ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FACULTY OF INFORMATICS

Component-Based Fault Tolerant Distributed Context Data Management for Context Aware Systems

BY

NEBYOU AZANAW

OCTOBER, 2009

ADDIS ABABA

Component-Based Fault Tolerant Distributed Context Data Management for Context Aware Systems

BY

NEBYOU AZANAW

A Thesis submitted to the School of Graduate Studies of Addis Ababa University in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

OCTOBER, 2009

ADDIS ABABA

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
FACULTY OF INFORMATICS

Component-Based Fault Tolerant Distributed Context Data Management for Context Aware Systems

BY

NEBYOU AZANAW

ADVISORS:

Girma Berhe (PhD)

Dejene Ejigu (PhD)

EXAMINING BOARD

1. Dejene Ejigu (PhD), Advisor _____
2. Dida Midekso (PhD) _____

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to sincerely thank all those people who have made this thesis possible. First and for most, I would like to express my heartfelt gratitude to my advisors, Dr. Girma Berhe and Dr. Dejene Ejigu, for their constant guidance, encouragement, comments and suggestion in the preparation of this thesis. Their incredible encouragement and support made me a better student in spite of myself.

I also extend my profound gratitude to all fellow classmates and Addis Ababa University Libraries staff members for their encouragement and cooperation in the course of my graduate study. I would like to thank Addis Ababa University, especially the Informatics Faculty, for allotting research budget to cover my expenses in the course of the research.

I equally wish to extend my appreciation and thanks to my friends Tewodros Befekadu, Netsanet Animut for their friendly support.

Finally, I would like to thank my family for their moral support and inspirational encouragements throughout my study.

Abstract

Context awareness allows applications to adapt themselves to their changing user circumstances or changing computing environment in order to better suit the needs of the user. Context information that supports such adaptations is provided by the underlying context management infrastructure, which gathers, pre-processes, manages and provides context information from a variety of context information sources. The distributed nature of the infrastructure is prone to failures which have a negative impact on the usability of context aware applications. In this thesis, we identified the shortcomings of existing works in relation to fault tolerance in distributed context data management system. The shortcomings are lack of adequate consideration for faults that arise from the distributed software components and lack of full-fledged fault management approach.

In order to overcome these limitations, we proposed a fault management architecture that wraps the software pieces in components in order to administer as component architecture. The architecture is composed of the manager system and the managed system. The manager system consists of fault manager component and supplementary modules. Fault manager component is responsible for monitoring components, detecting faulty components and repairing the faulty components. Supplementary modules provide the necessary functionalities to the fault manger to carry out its activities. On the managed system, the core components are context pre-processing component that collects and aggregates context data from potential context data sources like sensors, context storing component that stores context data in a database and context reasoning and decision component that performs ontology-supported context reasoning.

To prove the validity of the proposed architecture, we developed a prototype that implements the major components of the proposed architecture. Faults have been introduced manually into the context management system and results obtained from our experiment show that the fault manager allows the system to automatically and transparently recover from the induced failures.

Keywords: Context Aware Systems, Fault Tolerance, Component Based Computing, Distributed Context Data Management.

1. Introduction

1.1. Background

With the appearance and penetration of mobile devices such as notebooks, PDAs, and smart phones, pervasive (or ubiquitous) systems are becoming increasingly popular these days. In pervasive computing environment, networked computers are embedded in everyday objects and places; Sensors and actuators maintain rich connection between the physical and virtual worlds and computing is spread through the environment and yet gracefully integrates with it [1].

S. Ahmed [2] classifies pervasive computing environment into three categories:

- Fixed-infrastructure based: In fixed-infrastructure based pervasive computing environment, devices get support from powerful servers, access points, etc. These powerful devices handle complex computation and communication on behalf of the mobile devices. Smart/active spaces are examples of this type of environment.
- Infrastructure-less (pure ad-hoc): In an infrastructure-less pervasive computing environment, mobile devices are responsible for communication and computations. They communicate with each other using a short-range wireless network. An ad-hoc network in a stadium, conference room, and airport is example of this type of network.
- Hybrid Infrastructure: There are also hybrid networks that have some support from the infrastructure. Ad-hoc networks with the ability to communicate with the access point fall into this category.

One field in the wide range of pervasive computing is the so called context aware systems. Context aware systems are able to adapt their operations to the current context without explicit user interventions using context information derived from a collection of various information sources such as sensors. While the raw sensor data may be sufficient for some applications, many require the raw data to be transformed or fused with other sensor data before it is useful [3]. The context management system is responsible for collecting raw data from different sensors, process the data into context information, and distribute the information to different applications. The context management system can be deployed as a set of software components (services) to handle the processing of context information in a distributed manner i.e. either in infrastructure based or infrastructure-less approach.

To ensure the correct operation of context aware applications, the provision of context information to the applications must be reliable. i.e. context information must be processed properly and made available to the applications. Applications which can't be trusted or operated reliably will not be used [4]. The failure of software components to provide and process context information will have a negative impact on the ability of context aware applications to adapt their behavior and adjust to changing computing environments or changing user circumstances. In addition, sensors and controllers are also physical devices which may malfunction under different circumstances [5]. Any such failures should not go unnoticed, undetected and unsolved, so that provision of context information remains flawless, keeping the behavior of the system reliable.

Consider for example, context aware systems that are deployed in healthcare [6] for the elderly people. These systems monitor conditions of patients and automatically request assistance. In assisted-living facilities, context aware systems are used to identify age-related disorders of elderly people by observing their everyday activities. Therefore, users of such systems rely heavily on them. Failures in such systems can be hazardous and can result in loss of life.

In general, faults can lead to incorrect context sensing and processing, security and privacy breaches, and misuse of resources. For context aware systems to be sustainable, it should be unobtrusive and fault transparent to the user [6]. This requires the system automatically mask various kinds of faults and user intervention is sought only when absolutely required.

Different attempts to build infrastructure or architecture for context aware systems have been made, but these have provided only partial solutions; for instance, most have not adequately addressed issues such as fault tolerance or privacy [1][6]. Therefore it is important to develop a context aware infrastructure which comprises the mentioned limitations.

Since pervasive computing environments operate in the same physical (as well as virtual) space as humans [6], faults occurring in this environment should be tolerated with minimal user awareness. In order to satisfy this requirement, software deployed in this kind of infrastructure must be administered or managed autonomously. This can be done by autonomic manager which is responsible for monitoring the environment and react to events such as failures and reconfigure the system accordingly. Therefore introducing the concept of autonomic computing into the software components leads to have components with autonomic behavior that enables them for example to self-configuring and self-healing.

The main advantages of this approach are:

- Autonomic administration that allows the required fault management functionalities to be performed without human intervention
- Providing a high-level support for deploying and configuring software components which reduces errors and administrator's efforts.

This thesis work attempts to study, identify and research on the shortcomings of existing distributed context management infrastructure in relation to fault tolerance issues such as fault detection, fault notification, fault isolation, re-assignment in place of faulty components and fault healing in context aware systems and to incorporate a fault management functionality by taking the existing context management infrastructure as a foundation.

1.2. Motivation

Pervasive computing is a computing paradigm which allows any-time and any-where information access and processing for mobile users in dynamically changing heterogeneous computing environments [8]. Pervasive computing applications/services require seamless adaptation to the changing computing environment and/or changing user tasks (i.e. changing context of computation). The adaptation is triggered by evaluation of context information gathered from a variety of sources, including sensing devices. Some approximation of the current application's context can be captured from user provided information, hardware sensors and monitoring software. Context information is managed by the context management system which gathers context information from a variety of sources, and pre-processes, evaluates and disseminates this information [17].

Current applications/services are often distributed over heterogeneous computing devices connected by heterogeneous networks. Such computing environments are prone to node failures or network disconnections and this can be exacerbated by user mobility. Computing devices (servers, sensors, etc.) and services that participate in the applications can fail due to a number of reasons including low battery power, physical damage to the device, network disconnections.

Existing approaches to developing context aware applications mainly focus on how context information is gathered and processed to support application adaptation - problems of dynamic changes in the environment, like user mobility and node and component failures, have not been fully addressed [10]. A Frequent interaction between human and computer for

the purpose of configuring, tuning and maintenance are infeasible in such systems. For this reason and due to the complexity of such systems created by interoperability, scalability, and adaptability requirements, and also emerging behaviour problems, context aware systems need to be self-managed, i.e., able to self-configure, self-monitor, self-heal, self-adapt and self-protect. Consider for example the following scenario.

Emergency Services Scenario: A context-aware application is used to surveillance critical infrastructure and alerts the administrator any abnormal events. The application is supported by a distributed context management system for providing the required context data. Different types of sensors including cameras, audio sensors, temperature and light sensors, and weather stations are used to gather information about the infrastructure. The sensed information is then pre-processed using the nearby computing device before stored in the context repository. The context management system provides sophisticated processing of sensor information including face and scene recognition for detection of abnormal events using distributed software components. Assume one of the computing devices from the distributed nodes fails, for example by physical attack. In this case the components deployed in the failed node will also fail. Emergency services arrive equipped with the necessary node. The context management system then identifies new node and deploy and rebind the failed components on the new node. The reconfiguration is done dynamically without the help of human assistant.

Therefore, it is interesting to incorporate these services in context management systems that ensure availability and reliability of context provisioning services for context-aware applications i.e. the subject of this thesis work.

1.3. Statement of the Problem

In a pervasive computing environment with context-aware computing facility, the sustainability of context aware applications depends on reliable provisioning of context information provided by the context management system. The context management system must be fault tolerant in order to provide reliable context information. In addition, user intervention is sought only when absolutely required. Hence a fault tolerant issue is an important issue in distributed context data management.

The main theme of the research work is the realization of component-based fault tolerant distributed context management architecture for context-aware systems that is aware of issues that have not been comprehensively addressed by the existing models. These issues are fault

detection, fault notification, component isolation, re-assignment in place of faulty component and fault healing.

1.4. Objectives

1.4.1. General Objective

The general objective of the study is to come up with a component-based fault tolerant distributed context data management for context-aware systems.

1.4.2. Specific Objectives

To meet to the general objective of the study, the following specific objectives are identified

- To review literatures on existing context data management in context aware systems with respect to fault tolerance
- Identification of methods to solve fault tolerance issues using the concept of autonomic computing
- To design an architecture for component-based fault tolerant distributed context data management
- To test the performance to the proposed prototype

1.5. Scope

In general, context management system is responsible for gathering, pre-processing, managing and provisioning of context information to the context aware applications. However, we consider faults that originated at the pre-processing, managing and provisioning layer of the context management system. We didn't incorporate faults originated from the sensor network, as the topic is very wide and demands to undertake a separate research. In addition, we didn't entertain faults that have byzantine behaviour.

1.6. Methodology

In order to achieve the general and specific objectives mentioned in section 1.4, we use the methods with the following stages.

- Literature Review: related literatures from different sources (books, journals, research articles, etc.) are reviewed to understand fault tolerant issues in distributed context management system for context-aware system. The major activities carried out in this stage are :

- Studying major researches in the area of distributed context management
- Investigating the use of autonomic computing in fault tolerant context-aware systems.
- Studying existing distributed context management architecture with an emphasis on fault tolerant issues
- Reviewing the current existing methods of solving the component-based fault tolerant issues in the context aware systems.
- Examining properties of a full-fledged fault tolerant system

- Design Phase: design of component-based fault tolerant distributed context management model: the major activities undertaken in this stage are:
  - Specifying the requirements of the model to be developed based on the study made in the first phase.
  - Development of the model as per the outlined requirements. This involved:
    - Identifying the various components of the model
    - Defining the internal details of each component
    - Defining the means and scope of communication among the components
- Development of prototype architecture: major activities performed in this stage includes:
  - Specifying the requirements of the prototype
  - Choosing appropriate programming languages and technologies
  - Studying tools, algorithms, technologies and protocols needed for implementing the prototype.
- Testing of the prototype: in this stage the validity of the proposed architecture and performance of CPU and memory usage is tested

1.7. Thesis Organization

This thesis document is organized as follows: Chapter two presents introductory background about context aware systems and state-of-the-art in context management systems. In addition this chapter presents basic definition, classification and approaches in fault tolerance computing systems. Chapter three covers the summary of research works which are related to our work most. The fourth chapter presents our proposed fault management architecture for distributed context management system. This chapter presents details of the internal

components of the architecture with their associations. We have developed a prototype implementation as a proof of the concepts introduced in the architecture discussed in chapter four. The detail of the prototype is presented in the fifth chapter by incorporating the description of tools used in developing the prototype, the experimental setup and the results obtained when testing the prototype. Finally, the sixth chapter presents a general conclusion, thesis contributions and points out possible future works.

2. Literature Review

2.1. Overview

This chapter presents the state-of-the-art distributed context management approaches for context aware systems and fault tolerance techniques and describes the basic concepts behind context aware systems and fault tolerance systems.

2.2. Context Aware Systems: Definition, Services and Examples

2.2.1. Basic Concepts

Nowadays, the integration of mobile clients into a distributed environment and the ad-hoc networking of dynamic components are becoming ever more important in all areas of application [11]. According to F. Mattern [12] , the continuing technical advances in computing and communication are heading towards an all-encompassing use of networks and computing power named as ubiquitous or pervasive computing.

The essence of pervasive computing is to enhance the environment by embedding many computers that are smoothly integrated with human users [13]. To gracefully integrate a computation and communication saturated environment with human users, pervasive-computing applications need to be context-aware. We refer to two definitions in the literature for context aware systems. According to Robert Schmohl et al. [17] "*A system is said to be context-aware if its operations and services can be adapted to the current context without explicit user intervention and thus aims at increasing the usability and effectiveness by taking environmental context into account.* "

Dey et al. [15], define context-aware system as "*a system that uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task*".

To demonstrate the above definition, consider a simple restaurant finder application [16] which finds the nearest restaurant location based on the user context. The user subscribes her personal preferences e.g., she states that she likes French and Italian food at reasonable prices, dislikes fast food, etc. Then, whenever she feels hungry, she just invokes the restaurant finder service from her mobile, which delivers list of restaurants that are nearby and correspond to her preferences.

The history of context-aware systems started when Want et al. (1992) introduced their Active Badge Location System which is considered to be one of the first context-aware applications [14]. However, In literature the term context-aware appeared in Schilit and Theimer (1994) the first time [14][17]. The authors describe context as location, identities of nearby people, objects and changes to those objects. The most accurate definitions is given by Dey and Abowd [19] which is stated as follows.

"*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves*".

2.2.2. Context Aware System Service Classifications

Among the basic elements of services that context-aware systems provide to users are [19]

- Attaching context information for later retrieval: context information can be attached to existing pieces of information to give descriptive information about the subject. One example can be attaching context to desktop or networked files so as to enable easier retrieval later on.
- Presenting information and services: these are applications that present context information to the user or use context to propose appropriate services. For example, a nearby printer would be proposed when the user wishes to print something.
- Automatic execution of service: These kinds of applications trigger a command or reconfigure the system on behalf of the user as per changes that happen to context. For example, if the user is in a meeting and there is a phone call for him or her, then the call should be routed to voicemail.
- Collaborative computing services: voluntary based mission-oriented context-aware and dynamic communities of computing entities that perform tasks on behalf of users in an autonomous manner. Such applications are important in almost every sphere of our lives, such as campus management, health care, telemedicine, pervasive security, military, and crisis management.

2.3. Context Management Systems

Context-aware systems must use context information to adapt their behaviour and adjust to changing computing environments or changing user circumstances. Although identifying and deducing context information is a challenge, it is critical that context-aware systems should

operate by conveying the appropriate information to the right place at the right time by inferring the user's intention. In order to facilitate this objective, context aware systems need a context management system that collects, stores and manipulates context information. The context management system is responsible for [11]

- Gathering context information from the environment.
- Translating this information into the appropriate format.
- Combining or interpreting context information to generate a higher context.
- Automatically trigger actions based on the context information and monitoring of the actions.
- Make the information accessible to other applications and the neighbourhood.

A lot of research works has been done to implement the above functionalities which assists the development of context-aware applications [11][20][21]. These researchers classified Context Management Systems (CMS) focusing on different issues such as context acquisition, context accessing and sharing, as well as on architectural properties of the context management support system.

Chen [22] presents three possible approaches based on the context acquisition.

- Direct sensor access- the application gather context information directly from sensors, which means that there is no additional layer for gaining and processing sensor data. This tightly coupled approach is not particularly well suited for distributed systems.
- Context server - the availability of a server permits multiple clients access to possibly remote context data sources. The context server gathers sensor data and makes them available to client applications.
- Middleware: introduces a layered architecture in the design of context-aware systems. The middleware is responsible for collecting, storing, aggregating and distributing context information to context-aware applications.

Winograd [23] describes three different context management models for coordinating multiple processes and components. The classification is focused on context access and distribution rather than context acquisition.

- Widgets - derived from the concept of GUI, a widget is a software component that provides a public interface for a hardware sensor. Widgets hide low level details of sensing and ease application development. It is usually controlled by a widget

manager. The tight coupling of widget with their manager and single-manager control leads to lack of tolerance of component failure

- Networked service- in this architecture, a client needs to find the location of service (through pre-configuration or some kind of resource discovery process), and then set up a connection with it. A key feature of service-based architecture is the independence of the components. Each component contains appropriate code to manage connections, failures, error messages etc.
- Blackboard model- adopts a data-centric view rather than process-centric view of the widget and service-oriented model. In this approach, applications post messages to a shared media, called blackboard, and subscribe to it to be notified when some specified events occur. It is the most loosely coupled and therefore leads to tolerance of component failure.

Hong and Landay [24] propose a classification of software systems to support context management. The authors outline four main categories.

- Libraries- a library is a generalized set of related algorithms. Libraries are easy to use. However, they tend to be focused on low level context details and do not provide any support for application design.
- Frameworks - this approach more focused on design reuse by providing a basic structure for a certain class of applications, which can be customized according to the application requirements.
- Toolkit- toolkits are typically built on frameworks and offer a number of reusable components each one addressing specific functionalities, for example, a toolkit for accessing sensors and aggregating context information. Similar to frameworks, they typically depend on specific implementation platforms, operating systems and programming languages.
- Infrastructure- is a well established, pervasive, reliable, and publicly accessible set of technologies that act as a foundation for other system. Several infrastructures have been proposed to support context-aware applications. Middleware infrastructures are particularly well suited to support the development of context aware applications.

In general, the choice of an adequate model for context acquisition and management depends on specific application requirements and characteristics. However, an infrastructure based approach relying on the underlying network support for distributing and accessing context

bring several advantages. The most significant advantage is, it assists the separation of context management system from the context-aware application. This allows different issue like reconfiguration and self-healing to be pushed to the middleware layer, freeing the burden from the context-aware application. The distributed components that comprise the middleware will be discussed in the next section.

2.4. Components of a Context Aware System

Context aware systems are generally implemented using a variety of distributed components. Early systems were relatively simple, and were often constructed simply as distributed application components communicating directly with local or remote sensors. Today, it is widely acknowledged that additional infrastructural components are desirable, in order to reduce the complexity of context-aware applications, improve maintainability, and promote reuse [25].

An analysis of the architectures of multiple context-aware systems presented in various publications [26, 27, 28, 29] identifies the following abstract components shared by the majority of those approaches:

- Context sensors: Those sensors exist either as pieces of hardware sensing the physical environment, or as software component providing data from other context sources [20]. The primary task of both sensor types is the acquisition of raw data for further refinement into contextual information.
- Context capturing interface: Data acquired by sensors is usually uncertain and difficult to interpret by high-level components [14]. For this reason, sensor data needs to be refined for further processing by deriving a higher level of context from uncertain multi-modal sensor data
- Context repository: The context repository stores the current context. It consists of the according data structures used to represent the context model.
- Context reasoning: The context reasoning component is responsible for inferencing new context based on the current contextual information in the context repository and new contextual data acquired through the context capturing interface.
- Context API : The context API provides an interface for context-aware applications to actually utilize contextual information [17]. The employment of such an interface implements the commonly accepted paradigm of separating context management from application logic

- Context application: Applications accessing the context API can actually use the contextual information for their application-specific purposes. This includes the automatic execution of services based triggered by special contextual conditions, as well as the discovery and allocation of resources relevant to the current context
- Communication interface: Since a context-aware system is distributed in nature, communication needs to be handled appropriately.
- Actuators: Actuators are the counterparts of sensors that may be utilized as a result of a contextual update

The above components is abstracted into a hierarchy of layers and shown in figure 2.1.

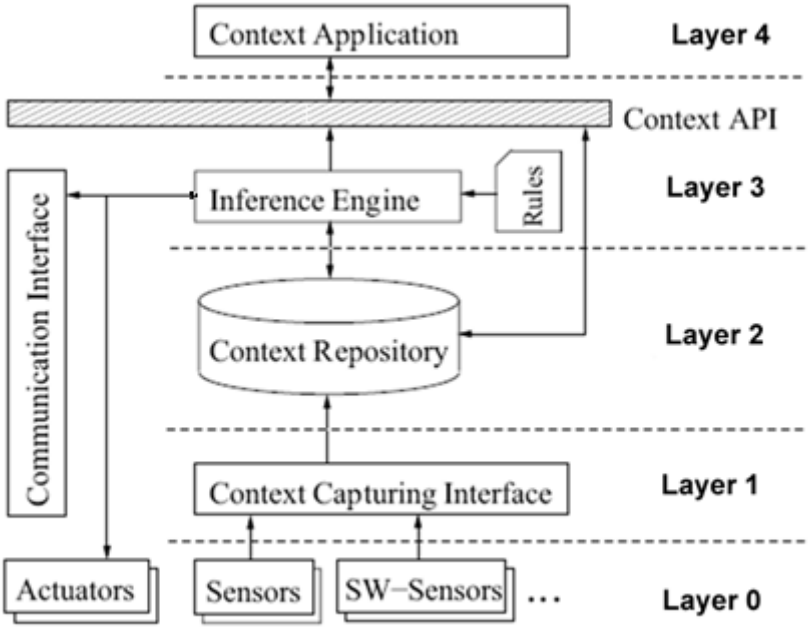


Figure 2.1: Context aware System Architecture

K. Henricksen et al. [25] refer the components that reside between the layer 4 application components and the layer 0 sensors and actuators - together with the communications interface that binds the distributed components together – as middleware for context-aware systems.

The large number of distributed components that are present in context-aware systems introduces a requirement for deploying, configuring and managing networks of sensors, actuators, context processing components, context repositories, and so on. Henricksen et al [25] presented a requirement for such kind of middleware. Summary of these requirements are:

- Support for heterogeneity: Hardware components ranging from resource-poor sensors, actuators and mobile client devices to high-performance servers must be supported.
- Support for mobility: All components (especially sensors and applications) can be mobile, and the communication protocols that underpin the system must therefore support appropriately flexible forms of routing. Context information may need to migrate with context-aware components.
- Scalability: Context processing components and communication protocols must perform adequately in systems ranging from few to many sensors, actuators and application components.
- Support for privacy: Flows of context information between the distributed components of a context-aware system must be controlled according to users' privacy needs and expectations.
- Traceability: The state of the system components and information flows between and control components should be open to inspection - and, where relevant, manipulation - in order to provide adequate understanding and control of the system to users, and to facilitate debugging.
- Tolerance for component failures: Sensors and other components are likely to fail in the ordinary operation of a context-aware system. Disconnections may also occur. The system must continue operation, without requiring excessive resources to detect and handle failures.
- Ease of deployment and configuration: The distributed hardware and software components of a context-aware system must be easily deployed and configured to meet user and environmental requirements, potentially by non-experts (for example, in "smart home" environments).

Even though approaches featuring a centralized context management component are generally realizable, almost all of the current research emphasizes either distributed systems or hybrid approaches, where centralized component act as a subordinate system supplement [17]. This fact implies that software providing context-awareness is deployed as middleware on the nodes in the distributed system. Numerous attempts to build middleware or infrastructure for context-aware systems have been made, but most of these satisfy some of the requirements stated above [25]. For instance, most have not adequately addressed issues such as fault tolerance or privacy.

2.5. Definition, characteristics and classification of faults

In previous sections we discuss the basic ideas and state of the art of context management system. In this section, we discuss basic definition, classification and characteristics of faults.

2.5.1. Definition

Correct service is delivered when the service implements the system function. The function of a system is what the system is intended to do and is described by the functional specification in terms of functionality and performance.

A service failure (failure): is an event that occurs when the delivered service deviates from correct service [31]. Since a service is a sequence of the system's external states, a service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an error. The hypothesized cause of an error is called a fault [31].

2.5.2. Characteristics and Classification of Faults

Algirdas Avi_zienis et al [31] present classification of faults that may affect a computing system during its life cycle. The authors classify all faults that may affect a system into eight fault classes. These are fault phase of creation or occurrence, system boundaries, phenomenological cause, dimension, objective, intent, capability, and persistence. The eight fault classes partially overlap into the following three groups:

- development faults that include all fault classes occurring during development,
- physical faults that include all fault classes that affect hardware,
- interaction faults that include all external faults

The tree representation of fault classes and their groupings is shown in figure 2.2.

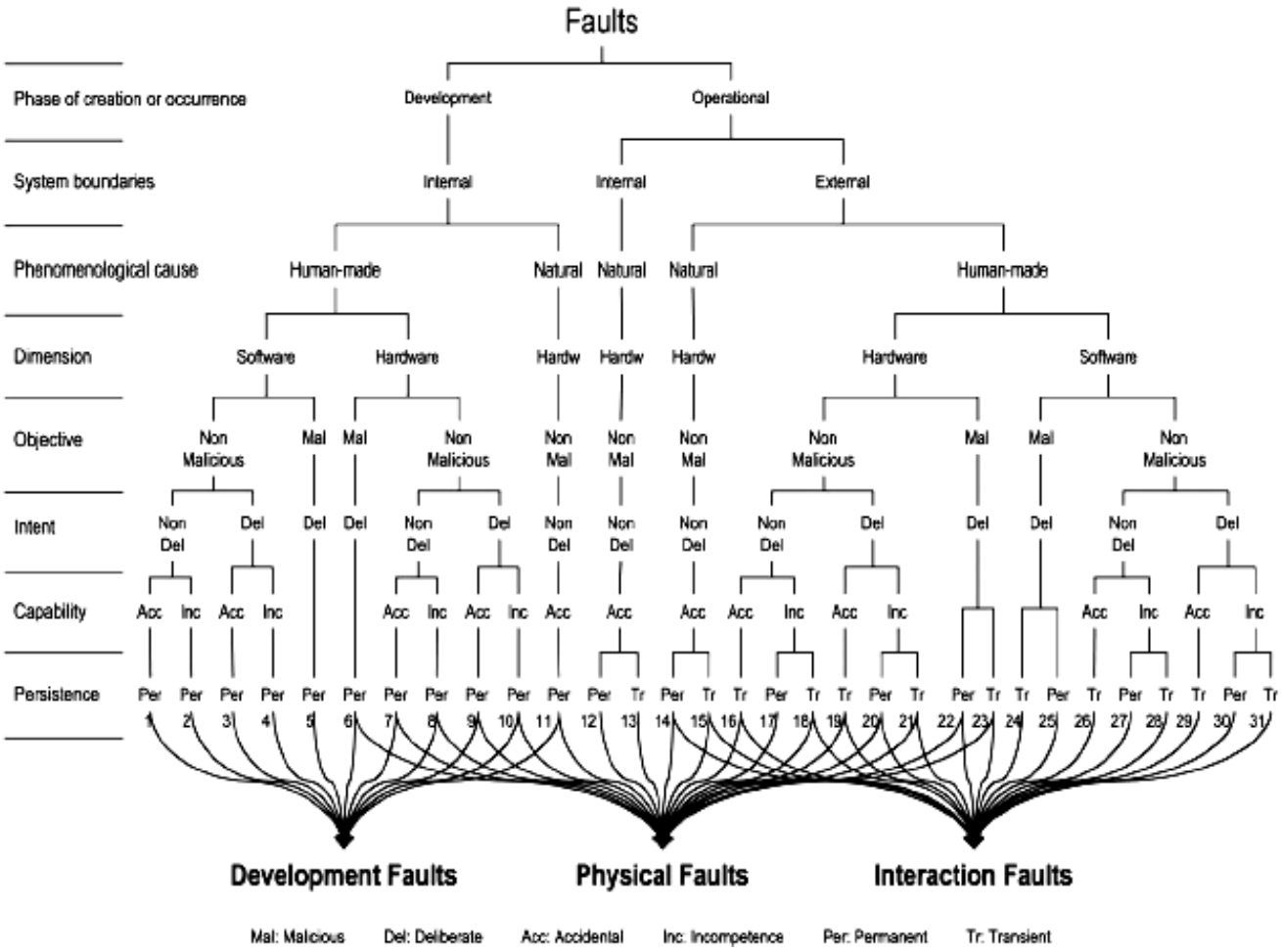


Figure 2.2: Classes of Faults in Tree Representation [31]

Chetan [8] broadly classify fault particularly in a pervasive system into device, application, network, and service failures.

- Application faults: include application crashes due to bugs, operating system errors, unhandled exceptions, and faulty usage. Pervasive applications are also likely targets for malicious software such as viruses and worms.
- Network faults: pervasive systems consist of wired and wireless devices. Therefore, a reliable pervasive system should account for network faults caused by low signal strength, devices going out of range and unavailability of communication channels due to heavy traffic. Network faults lead to unreachable devices that may be wrongly perceived as device faults.
- Service faults: a pervasive system is supported by various services that enable different functionalities. Some services include naming and context services that enable context-aware computing. Examples of service faults include service crashes

due to bugs and operating system errors, faulty operation of services like sensing incorrect context, wrong inferring, and lossy delivery of events. Service faults can potentially lead to the failure of the pervasive system.

- Device faults: a pervasive system consists of different kinds of devices such as desktops, laptops, handhelds, sensors, actuators, displays, etc. Each device is prone to its own set of faults that can potentially contribute to the failure of the pervasive system. For example, Mobile devices, such as laptops and handhelds, have physical constraints such as finite battery power and limited signal strength.

2.5.3. Failures

As defined in Section 2.5.1, a service failure is an event that occurs when the delivered service deviates from correct service. The different ways in which the deviation is manifested are a system's service failure modes [31]. The service failure modes characterize incorrect service according to four viewpoints which are presented below:

- Failure domain: this leads us to distinguish the following failures
    - Content failures: The content of the information delivered at the service interface (i.e., the service content) deviates from implementing the system function.
    - Timing failures: The time of arrival or the duration of the information delivered at the service interface (i.e., the timing of service delivery) deviates from implementing the system function.
    - Content and timing failures: when both content and timing are incorrect fall into two classes:
        - Halt failure, or simply halt, when the service is halted (the external state becomes constant, i.e., system activity, if there is any, is no longer perceptible to the users); a special case of halt is silent failure, or simply silence, when no service at all is delivered at the service interface (e.g., no messages are sent in a distributed system).
        - Erratic failures otherwise, i.e., when a service is delivered (not halted), but is erratic.
- Detectability of failures: addresses the signaling of service failures to the user(s). Signaling at the service interface originates from detecting mechanisms in the system that check the correctness of the delivered service. When the losses are detected and

signaled by a warning signal, then signaled failures occur. Otherwise, they are unsignaled failures. The detecting mechanisms themselves have two failure modes: 1) signaling a loss of function when no failure has actually occurred, that is a false alarm, 2) not signaling a function loss, that is an unsignaled failure.

- Consistency of failures: leads us to distinguish, when a system has two or more users:
    - Consistent failures: The incorrect service is perceived identically by all system users.
    - Inconsistent failures: Some or all system users perceive differently incorrect service (some users may actually perceive correct service); inconsistent failures are usually called, after [31], Byzantine failures.
- Consequences of failures on the environment: addresses the extent and severity of the failure on the system. Generally speaking, two limiting levels can be defined:
    - minor failures: where the harmful consequences are of similar cost to the benefits provided by correct service delivery
    - catastrophic failures: where the cost of harmful consequences is orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery

2.5.4. Relationship between Faults, Errors, and Failures

The creation and manifestation mechanisms of faults, errors, and failures can be summarized as follows [31]:

A fault is active when it produces an error; otherwise, it is dormant. An active fault is either 1) an internal fault that was previously dormant and that has been activated by the computation process or environmental conditions, or 2) an external fault. An error propagation within a given component (i.e., internal propagation) is caused by the computation process: An error is successively transformed into other errors.

A service failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. The failure of a component causes a permanent or transient fault in the system that contains the component. Service failure of a system causes a permanent or transient external fault for the other system(s) that receive service from the given system. Figure 2.3 shows the relationships between fault, error and failure.

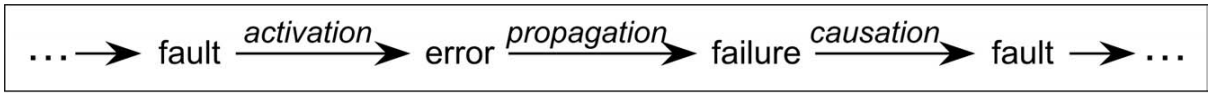


Figure 2.3: Relationships between Fault, Error and Failure [31]

2.6. Means of Achieving Fault Tolerance

2.6.1. General techniques

Fault tolerance which is aimed at failure avoidance, is carried out via error detection and system recovery.

- Error detection: used to identify the presence of an error. There are two detection mechanisms.
    - Concurrent detection: takes place during normal service delivery
    - Pre-emptive detection: takes place while normal service delivery is suspended; checks the system for latent errors and dormant faults
- System Recovery: transforms a system state that contains one or more errors and faults into a state without detected errors and without faults that can be activated again. It uses two approaches:
    - Error handling: eliminates errors from the system state. Three different approaches are identified for handling errors:
        - Rollback: brings the system back to a saved state that existed prior to error occurrence for example using saved state or checkpoint
        - Rollforward: brings the system to a state without detected errors in a new state
        - Compensation: the erroneous state contains enough redundancy to enable error to be masked
    - Fault handling: prevents the system from faults from being activated again. Fault handling uses four techniques to achieve its objective.
        - Fault diagnosis: identifies and records the causes of errors in terms of both location and type
        - Isolation: performs physical or logical exclusion of the faulty components from further participation in service delivery

- Reconfiguration: switches or reassign tasks among non-failed components
- Reinitializtion: checks, updates and records the new configuration and updates system tables and records

2.6.2. Fault Tolerance Approaches in pervasive computing

The goal of a fault-tolerant system is to mask faults and continue to provide service despite faults. The choice of error detection, error handling and fault handling techniques and of their implementation is directly related to and strongly dependent upon the underlying fault assumption i.e. the classes of faults that can actually be tolerated [31].

In this section, we discuss approaches that can be employed in pervasive computing environment for tolerating faults [8]. Each approach tackles one or more classes of faults.

- Substitute Application/ Device Usage: A common fault tolerance technique is to detect failure of a process and restart it. In order to minimize the loss of computation, the state of the process is periodically stored on a stable storage device. Upon failure, the process is restarted using the stored state. This technique can be used in a pervasive system to tolerate some application and device faults. If devices fail due to hardware problems, applications on the device cannot be restarted on the same device. So the solution is to find a surrogate device that can provide the same functionality as the failed device and restart the application on it. If the surrogate device cannot support the execution of the same application, an equivalent application that provides similar functionality can be used.
- Handling Errors in Sensing and Inferring Context: Detecting errors that may occur while sensing and inferring contexts is not easy. The primary way of detecting and handling such errors is by employing redundancy. Multiple sensors that sense the same (or similar) pieces of information can be used to overcome errors by one or more sensors. However, Peizhao Hu et al [32] argue that this approach can be very wasteful in terms of resource usage and it will not scale to large numbers of sensors.
- Alternate Notification Mechanisms: Pervasive computing can offer multiple ways of reaching users. For example, in an assisted-living facility, pervasive systems are used to monitor the status of facility users. In an emergency, the system notifies healthcare personnel for assistance. The system can notify the personnel through their cell phones, pagers, calling help lines, or as a text message on display devices or speakers

in an emergency monitoring station. So these “channels” of communication provide inherent redundancy in the pervasive system that the system can leverage to tolerate faults. If the system discovers that a notification device has failed, it should reroute the message through a different channel of communication.

2.7. Fault tolerance using self-manage approach

2.7.1. Component Base Autonomic Management

Computing systems are becoming very complex, highly heterogeneous and distributed. At the same time, the users of these systems are usually mobile and demand greater flexibility and efficiency in terms of response time, resource utilization, robustness, etc., to achieve critical business goals [33]. This implies that operating and maintaining computing systems in case of failure is expensive business. This high cost of ownership of computing systems has resulted in a number of industry initiatives to reduce the burden of operations and management by making computing systems - at least gradually - self-managing.

Self-management drives its basic principles from the autonomous nervous system, which governs our heart rate and body temperature, “thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions” [34]. A system is said to be self-managing if it exhibits one or more of the following characteristics: self-configuration, self-optimization, self-healing, and self-protecting [33]. This essential principle enables computing systems, whether acting individually or collectively, to receive higher-level objectives from their users but manage to maintain and adjust their operation in the face of changing components, workloads, demands, and external conditions as well as imminent hardware and software failures.

According to Kephart and Chess [34] a self-managing system contains autonomic manager software and a (hardware or software) managed element. The autonomic manager contains components that are used for monitoring the computing environment, analyzing emerging or foreseeable problems or potential causes of adaptation, planning component that accommodates workflows that specify a partial order of actions which should be carried out in accordance with the results of the analysis component and finally, the execution component that controls the execution of such workflows and provides coordination if there are multiple concurrent workflows.

A variety of frameworks, architectures and techniques have been proposed and used in the field of autonomic computing for self-management. Most of the works focus on one of the two design approaches, Externalization and Internalization [35]:

- In Externalization Approach, modules enabling self management lie outside the managed system.
- In Internalization Approach, application specific self management is done inside the managed system.

Externalization approach is more effective because it localizes problem detection and resolution in separate modules. It provides a more generic solution that can be used to enable autonomic behavior in existing systems [36]. This approach helps us to inject autonomic behavior in legacy and non-autonomic system. Component based autonomic management is one example of externalization approach.

Component-based autonomic management aims at providing a uniform view of a software environment composed of different types of servers. Each managed server is encapsulated in a component and the software environment is abstracted as component architecture [37]. Therefore, deploying, configuring and reconfiguring the software environment is achieved by using the tools associated with the used component-based management.

2.7.2. Techniques for Achieving Self-Healing

Self-healing refers to the ability of a system to detect, diagnose, and repair problems resulting from bugs or failures in software and hardware [33]. Static and dynamic survivability models have been introduced for enduring failures. Static survivability models employ redundancy to make systems resilient to failure. Dynamic survivability models replace erroneous components with dynamically created components. The following techniques are used to implement self-healing functionalities.

- Techniques for fault Detection/Diagnosis: fault detection involves identifying a fault that has occurred. Fault diagnosis involves finding the cause of the problem. Some techniques devised for this purpose are
    - The heartbeat failure detection algorithm [37] involves monitored systems sending messages “rising heart beat” to the monitor. Then the monitor replies by sending “falling heart beat”. Failure is detected when the monitor experiences a delay in receiving of message. The interval of the message is

adjusted dynamically, by sending “heart beat period” in the falling heart beat message by the monitor.

- Performance Query Systems (PQS) [38] can be used to enable user space monitoring of servers. PQS use advanced behavioral models thereby making accurate and fast decisions regarding server and service state. A sensor is placed in the host, which monitors host behavior and publishes significant events to an engine as observations. The engine fuses this information using custom process models and correlates these acquired events with events indicating host-level failure. Output of these process models helps administrators detect the state of nodes.

- Techniques for fault Repair: Some faults repair techniques that overcome failures and restore system to normal functioning state are described below [36]:
  - Component level rebooting is a technique that recovers from defects, without disturbing the rest of the application. However this technique requires that the components must be isolated and stateless. Data recovery is kept separate from process recovery in this technique. The important state information is kept separately in dedicated state stores. This technique provides fast recovery, reduction in functional disruptions and work lost compared to full reboot.
  - Another safe technique to quickly recover programs from deterministic and non deterministic bugs is to rolls back the program to a recent checkpoint upon a software failure, and then re-executes the program in a modified environment

3. Related Works

Several research works have been done since the early 1990's on the development of distributed context-aware systems [14]. The need for middleware to seamlessly bind the distributed components of a context-aware system together is well recognized. Numerous attempts to build middleware or infrastructure for context-aware systems have been made [25]. These works differ in support they provide to the context-aware applications, software architecture they follow, domain of application they utilize are some of the points. In this chapter, we present five works which are related to our work with an emphasis on the fault tolerance issue they addressed,

3.1. Context Fusion Network (CFN)

In this work, Chen proposes a Context Fusion Network (CFN) [13], a flexible and scalable context fusion infrastructure that collects aggregates and disseminates contextual information.

CFN was implemented by a prototype named Solar. Solar is a middleware infrastructure with two kinds of client: sensors as data sources and applications as data sinks. Sensors generate events that flow through a direct acyclic graph, passing one or more operators and is finally delivered to a subscribing application. Operator is an independent data processing module that takes one or more data sources as input and acts as another data source.

Solar uses a modified filter-and-pipe pattern for operator composition and XML-based language as composition language. The author mentions the advantages of the filter-and-pipe pattern, which helps to ensure low coupling between components which leads to easy handling of component failures.

Solar consists of a set of functionally equivalent hosts named Planets, which peer together to form a service overlay using Distributed Hash Table (DHT) base P2P routing protocol such as Pastry. Planets are used to host solar components.

Solar's dependency service is responsible for monitoring and recovering of component failure. Before using the dependency service, a Solar component must be registered by providing key configuration information. In case of component failure or lost, Solar restart the component using the configuration information. However, some components maintain state during operation, and may require that state to be restored after a crash. This facility is not implemented on the solar platform.

If an operator failed due to an internal exception, its hosting Planet can simply restart it. In case of a Planet failure, Solar implements a dedicated monitor to recover operators from a failed Planet. Solar uses bi-directional token message between the components and the dedicated monitor for monitoring purpose.

Since, installing one monitor per component would incur a large amount of monitoring traffic, Solar groups the operators on the Planet and monitor them as a whole. The monitor restarts all the operators in the group when a Planet fails.

3.2. Gaia Middleware

In this paper, Chetan et al[8] classified different type of faults in a pervasive system and pointed out various research challenges facing fault tolerance in the realm of pervasive computing. They also propose solutions to some of the challenges and develop a fault tolerance system based on the Gaia Middleware that makes use of context information to tolerate application and device faults.

The authors consider a fail-stop fault model. The fault model only considers application failures caused by transient errors such as device failure, network faults and failure due to faulty usage.

The fault management system is responsible for the fault tolerance issue in the architecture. Applications periodically save their states onto a checkpoint storage. If an application fails, i.e. stop sending a periodic heartbeat message to the Gaia operating system, the system notifies the fault manager. The fault manager then uses context information from the Gaia Context Infrastructure to infer contextually appropriate surrogate device on which the application can be restarted. The failed application is then restarted on the surrogate device using the saved state from the checkpoint storage.

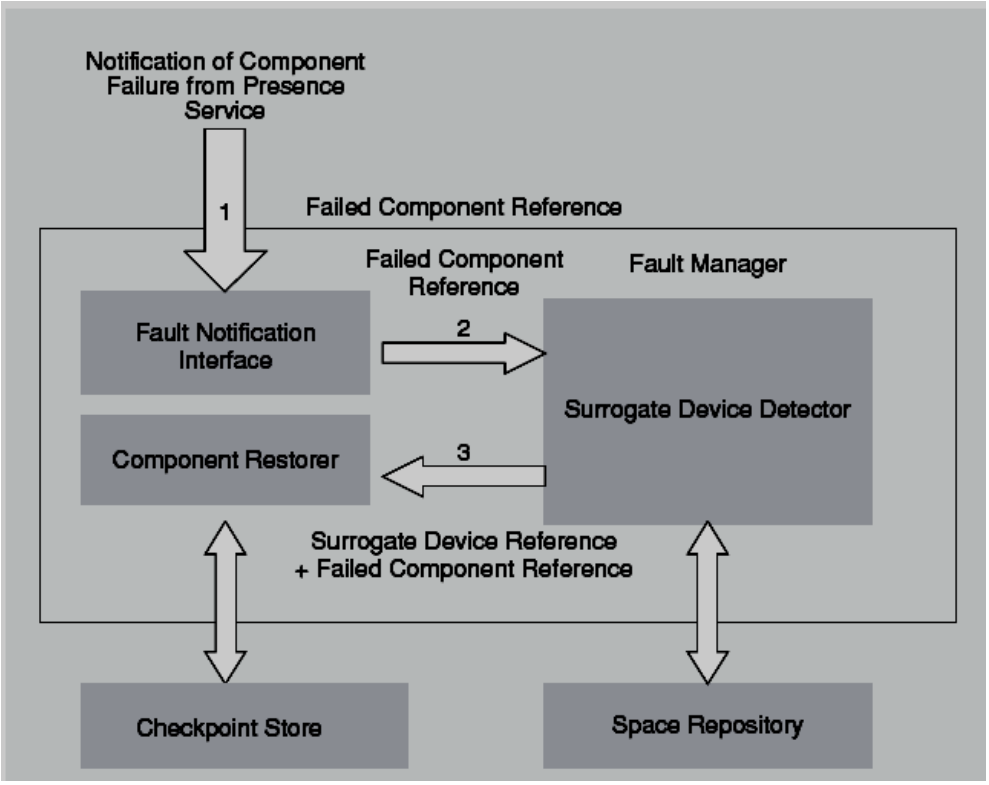


Figure 2.4: Architecture of Fault Manager [8]

Some of the drawbacks of the fault tolerance system are:

- Having redundant (surrogate device) and always active sources of context information leads to resource wastage and it will not scale to large number of resources.
- The proposed model doesn't address service failures like context services that enable context-aware computing, which may have potentially lead to the failure of the pervasive system

3.3. ACoMS (Autonomic Context Management System)

ACoMS (Autonomic Context Management System) [32] is a middleware that supports autonomic context management system for pervasive computing. The system supports dynamic configuration and re-configuration of a set of context information sources (sensors) to achieve fault tolerant provisioning of context information without relying on redundancy.

Eventhough, a common way to provide reliable context information is through redundancy of context sources, the authors claims it will leads to a very wasteful approach in terms of resource usage and it will not scale to large numbers of sensors. Instead, they use the elements of autonomy into the context management system for better monitoring of the system to support its fault tolerance. To achieve such autonomy they proposed context

management system which monitor and reconfigure the context information gathering and pre-processing tasks entirely by the middleware (context management system).

They incorporate context source manager and reconfiguration manager to their own previously developed context management system to handle replacement of failed/disconnected context sources at runtime and to self-configure context sources.

The monitoring service of the reconfiguration manager (RM) provides feedback information about the operational status of individual elements. When failure of these elements occurs, the RM notifies the context source manager (CSM) and provides it with information about failure of the specific element. Based on the notification, the CSM carries out reconfiguration procedure to replace the failed elements if alternatives exist.

The authors tried to solve the problem of fault tolerance in context management with particular interest on faults originated in sensors. However, many context-aware applications interact with higher level abstraction rather than use of context facts directly. Therefore, it is important to handle faults originated from higher level context processing components. The authors consider fault tolerant context management system by focusing on the problem of fault tolerant provisioning to context facts.

3.4. AMUN (Autonomic Middleware for Ubiquitous eNvironments)

Trumler et al. [45] proposes an Autonomic Middleware for Ubiquitous eNvironments (AMUN) which implements self-configuration, self-optimization and self-healing for distributed service-based ubiquitous applications in the Smart Doorplate application.

The architecture consists of four parts: transport interface, event dispatcher, service interface and service proxy and autonomic manager.

The autonomic manager is the major control instance of an AMUN node. It consists of

- Configurator- responsible for the configuration of the services on a node
- Control algorithm - evaluates the "actual state" of the node with the metrics from the metrics pool
- Metric pool - stores different metrics for evaluation
- Service infospace - represent the state repositories for the services
- Monitor infospace - represent the state repositories for the monitors

AMUN not only monitors on each level (System, transport interface, event dispatcher) but also uses monitor queues to add and remove monitors as needed.

Self-healing is used to guarantee that all given services are available anytime prior to performance. The self-healing works as follows. An Alive monitor assesses the availability of services in the event dispatcher by monitoring the message which are exchanged between the services. If the service doesn't communicate for a given period of time, the Alive monitor pings the service. If there is no answer, the middleware would first try to restart the service on the same node. If this fails, the other autonomic managers are asked to host the needed service notifying the need for a self-healing reconfiguration.

3.5. Middleware Adaptability for Resource Discovery, Knowledge Usability, and Self Healing (MARKS)

The authors, incorporated knowledge usability, resource discovery and self-healing properties of pervasive computing as a middleware services in MARKS [40].

The self-healing unit covers the fault detection and resource recovery issues. This unit exploits the process name rate of change of status of each device, to predict the fault and two different types of messages ("OK" and "SOS") to announce the condition to the healing manager.

Fault predication and detection: The healing manager periodically monitors, as well as assembles, the status of all of the running applications, memory, power, communication signals etc. Abrupt alternation in those values indicates the possibility of fault. By analyzing the rate of change of these parameter values, the devices self-healing unit detect faults.

Fault notification: If the device's self-healing unit detect the upcoming fault, then it sends an "SOS" message with the log status file and the important filenames that need to be saved in case of any collapse. Otherwise, it periodically sends "OK" message to notify the healing manager about the condition of the device.

Fault isolation: The isolation of faulty device is simply achieved by removing the service number from the corresponding position of the hash table. This is done by the "device isolation subunit" in the self-healing unit.

Fault correction: The healing manager explores alternate solution for the devices that are currently taking service from that faulty device by looking up the service provider's number

in the hash table. The authors also mention the significance of information distribution for resource recovery, to retrieve the proper functioning. Even though, the self-healing unit doesn't incorporate full-fledged self-healing approaches, it is able to detect, notify and isolate not only application faults but also device and communication faults in a transparent way.

3.6. Summary

We have seen some of the major works that incorporate fault tolerance techniques to achieve reliable provision of context to the context-aware applications. We mainly focus on the middleware approaches, as it allows fault tolerance to be handled by the context management system rather than the application itself.

Some of the approaches adopt redundancy of resources to achieve fault tolerance and other uses automatic reconfiguration and self-healing approaches. In addition, most of them focus to address device and communication failures. In our work, we focus on failures that arise from the software components that comprise the middleware itself. We will propose a fault tolerance architecture which helps to manage these components which leads to a fault tolerance context information provisioning to context aware systems.

4. Proposed Architecture

After studying the different approaches for handling fault management in distributed context aware systems, we envisioned a component based autonomic management model under the objective of incorporating self healing service for managing faults. It is important to have a management architecture that provide better monitoring of the system to support fault tolerance due to the fact that context pre-processing, managing and provisioning (context management) can be done in a distributed fashion using different services.

Therefore, we propose fault management architecture to provide fault tolerant provisioning of context information to context aware application. The fault management architecture is composed of different components located on a manager server. These components communicate each other in order to provide management functionality that makes the context management system dynamically reconfigurable and react to failure autonomously and accordingly. In order to have such autonomy, we encapsulate each context management services (i.e components used for gathering, pre-processing, managing and provisioning context information) in software components and administer them as component architecture. As a result, the main role of this architecture is to add management and administration layer on top of distributed context management system so that deploying, configuring and reconfiguring the software environment is achieved autonomously which leads to fault tolerant context provisioning infrastructure.

We describe the main requirements and underlying assumptions for our design in section 4.1. Overview and detailed discussions of the architecture is presented in section 4.2 and 4.3 respectively. Summary will be presented in section 4.4.

4.1. Requirements and Underlying Assumptions

Before describing the proposed architecture, we first define basic concepts and describe underlying assumptions.

Fault management domain: is a set of components under fault management control.

Node: represents an abstraction of a physical computer along with the sub-components that are executed by that node.

Components: represent the software components executing in the node.

Manager Node: is a node which has the ability to manage the distributed context data management software components.

Ordinary Node: is a node which is joined to the Fault Management domain in order to share its resources.

4.2. Overview of the Proposed Architecture

The proposed architecture consists of two major modules: managed system and manager system (see figure 4.1). Each module is composed of several components that are needed to carryout different activities.

The main principle we employ is externalization approach that enable us to provide self-management functionalities from outside the managed system. This approach helps us to incorporate new repair policy dynamically. In addition it helps us to implement a loosely coupled approach for managing faults. Hence, we encapsulate the distributed context management system in software components and manage the environment as component architecture. The software components are run-time entities that are encapsulated and have one or more interfaces (access points to a component that supports a finite set of methods). This abstraction helps manager system to benefit from the essential features of component model i.e. to deploy and configure managed system in a distributed environment. It can also monitor the environment and react to events such as failures and reconfigure accordingly and autonomously.

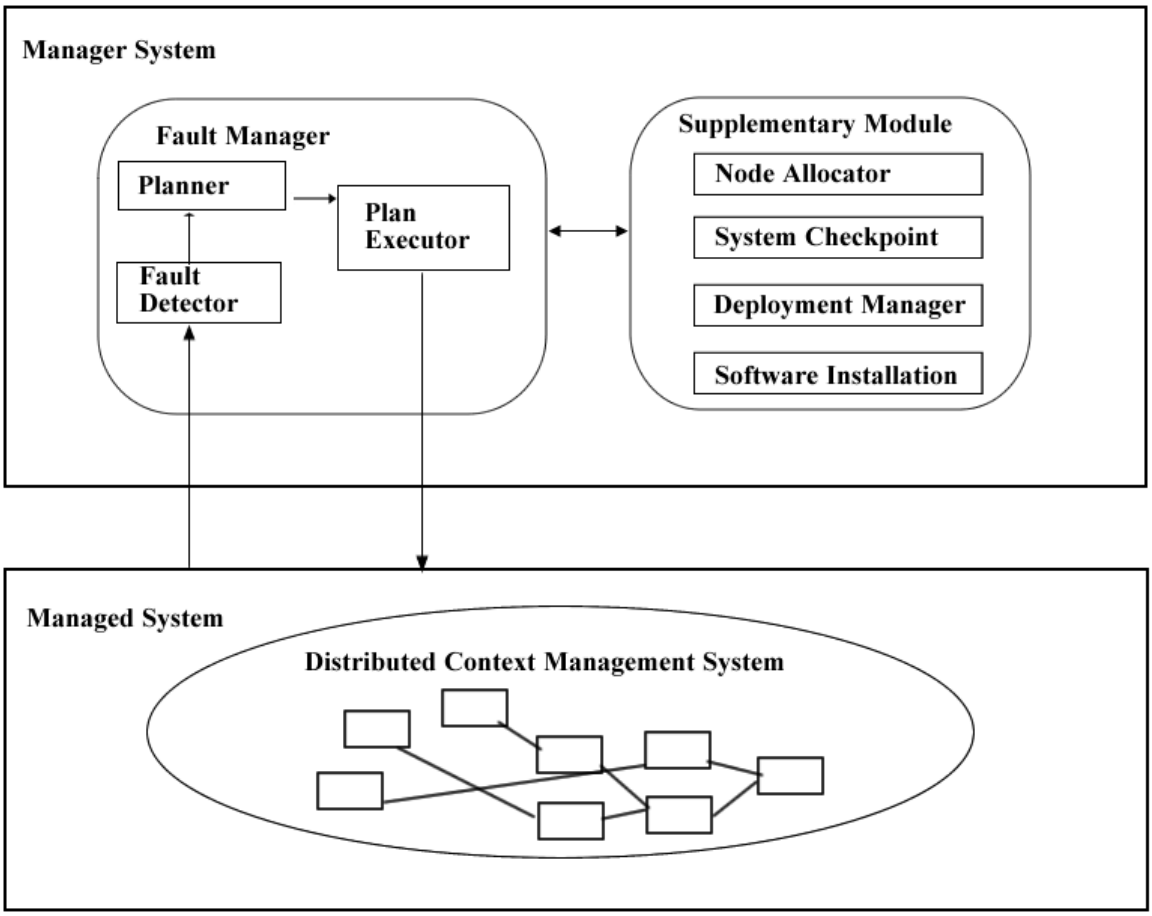


Figure 4.1: Proposed Fault Management Architecture

The managed system i.e. the distributed context management system consists of software components which are used to gather, pre-process, manage and provide context information for context aware applications. The detailed architecture of managed system is described in figure 2.1.

The distributed nature of the above components especially the context capturing, reasoning and aggregation components implies that software components providing context awareness are deployed as middleware on the nodes in the distributed system. Hence, we are interested to manage faults that arise from such infrastructure.

4.3. Details of Manager System

Manager system is composed of different components that are responsible for deploying and monitoring the managed system, then analyzing the monitoring information and deciding which reconfiguration operations need to be applied on the managed system, before actually applying those operations. In order to implement the specified activities, the manager system is composed of two main modules: Fault Manager and Supplementary module. Detail discussion of the modules is described below.

4.3.1. Supplementary Module:

This module consists of different services that provide the necessary functionalities to deploy managed system autonomously and help the fault manager component to carry out its repair policies. The sub-components of this module are discussed below.

4.3.1.1. Node Allocator

The set of nodes in the system can vary dynamically over time due to failures or introduction of new nodes. The node allocator is responsible to provide or reserve suitable nodes which suit to the particular requirement of the desired deployment. When one of the nodes in the fault management domain fails, the fault manger has to deploy software components that were found on the failed node into a new one so that the system will continue its normal activities. Hence, the fault manager uses the node allocator interfaces in order to get a list of idle nodes. For this operation we propose dynamic node allocation algorithm which is implemented and runs on both manager node and on ordinary node.

For the implementation of the algorithm, we assume that each node in the neighborhood set of the manager node has the information of their neighbors by calling the proximity functions of the underlying network system.

Node allocation algorithm on the Manager Node

The finite state machine of the algorithm for the manager node is shown in figure 4.2.

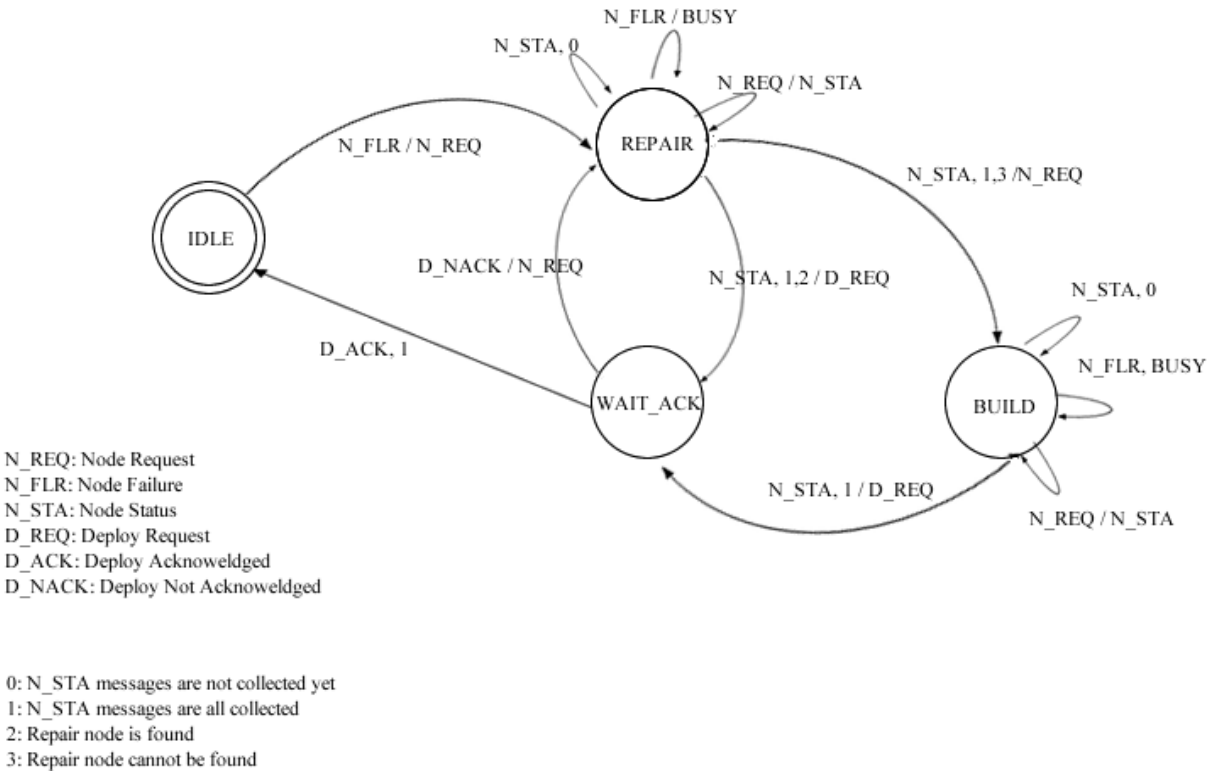


Figure 4.2: Finite State Machine of Manager Node

The above finite state machine is discussed as follows:

- Initially manager node is in the IDLE state
- Fault manager component sends a N_FLR message in case of node failure
- When N_FLR message is received the node allocator changes its state to REPAIR and multicasts a N_REQ message in order to collect neighborhood information.
- When all N_STA messages are collected, the node allocator changes its state to WAIT_ACK
- If a suitable node is found for the replacement a D_REQ message is sent to this node.
- If the node replies with a D_ ACK message then node allocator changes its state to IDLE and returns the repaired structure. If a D_NACK message is received by the requested node, then the node allocator repeats the node replacement operation excluding the tried nodes.
- If a suitable node cannot be found, the node allocator changes its state to BUILD and starts to build a new structure. After the build operation the newly allocated node set is returned and the fault manager is informed about the new allocation of the whole structure.

Algorithm on the ordinary node:

The finite state machine of the algorithm for the ordinary node is shown in figure 4.3.

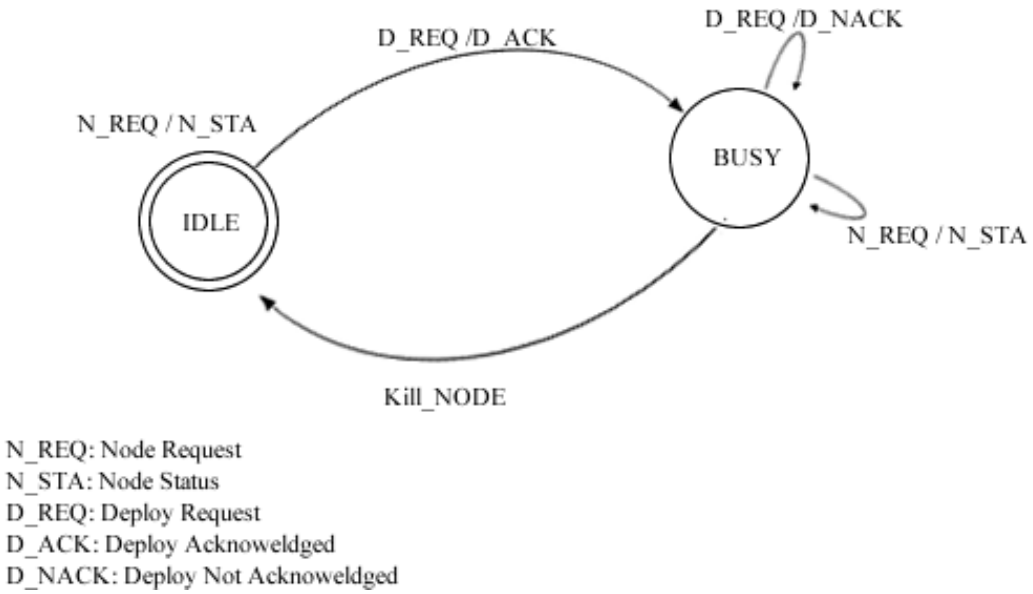


Figure 4.3: Finite State Machine of Ordinary Node

The above finite state machine is discussed as follows:

- Initially ordinary node is in the IDLE state. As long as the node is not reserved for a deployment it will be in the IDLE state
- An ordinary node may receive N_REQ, D_REQ. When a N_REQ message is received the node replies it with a status indicator, N_STA, to the message, indicating that if it is available for a deployment or not. If it is in the IDLE state it sets this indicator as available, otherwise it sets as not available.
- When a D_REQ message is received, it means that a manager node wants to deploy failed components on this node. In such a case if the node is in the IDLE state, it changes its state to BUSY and replies a D_ACK message indicating that it is ready to share its resources. If the D_REQ message is received when the node is in the BUSY state, it replies with a D_NACK message indicating that it is already reserved by a node allocator. The only way that a node changes its state from BUSY to IDLE is to receive a Kill_NODE message which indicates that the deployments are terminated on this node. In this case it gives up its duty and changes its state to IDLE.

4.3.1.2. System Checkpoint

This component is responsible for maintaining the runtime system architecture of the managed system. The System checkpoint is initially deployed at deployment time through deployment manager (see section 4.3.1.3) and modified by reconfiguration functions of the fault manager.

The overall organization of the distributed software architecture that will be deployed is represented as a deployment schema. The schema is describes as a meta-model which consists of software elements, wrapper and attributes as shown in figure 4.4. Software elements represent the managed system which will be deployed. A software element is parameterized by a set of attributes and it references its wrapper.

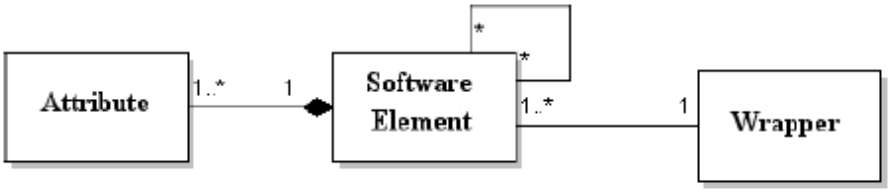


Figure 4.4: Meta-model for a Deployment Schema

Upon deployment time, the deployment schema is parsed and for each software element, a number of runtime components which encapsulates the software element will be created. The system checkpoint represents this component architecture and shown in figure 4.5.

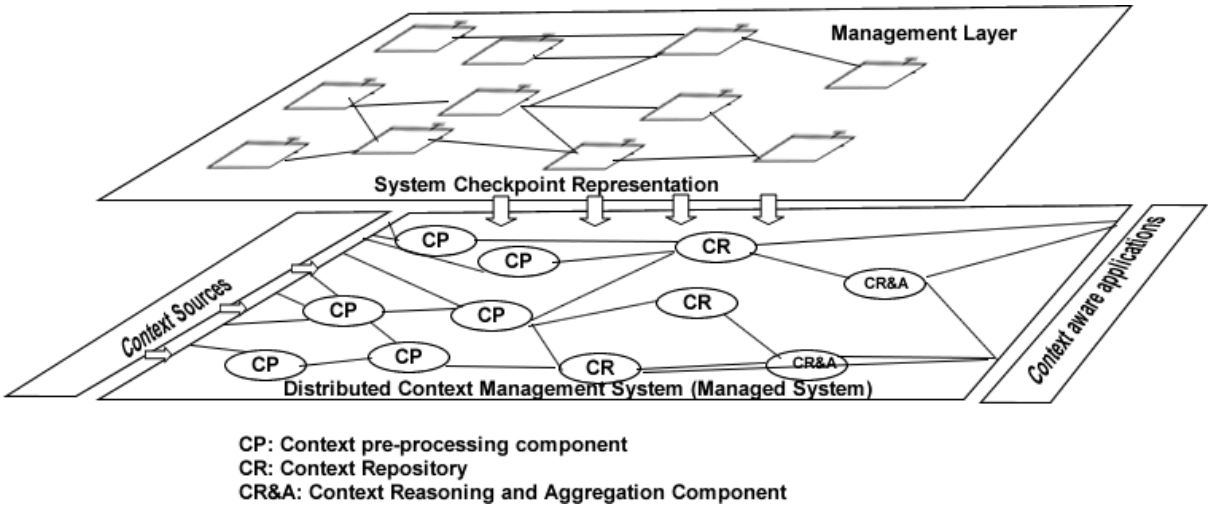


Figure 4.5: System Checkpoint Model Representation

The components shown at the management layer (see Figure 4.5) implement wrappers which provide control over the deployed software components. A wrapper is composed of a list of administration methods. Each method has an implementation and a list of parameters (values) that must be passed to the method upon invocation. One example of administration methods is a start method which can be invoked to launch a software component by taking parameters like the shell command that launch the server, and environment variables. Meta-model representation of the wrapper is described in figure 4.6.

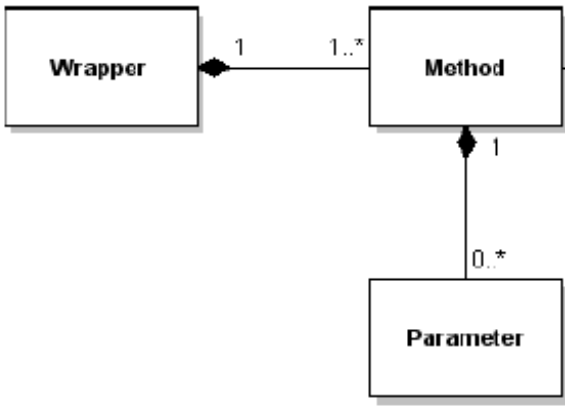


Figure 4.6: Meta-model for Wrapper Description

When one of the components fails in the managed system, fault manager component will use the system checkpoint in order to find the configuration of the failed component prior to failure. The failure manager uses this configuration information and reconfigures the failed component according to the repair policy generated. The system checkpoint is then automatically updated according to the new configuration

4.3.1.3. Deployment Manager

Deployment manager is responsible for deploying the architecture of the managed elements i.e. distributed context management components. It communicates with the software installation service to deploy the necessary software on the allocated node, if is not already installed. It also used to generate the initial system checkpoint model.

4.3.1.4. Software Installation

The Software Installation component allows installation of software on a node if it does not already exist. This component implements a repository that stores software libraries needed by managed systems and libraries of the wrappers associated with these managed systems.

4.3.2. Fault Manager

The fault manager module is the core component that plays the most important role in the fault management architecture. This component is responsible for detecting faults in the managed system, identify the faulty component, prepare repair policy and carry out actions to resolve the faults. The sub-components are discussed below

Fault Detector: is in charge of detecting faults in the system. High-quality fault detection, the first step of fault tolerance process, not only prevents loss of resources but also lessens correction time. To this purpose, the fault detector monitors the health of the managed system through probe components installed on the node hosting the managed system. These probes are implemented using heartbeat techniques and periodic observations include software component failure and node failure observations.

According to figure 4.7, each managed component (MC_i) will be monitored by its respective probing component which sends alive message to the fault detector after a specific time period. This period is called the heartbeat period (Hp). For each managed component, it takes Δt_i time to send alive message to the probe component, where i is the component number. So, the detector will get the message after (Hp+ Δt_i) time.

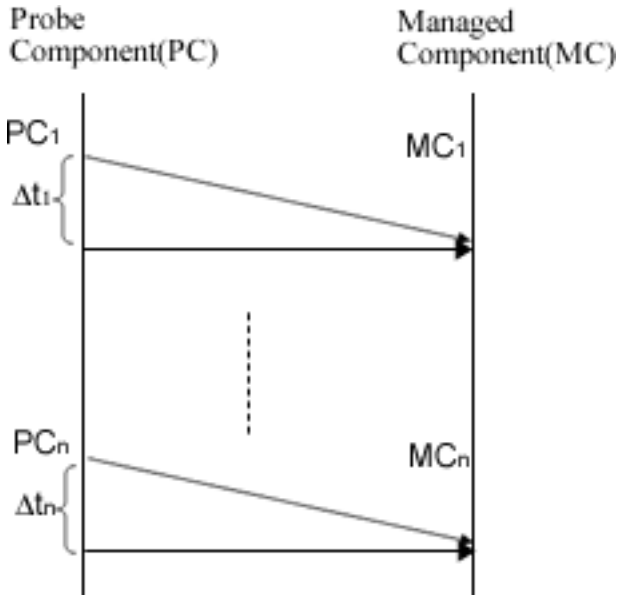


Figure 4.7: Component Monitoring Mechanism

In this scheme, each probe component will send any one of the following messages to the detector.

- OK message: It simply sends a packet containing “OK” string. It’s nothing but a heart beat message.
- If the detector doesn’t get such message for a pre-fixed threshold period of time, right away it will generate an event. An event is defined as an event type (node or component failure) and the name of the component which generated the fault. The event is then transmitted to the Planner component in order to prepare a repair policy accordingly.

Planner: is responsible for accepting the event from the fault detector component and extract a repair policy. A repair policy is an ordered set of actions (self-configuring, self-healing) used to repair detected faults. The functionalities of this sub-component can be described through the following steps.

- accept failure events from fault detector component
- extract a repair policy which is used to repair the detected faults. The Planner uses the name of the component which generates the fault to extract the best repair policy.
- send the repair policy to the policy executor

Policy Executor: is responsible for implementing repair policy. We aim at autonomously repairing the managed system by replacing the failed component by a new one. The policy executor implements the repair policy to rebuild the failed managed system as it was prior to the occurrence of the failure. To this purpose, the policy executor uses system checkpoint to retrieve the necessary information about the failed component.

The functionalities of this sub-component can be described through the following steps.

- Accept a repair policy from the planner sub-component
- introspect the failed component configuration and create a new instance of the failed component from the system checkpoint
- rebind the new component with the existing component architecture.

To illustrate the functionalities, we present two repair policy examples as follows.

Example 1. Failure Type: Software component failure

Repair policy:

- stop bindings to the failed component
- start a new (possibly the same) component replacing the previous one in the same node in consultation with the system checkpoint
- reinstate failed bindings

Example 2. Failure Type: Node failure

Repair policy:

- stop bindings to all components deployed in the node
- request new node from the node allocator (if the node allocator can't find any node for deploying components, node availability fault will be generated)
- install new components in consultation with the system checkpoint that are isomorphic to the failed one on the newly selected node
- reinstate all failed bindings

The repair policy can be described as an activity diagram shown in figure 4.8.

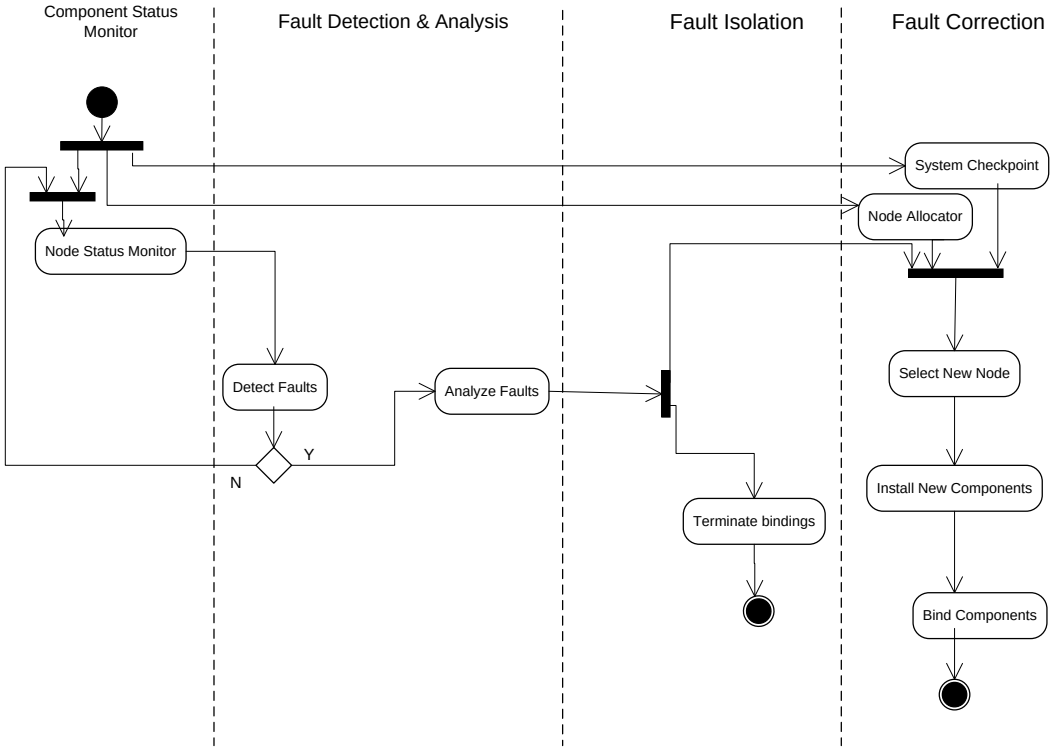


Figure 4.8: Activity Diagram for Node Failure Management

5. Prototype Implementation and Evaluation

In this thesis work, we have proposed architecture aimed at supporting fault tolerant context information provisioning to context aware systems. In chapter four we presented the details of our architecture. This chapter discusses the implementation of the architecture by stating the overview of the prototype, tools and technologies used for implementation, the implementation details and evaluation results.

5.1. Overview

The prototype implementation developed in this thesis is basically used as a proof of concepts introduced in the explanation of the proposed fault management architecture. To achieve this, the majority of the components in the architecture have been implemented as per the theoretical specifications formulated for each component.

The prototype implementation lets us to deploy the distributed components of the context management system autonomously into four nodes. The nodes are used as context server, ontology server, rule server and context management component (CoCAsystem). Then the fault manager component which is installed on the fault manager node will continuously monitor each of the deployed components and repair the faulty component together with the system checkpoint and node manager component.

5.2. Tools and Technologies Utilized for Implementation

The development tools that have been used to implement the fault management system are:

- **FRACTAL component model:** The FRACTAL component model is a general component model that is intended to implement, deploy, and manage (i.e. monitor, control, and dynamically configure) complex software systems, including in particular operating systems and middleware [41]. The main features of the model are:
    - Composite components (components that contain sub-components): in order to have a uniform view of applications at various levels of abstraction.
    - Shared components (sub-components of multiple enclosing composite components): in order to model resources and resource sharing while maintaining component encapsulation.

- Introspection capabilities: in order to monitor and control the execution of a running system.
- Re-configuration capabilities: in order to deploy and dynamically configure a system.

A FRACTAL component (Figure 5.1) can be understood generally as a run-time entity that is encapsulated and has one or more interfaces. Interfaces can be of three kinds: server interfaces, which correspond to access points accepting incoming method calls; and client interfaces, which correspond to access points supporting outgoing method calls and the control interfaces allow managing the element's attributes and bindings with other elements. Collocate

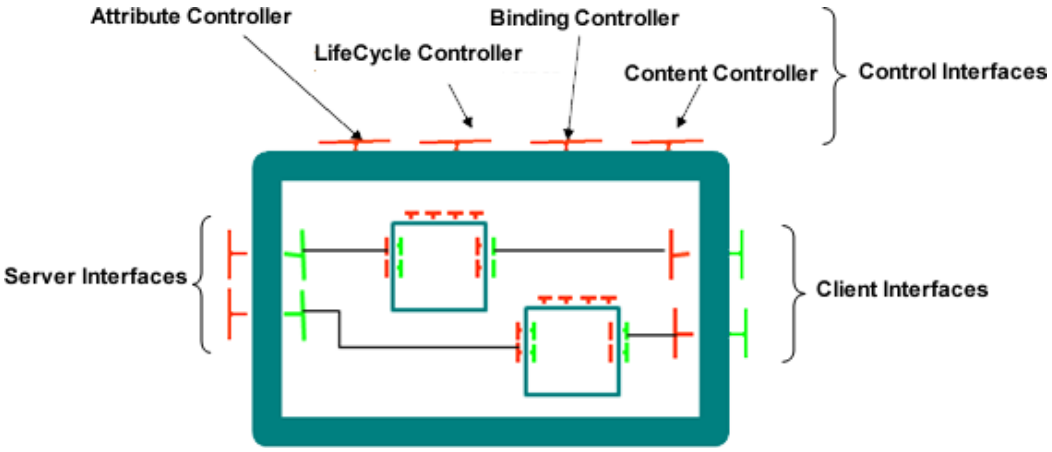


Figure 5.1: A FRACTAL Component

The FRACTAL specification provides several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features [41].

- Attribute controller: An attribute is a configurable property of a component. A component can provide an AttributeController interface to expose getter and setter operations for its attributes.
- Binding controller: A component can provide the BindingController interface to allow binding and unbinding its client interfaces to server interfaces by means of primitive bindings.
- Content controller: A component can provide the ContentController interface to list, add, and remove sub-components in its contents.

- Life-Cycle controller: A component can provide the LifeCycleController interface to allow explicit control over its main behavioral phases, in support for dynamic reconfiguration. Basic lifecycle methods supported by a LifeCycleController interface include methods to start and stop the execution of the component.

- TUNe (Toulouse University Network) used for as administration software which provides a higher level interface for describing the encapsulation of software elements in fractal components and the deployment of context management system architecture in a distributed environment.

- Topcased[1]: is a software environment primarily dedicated to the realization of critical embedded systems including hardware and/or software. It is used for creating UML models which will be used in the deployment and reconfiguration phase. It produce XML files from the UML models which later on used by TUNe to create fractal components.

- Java 2 Standard Edition (J2SE) version 1.5.0_16 is used for developing the java classes used in the prototype. Java RMI technologies are used as communication method between the distributed software components.

- MySql database server version 5.0.22 is used for persistent context data management on the context management server

- Jena Semantic web framework version 2.6.0 is used for generating an RDF model and implementing the ontology and user-defined reasoning rules

- Ubuntu 9.0 operating system used as a working operating system

5.3. Implementation Details

5.3.1. Modules on Manager System

Node Allocator: this module handles the representation and allocation of nodes which participate in the distributed context management system. We deploy and administer the context management system on a distributed nodes assigned by the node allocator. The Node Allocator is an important component that manages the physical distribution of the software

[1] http://www.topcased.org/

elements based on preferences of each class element. We represent a node in UML class as shown in figure 5.2.

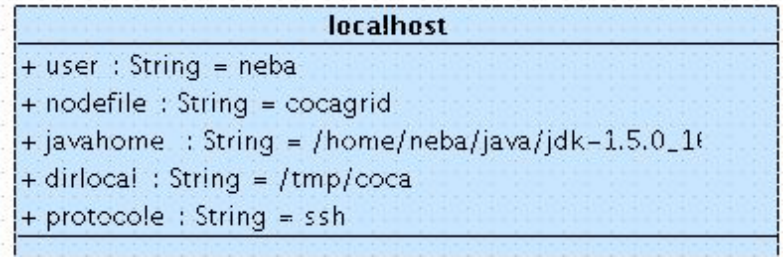


Figure 5.2: UML Class for Node Component

The Node component representation consists of the following attributes:

- user: is an attribute which gives the login username while accessing the remote node
- nodefile: corresponds to the list of machines in the distributed system. The node allocator uses this set of machines and allocate according to the repair policy. In our prototype the text files consists of nodes represented by their IP address. These are 10.6.20.229 used as a context server, 10.6.20.88 used as an ontology repository server, 10.6.20.68 used as a rule repository server, 10.6.20.123 used as the context storing and reasoning server.
- javahome: is an attribute which gives the path for java virtual machine
- protocol: is an attribute which gives login protocol to access the remote node
- dirlocal: : is the directory that will be used locally on machines to decompress the archive files.

Using topcased software, we generate an XML representation of the node which later used in TUNe to create a fractal component for each node lists in the attribute nodefile. Using the generated fractal component control interface, we manage the node elements. XML representation of the node diagram is described in figure 5.3.

```
<packagedElement xmi:type="uml:Package" xmi:id="_w8IxIM37EdqwVrslYOdUDA" name="grid">
        <packagedElement xmi:type="uml:Class" xmi:id="_5Zh6IIhiEdyfT8U81_DOUA" name="localhost">
                <ownedAttribute xmi:id="_7ndC4IhlEdy85rL0FCbYhw" name="user"
                type="_HN4TcYhmEdy85rL0FCbYhw">
                        <defaultValue xmi:type="uml:LiteralString" xmi:id="_KGRc4IhmEdy85rL0FCbYhw"
                        value="neba"/>
                </ownedAttribute>
                <ownedAttribute xmi:id="_9Ay0YohlEdy85rL0FCbYhw" name="nodefile"
                type="_HN4TcYhmEdy85rL0FCbYhw">
                        <defaultValue xmi:type="uml:LiteralString" xmi:id="_NMDiEIhmEdy85rL0FCbYhw"
                value="cocagrid"/>
                </ownedAttribute>
                <ownedAttribute xmi:id="_9_DzcIhlEdy85rL0FCbYhw" name="javahome"
                type="_HN4TcYhmEdy85rL0FCbYhw">
                        <defaultValue xmi:type="uml:LiteralString" xmi:id="_P2YaUIhmEdy85rL0FCbYhw"
                        value="/home/neba/java/jdk1.5.0_16"/>
                </ownedAttribute>
                <ownedAttribute xmi:id="__bNoYIhlEdy85rL0FCbYhw" name="dirlocal"
                type="_HN4TcYhmEdy85rL0FCbYhw">
                        <defaultValue xmi:type="uml:LiteralString" xmi:id="_RqLTMIhmEdy85rL0FCbYhw"
                        value="/tmp/coca"/>
                </ownedAttribute>
                <ownedAttribute xmi:id="_APoH4ohmEdy85rL0FCbYhw" name="protocole"
                type="_HN4TcYhmEdy85rL0FCbYhw">
                        <defaultValue xmi:type="uml:LiteralString" xmi:id="_V0Yp0IhmEdy85rL0FCbYhw"
                        value="ssh"/>
                </ownedAttribute>
        </packagedElement>
</packagedElement>
```

Figure 5.3: XML representation for the node diagram

System Checkpoint

In presence of failures, parts of the managed system information i.e. architecture, bindings between components and configuration information are lost. Therefore, in order to correct the fault, the fault manager needs to represent and maintain knowledge on the system. This representation provides a view of the runtime system architecture and can be introspectable and causally connected to runtime components. We will represent this view in a UML class diagram so that we can easily describe the overall organization of a software infrastructure to be deployed.

The UML specification defines only the initial architecture of the managed system which can later evolve within the framework of the architectural model and is used for failure recovery. Figure 5.4 represents a UML class representation for the system checkpoint.

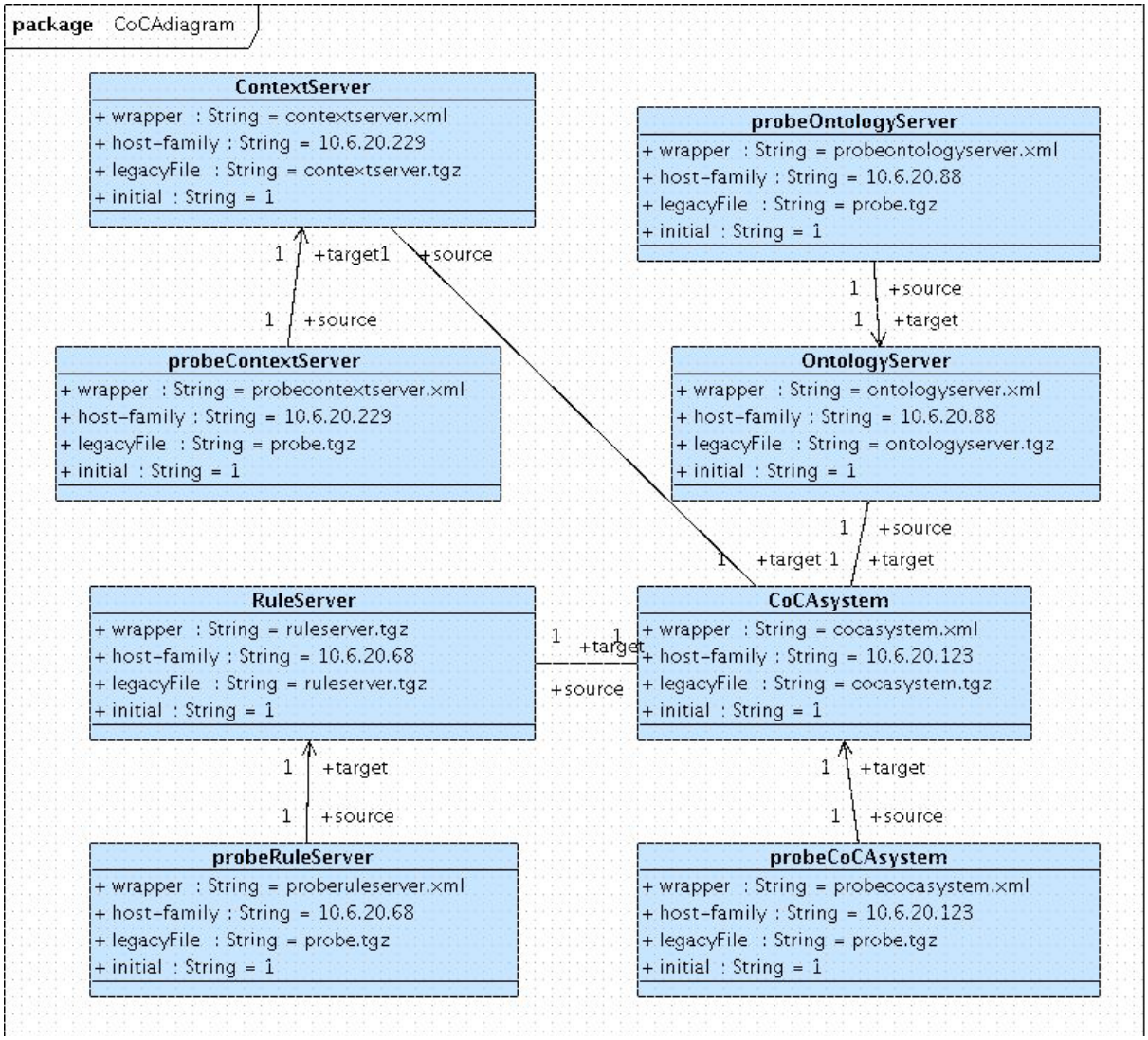


Figure 5.4: System checkpoint class diagram of the Managed System

As shown in the figure 5.4, the diagram consists of software element which corresponds to software which can be instantiated in several component replicas.

The binding between probe components and software component allows us to navigate in the component architecture. A software element includes a set of configuration attributes (all of type String) which are used during deployment time. The descriptions of each attributes used in the above diagram are

- **wrapper**: is an attribute which gives the name of the wrapper description file of the wrapper
- **legacyFile**: is an attribute which gives the name of the archive file which contains the managed software components and configuration files

- **hostFamily**: is an attribute which gives a hint regarding the dynamic allocation of the nodes where the software should be deployed
- **initial**: is an attribute which gives the number of instances which should be deployed. The default value is 1.

Similarly to the node class diagram, we generate an XML file using topcased software tool for the system checkpoint. Figure 5.4 describes part of the XML file which shows the CoCAsystem software component. The full list of the XML file is described in Appendix B.

```
<packagedElement xmi:type="uml:Package" xmi:id="_DpGvsIhnEdy85rL0FCbYhw" name="deploiement">
        <packagedElement xmi:type="uml:Class" xmi:id="_LiAloIhnEdy85rL0FCbYhw"
name="CoCAsystem">
                <ownedAttribute xmi:id="_NFYc4IhnEdy85rL0FCbYhw" name="wrapper"
                type="_HN4TcYhmEdy85rL0FCbYhw">
                        <defaultValue xmi:type="uml:LiteralString" xmi:id="_S5ctQIhnEdy85rL0FCbYhw"
                        value="cocasystem.xml"/>
                </ownedAttribute>
                <ownedAttribute xmi:id="_Od9ZYIhnEdy85rL0FCbYhw" name="legacyFile"
                type="_HN4TcYhmEdy85rL0FCbYhw">
                        <defaultValue xmi:type="uml:LiteralString"
                        xmi:id="_X3UVcIhnEdy85rL0FCbYhw" value="cocasystem.tgz"/>
                </ownedAttribute>
                <ownedAttribute xmi:id="_M_tdQIhoEdy85rL0FCbYhw" name="host-family"
                type="_HN4TcYhmEdy85rL0FCbYhw">
                        <defaultValue xmi:type="uml:LiteralString"
                xmi:id="_P6vD8IhoEdy85rL0FCbYhw" value="10.6.20.123"/>
                </ownedAttribute>
</packagedElement>
.
.
.
```

Figure 5.5: XML Representation for System Checkpoint Architecture

Upon deployment, the above XML schema is parsed and for each element, a number of Fractal components are created. These Fractal components implement the wrappers for the deployed software, which provide control over the software.

A wrapper is written in an XML format which is composed of a list of administration methods. Each method has an implementation and a list of parameters (values) that must be passed to the method upon invocation. We implement start and stop methods for controlling

the activity of the software, and a configure method for reflecting the values of the attributes defined in the system checkpoint diagram and in the configuration files of the software.

Figure 5.6 shows an example of wrapper specification which wraps a ContextServer software component. It defines start and stop methods which can be invoked to launch/stop the deployed ContextServer component, and a configure method which reflects configuration attributes in the configuration file of the ContextServer software.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
        <wrapper name="ContextServer">
                <method name="start" key="appli.wrapper.util.GenericStart"
        method="start_with_pid_linux">
                <param value="java –cp –Djava.security.manager –Djava.security.policy=java.policy
        ContextServer"/>
                <param value="LD_LIBRARY_PATH=$dirLocal"/>
        </method>

        <method name="configure"  key="appli.wrapper.util.ConfigurePlainText="configure">
                <param value="$dirLocal/$compName-cfg"/>
                <param value="="/>
                <param value="parentName:$CoCAsystem.compName"/>
                <param value="name:$compName"/>
        </method>

        <method name="stop" key="appli.wrapper.util.GenericStop" method="stop_with_pid_linux">
                <param value="$PID"/>
        </method>
</wrapper>
```

Figure 5.6: Wrapper Specification for ContextServer Component

The start method takes as parameters the shell command that launch the ContextServer software component, and the environment variables that should be set:

- <param value="java –cp –Djava.security.manager –Djava.security.policy=java.policy ContextServer"/> passes the java command to start ContextServer on the respective node
- LD_LIBRARY_PATH=$dirLocal is an envionment variable

The configure method is implemented by the ConfigurePlainText java class of the TUNe system. The configuration method generates a configuration file composed of <attribute,

value> pairs. Finally, the stop method takes the process Id as parameter to stop the running component.

Fault Detector: this component handles detection of faults in the managed system. The detector is implemented as probe components which monitor the nodes and components of the managed infrastructure. These probe components are attached to each components of the managed system that control and detect failures and issue events. The probe components monitor the health of managed system by executing an ssh(Secure Shell) command and checks the process id of the managed software. If it can't get the process id it produces a not alive message. This error message will be sent to the planner along with the component name which generates the error. Figure 5.7 shows code excerpt for isAlive() method of the probe component.

We define probe components in the UML diagram (as shown in figure 5.4) by linking a probe element with a software element (component) in the deployment architecture.

```
      public boolean isAlive() throws RemoteException
      {
            String commande="ssh "+this.pid_node+" ps -p "+pid_to_check+" -o
command=";
            String str = "";
            try
            {
                  BufferedReader bf = new BufferedReader(new
InputStreamReader(Runtime.getRuntime().exec(commande).getInputStream()));
                  try
                  {
                        String tmp = bf.readLine();
                        while (!tmp.equals(null))
                        {
                              str=str+tmp;
                              tmp = bf.readLine();
                        }
                  }
                  catch (Exception e)
                  {
                  }
                  if(str.equals(""))
                        return false;
                  else
                        return true;
            }
            catch(Exception e)
            {
                  return false;
            }

      }
```

Figure 5.7: Java Code Excerpt from the Probe Component

Planner: is responsible to extract a repair policy based on the event received from the fault detector component. We use UML state diagrams to specify repair policy for faulty components. Such a state diagram defines the workflow of operations that must be applied in reaction to an event. Figure 5.8 shows a repair policy which should be followed in reaction to a ContextServer software component failure in context management system.

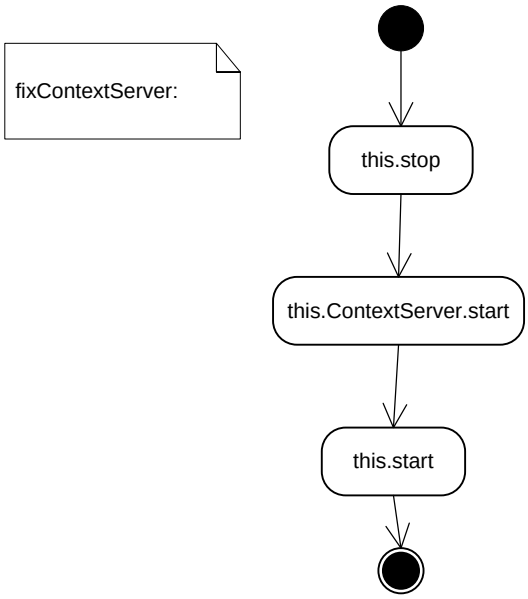


Figure 5.8: Reconfiguration Policy for ContextServer Component Failure

The event name (fixContextServer) which is generated by a probeContextServer component instance is used by the Planner to extract the repair policy. In figure 5.8, the *this* variable is the name of probeContextServer component instances. The details of the above state chart is describes as follows.

- this.stop will invoke the stop method on the probing component to prevent generation of multiple events
- this.ContextServer.start will invoke the start method on the ContextServer component instance which is linked with the probe. This is the actual repair of the faulty ContextServer component.
- this.start will restart the probe associated with the ContextServer.

Likewise a start chart will be used to start the deployed context management environment, as illustrated in Figure 5.9. The start chart diagram first ensures that configuration files must be generated, and then the servers must be started following the order ContextServer, OntologyServer, RuleServer and CoCAsystem. For each type of server, the server is started before its probe component.

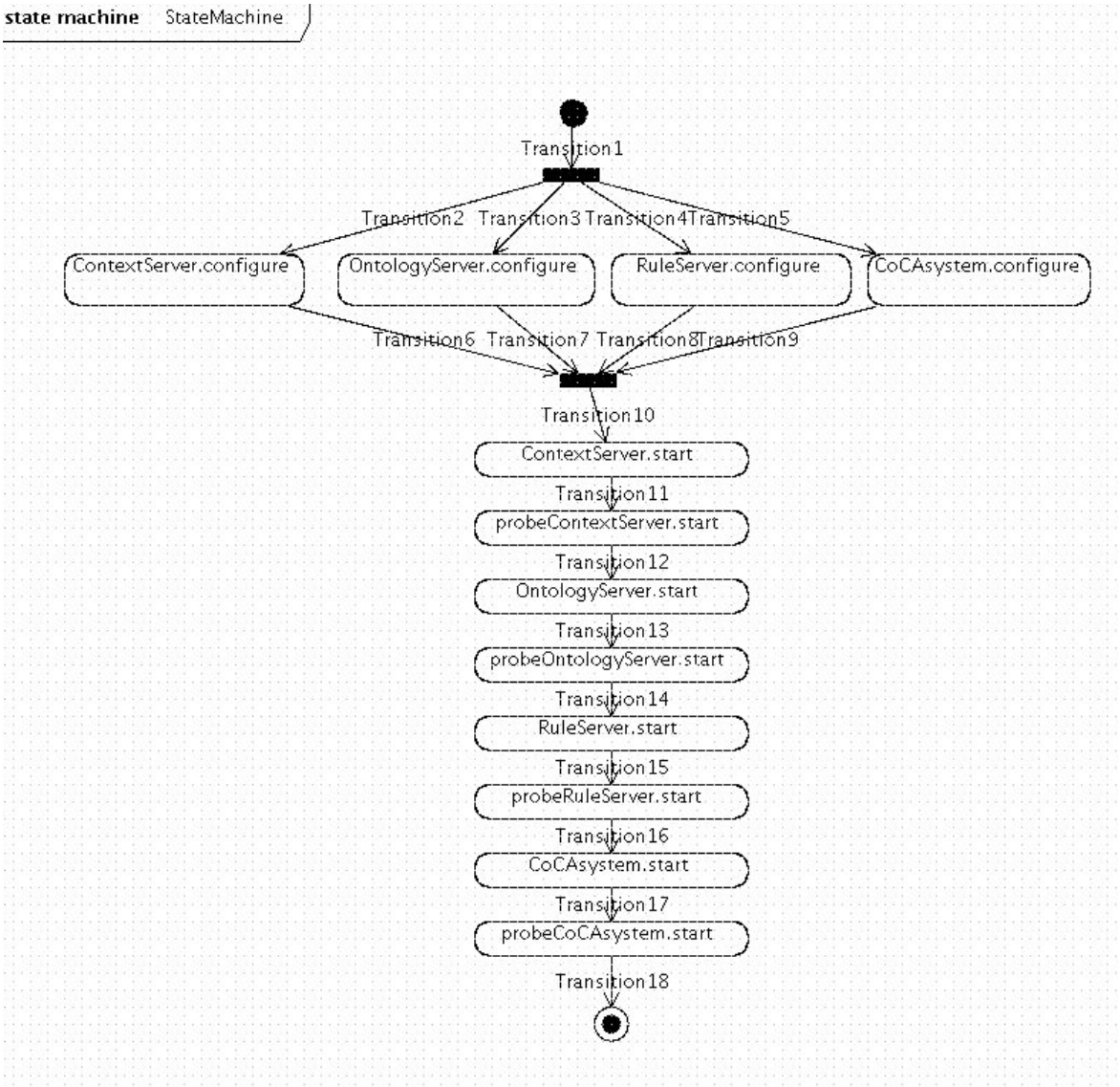


Figure 5.9: Start activity diagram to deploy the distributed software components

5.4. Evaluation

This section first describes our experimental environment, and then we present the evaluation results for our proposed fault tolerance architecture.

5.4.1. Experimental Environment

The experiment was performed on five nodes consisting of a context server, ontology server, rule server, CoCAsystem server and a dedicated fault manager node. The nodes have Ubuntu operating system and with specification x86-compatible machines, with 2GB RAM and 3.4GHz processor, connected via a 100Mb/s Ethernet LAN to form a distributed environment. Figure 5.10 shows the experimental environment setup.

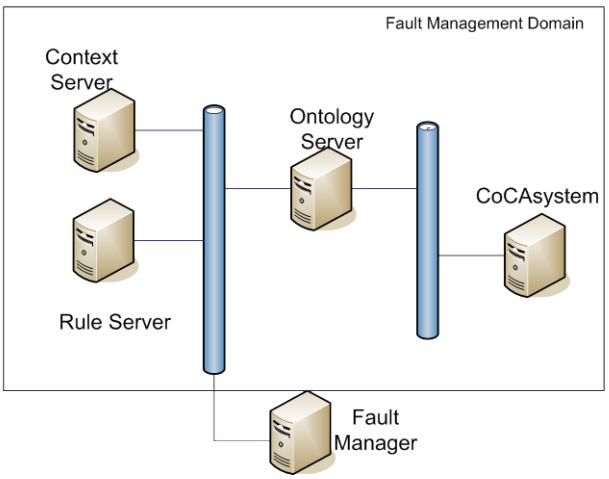
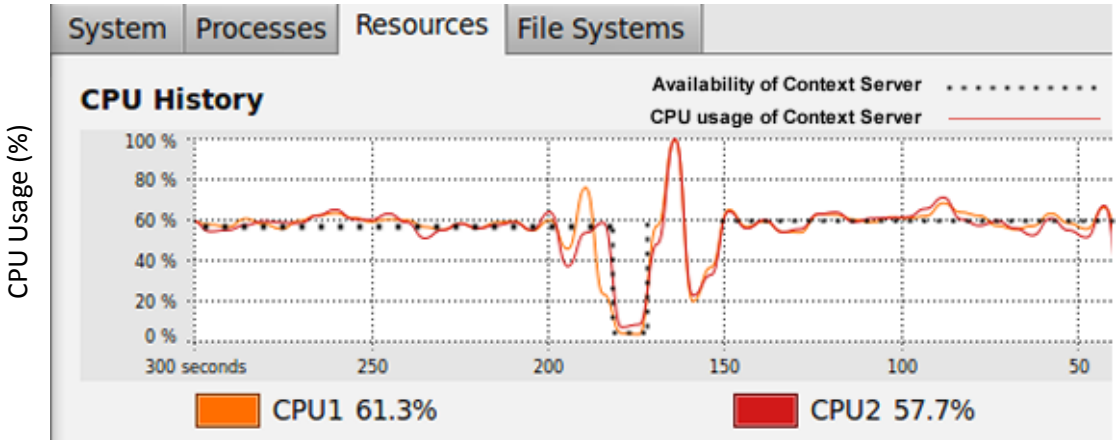
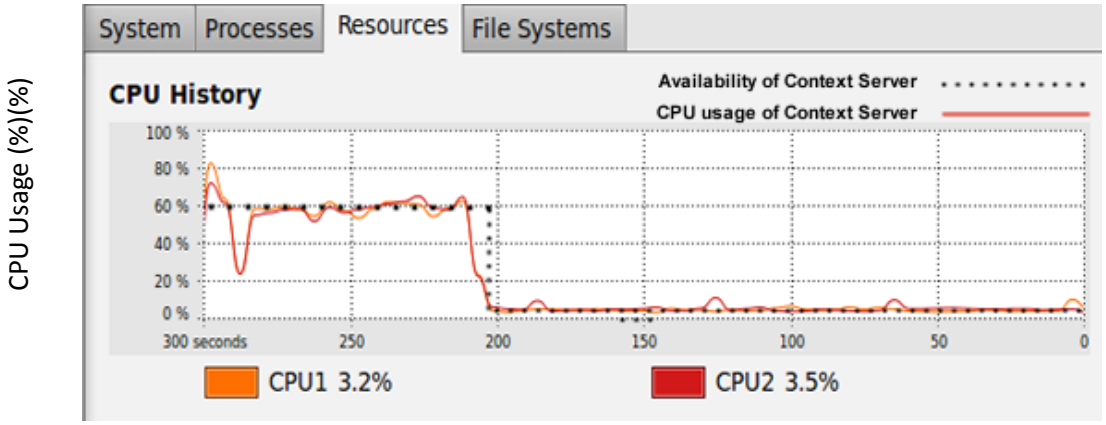


Figure 5.10: Experimental Environment Setup

We conducted two types of experiments on the distributed context data management system with and without the fault manager node i.e. Self-managed system versus Manual-managed system to demonstrate our proposed architecture. In the first type of experiment, without the fault manager node, when we simulate a fault by killing the process ID of the context server, the CoCAsystem server can't find any context data and can't continue providing context data properly. However, in the second test when Fault manager is used, the failure is automatically repaired by replacing the failed server component by a new one, and thus guaranty service continuity. Figure 5.11 shows the results of the two experiments in a graph form.



(a)



(b)

Figure 5.11: Experimental results (a) with fault manager and (b) without fault manager

5.4.2. Performance Overhead

In order to measure the possible performance overhead induced by the fault management architecture, we compared two executions on the same context management system: when it is run with the fault manager and when it is run without fault manager. During the experiments, the managed system has been assigned a certain task so that its execution under the control of fault manager induced no dynamic reconfiguration. Table 5.1 and Table 5.2 show the results we got while conducting the experiment.

Table 5.1. Performance Overhead on CPU Usage (%)

No	With Fault Manager	Without Fault Manager
1	51.60	47.05
2	67.76	54.75
3	65.77	61.77
4	58.37	66.89
5	45.54	55.21
6	66.22	56.90
7	71.04	51.22
8	77.09	66.71
9	57.65	59.65
10	47.41	55.08
Average	**60.845**	**57.523**

Table 5.2. Performance Overhead on Memory Usage (%)

No	With Fault Manager	Without Fault Manager
1	19.7	16.98
2	21.9	16.3
3	23.3	16.96
4	27.7	23.21
5	25.3	17.67
6	22.7	17.26
7	25.5	24.59
8	24.2	22.06
9	23.4	18.43
10	25.6	15.25
Average	**23.9**	**18.9**

From our experiment, we noticed a slight memory overhead (23.9% vs. 18.9%) that can be linked with the creation of internal software components by the fault manager. However, the fault manager does not induce an observable overhead on CPU usage.

6. Conclusion and Future Work

In this thesis, we have proposed a fault management architecture which includes fault detector, fault repair planner, and fault repair executor components for a distributed context data management environment. In this chapter, first we give a brief summary of our approach to fault detection, fault notification, and fault correction. This summary has been portrayed in section 6.1. The contribution of our thesis has been described in section 6.2. Finally, we give some directions for future work in section 6.3.

6.1. Conclusion

We have proposed a solution for fault detection, notification and correction in distributed context data management system. Our approach is based on wrapping the existing distributed software components (legacy entities) in Fractal components and deploys them in a distributed environment. This wrapping provides us a management layer which has a uniform view of management interfaces, introspection capabilities and architectural view of legacy software. Then we administer the overall software environment as a component architecture using the manager system. The manager system is composed of fault manager and supplementary services.

We implement the fault manager using three components: the fault detector, fault repair planner and plan executor. The fault detection mechanism uses a probing method which continuously monitors the live of the managed components through the wrapped component interfaces and produces an event in case of failure. An event is composed of failure type and the name of the component which generated the fault.

Based on the produced event, the fault repair planner introspect the failed component configuration from the system checkpoint and create a new instance from it. Then it prepares an equivalent architecture in a repair plan. The repair plan is then used by the fault repair executor to rebind the new component with the existing component architecture. The system checkpoint is also updated by the executor to reflect the current system architecture view.

In order to demonstrate the validity of the proposed fault tolerance architecture, we use a distributed context data management system for which we developed a prototype implementation (discussed in chapter 5). In the prototype, we have implemented the major components of the proposed architecture and conducted an evaluation of how the architecture

detects and repairs faulty components. Results obtained from our experiment show that the system has been automatically and transparently recover from failures.

6.2. Contributions

In this thesis, we first identified the shortcomings that the existing distributed context management systems for context aware systems experience with regards to handling failures occurred in the system. These shortcomings were lack of a full fledged fault management approach and lack of adequate consideration for faults that arise from the distributed software components.

To address these shortcomings, we proposed (as explained in chapter 4) a component base fault tolerance architecture that aims at detecting and correcting failures occurred in the distributed context management system autonomously. Our major contributions in this work can be summarized as follows.

- Full-fledged fault management architecture: we have developed a full-fledged fault management architecture which not only detect faults in the system but also identify the faulty component, prepare a repair policy based on the detected fault type and execute the policy to make the context management system fault free. We use a system checkpoint to provides a view of the runtime system architecture
- In addition we also introduced mechanism of handling faults that originates from the distributed software components. We used a component based approach which provides a uniform view of the software environments and administration facility for managing and administering so that the managed architecture will be dynamically reconfigurable when faults occurred.

6.3. Future Works

The followings are some of the envisaged future works that can be considered as a continuation to this work:

- Handling state of faulty component: currently our approach didn't have functionality to handle the state of a faulty component. In case of component failure, we redeploy the component using our fault manager. However, some components maintain state during operation, and may require that state to be restored after a crash. The use of log

file to record the state of the component can be a starting point towards achieving the problem.

- Failures in the management system: our approach is mainly focused on repairing failures that occurred in the managed system. Faults that may occur in the manager system will not be detected and hence can’t be repaired. We believe that replicating the manager system in the distributed environment can alleviate the problem.
- We also intend to apply the fault management architecture on other use cases such as peer-to-peer environments. Such use cases would confirm the generality of the proposed approach.

7. References

[1] Hui Lei, ‘Context Awareness: a Practitioner’s Perspective’, Proceedings of the 2005 International Workshop on Ubiquitous Data Management, 2005

[2] Shameem Ahmed, ‘Self-healing for Autonomic Pervasive Computing’, master’s thesis, Marquette University, USA, 2006

[3] Guanling Chen; Ming Li; Kotz D., ‘Solar: An open platform for context-aware mobile applications', In Proceedings of the First International Conference on Pervasive Computing, 2002

[4] Rice, A.C.; Beresford, A.R.; ‘Dependability and accountability for context-aware middleware systems’ , Pervasive Computing and Communications Workshops, PerCom Workshops 2006.

[5] Ahmed, B.; Young-Koo Lee; Sungyoung Lee; Yonil Zhung, ‘Scenario Based Fault Detection in Context-Aware Ubiquitous Systems using Bayesian Networks’, Computational Intelligence for Modeling, Control and Automation, and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on Volume 1, 28-30 Page(s):414 - 420

[6] Shiva Chetan; Anand Ranganathan; Campbell, R.; ‘Towards fault tolerance pervasive computing’, Technology and Society Magazine, IEEE Volume 24, Issue 1, 2005 ,Page(s):38 - 44

[7] Mohammed Toure, Girma Berhe, Patricia Stolf, Laurent Broto, Noel Depalma, Daniel Hagimont, "Autonomic Management for Grid Applications," pdp, pp.79-86, 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), 2008

[8] Anand Ranganathan, Jalal Al-Muhtadi, Jacob Biehl, Brian Ziebart, Roy H. Campbell, Brian Bailey, "Towards a Pervasive Computing Benchmark," percomw, pp.194-198, Third IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'05), 2005

[9] Karen Henricksen, Jadwiga Indulska, Andry Rakotonirainy, ‘Modeling Context Information in Pervasive Computing Systems’, Proceedings of the First International Conference on Pervasive Computing, Pages: 167 - 180 , 2002

[10] Tang Xiaosheng Shen Qinghua Zhang Ping , ’A Distributed Context Aware Model for Pervasive Service Environment’, Wireless Pervasive Computing, 2006 1st International Symposium on, 2006, 5pp

[11 Dejene Ejigu Dedefa, ‘Context Modeling and Collaborative Context-Aware Services for Pervasive Computing’, ph.d thesis, 2007, INSA de Lyon, France

[12] Mattern F., Sturn P. ‘From Distributed Systems to Ubiquitous Computing – State of the Art’. Trends and Prospects of Future Networked systemspp. 3-25. Fachtagung "Kommunikation in Verteilten Systemen" (KiVS), Leipzig, , 2003. Springer-Verlag, Berlin, 2003

[13] Guanling Chen , ‘Solar: Building A Context Fusion Network for Pervasive Computing’, ph.d thesis, DARTMOUTH COLLEGE

[14] Baldauf, M., Dustdar, S. and Rosenberg, F. ‘A survey on context-aware systems’, Int. J. Ad Hoc and Ubiquitous Computing, Vol. 2, No. 4, 2007, Page(s):263–277

[15] Dey A. K., Abowd G. D. ‘Towards a Better Understanding of Context and Context-Awareness’. In: Proceedings of the CHI Workshop on the What, Who, Where, and How of Context-Awareness, April 2000, The Hague, The Netherlands.

[16] Thomas Buchholz, Michael Krause, Claudia Linnhoff-Popien, Michael Schiffers, "CoCo: Dynamic Composition of Context Information," mobiquitous, pp.335-343, First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04), 2004

[17] Robert Schmohl, Uwe Baumgarten , ‘Context-aware Computing: a Survey Preparing aGeneralized Approach’, Proceedings of the International MultiConference of Engineers and Computer Scientists 2008 Vol I IMECS 2008, 19-21 March, 2008, Hong Kong

[18] Anind K. Dey , ‘Understanding and using context’, Personal and Ubiquitous Computing, Volume 5 , Issue 1 (February 2001),Pages: 4 - 7

[19] Dey, A.K., Salber, D., Abowd, G.D.: 'A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications'. Human-Computer Interaction 16 (2001) 97–166

[20] Romain Rouvoy, Denis Conan, and Lionel Seinturier, "Software Architecture Patterns for a Context-Processing Middleware Framework," IEEE Distributed Systems Online, vol. 9, no. 6, 2008,

[21] Pyrros Bratskas, Nearchos Paspallis, and George Papadopoulos, ‘An Evaluation of the State of the Art in Context-aware Architectures’, 16th International Conference on Information Systems Development (ISD), Galway, Ireland, August 29-31, 2007, Springer Verlag

[22] Harry Chen. 'An Intelligent Broker Architecture for Pervasive Context-Aware Systems.' PhD thesis, University of Maryland, Baltimore County, Baltimore MD, USA, 2004.

[23] Terry Winograd. 'Architectures for context'. Human Computer Interaction, 16 (2-4), 2001.

[24] Jason I. Hong and James A. Landay. 'An infrastructure approach to context-aware computing'. Human-Computer Interaction (HCI) Journal, 16(2-3), 2001

[25] Karen Henricksen1, Jadwiga Indulska2, Ted McFadden1, and Sasitharan Balasubramaniam, ‘Middleware for Distributed Context-Aware Systems’, Lecture notes in computer science ISSN 0302-9743

[26] Ricardo Couto A. da Rocha and Markus Endler. ‘Evolutionary and efficient context management in heterogeneous environments’. In MPAC ’05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, pages 1–7, New York, NY, USA, 2005. ACM Press

[27] Gregory Biegel and Vinny Cahill. ‘A framework for developing mobile, context-aware applications’, percom, 00:361, 2004.

[28] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K. S. Gupta. ‘reconfigurable contextsensitive middleware for pervasive computing’. IEEE Pervasive Computing, 1(3):33–40, 2002

[29] Dejene Ejigu, Marian Scuturici, Lionel Brunie, "CoCA: A Collaborative Context-Aware Service Platform for Pervasive Computing," itng, pp.297-302, International Conference on Information Technology (ITNG'07), 2007

[30] Christos B. Anagnostopoulos, Athanasios Tsounis, and Stathes Hadjiefthymiades. 'Context awareness in mobile computing environments'. Wirel. Pers. Commun., 42(3):445–464, 2007

[31] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004

[32] Peizhao Hu; Indulska, J.; Robinson, R.; 'An Autonomic Context Management System for Pervasive Computing', Pervasive Computing and Communications, Sixth Annual IEEE International Conference on 17-21 ,2008 Page(s):213 - 223.

[33] Waltenegus Dargie , 'The Role of Context-Aware Computing in Developing Self-Managing Systems', Context-Aware Computing and Self-Managing Systems, March 25, 2009 pp 405

[34] Kephart, J.O.; Chess, D.M., 'The vision of autonomic computing', Computer, Volume 36, Issue 1, Jan. 2003 Page(s):41 – 50

[35] Shang-Wen Cheng; An-Cheng Huang; Garlan, D.; Schmerl, B.; Steenkiste, P.; , 'Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure', Autonomic Computing, 2004. Proceedings. International Conference on 17-18 May 2004 Page(s):276 – 277

[36] Khalid, A.; Haye, M.A.; Khan, M.J.; Shamail, S.; 'Survey of Frameworks, Architectures and Techniques in Autonomic Computing, Autonomic and Autonomous Systems', 2009. ICAS '09. Fifth International Conference on 20-25 April 2009 Page(s):220 - 225

[37] Broto, L.; Hagimont, D.; Annoni, E.; Combemale, B.; Bahsoun, J.-P.; , 'Towards a Model Driven Autonomic Management System', Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on 7-9 April 2008 Page(s):63 - 69

[38] Roblee, C.; Berk, V.; Cybenko, G.; , ‘Implementing large-scale autonomic server monitoring using process query systems.’, Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on 13-16 June 2005 Page(s):123 - 133

[39] W. Trumler, J. Petzold, F. Bagci, and T. Ungerer, 'AMUN – An Autonomic Middleware for the Smart Doorplate Project' System Support for Ubiquitous Computing Workshop at the Sixth Annual Conference on Ubiquitous Computing (UbiComp 2004), Nottingham, England, September 7, 2004

[40] Moushumi Sharmin, Shameem Ahmed, and Sheikh I. Ahamed, ' MARKS (Middleware Adaptability for Resource Discovery,Knowledge Usability and Self-healing) for Mobile Devices of Pervasive Computing Environments.', Proceedings of the Third International Conference on Information Technology: New Generations, Pages: 306 - 313, 2006

[41] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, ‘The FRACTAL component model and its support in Java’, Software – practice and experience, Pages: 1257-1284, 2006

Appendix

Java Class for probing a software component in a distributed environment

```
import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.rmi.RemoteException;

public class OneDistributedProbe implements Runnable
{

	private String pid_to_check;
	private String probedEntity;
	private String tubeAddr;
	private String notificationName;
	private String pid_node;

	public void run()
	{
		try
		{
			while(true)
			{
				if(!isAlive())
				{

					try
					{

						new PrintStream(new
FileOutputStream(tubeAddr)).println(notificationName+";this;"+probedEntity)
;
					}
					catch(Exception e)
					{
						e.printStackTrace();
					}
				}

				Thread.sleep(1000);

			}
		}
		catch(Exception e)
		{

		}

	}


	public OneDistributedProbe(String pid_to_check, String tubeAddr,
String probedEntity, String notificationName, String pid_node)
	{
		this.pid_to_check=pid_to_check;
		this.tubeAddr=tubeAddr;
		this.probedEntity=probedEntity;
```

```
            this.notificationName=notificationName;
            this.pid_node=pid_node;

            System.out.println("OneDistributedProbe probing the PID:
"+pid_to_check+" on the node "+pid_node);

            new Thread(this).start();
      }

      public boolean isAlive() throws RemoteException
      {
            String commande="ssh "+this.pid_node+" ps -p "+pid_to_check+" -
o command=";
            String str = "";
            //System.out.println("cmd : "+commande);
            try
            {
                  BufferedReader bf = new BufferedReader(new
InputStreamReader(Runtime.getRuntime().exec(commande).getInputStream()));
                  try
                  {
                        String tmp = bf.readLine();
                        while (!tmp.equals(null))
                        {
                              str=str+tmp;
                              tmp = bf.readLine();
                        }
                  }
                  catch (Exception e)
                  {

                  }

                  if(str.equals(""))
                        return false;
                  else
                        return true;
            }
            catch(Exception e)
            {
                  return false;
            }

      }




}
```

Declaration

I, the undersigned, declare that this thesis is my original work and has not been presented for a degree in any other university, and that all source of materials used for the thesis have been duly acknowledged.

_____

NEBYOU AZANAW

This Thesis has been submitted for examination with my approval as a university advisor.

_____

DEJENE EJIGU (PhD)

Place and date of submission: Addis Ababa, October, 2009