node
A	The Primary try block
B	The Alternate try block
$Primary_{Clock}$	The Primary node clock
$Backup_{Clock}$	The Backup node clock
AT	The acceptance test expression
T_v	The average temperature
Successor	The successor node
PrimaryFail	Boolean that refers the failure of the Primary node
S	Temperature Sum
N	Number of temperatures needed for average calculation
Temp	Table for saving the received temperatures
TimeOut	The execution delay set by the system monitor

Algorithm3: Masking Cloud node 1 (using DRB scheme).

Input: temperatures T;
Output: temperature Average T_v;

```
[1] PrimaryNodeCode()
[2]  Clock_Primary := 0
[3] Execute (PrimaryBlockCode())
[4] If AT=true then
[5]  Send (T_v, Successor)
[6]  Save(T_v)
[7]   Go to PrimaryNodeCode()
[8] Else
[9]  PrimaryFail:=True
[10]  Execute (PrimaryNodeFailure(), Recovery())
[11] End if
[12]End PrimaryNodeCode()
[13]BackupNodeCode()
[14]  Clock_Backup := 0
[15] Execute (PrimaryBlockCode())
[16] If AT=true then
[17]  If PrimaryFail=True then
[18]    Send (T_v, Successor)
[19]  End If
[20]  Save(T_v)
[21] Go to BackupNodeCode()
[22] Else
[23]  Recovery()
[24] Enf if
[25]End BackupNodeCode()
[26]PrimaryNodeFailure()
[27]Exchange (X,Y)
[28] Exchange(A,B)
[29] execute  (PrimaryNodeCode(), BackupNodeCode )
[30]End PrimaryNodeFailure
[31]Recovery ()
```

```
[32] Execute (AlternateBlockCode)
[33] If AT=True then
[34]  Save (T_v)
[35] End If
[36]End Recovery()
```

```
[37]A()
[38] S = 0;
[39] For i=0 to N-1 do
[40]   S=S+Temp[i];
[41] End for
[42] T_v= S / N;
[43]End PrimaryBlockCode()
```

```
[44]B()
[45] T_v= (Temp[0]+Temp[N-1]) / 2;
[46]End B()
```

```
[47]AT():
[48] Return    (averageCalcul(a,b,c,d) = e) &&(clock ≤ Timeout) = True
[49]End AT ()
```

```
[50]Initialization ()
[51] PrimaryBlockCode:=A
[52] AlternateBlockCode:=B
[53] PrimaryNodeCode:=X
[54] X.PrimaryBlockCode:=A
[55] X.AlternateBlockCode:=B
[56] BackupNodeCode:=Y
[57] Y.PrimaryBlockCode:=B
[58] Y.AlternateBlockCode:=A
[59]Execute(PrimaryNodeCode,  BackupNodeCode)
[60]End Initialization
```

As said before, the main role of the Cloud node1 is calculating the average of N temperatures and sending the result to the successor node 2. The algorithms 2 presents the Fault-masking behavior of the Cloud node1. In which, the calcul of the temperature average will be done using two different try blocks (i.e., the Primary try block and the Alternate try block) on two different nodes (the Primary node and the Backup node).

The Primary try block [lines 37-43], use N temperatures to provide the average, whereas the alternate try block [line 44-46] needs only the first and the last temperatures to calculate the average. The Primary node execute first the Primary try block while the Backup node execute the Alternate try block. Both nodes (i.e., the Primary and the Backup) have the same acceptance test [line 48]. As said before, the Primary and the Backup nodes perform their Primary try blocks in the same time on different machines. In the case of a failure, the node performs a recovery using its Alternate try block. At the beginning, A is considered as the Primary try block [line 51], whereas B as the Alternate try block [line 52]. X in the Primary node [line 53] and Y as the Backup node [line 56]. X will perform A as the Primary try block and B as the Alternate try block [line 53-54], Y will perform B as Primary try block and A as the Alternate try block. X and Y are performed on two different nodes in the same time [line 59]. The Primary node will send T_v to the successor node if its behavior satisfies the acceptance test [line 5]. In the case where the acceptance test is not validated by the Primary node [line 8-9], the Backup node will send T_v to the successor in order to avoid that the response time exceed the TimeOut [line 18]. At the same moment, the Primary node performs a recovery task using the Alternate try block B [line 32] and a role reverse will be performed by exchanging the failed Primary node X to be the Backup node and the Backup node Y as the new Primary node [line 27]. Another Exchange will be provided on the try blocks [line 28]. In which the Primary try block A will be the new Alternate try block and B will be the new Primary try block. A next execution cycle will be provided using Y (A, B) as Primary node and X (B, A) as backup one [line 29]. This roles reverse will be doing after any fault detection on the Primary node.

5.3.3.2 Liveness Verification using model-checker

The Figure 5.7, presents a simulation using UPPAAL tool for the DRB Fault-Masking model of the Cloud node1. In order to prove the effectiveness of the Fault-Masking model, a set of Liveness properties of the node1 model are presented in Table 5.2. We will verify the validation of these properties in both Fault-Free and failed models.

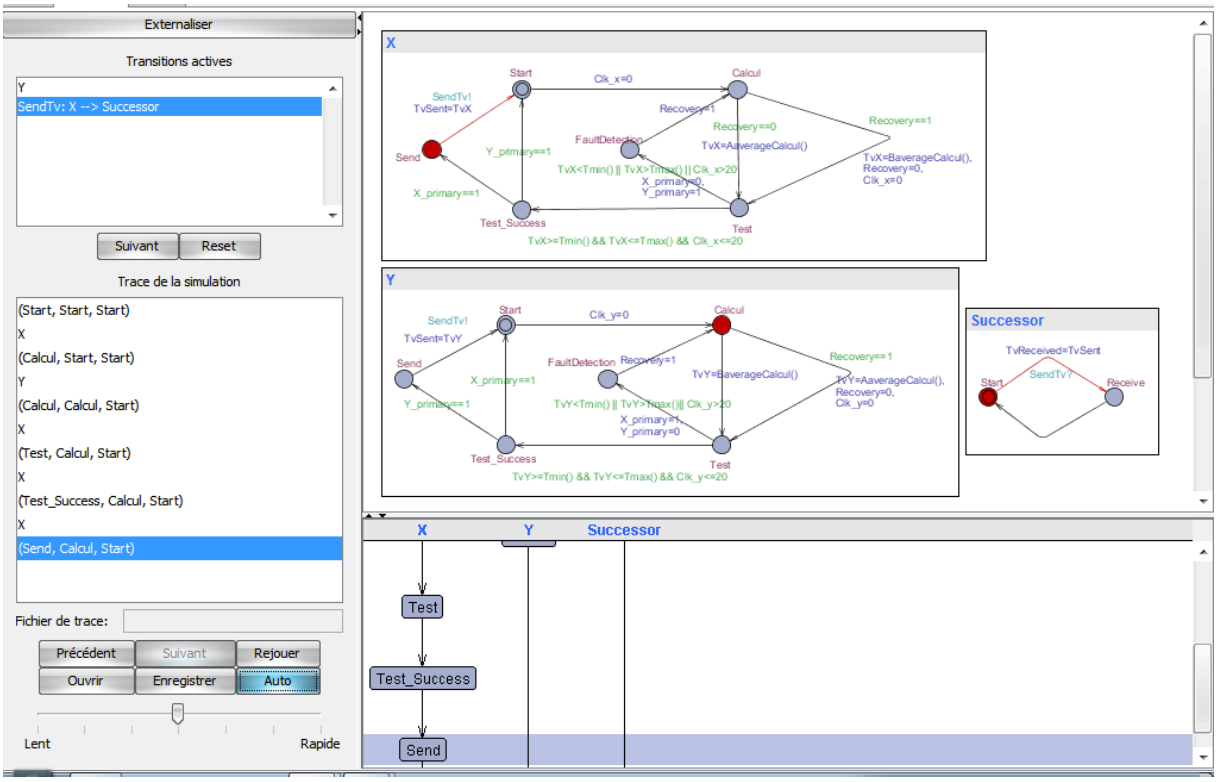


Figure 5.7 Fault-Masking model of Cloud node1- Fire Control system

Table 5.2. Liveness properties of the Cloud node1 model.

Liveness Properties	**Liveness Request**
1. The average temperature is always sent by the Primary node.	A[](X.Send imply X_primary==1)
	A[] (Y.Send imply Y_primary==1)
2. Always there is only one primary node.	A[] (not (X_primary==1 and Y_primary==1))
3. Only one node sends a response to the successor.	A[] (X.Send imply not Y.Send) and (Y.Send imply not X.Send)
4. The output sent to the successor is that validates the Acceptance Test.	E[] (X.Send imply X.Test_Success)
	E[] (Y.Send imply Y.Test_Success)
5. The output is always sent by a fault free node.	A[] (X.FaultDetection imply not X.Send)
	A[] (Y.FaultDetection imply not Y.Send)
6. The system never reaches a deadlock state.	A[] not deadlock

7. Always the successor receives a response.	E<>(Successor.Receive)
8. The output sent to the successor is usually that calculated by the Primary try block.	E[](Successor.TvReceived ==PrimBlock)

a. Liveness verification on the fault-free model

As shown in the Figure 5.8, Liveness properties are satisfied in the fault free model. Now, some faults will be injected in the model to make it failed and the same *Liveness* properties will be checked on the failed model. This aims to check whether *Liveness* properties are saved or not with the presence of failures.

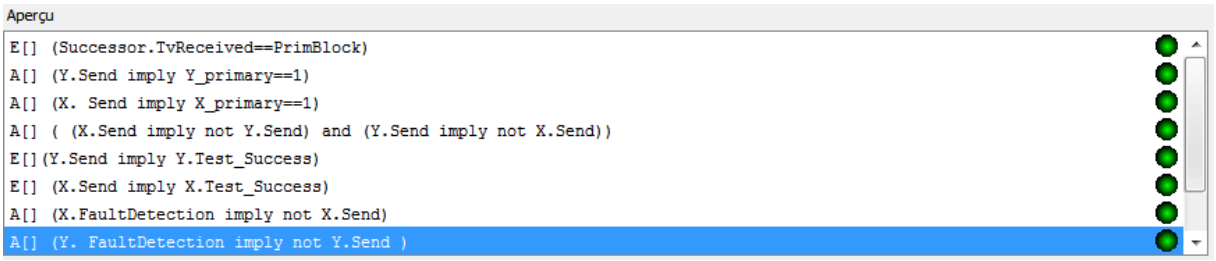


Figure 5.8 *Liveness* properties verification on the fault free model

b. Liveness verification on the failed model

Now, a set of failures are Injected on the initial model and the Liveness properties (Table 5.2) are checked. The set of injected faults are presented in Table 5.3. For each faults, the type and the injection manner are described.

Table 5.3: Faults injected in the fault free model

Fault	Fault Type	Injection
Production of random outputs	Transient hardware	TvX=50
		Tmin = 60
Incorrect calculation	Software	Algorithm3-line39, i:=1 instead of i:=0
		Algorithm3-line 38, the instruction S=0 is deleted

		Algorithm3- line 40, the instruction S=S+Temp[i] is replaced by S=Temp[i]
Component Time-out	Response-Time	Add a loop on the state X.Test (see Figure 5.7) that produces +10 of execution time

After injection of faults, the same liveness properties of the Table 5.2 will be verified on the failed model. The verification results are presented in the Figure 5.9.

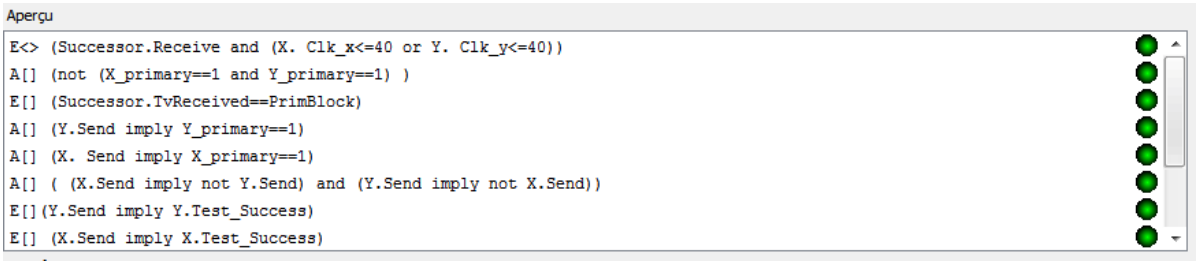


Figure 5.9 Liveness properties verification on the failed model

We can see that all liveness properties are satisfied in the model in spite of hardware and software faults. We can conclude that the use DRB Fault- Masking scheme for cloud environment is efficient compared with the existent strategies for Fault Tolerance in cloud computing systems.

5.4 Comparative Analysis

A comparative analysis between the Fault-Masking strategy and the existent techniques for fault tolerance are summarized in the Table 5.4.

Table 5.4. Comparison between fault tolerance techniques in cloud systems

Technique	Strategy	Principle	Difficulties	Advantages	Restrictions
Self-Healling Job Migration Replication Task Resubmission	Space redundancy	-Execution of the same version on multiple machines.	-Needs complicated protocols for managing the replication groups.	-Efficient for hardware fault tolerance. -Convenient for real-time applications. -Response time delays are reduced.	-Cost more physical components.
Checkpointing Software rejuvenation SGuard Retry	Time Redundancy (Rollback recovery)	-Execution of the same program many times.	-Needs to create a coherent state for rollback recovery. -The creation of a coherent recovery state is a difficult task. - Needs to precise a checkpointing frequency.	-Efficient for transient hardware fault tolerance.	-A bad checkpointing process can remains the system to a domino effect. -Not supportable by real-time applications.

Fault Masking	Design Diversity &Space redundancy.	-Execution of multiple versions for the same program on different machines.	-Difficulty of designation of a rigorous acceptance test. -the development of different program versions from the same specification is not easy. -The management of the Primary and the Alternate nodes needs some complicated protocols.	-Efficient for hardware and software fault tolerance. -Response time delays are very reduced.	-Tolerate only a set of known faults. -Cost more physical components.

5.5 Conclusion

In this chapter, we propose a Fault masking framework for Component-based cloud computing by using *Recovery Blocks*' scheme. This framework aims to construct Fault Masking cloud modules which have the ability of Self-fault detection and fault recovery. The proposed strategy is performed on a case study, time and space complexities are estimated and the efficiency of the proposed schemes is proved using *Liveness* verification by using model-checker.

Conclusion

Embedded systems are often installed in large areas. The application scenarios range from the domestic sector to military sector through the use of high-tech computing devices in various fields like transport, space, industry, public health, and so on. Indeed, the growing interest in the deployment of these applications with hundreds of computing operations has created frenzy in the research area in the related promising fields.

Cloud computing systems are largely used in embedded systems. They have attracted much interest recently for numerous Internet applications and services. However, reliability in cloud applications remains as a very crucial issue. This issue is especially difficult to deal with since cloud computing is a combination of hardware and software in a dynamic setting. The majority of these systems are *Safety-Critical*, either because they are at the heart of the behavior of a device or because they interact with the human life in various critical situations. In such cases, the assurance of their *Safety and Liveness* assumes a major objective. Thus, in order to grant them the ability to carry out their missions despite faults that occur, fault detection and fault masking is the way to accomplish this stated goal.

The aim from this thesis is the improvement of Cloud systems reliability. For this target, Recovery Blocks principle for fault tolerance is adopted. Two new schemes were proposed. They allow a uniform treatment of hardware and software faults in Cloud systems. The first scheme is dedicated for fault detection. It is based on the use of the Acceptance Test which can be presented as an internal temporal and logic audit. The component that has the ability of self-fault detection using the acceptance test is called The Fail-Silent component. It operates correctly and stops at just fault detection. The second scheme in this thesis is dedicated for Fault-Masking. It performs the fault detection by using the acceptance test and the forward recovery by using the try blocks. A Fault-Masking component is able to detect and recover from faults without perturbing the Cloud services. It can ensure the reliability and the availability of the Cloud services.

In this thesis, the construction of Fail-Silent and Fault-Masking components was well described. Furthermore, the proposed approaches were applied on the case study Fire Control System and time and space complexities were analyzed. A Safety and a

Liveness verification were provided using UPPAAL model-checker to prove the efficiency of the deduced models. Finally, comparative analysis was provided to highlights the main differences between the proposed approaches and the existent techniques.

The incorporation of Fault Detection and Fault-Masking capabilities in Cloud systems will provide high reliable Cloud nodes but with more complexity. The node that was only offer a service, from now, it must detect and recover from failure in addition to its initial main role. Beside, Forward recovery can only remove predictable and known errors from the system state. Hence, initiation of the recovery actions depends on the ability to accurately detect the occurrence of faults. Furthermore, the design of an effective acceptance test and a non fault-correlated set of try blocks is not an easy task because that depends on the application complexity. As an interesting future research direction, we would like to design and develop methods that can automatically provide efficient acceptance test and non-correlated try blocks.

Bibliography

[1] N. Fernando, S. W. Loke, W. Rahayu, “Mobile Cloud Computing: A Survey”, Future Generation Computer Systems, Jan 2013, Vol. 29, Issue. 1, pp. 84-106.

[2] P. A. Cox, “Mobile Cloud Computing: Devices, Trends, Issues and the enabling technologies”, DeveloperWorks, IBM Corporation, March 2011, pp. 1-9, ibm.com/developerWorks/.

[3] A. Ahmed, E. Ahmed, “A Survey on Mobile Edge Computing”, 10th IEEE International Conference on Intelligent Systems and Control (ISCO 2016), India, January, 2006.

[4] E. Ahmed, A.Gani, M-K. Khan, R. Bayya, S.U.Khan, “Seamless Application Execution in Mobile Cloud Computing: Motivation, Taxonomy, and Open Challenges”, Journal of Network and Computer Applications, Vol. 52, Pages 154-172, June 2015.

[5] A. C. Donald, S. A. Oli, L. Arockiam, “Mobile Cloud Security Issues and Challenges: A Perspective”, International Journal of Engineering and Innovative Technology (IJEIT), Vol.3, Issue.1, July 2013, pp. 401-406.

[6] RNewswire.org, http://www.reportlinker.com/, 2012.

[7] H. T. Dinh, C. Lee, D. Niyato, P. Wang, “A Survey of Mobile Cloud Computing: Architecture, Applications, and approaches”, Wireless Communications and Mobile Computing Journal, 2013, Vol.13, Issue.18, pp. 1587-1611.

[8] R. Buyya, C.S. Yeo, S.Venugopal, J. Broberg, and I.Brandic, “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility”, Future Generation Computer Systems, Vol. 25, No.6, 2009, pp. 599- 616.

[9] M. M. Hassan, A.-S. K. Pathan, E.-N. Huh, and J. Abawajy, “Emerging Sensor-Cloud Technology for Pervasive Services and Applications”, Guest Editorial of the Special Issue of International Journal of Distributed Sensor Networks, Volume 2014, Article ID 610106, DOI:10.1155/2014/610106, Hindawi Publishing Corporation, 2 pages.

[10] L. Youseff, M. Butrico, and D. Da Silva, “Toward a Unified Ontology of Cloud Computing”, Grid Computing Environments Workshop (GCE’08), 12-16 Nov. 2008, pp. 1-10.

[11] M. Ramachandran, "Component-based Development for Cloud Computing Architectures", Cloud Computing for Enterprise Architectures, Springer, 2011, pp. 91-114.

[12] G. Gossler and J. Sifakis, "Composition for Component-based Modeling", Science of Computer Programming, Vol.55, Issues.1-3, 2005, pp.161-183.

[13] D. Patcu and C. Sandru, "Towards component-based Software Engineering of Cloud Applications", Working IEEE/IFIP Conference on software architecture (WICSA), Venice, Italy, 2012, pp. 80-81.

[14]. A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-Time Components in BIP", IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), 2006, pp 3-12.

[15]. M. Bozga, V. Sfyrla, and J. Sifakis, "Modeling Synchronous Systems in BIP", ACM International Conference on Embedded Software (EMSOFT'09), 2009, pp. 77-86.

[16]. A. Basu, L. Mounier, M. Poulhies, J. Pulou, and J. Sifakis, "Using BIP for Modeling and Verification of Networked Systems –A Case Study on TinyOS-based Networks", IEEE International Symposium on Network Computing and Applications, NCA, Cambridge.MA, 2007, pp. 257-260.

[17] L. de Silva, R. Yan, F. Ingrand, R. Alami, and S. Bensalem, "A Verifiable and Correct-by-Construction Controller for Robots in Human Environments", Extended Abstract, ACM/IEEE International Conference on Human-Robot Interaction, New York, USA, 2015, pp.281- 281.

[18]S. Bensalem, L. de Silva, F. Ingrand, and R. Yan, "A Verifiable and Correct-by-Construction Controller for Robot Functional Levels", Journal of Software Engineering for Robotics, Vol.2, No.1, 2011, pp.1-19.

[19] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and T.H. Nguyen, "Toward a more dependable software architecture for autonomous robots", IEEE Robotics and Automation Magazine, Vol.16, No. 1, 2009, pp. 67-77.

[20] A. Avizienis, J.C. Laprie, B. Randell, and C.Laudwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, 2004, pp. 11-33.

[21] Y.S. Dai, D.Yang, J.Dongarra, and G.Zhang, “Cloud Service Reliability: Modeling and Analysis”, IEEE Pacific Rim International Symposium on Dependable Computing, 2009.

[98][22] A. Ganesh, M. Sandhya, and S. Shankar, “A study on fault tolerance methods in Cloud Computing”, IEEE International Advance Computing Conference (IACC), 2014, Gurgaon, pp. 844-849.

[23] D. P. Chandrashekar, Robust and Fault-Tolerant Scheduling for Scientific Workflows in Cloud Computing Environments. PhD dissertation, The University of Melbourne, August 2015. Available at: http://www.cloudbus.org/students/DeepakPhDThesis2015.pdf [last accessed: 27 October 2015].

[24] J. Guitart, M. Macias, K. Djemame, T. Kirkham, M. Jiang, and D. Armstrong, "Risk-Driven Proactive Fault-Tolerant Operation of IaaS Providers", 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), Volume: 1, 2-5 Dec., 2013, pp. 427-432.

[25] E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson.“A Survey of Rollback Recovery Protocols in Message-Passing Systems”, ACM Computing Surveys, Vol. 34, No. 3, 2002, pp. 375-408.

[26] H. Mansouri, N. Badache, M. Aliouat, and A.-S. K. Pathan, “A New Efficient Checkpointing Algorithm for Distributed Mobile Computing”, Journal of Control Engineering and Applied Informatics, Vol.17, No.2, 2015, pp. 43-54.

[27] R. Guerraoui and A. Schiper, “Fault-Tolerance by Replication in Distributed Systems”, International Conference on Reliable Software Technologies, Switzerland, 1996, pp. 38-57.

[28] A. Avizienis, “The Methodology of N-Version Programming”, Chapter 2 of Software Fault Tolerance. M. R. Lyu (ed.), Wiley, 1995, pp.23-46.

[29] B. Randel, “System Structure for Software Fault Tolerance”, ACM SIGPLAN Notices, Vol. 10, No.6, 1975, pp.437-449.

[30] J.J Horning, H.C Lauer, P.M. Melliar-Smith, and B. Randell. “A Program Structure For Error Detection and Recovery”, Operating Systems, International Symposium held at Rocquencourt, Springer Berlin Heidelberg, Vol.16, 1974, pp.171-187.

[31] M. Smara, M.Aliouat, Z.Aliouat, “Fault Detection in Component-based models: Using BIP Models”, the 12th International Symposium on Programming and Systems (ISPS), April 2015, pp.1-9.

[32] Mahendra.-K. Aharwir, Manish.-K.Ahirwar, U.Chourasia, “Anomaly Detection in the Services Provided by Multi Cloud Architectures; A Survey”, International Journal of Research in Engineering and Technology, Vol.3, Issue.9, Sep 2014, pp.196-200.

[33] A.Sari, “A Review of Anomaly Detection Systems in Cloud Network and Survey of Cloud Security Measures in Cloud Storage Applications”, Journal of Information Security, Vol.6, No.2, 2015, pp.142-154.

[34] I. M. Osman, H. T. Elshoush, “Alert Correlation in Collaborative Intelligent Intrusion Detection Systems- a Survey”, Applied Soft Computing Journal, Vol.11, Issue.7, 2011, pp.4349-4365.

[35] S. X. Wu, W. Banzhaf, “The use of Computational Intelligence in Intrusion Detection Systems; A Review”, Applied Soft Computing Journal, 2010, Vol.10, Issue. 1, pp.1-35.

[36] A.-S.K.Pathan. The State of the Art in Intrusion Prevention and Detection. ISBN 9781482203516, CRC Press, Taylor & Francis Group, USA, January 2014.

[37] J. Lee, S. Moskovics, L. Silacci, “A Survey of Intrusion Detection Analysis Methods”, 1999.

[38] S. Agrawal, J. Agrawal, “Survey on Anomaly Detection Using Data Mining Techniques”, International Conference on Knowledge and Intelligent Information and Engineering Systems, 2015, pp.708-713.

[39] J. Singh, M.J. Nene, “A survey on Machine Learning Techniques for Intrusion Detection Systems”, International Journal of Advanced Research in Computer and Communication Engineering, Vol. 2, Issue. 11, November 2013.

[40] P. Felber, X. Défago, R. Guerraoui, P. Oser, “Failure Detector as First Class Objects”, International Symposium on Distributed Objects and Applications (DOA), Computer Society, Edinburgh-Scotland, September 1999, pp. 132-141.

[41] M. Gander, M. Felderer, B. Katt, A. Tolbaru, R. Breu, A. Moschitti, “Anomaly Detection in the Cloud: Detecting Security Incidents via Machine Learning”, In Trustworthy Eternal Systems via Evolving Software, Data and Knowledge, Springer Berlin Heidelberg, 2013, pp. 103-116.

[42] N. F. Haq, M. Rafni, A.-R.Onik, F. M. Shah, Md. A. K. Hridoy, D. Md. Farid, "Application of Machine Learning Approaches in Intrusion Detection System: A Survey", International Journal of Advanced Research in Artificial Intelligence, 2015, Vol. 4, No. 3, pp.9-18.

[43] I. Gupta, T. D. Chandra, and G.S. Goldszmidt, "On scalable and efficient distributed failure detectors", ACM symposium on Principles of distributed computing, August 2001, pp. 170-179.

[44] C. Dobre. F. Pop, A. Costan, M. I. Andreica, V. Cristea , "Improving Network Traffic Anomaly Detection for Cloud Computing Services", The Ninth International Conference on Systems and Networks Communications ICSNC, 2014, pp.107-113.

[45] B.Bonakdarpour, M.Bozga, and G.Gossler, "A Theory of Fault Recovery for Component-Based Models", International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'12), Toronto, Canada, 2012, pp.314-328.

[46] G. Fan, H. Yu. L. Chen, and D. Liu, "Model-based Fault Detection Technique in Cloud Computing", IEEE Asia-Pacific Services Computing Conference (APSCC), Guilin, 2012, pp.249-256.

[47] T. Wang, W. Zhang, J. Wei, and H. Zhang, "Fault Detection for Cloud Computing Systems with Correlation Analysis", International Symposium on Integrated Network Management (IM), Ottawa, ON, May 2015, pp. 652-658.

[48] S.Barhuiya, Z.Papazachos, P.Kilpatrick, D.S. Nikolopoulos, "A Lightweight Tool for Anomaly Detection in Cloud Data Centers", International Conference on Cloud Computing and Services Science CLOSER, Lisbon, Portugal, 2015, pp. 343-351.

[49] T. Wang, J. Wei, F. Qin, W. Zhang, H. Zhong, T. Huang, "Detecting performance anomaly with correlation analysis for Internetware", Science China Information Sciences, Vol.56, Issue.8, 2013, pp. 1-15.

[50] C. Wang, K. Viswanthan, L. Choudur, V. Talwar, W. Satterfield, K. Schwan, "Statistical Techniques for Online Anomaly Detection in Data Centers", International Symposium on Integrated Network Management (IM), Duplin, May 2011, pp. 385-392.

[51] M. Kumar and R. Mathur, "Outlier Detection Based Fault-Detection Algorithm for Cloud Computing", International Conference for Convergence of Technology (I2CT), Pune, April 2014, pp.1-4.

[52] Y. A. Siva Prasad, G. R. Krishna, "Statistical Anomaly Detection Technique for Real Time Datasets", International Journal of Computer Trends and Technology (IJCTT), Dec 2013, Vol. 6, No. 2, pp. 89-94.

[53] R. Ranjan and G. Sahoo, "A New Clustering Approach for Anomaly Intrusion Detection", International Journal of Data Mining & Knowledge Management Process (IJDKP), Vol.4, No.2, March 2014, pp. 29-38.

[54] D. Singh, D. Patel, B. Borisaniya, C. Modi, "Collaborative IDS Framework for Cloud ", International Journal of Network Security, Vol. 18, No. 4, 2015, pp. 699-709.

[55] N. Pandeeswari, G. Kumar, "Anomaly Detection System in Cloud Environment Using Fuzzy Clustering Based ANN", Mobile Network and Applications, Springer US, August 2015.

[56] W. Sha, Y. Zhu, M. Chen, T. Huang, "Statistical Learning for Anomaly Detection in Cloud Server Systems: A Multi-Order Markov Chain Framework", IEEE transaction on Cloud Computing, January 2015, pp. 1-14.

[57] T. F. Ghanem, W. S. Elkilani, H. M. Abdul-kader, "A hybrid approach for efficient anomaly detection using metaheuristic methods", Journal of Advanced Research, Vol.6, Issue.4, July 2015, pp. 609-619.

[58] L. Arockiam and G. Francis E, "FTM-A Middle Layer Architecture for Fault Tolerance in Cloud Computing", Special Issue in International Journal of Computer Applications on Issues and Challenges in Networking, Intelligence and Computing Technologies (ICNICT, pp. 12-16), November 2012.

[59] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching network security analysis with time travel". ACM SIGCOMM Computer Communication Review, Vol. 38, No. 4, August, 2008, pp. 183-194.

[60] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors", IEEE Transactions on Computers, Vol. 51, No. 5, 2002, pp. 561-580.

[61] N. Hayashibara, X. Défago, R. Yared, T. Katayoma, "The φ accrual failure detector", Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004, pp. 66-78.

[62] A. Lavinia, C. Dobre, F. Pop, V. Cristea, "A Failure Detection for Large Scale Distributed Systems", International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), 2010.

[63] A. Arora and M.G. Gouda, “Closure and convergence: A foundation of fault tolerant computing”, IEEE Transactions on Software Engineering, Vol. 19, Issue 11, 1993, pp. 1015-1027.

[64] P. Alko and J.Mattila, “Software Fault Detection and Recovery in Critical Real-Time Systems: An Approach based on Loose Coupling”, Journal of Fusion Engineering and Design, Vol. 89, Issues. 9-10, 2014, pp. 2272-2277.

[65] A. Arora and S. S. Kulkarni, “Detectors and correctors: A theory of fault-tolerance components”, International Conference on Distributed Computing Systems (ICDCS), USA, 1998, pp. 436-443.

[66] B. Bonakdarpour, S. S. Kulkarni, and A. Arora, “Disassembling real-time fault tolerant programs”, ACM International Conference on Embedded Software (EMSOFT), New York, USA, 2008, pp. 169-178.

[67] M. Roohitavaf and S. Kulkarni, “Stabilization and Fault-Tolerance in Presence of Unchangeable Environment”, 2015, available at: http://arxiv.org/abs/1508.00864 [last accessed: 8 October, 2015].

[68] S. Bensalem, M. Bozga, T-H. Nguyen and J. Sifakis, “D-Finder: A Tool for Compositional Deadlock Detection and Verification”, Lecture Notes in Computer Science, Volume 5643, Springer Berlin Heidelberg, pp. 614-619.

[69] T.T. Pham and X. Défago, “Reliability Prediction for Component-based Systems: Incorporating Error Propagation Analysis and Different Execution Models”, International Conference on Quality Software (ICQS 2012), Shaanxi, 2012, pp. 106-115.

[70] T.T. Pham and X. Défago, “Reliability Prediction for Component-based Software Systems with Architectural-level Fault Tolerance Mechanisms”, International Conference on Availability, Reliability and Security (ARES 2013), Germany, 2013, pp 11-20.

[71] T.T. Pham, X. Défago, and Q.T.Huynh, “Reliability prediction for component-based software systems: Dealing with concurrent and propagating errors”, Journal of Science of Computer Programming, Vol. 97, Part. 4, Elsevier, 2015, pp. 426-457.

[72] P. Mathur and N. Nishchal, “Cloud Computig : New challenges to the entire computer industry”, International Conference on Parallel and Distributed Grid Computing (PDGC), Solan, October 2010, pp. 223-228.

[73] A. Ebnenasir and B.H.C. Cheng, "Pattern-Based Modeling and Analysis of Failsafe Fault-Tolerance in UML", IEEE High Assurance Systems Engineering Symposium (HASE'07), 2007, pp. 275-282.

[74] A. Ebnenasir and B. H. C Cheng, "Architecting Dependable Systems IV, chapter A Pattern-Based Approach for Modeling and Analyzing Error Recovery", Lecture Notes in Computer Science, Volume 4615, Springer Berlin Heidelberg, 2007, pp. 115-141.

[75] T. Xu, Z. Liu, T. Tang, W. Zheng, and L.Zhao, "Component Based Design of Fault Tolerant Devices in Cyber Physical System", International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW 2012), Guangdong, 2012, pp. 37-42.

[76] Y. Wu, G. Huang, H. Song, and Y. Zhang, "Model Driven Configuration of Fault Tolerance Solutions for Component-Based Software System", International Conference on Model Driven Engineering Languages and Systems (MODELS'12), Innsbruck, Austria, 2012, pp. 514-530.

[77] S. Tambe, A. Dabholkar, and A. Gokhale, "Fault-tolerance for Component-based Systems -An Automated Middleware Specialization Approach", International symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2009), Tokyo, 2009, pp. 47-54.

[78] M.Y. Jung and P. Kazanzides. "Run-time Safety Framework for Component-based Medical Robots". Medical Cyber Physical Systems Workshop (formerly known as HCMDSS (High Confidence Medical Devices, Software, and Systems)), CPSWeek, 2013, Philadelphia-USA, 2013.

[79] Z. Liu and M. Joseph, "Transformation of programs for fault-tolerance. Formal Aspects of Computing, Volume 4, Issue 5, 1992, pp. 442-469.

[80] Z. Liu and M. Joseph, "Specification and verification of fault-tolerance, timing, and scheduling", ACM Transactions on Programming Languages and Systems (TOPLAS), New York-USA, Vol.21, Issue. 1, 1999, pp. 46-89.

[81] M. Bozzano, A.Cimatti, M.Gario, and S.Tonetta, "Formal Design of Fault Detection and Identification Components Using Temporal Epistemic Logic", European Joint Conferences on Theory and Practice of Software (ETAPS, 2014), pp. 326-340.

[82] S. W. Loke, “Supporting ubiquitous sensor-cloudlets and context-cloudlets: Programming compositions of context-aware systems for mobile users”, Future Generation Computer Systems, Vol. 28, No. 4, 2012.

[83] T. Liu, F. Chen, Y. Ma, Y. Xie, “An energy-efficient task scheduling for mobile devices based on cloud assistant”, Future Generation Computer Systems, Vol. 61, 2016.

[84] Y.S. Chang, C.T. Fan, W.T. Lo, W.C.Hung, S.M.Yuan, “Mobile cloud-based depression diagnosis using an ontology and a Bayesian network”, Future Generation Computer Systems, Vol. 43-44, 2015.

[85] E. Ahmed, S. Khan, I. Yaqoob, A. Gani, F. Saleem, "Multi-Objective Optimization Model for Seamless Application Execution in Mobile Cloud Computing", Proceedings of 5th International Conference on Information and Communication Technologies, (ICICT'13), Karachi, Pakistan.

[86] Y. Saleem, F. Salim, M. H. Rehmani, “Resource Management in Mobile Sink based Wireless Sensor Networks through Cloud Computing”, Book Chapter at Resource Management in Mobile Computing Environments, Springer-Verlag Handbook, Volume 3, 2014, pp 439-459.

[87] M. Sookhak, H. Teleban, E. Ahmed, A. Gani, M-K Khan, “A Review on Remote Data Auditing in Single Cloud Server: Taxonomy and Open Issues”, Journal of Network and Computer Applications, Vol. 43, August 2014, pp. 121-141.

[88] S. Ghafoor, M. H Rehmani, S. Cho, and S. H. Park, “An Efficient Trajectory Design for Mobile Sink in a Wireless Sensor Network”, Elsevier Computers and Electrical Engineering Journal, Volume 40, Issue 7, Oct 2014, pp. 2089-2100.

[89] E. Ahmed, A. Gani, M. Sookhak, S.H. Ab Hamid, F. Xia, “Application Optimization in Mobile Cloud Computing: Motivation, Taxonomies, and Open Challenges”, Journal of Network and Computer Applications, Vol. 52, June 2015, pp. 52-68.

[90] Y. Saleem, F. Salim, M. H. Rehmani, “Integration of Cognitive Radio Sensor Networks and Cloud Computing: A Recent Trend, Cognitive Radio Sensor Networks: Applications, Architectures, and Challenges”, This book is part of the Advances in Wireless Technologies and Telecommunication (AWTT) series., Editors: Mubashir Husain Rehmani and YasirFaheem, IGI Global USA, 2014.

[91) U. Shaukat, E. Ahmed, Z. Anwar, F. Xia, “Cloudlet Deployment in Local Wireless

Area Networks, Motivation, Taxonomies, and Open Research Challenges", Journal of Network and Computer Applications, Vol. 62, 2016.

[92] M-H Rehmani, A.-S. K Pathan, "Emerging Communication Technologies based on Wireless Sensor Networks: Current Research and Future Applications, CRC Press, Taylor and Francis Group, USA, 2015.

[93] Ejaz Ahmed, Adnan Akhanzada, Md. Wahiduzaman, Abdullah Gani, SitiHafizahAb Hamid, RajkumarBuyya, Network-centric Performance Analysis of Runtime Application Migration in Mobile Cloud Computing, Simulation Modelling Practice and Theory, Vol. 50, January, 2015, pp. 42-56.

[94] M. N. Cheraghlou, A. K. Zadeh, M. Haghparast, " A survey of fault tolerance architecture in Cloud Computing", Journal of Network and Computer Applications, Vol. 16, 2016, pp.81-92.

[95] NIST Definition of Cloud Computing v15, http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf.

[96] Q. Zhang, L. Cheng and R. Boutaba , "Cloud Computing: state-of-the-art and research challenges", Journal of Internet Services and Applications, May 2010, Vol. 1, Issue.1, pp. 7-18.

[97] P. K. Patra, H. Singh, G. Singh, " Fault Tolerance Techniques and Comparative Implementation in Cloud Computing", International Journal of Computer Applications, 2013, Vol. 64, No. 14, pp. 37-41.

[98] A.Ganesh, M.Sandhya and S. Shankar, "A Study of Fault Tolerance Methods in Cloud Computing", IEEE International Advance Computing Conference (IACC), February 2014, India, pp. 844-849.

[99] J. L. M. Humphrey, Y.-W. Cheah, Y. Ryu, "Fault Tolerance and scaling in e-Science Cloud Applications: Observations from the Continuing Development of MODISAzure", IEEE e-Science Conference, 2010, Australia, pp. 1-8.

[100] Y. S. Dai, B. Yang, J. Dongarra, G. Zhang, "Cloud Service Reliability: Modeling and Analysis" , IEEE Pacific Rim International Symposium on Dependable Computing, 2009, China , pp.1-17.

[101] S. K. Chamoli, D. S. Rana, "Fault Tolerance and Load Balancing algorithm in Cloud Computing : a survey", International Journal of Advanced Research in Computer and Communication Engineering, Vol. 4, Issue. 7, 2015, pp. 92-96.

[102] Z. Amin , N. Sethi, H. Singh, "Review on Fault tolerance techniques in Cloud Computing", International Journal of Computer Applications, Vol. 116, No.18, 2015, pp. 11-17.

[103] S. M. Hosseini, M. G. Arani, "Fault Tolerance Techniques in Cloud Storage : a Survey", International Journal of Database Theory and Application , Vol. 8, No. 4, 2015, pp. 183-190.

[104] J. –C. Laprie, " Dependability-its attributes, impairments and means", in B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, editors, Predictability Dependable Computing Systems, ESPRIT Basic Research Series, Springer, 1995, pp. 3-24.

[105] A. Avizienis, "Fault-Tolerant Systems", IEEE Transactions on Computers, Vol.25, No.12, 1976, pp. 1304-1312.

[106] P. Jalote, "Fault-Tolerance in Distributed Systems", 1998, by Prentice Hall, Inc. A Pearson Education Company, Upper Saddle River, New Jersey 07458.

[107] B. Randel, "System Structure for Software Fault Tolerance", ACM SIGPLAN Notices, Vol. 10, No.6, 1975, pp.437-449.

[108] J.J Horning, H.C Lauer, P.M. Melliar-Smith, and B. Randell. "A Program Structure For Error Detection and Recovery", Operating Systems, International Symposium held at Rocquencourt, Springer Berlin Heidelberg, Vol.16, 1974, pp.171-187.

[109] K. H. Kim, H. O. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications", IEEE Transactions on Computers, Vol. 38, No.5, 1989, pp. 626-636.

[110] M. Abadi, L. Lamport, "The existence of refinement mappings", Theoritical Computer Science, Vol. 82, Issue. 2, May 1991, pp. 253-284.

[111] H. P. Zima, A. Nikora, "Fault Tolerance", from Encyclopedia of Parallel Computing, Springer US, 2011, pp.645-658.

[112] L. Lamport, "Proving the correctness of multiprocess programs", IEEE Transaction in Software Engineering, USA, Vol. 3, No. 2, March 1977, pp. 125-143.

[113] B. Alpern, F.B. Schneider, "Defining Liveness", 1985, Inf Process Left, Vol.21, No. 4, pp. 181-185.

[114] Y. Zhang, Z. Zheng, M. R. Lyn, BFT Cloud, "A Byzantine Fault Tolerance Framework for Voluntary-Ressource Cloud Computing", IEEE International Conference on Cloud Computing, Washington, US, 2011, pp. 444-451.

[115] Z. Jia, R. Chen, X. Xing, J. Xu, Y. Xie, "SFDCloud : top-k Service faults diagnosis in Cloud Computing", Automated Software Engineering, 2014, Vol. 21, Issue. 4, pp. 461-488.

[116] S. K. Choi, K. Chung, H. Yu, "Fault Tolerance and QoS Scheduling using CAN in mobile social Cloud Computing", Cluster Computing Journal, 2014.

[117] D. Jing, H. Scott, H. Yunghsiang, D. Julia, "Fault-Tolerant and Reliable Computation in Cloud Computing ", Globecom Workshops, 2010, pp. 1601-1605.

[118] D. Sun, G. Chang, C. Miao, X. Wang, "Analyzing, Modeling and Evaluating Dynamic Adaptive Fault Tolerance strategies in Cloud computing environments", Journal of Supercomputing 2013, Vol. 66, Issue. 1, pp. 193-228.

[119] H. Yi, G. Bin, W. Fengyu, "Cloud model-based Security-aware and Fault Tolerant Job Scheduling for computing Grid", ChinaGrid, 2010, pp. 25- 30.

[120] S. Malik, F. Huet, "Adaptive Faut Tolerance in Real-time Cloud Computing", IEEE World Congress on services, Washington, USA, 2011, pp. 280-287.

[121] Y. Wu, Y. Yuan, G. Yang, W. Zheng, "An adaptive task-level fault tolerant approach to Grid", Journal of SuperComputing, 2010, Vol. 51, Issue. 2, pp. 97- 114.

[122] J. B. Lim, J. M. Gip, K. S. Chung, J. Kang, D. Lee, H. Yu, "Gossip Membership Management with social graphs for Byzantine Fault Tolerance in Clouds", Network and Parallel Computing, 2014, LNCS 8707, pp. 321-332.

[123] K. H. Kim, "Distributed Execution of Recovery Blocks: an Approach to uniform Treatment of Hardware and Software faults", Proceeding Fourth International Conference on Distributed Computing Systems, 1984, pp. 526-532.

[124] K. H. Kim, H. O. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications", IEEE Transactions on Computers, Vol. 38, No.5, 1989, pp. 626-636.

[125] K. H. Kim, "The distributed Recovery Block Scheme" in M. R. Lyu (ed.), Software Fault Tolerance, New York: John Wiley & Sons, 1995, pp. 189-209.

[126] L. L. Pullum, "Software Fault Tolerance Techniques and Implementations", Artech house, Inc. Norwood, MA, USA, 2001. ISBN: 1-58053-137-7.

[127] B. Randel, “System Structure for Software Fault Tolerance”, ACM SIGPLAN Notices, Vol. 10, No.6, 1975, pp.437-449.

[128] J.J Horning, H.C Lauer, P.M. Melliar-Smith, and B. Randell. “A Program Structure For Error Detection and Recovery”, Operating Systems, International Symposium held at Rocquencourt, Springer Berlin Heidelberg, Vol.16, 1974, pp.171-187.

[129] M. Abadi, L. Lamport, “The existence of refinement mappings”, Theoritical Computer Science, Vol. 82, Issue. 2, May 1991, pp. 253-284.

[130] J. C. Laprie, “Dependability, Basic Concepts and Terminology”, Number 5 of Dependable Computing and Fault-Tolerant System, Springer-Verlag, 1992.

[131] E. Bauer, R. Adams, “Reliability and Availability of Cloud Computing”, John Wiley & Sons, July 2012, DOI: 10.1002/9781118393994. Ch1.

[132] K. H. Kim, “The distributer Recovery Block Scheme”, Chapter 8 in Software Fault Tolerance, Edited by Lyu, 1995, John Wiley & Sons Ltd.

[133] M. Stanisavijevic, A. Schmid, Y. Leblebici, “Reliability, Faults and Fault Tolerance”, Chapter 2 from “Reliability of Nonoscale Circuits and Systems Methodologies and Circuits Architectures”, Hardcover, 195p, Springer, 2011.

[134] A. Avizienis, J.-C. Laprie, B. Randell, C. Landweht, “Basic Concepts and Taxonomy of Dependable and secure Computing”, IEEE Transactions on Dependable and Secure Computing, Vol.1, No.1, March 2004, pp.1-23.

[135] J. Liu, J. Zhou, R. Buyya, “Software Rejuvenation based Fault Tolerance Scheme for Cloud Applications ”, IEEE International Conference on Cloud Computing, 2015,New York-USA, pp. 1115-1118.

[136] a. Garg, S. Bagga, “An Autonomic Approach for Fault Tolerance using Scaling, Replication and Monitoring in Cloud Computing”, International Conference on MOOCs, Innovation and Technology in Education (MITE), 2015,Amritsar-Punjab, pp. 129-134.

[137] P. Garraghan, P. Townend, J.Xu, X.Yang, P. Sui, “Using Byzantine Fault-Tolerance to improve Dependability in Federated Cloud Computing ”, International Journal of Software and Informatics, Vol. 7, Issue. 2, 2013, pp. 221-237.

[138] M.O. Alannsary, J. Tian, "Measurement and Prediction of SaaS Relability in the Cloud ", IEEE International Conference on Software Quality, Reliability and Security companion, 2016, Vienna-Austria, pp. 123-130.

[140] B. Mohamed, M. Kiran, I.-U. Awan, K. M. Maiyama, "Optimizing Fault Tolerance in Real-Time Cloud Computing IaaS Environment ", IEEE international Conference o Future Internet of Things and Cloud, 2016, Vienna-Austria, pp. 363-370.

[141] C. M. Reddy, N. Nalini, "FT2R2Cloud: Faault Tolerance using time-out and retransmission of requests for Cloud applications", in International Conference on Advances in Electronics, Computers and Communications (ICAECC), 2014.

[142] Z. Zheng, M. R. Lyu, "Selecting an Optimal Fault tolerance strategy for reliable service-oriented systems with local and global constraints", IEEE Transaction-compt. 64(1), 219-232, 2015.

[143] G. Chen et al. "A Lightweight Software Fault Tolerance System in the Cloud Environment", Concurrency Comput. Pract. Experience, 27(12), 2015, 2982-2998.

[144] A. Moghtadaeipour, R. Tavoli, "A New Approach to Improve Load Balancing for Increasing Fault tolerance and Decreasing Energy Consumption in Cloud Computing", in 2nd International Conference on Knowledge-based Engineering and innovation (KBEI), 2015.

[145] S. Shankland, "Amazon suffers u.s", Outage on Friday- Retrieved on December 15, 2013.

[146] B. Winterford , "Stress tests rain on Amazon's Cloud", IT News, Retrieved on December 15, 2013, August 2009.

[147] Z. Zheng, T. Zhan, M. Lyu and T. King, "Component Ranking for Fault Tolerant Cloud Applications", IEEE Transactions on Services Computing, Issue 4.Vol.5, Fourth Quarter 2012, PP. 540-550, DOI 10.1109/TSC.2011.42.

[148] F. Khomh, "On Improving the Dependability of Cloud Applications with Fault Tolerance", WISCA 2014, Sydney, Australia, Vol. 2, 2 pages.

[149] ITProPortal, "ITProPortal.com: 24/7 Tech Commentary & Analysis", 2012 [Online]. Available : http://www.itproportal.com/.

[150] K.Bilal, O, Khalid, R.Malik, M.Usman and S.Khan, "Fault Tolerance in the Cloud", no. ITProPortal, pp. 1-13, 2012.

Appendix A

Theorem1: A *Fail-Silent* atomic component, $FS_B = (Q,\ P,\ \rightarrow,\ X,\ q^0,\ Hs,\ AT_B)$can insure *Safety* property using the *acceptance test* AT_B. The AT can validate final results and decide their correctness. In the case of Fault Detection, the atomic component FS_Bwill pass to a Deadlock state till failure correction.

Proof. In order to prove Theorem 1, we will use demonstration by the absurdum. Let$B = (Q,\ P,\ \rightarrow,\ X,\ q^0)$be an atomic component which has no acceptance test.

At the moment t, initial variable value of B isv_0. The expected results are: v_0 at the moment$t$,v_1 at the moment t',v_2 at t'' and v_3 at t''' where:$t < t' < t'' < t'''$. The atomic component B will be infected by a fault at the moment $t''$and B has no acceptance test which can validate its final results and detect the failures.$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$are the correct and expected final results of the atomic component B.

At the moment$t'(t' > t)$,$v_0 - Beh_{Cy}(B) \rightarrow v_1$. The atomic component *B* produces the correct result and it is considered that it is correct (c).

At the moment $t''\ (t'' > t' > t)$,$v_1 - Beh_B(Cy) \rightarrow v_2'$. At that moment, the atomic component *B* offers the result v_2' which is different from the expected value v_2 because of a failure that infects the atomic component *B*. *B* offers a failed result ($v_2' \neq v_2$)but it earns its execution.

Hence, the atomic component continues its execution and at the moment $t'''(t''' > t'' > t')$, $v_2' - Beh_{Cy}(B) \rightarrow v_3'$, ($v_3' \neq v_3$). We can see that since the first failure that happens at the moment t'', the component *B* has deviated from its exact behavior and it continues to produce faulty results that may be transferred to other components without failure detection. This kind of failure is very difficult to deal with because it may include the entire system in Byzantine failure which threatens the *Safety* property. Therefore, because of the absence of the AT, the *Safety* property will be violated.

We can see clearly that the *Safety* property will be ensured using the AT. If *B*'s outcomes do not satisfy *B*'s AT, *B* will be considered as *failed* and it will be blocked immediately. Using the AT, we can ensure that the values which are used in the entire system are the correct ones. In other way, we have ensured *Safety* property by adopting the *Fail-Silent* behavior.

Lemma1. A rendezvous connector, $\beta = \{p_{i=1..m}, p_i \subset B_i\}$ which involves a set of Fail-Silent atomic component is Fail-Silent rendezvous connector: $FS_{Rendezvous(\beta)} = \{p_{i=1..m}, p_i \subset FS_{Bi=1..m}\}$.

Proof. We will demonstrate by the absurdum that:

$if\ \exists\ B_{1\leq i\leq m} \in rendezvous\ connector\ \beta$; and$B_i$ is not Fail-Silent, then the rendezvous connector β is not Fail-Silent either.

Let $\beta = \{p_{i=1..m}, p_i \subset B_i\}$ be a rendezvous connector which contains m atomic components. All The atomic components are Fail-Silent except B_j. The atomic components involved in $\beta$are synchronized to exchange information via this connector. Each atomic component $FS_{Bi\neq j}$ is Fail-Silent atomic component, and it has an AT that can validate its outputs and decide their correctness. In such a scenario, in case of detection of any failure, all components will be blocked immediately except the atomic component B_j (that is not Fail-Silent) and thus, B_j will still be executed and fail. The initial value of B_j is $v_j^0$and the expected results of B_j are: $v_j^1 \rightarrow v_j^2 \rightarrow v_j^3$at the moments: t', t'' and t'''such that ($t' < t'' < t'''$). The initial value of the entire group of atomic components is V_0 and the expected values of the entire group are: $V_1 \rightarrow V_2 \rightarrow V_3$ at the moments: t', t'' and t''' where: ($t' < t'' < t'''$).

At the moment t', B_j returns the value $v_j^1$and it exchangesv_j^1 with the other atomic components in the same group to produce the global value V_1.

At the moment t'', B_j will be infected by a failure but it earns execution and returns the value $v_j^{2'}$ instead of v_j^2. This value will be transferred to other atomic components and the global result of the entire group will be V_2' instead of V_2 .

At the moment t''', faulty B_j will produce the value $v_j^{3'}$ instead of v_j^3. This value will be transferred to the others atomic components and the global result of the group will be V_3' instead of V_3.

If the failure is not be detected by another Fail-Silent atomic component, the entire group will earn execution; hence, would show failed behavior and as a consequence, would produce incorrect results. As noted earlier, this type of failure is Byzantine failure which disrupts the *Safety* of the composite component. Hence, we can see that if there exist even a single atomic component involved in the rendezvous connector that is not Fail-Silent, the rendezvous connector will not be considered as Fail-Silent. From

this, we conclude that to construct a Fail-Silent rendezvous connector, all its involved atomic components must be Fail-Silent as well. $FS_{Rendezvous(\beta)} = \{FS_{B1}, FS_{B2}, \dots, FS_{Bm}\}$.

Lemma 2. A broadcast connector $\beta = \{p_{i=1..m}, p_i \subset B_i \; and \; B_k \; is \; Broadcast \; initiator\}$ which involve at least a Fail-Silent broadcast initiator is a Fail-Silent broadcast connector: $FS_{Broadcast(\beta)} = \{FS_{Bk}, B2, \dots, B_m\}$..

Proof. Let suppose that $B = \gamma(B_1, B_2, B_3)$ is a composite component and γ is a broadcast connector that relies the set of atomic components $B_1, B_2 \; and \; B_3$ where B_1 is the broadcast initiator. Let's suppose the following scenario, in which $B_1, B_2 \; and \; B_3$ are not Fail-Silent components.

At the moment t, the atomic components $B_1, B_2 \; and \; B_3$ have a correct behavior and no faults are detected.

If we suppose that at the moment $t' > t$, both $B_2 \; and \; B_3$ failed. They would stay in operation but produce wrong results without affecting the broadcast initiator B_1.

We take another scenario when the broadcast initiator B_1 fails first. In this case B_1 will produce and send a wrong result to $B_2 \; and \; B_3$.

We can see that the failure of B_1 affect immediately the others atomic component involved in the same broadcast connector (i.e., $B_2 \; and \; B_3$). But, if $B_2 \; or \; B_3$ fails, they do not perturb the broadcast initiator B_1 . Hence, if B_1 is not Fail-Silent, Safety of the entire composite component can be violated.

We can conclude that to construct a Fail-Silent connector γ, at least the broadcast initiator must be Fail-Silent as well: $FS_{Broadcast(\beta)} = \{FS_{Bk}, B2, \dots, B_m\}$.

Lemma3. A composite component which contains *Fail-Silent* connectors (rendezvous and/or broadcast) is *Fail-Silent* composite component. The $\omega - regular$ expression of a *Fail-Silent* composite component is: $[(C \; / \; A / \; F)^* F]$.

Proof. Let $B = \gamma(B_1, \; B_2 \dots, \; B_n)$ be a composite component which is composed of a set atomic components relied on by a set of *Fail-Silent* connectors $\gamma = \{FS_{\beta 1}, \; FS_{\beta 2}, \dots, \; FS_{\beta l}\}$.

At moment t, the behavior of all the connectors is correct: $\forall\, FS_{\beta i=1..l} \subset B;\ FS_{\beta i}\ is\ correct(c)$ thus the values used on the composite component B are the correct one, C.

At moment $t' > t$; some connectors have a correct or accepted behavior and other connectors have failed behavior and thus it will remain at a Deadlock state because of a failure detection: $\forall\, FS_{\beta i=1..l} \subset B;\ FS_{\beta i}\ is\ correct(c)\ or\ accepted(a)\ or\ failed(f)$ state. As some of the connectors are still operating, the entire behavior of the composite component would be correct C or accepted A or failed F: (C/A/F).

At the moment $t''' > \cdots > t' > t$; all the *Fail-Silent* connectors are failed and they will be in Deadlock state because of failure detection: $\forall\, FS_{\beta i=1..l} \subset B;\ FS_{\beta i}\ is\ failed(f)$, thus the entire composite component will be considered as failed (F) and will pass to a Deadlock state.

We can conclude that the behavior of the composite component is$[(C/A\ /F)^{*}F]$. A composite component which contains *Fail-Silent* connectors is therefore *Fail-Silent* composite component.

Theorem 2. A composite component which is composed of a set of Fail-Silent atomic component is Fail-Silent composite component: $FS_B = \gamma(FS_{B1},\ FS_{B2}, \ldots,\ FS_{Bn})$.

Proof. Let $B = \gamma(FS_{B1}\ ,\ FS_{B2}, \ldots,\ FS_{Bn})$ be a composite component. B is composed of the set of Fail-Silent atomic components $FS_{Bi=1..n}$ which are connected by the set of connectors $\gamma = \{\beta_{i=1..l},\ /\ \beta_i\ is\ rendezvous\ or\ broadcast\}$ Each connector $\beta_{i=1..l} = \{p_{i=1..m},\ p_i \subset FS_{Bi=1..m}\}$is a rendezvous or broadcast connector. According to *lemma.1* and *lemma.2*, β_i is Fail-Silent connector and we write$FS_{\beta i=1..l} = \{p_{i=1..m},\ p_i \subset FS_{Bi=1..m}\}$.

On the other hand, we have the composite component $B = \gamma(FS_{B1}\ ,\ FS_{B2}, \ldots,\ FS_{Bn})$ which is composed of a set of Fail-Silent atomic components relied on by a set of Fail-Silent connectors $\gamma = \{FS_{\beta i=1..l}, /\ \beta_i\ is\ rendezvous\ or\ broadcast\}$. According to *Lemma3*, the composite component B is Fail-Silent and we write: $FS_B = \gamma(FS_{B1},\ FS_{B2}, \ldots,\ FS_{Bn})$. Therefore, we conclude that a composite component which is constructed by a set of Fail-Silent atomic components is Fail-Silent as well.

Appendix B

Lemma 1: A Fail-Silent atomic component that use an Alternate behavior ($i.e., Beh_{Alternate}$) can ensure *Liveness* property even in the presence of faults.

Proof: let suppose the Fail-Silent atomic component $FS_B = (Q,\ P, Beh_{primary},\ X,\ AT_B)$.

FS_B has an acceptance test that can validate $B's$ internal behavior. We have seen before that the ω-regular expression of any *Fail-Silent* atomic component is:$(c/a\ )^*f$. Therefore, at the moment of a failure detection on the Primary behavior, B will be blocked immediately and the component will enter in a deadlock state. It is very clear that the use of a *Fail-Silent* atomic component that has only one behavior leads to a deadlock state in the case of fault detection. Hence, in order to avoid this case an Alternate behavior for the atomic components have to be used. It helps the component to skip the abnormal state without entering in the deadlock state. We can say that the use of an Alternate behavior in the atomic component can save *Liveness* of the atomic component.

Theorem 1: the use of an Alternate behavior in the *Fail-Silent* atomic component can produces a *Fault-Masking* component that can ensure both *Safety* and *Liveness* properties in the same time. The ω-regular expression of the *Masking* atomic component is$[(c/a)^*f\ r^*\ (c/a)]^\omega$.

Proof: Let suppose that we have a Fail-Silent atomic component $FS_B = (Q, P, Beh_{primary}, Beh_{Alternate},\ X,\ AT_B)$. It uses the acceptance test AT for self-checking of final outputs. The ω-regular expression of the Fail-Silent B in this case is : $(c/a\ )^*f$. At the moment of failure detection, FS_B will be blocked and thereby the *Liveness* of the component is violated. At that moment FS_B will perform its Alternate behavior. This last will perform a forward recovery task to save Liveness proprties. The ω-regular expression of the Alternate behavior in the Fail-Silent atomic component is $r^*(c/a)$. Therefore, the ω-regular expression produced from the conjunction of the Primary and the Alternate behaviors is $[(c/a\ )^*fr^*(c/a)]^\omega$. We can observe that it is

the same expression of the Fault-Masking component. Finally, we can conclude that the use of an Alternate behavior in a Fail-Silent atomic component provide a Fault-Masking component that ensure Safety and Liveness properties in the same time.

Lemma2: Let $B = \gamma(B_1 \dots B_n)$ a composite component. To construct a Fault-Masking rendezvous connector γ, all the atomic components $B_1 \dots B_n$ involved in it must be Fault- *Masking* as well. $FM_{Rendezvous(\gamma)} = \{FM_{B1}, FM_{B2}, \dots, FM_{Bn}\}$.

Proof, Let suppose that $B = \gamma(B_1, B_2, B_3)$ is a composite component and γ is a rendezvous connector that relies the set of atomic components $B_1, B_2 \; and \; B_3$. Both $B_1 \; and \; B_2$ are Fault-*Masking* components whereas B_3 is only *Fail-Silent*. $B_1, B_2 \; and \; B_3$ are synchronized to exchange information via the connector γ. Let's suppose the following scenario:

At the moment t, the components $B_1, B_2 \; and \; B_3$ have a correct behavior and no faults are detected.

At the moment $t' > t$, the atomic component B_1 is failed. It stops operating immediately and perform a forward recovery using its Alternate behavior. After a period of time, B_1 will reach a stable state and achieve the rendezvous connection with $B_2 \; and \; B_3$.

At the moment $t'' > t'$, the atomic component B_2 is failed. Because it is Fault-*Masking* component, it can detect and tolerate the fault after a period of time and earn the rendezvous connection with $B_1 \; and \; B_3$.

At the moment $t''' > t''$, the atomic component B_3 detects a failure. It stops immediately the operation and stays in a deadlock state. Because it has no mechanism for fault recovery, the atomic component B_3 still blocked and it would enter the entire atomic components involved in the rendezvous connector γ (i.e.,$B_1 \; and \; B_2$) in a deadlock state even though they are *Fault-Masking* and hence violate the *Liveness* of the entire composite component.

From this scenario, we can conclude that to construct a *Fault-Masking* composite component $B = \gamma(B_1, \dots B_n)$ with a rendezvous connector γ; all its inner atomic components must be *Fault- Masking* as well: $FM_{Rendezvous(\gamma)} = \{FM_{B1}, FM_{B2}, \dots, FM_{Bn}\}$.

Lemma3: Let $B = \gamma(B_k \dots B_n)$ a composite component where γ is a broadcast connector. To construct a Fault-M*asking broadcast* connector γ, at least the broadcast initiator B_k must be *Fault-Masking*: $FM_{Broadcast(\gamma)} = \{FM_{Bk}, B2, \dots, B_m\}$.

Proof, Let suppose that $B = \gamma(B_1, B_2, B_3)$ is a composite component and γ is a broadcast connector that relies the set $B_1, B_2\ and\ B_3$ where B_1 is the broadcast initiator. Let's suppose the following scenario in which $B_1, B_2\ and\ B_3$ are only *Fail-Silent* components and not Fault-Masking.

At the moment t, the atomic components $B_1, B_2\ and\ B_3$ have a correct behavior and no faults are detected.

At the moment $t' > t$, the atomic component B_2 is failed. It stops operating immediately and stay in a deadlock state. But the entire composite component stills in operation because the fault cannot be transferred to $B_1\ or\ B_3$.

At the moment $t'' > t'$, the atomic component B_3 is failed. It will be blocked too without perturbing the entire composite component.

We can say that the failure of $B_2\ and\ B_3$ has no effect on the entire composite component while the broadcast initiator is not failed.

Now, let's take another scenario when the broadcast initiator B_1 fails first. In this case B_1 will stay in a deadlock state and $B_2\ and\ B_3$ will even enter in a deadlock state waiting a broadcast message from B_1. We can see that if B_1 fails, the others atomic component involved in the same broadcast connector will be infected by the failure. But, if $B_2\ or\ B_3$ fails it does not affect the broadcast initiator B_1. Hence, if B_1 is only *Fail-Silent* and not *Fault-Masking component* , *Liveness* of the entire composite component can be violated.

We can conclude that to construct a M*asking broadcast* connector γ, at least the broadcast initiator must be *Fault-Masking*: $FM_{Broadcast(\gamma)} = \{FM_{Bk}, B2, \dots, B_m\}$.

Theorem 2. A composite component that is composed of a set of *Fault-Masking* atomic component is *Fault-Masking* composite component: $FM_B = \gamma(FM_{B1},\ FM_{B2}, \dots,\ FM_{Bn})$.

Proof. Let $B = \gamma(B_1,\ B_2, B_3)$ be a composite component which is composed of a set of atomic components relied on by a set of *Fault-Masking* connectors $\gamma = \{FM_{\beta 1},\ FM_{\beta 2}\}$.

At the moment t, the behavior of $B_1,\ B_2, and B_3$ is correct (c/a) thus we can say that entire composite component B is correct, C/A.

At the moment $t' >> t$; the atomic component B_1, B_2 and B_3 have a failed behavior (f). the entire behavior of the composite component will be failed F.

At the moment $t'' > t'$; all the *Fault-Masking* atomic component B_1, B_2 and B_3 perform a recovery task to reach a stable state after the fault detection. Thus the entire composite component will be considered in a recovery state R.

At the moment $t''' > t''$; the component B_1, B_2 and B_3 recover from the failure and reach a correct or acceptable behavior(c/a) therefore the behavior of the composite component is (C/A) as well.

We can conclude that the behavior of the composite component is $[(C/A)^* F R(C/A)]$. We can see that the the ω-regular expression of the composite component is the same of the Fault- Masking component. We conclude that a composite component which is composed of a set of *Fault- Masking atomic components* is *Fault-Masking* composite component $FM_B = \gamma(FM_{B1}, FM_{B2}, \dots, FM_{Bn})$.