Autonomic resilience of distributed IoT applications in the Fog

Umar Ibn Zaid Ozeer

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Umar OZEER

Thèse dirigée par **M. Gwen SALAÜN**, et
Co-encadrée par **M. Loïc LETONDEUR, M. François-Gaël OTTOGALLI** et **M. Jean-Marc VINCENT**

préparée au sein d'**Orange Labs** et de l'**INRIA**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Gestion Autonome de la Résilience des Applications IoT Distribuées en Bordure de Réseau (Fog)

Autonomic Resilience of Distributed IoT Applications in the Fog

Thèse soutenue publiquement le **11 décembre 2019**,
devant le jury composé de :

Monsieur Thomas LEDOUX
Enseignant-chercheur, IMT Atlantique, Rapporteur
Monsieur Pierre SENS
Professeur, Sorbonne Université, Rapporteur
Madame Sara BOUCHENAK
Professeur, INSA Lyon, Président
Monsieur Charles CONSEL
Professeur, Bordeaux INP, Examinateur
Monsieur Gwen SALAÜN
Professeur, Université Grenoble Alpes, Directeur de thèse
Monsieur Loïc LETONDEUR
Ingénieur de recherche, Orange Labs, Co-encadrant de thèse
Monsieur François Gaël OTTOGALLI
Ingénieur de recherche, Orange Labs, Co-encadrant de thèse
Monsieur Jean-Marc VINCENT
Maître de conférences, Université Grenoble Alpes, Co-encadrant de thèse

Acknowledgements

Here I am, at the beginning of this manuscript and paradoxically at the end of my Ph.D. This journey would not have been possible without the support, guidance, and feedbacks from numerous people around me.

First and foremost, I would like extend my sincere gratitude to my supervisors for their continuous advice and feedbacks, for giving me the freedom to explore areas which align with my research interests, and for trusting my decisions. I am grateful to Gwen Salaün and Xavier Etchevers for (en)trusting me with this Thesis subject. I am thankful to Loïc Letondeur for taking interest in this project, for being present to advise me, and also for his (sometimes nerdy) humour. I am also thankful to François-Gaël Ottogalli and Jean-Marc Vincent for their support and for trusting that I can always aim higher.

I would like to thank Pierre Sens and Thomas Ledoux for accepting to review this Thesis as well as for their valuable insights. I would also like to express my sincere gratitude to the Jury members Sara Bouchenak and Charles Consel for their challenging questions and appreciations during my Thesis defense.

I have to thank all the members of the COPS and Polaris teams for their welcome. I am thankful to my office mate, Franck Roudet, for the long discussions on development and technological aspects. I would like to thank Sabbir Hasan for the geeky discussions and for sharing useful tips during lunch. I have also been blessed to be part of the Convecs team. I am grateful to Radu Mateescu for welcoming me in the team, to Ajay Krishna, Pierre Bouvier, and Philippe Ledent for the thoughtful discussions during lunch and coffee breaks. A special thank to Lina Marsso for being lively, reliable, and always ready to listen and help. It has been a pleasure to be part of such an amazing team and I believe we are family. I sincerely hope that our paths will cross again.

Merci to Antonin Chazalet for encouraging me to do a Ph.D, *Mersi* to my travel buddies Nikhill and Youssouf for accompanying me in my last minute travel plans, *Danke schön* to Samira for being an inspiration of hard work. My appreciation also goes to all the wonderful and amazing people I met in Grenoble that made this journey nicer and more adventurous.

Last but not the least, I want to wholeheartedly thank my parents for their support which made this journey easier. Thank you for teaching me patience, for believing in me, and encouraging me to follow my dreams. I rejoice in knowing that your prayers have been answered.

To my parents

"The oak fought the wind and was broken, the willow bent when it must and survived."

– Robert Jordan

Abstract

Recent computing trends have been advocating for more distributed paradigms, namely Fog computing, which extends the capacities of the Cloud at the edge of the network, that is close to end devices and end users in the physical world. The Fog is a key enabler of the Internet of Things (IoT) applications as it resolves some of the needs that the Cloud fails to provide such as low network latencies, privacy, QoS, and geographical requirements. For this reason, the Fog has become increasingly popular and finds application in many fields such as smart homes and cities, agriculture, healthcare, transportation, etc.

The Fog, however, is unstable because it is constituted of billions of heterogeneous devices in a dynamic ecosystem. IoT devices may regularly fail because of bulk production and cheap design. Moreover, the Fog-IoT ecosystem is cyber-physical and thus devices are subjected to external physical world conditions which increase the occurrence of failures. When failures occur in such an ecosystem, the resulting inconsistencies in the application affect the physical world by inducing hazardous and costly situations.

In this Thesis, we propose an end-to-end autonomic failure management approach for IoT applications deployed in the Fog. The approach manages IoT applications and is composed of four functional steps: (i) state saving, (ii) monitoring, (iii) failure notification, and (iv) recovery. Each step is a collection of similar roles and is implemented, taking into account the specificities of the ecosystem (e.g., heterogeneity, resource limitations). *State saving aims* at saving data concerning the state of the managed application. These include runtime parameters and the data in the volatile memory, as well as messages exchanged and functions executed by the application. *Monitoring* aims at observing and reporting information on the lifecycle of the application. When a failure is detected, *failure notifications* are propagated to the part of the application which is affected by that failure. The propagation of failure notifications aims at limiting the impact of the failure and providing a partial service. In order to *recover* from a failure, the application is reconfigured and the data saved during the state saving step are used to restore a cyber-physical consistent state of the application. Cyber-physical consistency aims at maintaining a consistent behaviour of the application with respect to the physical world, as well as avoiding dangerous and costly circumstances.

The approach was validated using model checking techniques to verify important correctness properties. It was then implemented as a framework called F^{3}ARIoT. This framework was evaluated on a smart home application. The results showed the feasibility of deploying F^{3}ARIoT on real Fog-IoT applications as well as its good performances in regards to end user experience.

Résumé

Les dernières tendances de l'informatique distribuées préconisent le *Fog computing* qui étend les capacités du Cloud en bordure du réseau, à proximité des objets terminaux et des utilisateurs finaux localisé dans le monde physique. Le Fog est un catalyseur clé des applications de l'Internet des Objets (IoT), car il résout certains des besoins que le Cloud ne parvient à satisfaire, tels que les faibles latences, la confidentialité des données sensibles, la qualité de service ainsi que les contraintes géographiques. Pour cette raison, le Fog devient de plus en plus populaire et trouve des cas d'utilisations dans de nombreux domaines tels que la domotique, l'agriculture, la e-santé, les voitures autonomes, etc.

Le Fog, cependant, est instable car il est constitué de milliards d'objets hétérogènes au sein d'un écosystème dynamique. Les objets de l'IoT tombent en pannes régulièrement parce qu'ils sont produits en masse à des couts très bas. De plus, l'écosystème Fog-IoT est cyber-physique et les objets IoT sont donc affectés par les conditions météorologiques du monde physique. Ceci accroît la probabilité et la fréquence des défaillances. Dans un tel écosystème, les défaillances produisent des comportements incohérents qui peuvent provoquer des situations dangereuses et coûteuses dans le monde physique. La gestion de la résilience dans un tel écosystème est donc primordiale.

Cette Thèse propose une approche autonome de gestion de la résilience des applications IoT déployées en environnement Fog. L'approche proposée comprend quatre tâches fonctionnelles: (i) sauvegarde d'état, (ii) surveillance, (iii) notification des défaillances, et (iv) reprise sur panne. Chaque tâche est un regroupement de rôles similaires et est mise en oeuvre en tenant compte les spécificités de l'écosystème (e.g., hétérogénéité, ressources limitées). *La sauvegarde d'état* vise à sauvegarder les informations sur l'état de l'application. Ces informations sont constituées des données d'exécution et de la mémoire volatile, ainsi que des messages échangés et fonctions exécutées par l'application. *La surveillance* vise à observer et à communiquer des informations sur le cycle de vie de l'application. Il est particulièrement utile pour la détection des défaillances. Lors d'une défaillance, des *notifications* sont propagées à la partie de l'application affectée par cette défaillance. La propagation des notifications vise à limiter la portée de l'impact de la défaillance et à fournir un service partiel ou dégradé. Pour établir une *reprise sur panne*, l'application est reconfigurée et les données enregistrées lors de la tâche de sauvegarde d'état sont utilisées afin de restaurer un état cohérent de l'application par rapport au monde physique. Cette réconciliation entre l'état de l'application et celui du monde physique est appelé *cohérence cyber-physique*. La procédure de reprise sur panne en assurant la cohérence cyber-physique évite les impacts dangereux et coûteux de la défaillance sur le monde physique.

L'approche proposée a été validée à l'aide de techniques de vérification par modèle afin de vérifier que certaines propriétés importantes sont satisfaites. Cette approche de résilience a été mise en oeuvre sous la forme d'une boîte à outils, F^3ARIoT, destiné aux développeurs. F^3ARIoT a été évalué sur une application domotique. Les résultats montrent la faisabilité de son utilisation sur des déploiements réels d'applications Fog-IoT, ainsi que des performances satisfaisantes par rapport aux utilisateurs.

Contents

Chapter 1

Introduction

Contents

The idea behind connecting devices to the internet originates in the 1980's where a coke machine at the Carnegie Mellon University was connected to the ARPANET [102] by a group of students. This allowed them to remotely check the machine's status and the availability of sodas before going for a refreshment [13]. It is not until later in 1999 that the term *Internet of Things (IoT)* was coined, which at the time, aimed at promoting the newly developed RFID technology [12]. Since then, with the increased interest in the capabilities of the IoT and progress in technology as well as the evolution of requirements and human needs, the concepts and ideas behind the IoT have evolved.

There is no standard or official definition for the IoT. Some of the various descriptions proposed can be found in [55]. It is sometimes described with respect to the *internet* and the *Internet Protocol (IP)*. For instance, in a special report on the Internet of Things [55], IEEE describes the IoT as "A network of items - each embedded with sensors - which are connected to the Internet". However, the word *internet* in IoT is misleading since it indulges an incorrect perception that devices should necessarily be connected to the internet and that the latter is a requirement in order to deploy IoT applications. The IoT should rather be described as a **network** of devices or *objects* having a cyber-physical presence as they lie at the frontier between the digital and the physical world. The physical world refers to spaces which have the presence and interaction of living things as opposed to

data-centers and software elements which are located in the digital space. The IoT is thus composed of devices which have a presence both in the physical and digital world, having processing and communicating capabilities, and interact with both spaces: sensors report information on the physical world and actuators act on the physical world by automating physical actions. This network of devices may be of varying spread (e.g., a personal, local or wide area network and potentially the internet).

The capabilities offered by the IoT have been defining how we perceive and interact with our surroundings. These unprecedented possibilities have given rise to a plethora of application domains aimed at improving daily consumer life and increasing productivity in industry through the automation of tasks. As a consequence, the IoT has been influencing both business and economic models. For instance, Amazon Go has revolutionise the way we shop by introducing *grab-and-go* supermarkets that do not require any checkouts [128]. Likewise, the IoT has been gaining momentum in other application domains including smart homes and cities [126], agriculture [89], mining [100], healthcare [51], transportation [136] and many more [81]. It is estimated that the IoT will have a total potential economic impact of up to $11.1 trillion a year by 2025 which is around 11% of the total economy of the world [19].

One of the major drawbacks of IoT devices, however, is that they suffer from constrained capabilities. These limitations imply that the functions of these devices cannot be extended (e.g., firmware, APIs), their processing units are not accessible to carry out unintended computations and their storage capacities are limited or not accessible. Therefore, in order to store and analyse data sensed as well as make decisions and push them to actuators, a more capable infrastructure is needed. The Cloud and the Fog have been proposed to play this role by providing the necessary resources for the execution of IoT applications.

This chapter introduces this Thesis and consists of the following sections. Section 1.1 discusses the limitations of the Cloud with respect to the IoT. Section 1.2 presents the concepts behind Fog computing and the specificities of the Fog-IoT ecosystem. Section 1.3 describes the motivations and scope of this work in terms of resilience and failure management. Section 1.4 summarises our approach to handle failures. The contributions of this Thesis and the associated publications are summarised in Section 1.5. Finally, Section 1.6 concludes this chapter by presenting the organisation of this Thesis.

1.1 The Cloud and the Internet of Things

Cloud computing [62, 160] has been prolific for more than a decade now. It advocates for a new paradigm to access on-demand resources (such as storage, computing power, development environments, or web services) located within the core network, accessible via the internet. The Cloud environment is usually considered to have infinite resources [73]. This allows the construction of highly available IoT applications. However, by being located far from end devices in terms of network and geo-location, the Cloud fails to resolve some

of the requirements of IoT applications.

Internet Connectivity. IoT applications that rely on the Cloud are dependent on an internet connection. This is typically the case of vocal assistant devices, such as *Amazon Alexa* and *Google Home Assistant*, which rely on resources provided by the Cloud for voice recognition, natural language processing (NLP), and actuation of other IoT devices in a smart home. These devices therefore need the internet to function properly and are crippled when the internet connection is lost.

Time Sentivity. The Cloud is unsuitable for time-sensitive IoT applications because the delays in communication are not bounded over the internet. The communication latencies between IoT devices and the Cloud are relatively high and can be up to several hundreds of milliseconds [166]. Even a good internet connection may have unpredictable variations. In application domains such as remote surgery or virtual and augmented reality (VAR), a variation of a few milliseconds matters. These types of applications may not afford to access services in the Cloud since the high response time and variations induced by the internet network cannot be tolerated.

Privacy. Cloud providers still do not guarantee strict privacy and security policies [56, 147] even though IoT applications collect privacy-sensitive information. Personal data sent to the Cloud may potentially be, legitimately or illicitly, shared to third parties for political or advertising purposes. One recent example is the *Facebook-Cambridge Analytica data scandal* [84] where millions of users' personal information were harvested from Facekook accounts because of weak enforcement of privacy rights. Moreover, personal data stored in the Cloud can be exposed due to mediocre security policies. In these cases, a breach has a substantial impact since it exposes massive amount of data at once. In June 2019, a team of researchers found a leak that exposed two billion smart home records in Orvibo's IoT management platform. The breach was not patched until several days after the alert [9].

Data Volume. There will be around 26 billion IoT devices by 2020 and this number is set to explode in the coming years [112]. This upward trend is not unexpected since the number of connected objects easily exceeds the world's human population: one person can carry multiple mobile phones (for personal and professional use), wearables (e.g., earphones, VR headsets, smart watch, pacemakers, fitness trackers) in addition to connected devices at home (e.g., connected lamps, door locks, motion / temperature / humidity sensors, cameras). If the large volumes of data generated by these devices are sent to the Cloud without any pre-processing and filtering, the internet core network would be congested.

Location Awareness. Finally, IoT applications have geographical constraints because devices such as sensors and actuators are tied to their geo-location. They have to be deployed in the physical world in order to ensure their services. Unlike other application domains where devices can be outsourced to the Cloud and the deployment of services in the data-centers may be enough, it is impossible to relocate such IoT devices to the Cloud.

In order to mitigate the limitations of the Cloud, we are now witnessing the advent of more distributed paradigms to extend the capabilities of the Cloud for IoT applications.

1.2 The Fog and the Internet of Things

Recent computing trends are promoting more distributed paradigms, namely Fog computing [35, 36, 134, 154] which extends the capacities of the Cloud at the edge of the network, that is close to end devices and end users in the physical world. The Fog offers the possibility of having real-time processing and decision making due to geographical proximity to end devices and subsequent low latencies. Moreover, the Fog relieves the core network by providing the means for storing, analysing, filtering and performing other pre-processing of data at the edge of the network before sending them to the Cloud. Fog devices can be placed within one's private physical environment so that end users can have full control over their data, avoiding privacy exposures by the Cloud. In this way, the geographical requirements of IoT applications can also be met.

The Fog infrastructure is therefore a key enabler of IoT applications. In the next subsections, the structure and specificities of the Fog-IoT ecosystem are detailed.

1.2.1 The Fog Infrastructure

As illustrated in Figure 1.1, the Fog is hierarchically organised [134]. As a general rule, as Fog devices get closer to the physical world, the communication latencies to reach IoT end devices decrease at the price of lower computing and storage capabilities. IoT end devices are composed of sensors and actuators. These are connected to Fog devices which host software elements and services for (i) configuring the devices, (ii) collecting and storing data reported by sensors, (iii) computing data including pattern analysis, filtering and auditing, (iv) decision making, and, (v) controlling actuators.

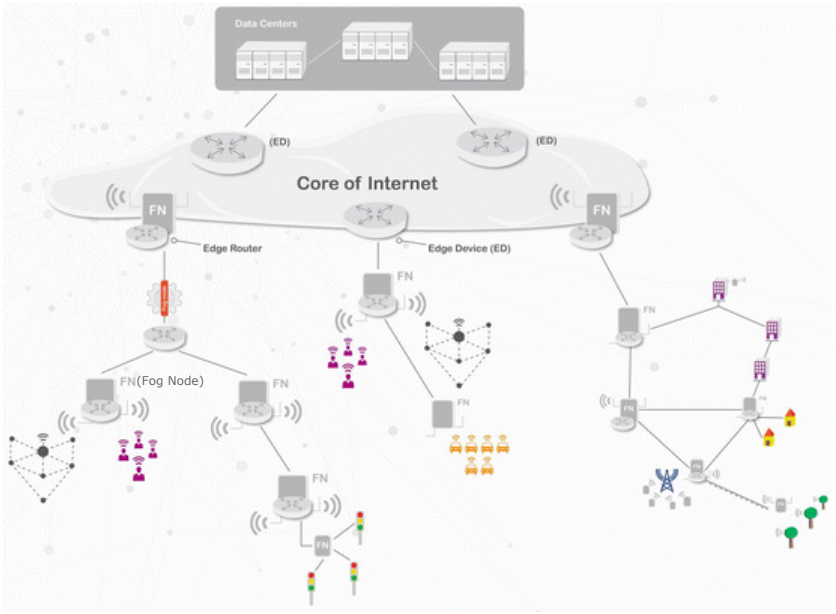


Figure 1.1: The Hierarchical Structure of the Fog [134]

The Fog infrastructure at the edge makes use of devices from the telecommunication and network infrastructures as well as devices deliberately deployed to provide computation and storage resources. For instance, in [99] the authors showed the feasibility of deploying IoT applications on an Orange home internet gateway. The gateway can also be equipped with a 1TB hard disk for storage purposes. In this case, the device enforces data privacy since the users have full ownership of their data and are the only one to have access to them. This service also limits the coverage of potential attacks since the user's data remain in the local network and do not transit to the core network. For more intensive computing, for instance within a corporation or across a neighbourhood, a *RuggedPod* [135] can be used. It is a vicinity micro-datacenter, composed of four 8 cores Xeon CPU, which can be deployed for example near mobile base stations. *Raspberry Pis* [33, 65, 124], a single board computer, have also extensively been used to populate the Fog infrastructure. Raspberry Pis are very attractive because they are readily available at low prices and have good online support and documentation. They can easily run software on a linux operating system and have various modes of connection: cable (USB, GPIO, DPI), wireless (Wi-Fi, Bluetooth and can be extended to support Z-wave and Zigbee). For instance, the University of Malaga has deployed Raspberry Pis to build a smart campus [70]. Other devices such as the *Onion Omega* [5] or *Arduino* [1, 45] have also been used to create a Fog infrastructure.

The Fog presents enormous advantages for the deployment of IoT applications since it addresses the issues that are not resolved by the Cloud. It therefore has business and commercial stakes for industries such as Telcos.

1.2.2 Specificities of the Fog-IoT Ecosystem

The Fog-IoT ecosystem is highly distributed and is composed of billions of devices which exhibit the following challenging specificities: cyber-physical interactions, heterogeneity and constraint diversity, as well as dynamicity.

a . Cyber-Physical Interactions

The IoT is cyber-physical: it is an ecosystem located at the frontier between the digital and physical worlds. Sensors enable the digitalisation of physical world by collecting information about its properties and how they evolve. These data are sent to the Fog for storage and analysis so as to build a dynamic model of the physical world. The actuating capabilities of the IoT rely on this dynamic model created to transform logical decisions into physical actions that we perceive.

The cyber-physical property of the IoT leads to additional challenges since decisions regarding physical actuations may have a direct impact on the physical world. A wrong assumption or an incorrect interpretation of sensed data can have dangerous repercussions. For instance, in a patient monitoring application, the incorrect measurements of the blood sugar level reported by sensors may trigger an insulin injection which can be fatal for the patient. Cyber-physical interactions also introduce non-deterministic events that

may be tied to context such as physical world time, temperature and geo-location. These events result from the unpredictable nature of the physical world and the arbitrary ways in which humans interact with the application.

b . Heterogeneity and Constraint Diversity

The Fog-IoT ecosystem is highly heterogeneous in terms of hardware, software, functions, administrability, network and communication models.

The processing architectures and their capacities are diverse. For instance, a Raspberry Pi has a an ARM architecture whereas most modern desktop computers, laptops and servers have a x86 based hardware. The Onion Omega is based on a MIPS architecture [5]. Moreover, mirco-controllers such as *Arduino*, *ARM Cortex M*, or *ESP32* have constraints also on the programming languages and code instructions. These fundamental differences imply that a software element running on a device may not be compatible to run on another device. In addition, these devices come with different computing powers as well as storage capacities. This means that the design of a software and the choice of the device on which the software will run should take into account the underlying resources of the device. For example, a Raspberry Pi might be able to store data and perform some CPU intensive computations that an Arduino might fail to perform. On the contrary to the Raspberry Pi, the Arduino may be able to control some embedded sensors and actuators.

Devices in the Fog have different degrees of observation and administration. Some proprietary devices have deliberately hidden and obfuscated APIs. The functions of such constrained devices are non-extensible and they cannot be used to perform unintended operations. They usually provide sensing and actuating services and have dedicated computing and storage resources solely for their function. On the other hand, some devices may provide physical resources as well as offer a full range of operations for their lifecycle administration such as hosting and deployment of software elements, reconfiguration, updates, resource management, etc.

The type of network in the Fog-IoT environment varies from wired to wireless. The network connection of IoT devices is generally dominated by wireless protocols which differ in coverage, bandwidth and stability. Some of these protocols are Wi-Fi, Bluetooth, Bluetooth Low Energy (BLE) [148], Zigbee [151], Z-wave [164], Sigfox [111], 6LoWPAN [92], NarrowBand IoT Protocol (NB-IoT) [133], LoRa [156] and more [106]. These numerous protocols are incompatible with each other and the Fog should therefore provide interpretability between these heterogeneous protocols [121]. Regardless of the network protocol, the communication model implemented at the application layer also varies. It can be based on multiple implementations of synchronous or asynchronous message exchange and function calls such as MQTT, websocket and RESTFUL APIs [90] which use different communication paradigms and data formats.

c . Dynamicity

Devices in the Fog-IoT ecosystem may appear and disappear with or without synchronisa-

tion. This may be due to various reasons such as reconfiguration, updates, (unannounced) mobility, instability of wireless network links and failure occurrences.

Since the concepts behind the IoT are fairly new and keep evolving, there is a multitude of projects in IoT. Creating interoperable software and stable architectures are thus hard. The application also evolves with respect to new requirements, human needs and technology changes. This creates a dynamic environment where part of the application may be temporarily brought down for performing updates during which a partial service may be provided.

The dynamic nature of the Fog-IoT ecosystem implies that applications' topology, size and available resources evolve with time. IoT applications are thus designed taking into account these characteristics of the environment allowing support for on-the-fly (re)placement, (re)deployment and reconfiguration so as to adapt to the changing environment.

1.3 Motivations and Scope of this Thesis

Recent research trends have seen many endeavours towards the development of IoT applications in urban environment by industrial, academic and political stakeholders [17]. Smart urban environment is getting more and more attention because it responds to important societal needs and trends [165] including smart home, smart lightning, crime prevention, air quality, noise monitoring, waste management and traffic congestion. Major corporations such as IBM, Google and Cisco have been investing in smart cities in the USA, Europe and Asia [63]. SmartSantander [18] is a European project in which more than 15,000 sensors have been deployed in the city of Santander in Spain in order to make a smart city testbed. Songdo in South Korea is another smart city project in which Cisco invested $47 million [144, 105]. To achieve smart urban environments, a wide range of sensors and actuators are deployed in indoor and outdoor environments. The sensors report data on the physical world such as temperature, humidity, motion, light and air quality. Computing and storage devices retrieve, analyse and store these data and turn them into relevant and actionable insights. Actuators are then used to transform logical decisions into physical actions. For example, in a smart home, the IoT allows the automation of lightning with respect to ambient light intensity and presence in the rooms. It can prevent intrusions in the home by the means of devices like connected door lock, alarms and motion detectors. Similarly, the activities of patients and elderly persons can be monitored inside the home and their tasks can be automated so as to improve their quality of life. The data collected within a home can be shared to have a broader view into the actionable insights which give rise to services like neighbourhood security.

The Fog-IoT ecosystem gives rise to many important applications in smart homes and smart buildings. However, this ecosystem suffers from a low stability and is prone to failures because of bulk production of devices and cheap design. Bugs and non-interoperability of software also make applications unstable. IoT devices are commonly connected by wireless

network links because it brings convenience and flexibility to end users. Such networks are however volatile because the propagation of radio signals are easily obstructed and the reflection of the signals on obstacles (e.g., door, wall, furniture) causes interference [159]. Motion of wireless devices causes fluctuations and attenuations in the level of received signals [161]. Moreover, cyber-physical devices suffer from environmental conditions of the physical world which cause wear-out and increase the occurrence of failures. IoT devices may also suffer from accidental damages from human interactions and even vandalism.

The management of failures of IoT applications in the Fog remains a key challenge because of the specificities of the environment. Heterogeneity implies that one uniform mechanism for dealing with failures is impossible. Since devices are resource-constrained, the failure management protocol should consume a minimum of resources to avoid disrupting the application. The recovery procedure in cyber-physical IoT should take into account the impacts of failures and subsequent repairs on the physical world. Moreover, failure management should be done in an autonomous way because the Fog-IoT ecosystem is increasingly complex and manual intervention is prone to errors or may be impossible. Handling failures autonomously saves time and avoid the cost associated with the intervention of a technician. On the end user side, it gives a smoother user experience as failures of the application is handled in a transparent way.

In classical large-scale internet systems, a significant portion of software failures are cured by rebooting [46]. Even today, in applications that do not handle failures autonomously, end users are inclined to adopt this approach to attempt to cure failures. Such a solution is not feasible in the Fog-IoT ecosystem. The non-containment of failures or merely rebooting the application may result in costly and hazardous situations:

- When a software fails, it loses its volatile memory. By rebooting the software, its state of execution becomes inconsistent with the rest of the application because the software processes events differently and has consequently an unexpected behaviour. This causes disruptions within the application that can induce more failures and potentially cause the failure of the whole application. The resulting service downtime can be very costly for large corporations.
- Since the Fog-IoT ecosystem is cyber-physical, failures may have impacts on the physical world by inducing life-threatening situations. For instance, if the failure of the autopilot software in an autonomous car or the failure of a connected drug injection device of a patient is not repaired, it may be fatal. Even failures which are a priori insignificant, such as the failure of a lamp, may prove to be hazardous in the context of a smart home for elderly and medicated people. Merely rebooting is also not enough to repair such applications. For example, in a crop irrigation application, the reboot of the application may deliver additional pesticides to the crops since the information on the amount of pesticide already delivered may be lost. In this case, additional cost is incurred and the impact on the physical world includes the damage and contamination of the crops. We refer to these types of impact on the physical world as a *cyber-physical inconsistency*.

This Thesis deals with the management of failures of IoT applications in the Fog. Failures in such an ecosystem cause service disruptions as well as induce hazardous and costly situations. Rebooting as a means of curing failures is not feasible as it may cause an inconsistent behaviour of the application. Therefore, in this work, we propose an autonomic failure management approach for IoT applications that deals with cyber-physical inconsistencies. The design of the failure management approach takes into account the specificities of the ecosystem such as cyber-physical interactions, heterogeneity, constraint diversity, and dynamicity. The approach focuses on Smart Home / Building environments. The following section gives a succinct presentation of our approach.

1.4 Failure Management Approach

This research work aims at providing an autonomic end-to-end failure management approach for IoT applications in the Fog.

IoT applications are modelled as a set of applicative entities composed of software elements and appliances which communicate to ensure the functions and services of the application. Software elements are hosted on fog nodes. They give access to the execution resources and provide the runtime environment to run software elements. Appliances are cyber-physical IoT end devices which provide sensing and actuating services. When software elements and appliances interact with each other, their state changes upon processing of events. This change in state can be characterised by their respective behavioural models. The Fog infrastructure is composed of devices and network links that provide the resources for applicative entities to execute their behaviour.

In order to provide failure management, the infrastructure and application should be monitored for failure detection. When an entity fails, it can no longer provide its services. This failure can also propagate to failure-free entities that makes use of (i.e., have functional dependencies on) the service provided by the failed entity. A failure can thus impact the overall service provided by the application. A failure can also have harmful consequences on the physical world because of the cyber-physical nature of the Fog-IoT ecosystem. Failures should therefore be contained in order to limit their impacts on the application and on the physical world. Finally, a recovery procedure has to be implemented so that the application becomes functional again. The recovery procedure should restore the state of execution of the application so that a correct behaviour is re-established. Hence, our proposed failure management approach is composed of four functional steps: (i) state saving, (ii) monitoring, (iii) failure notification, and (iv) recovery. Note that these steps are not successive. Each functional step corresponds to a set of similar roles ensured by the failure management approach.

In the first step, the state of the application is saved in an uncoordinated way. It aims at saving data from which the state of the application can be restored after a failure. State saving policy is proposed in order to cope with the heterogeneous nature of the

ecosystem. A state saving policy defines, for each entity participating in the application, the data that have to be saved (i.e., the state data) as well as the technique, frequency and storage location of these data. The state data include checkpoints and events (e.g., events emitted/received, function calls, and internal computations). The techniques for state saving are based on uncoordinated checkpoint, message logging and function call logging. The frequency of checkpoints depends on the time of execution as well as the number and types of events processed. State data can be stored locally and/or on a stable storage.

In the second step, devices and applicative entities are monitored for failure detection. This is a continuous process. It is achieved through various techniques such as heartbeat, ping-acks, applicative message observation and local system observation. A monitoring technique is chosen so as to induce a minimal disruption in the execution of the application. For instance, sensors that communicate at regular intervals are monitored through applicative message observation to avoid additional overhead on the network.

When a failure is detected, the state saving step of the failed entity stops. The third and fourth steps are subsequently triggered. In the third step, failure notifications are propagated so that the part of the application that is impacted by the failure is notified. Upon the reception of a failure notification, an entity adapts it behaviour (e.g., pause or filter incoming/outgoing events) with respect to the failure. This procedure aims at limiting the impact of the failure and providing a partial service.

The recovery step is the final step which is composed of a reconfiguration of the application and state restoration where the state data saved in the first step are used to restore a consistent state of the application. Reconfiguration depends on the type of entity that has failed. It consists of performing operations on the architecture of the application such as placement, redeployment and restart of software elements. For instance, the failure of a hosting device induces the failure of the hosted software elements. A new hosting device has to be found in order to redeploy the failed software elements so that the architecture of the application can be repaired. If an appliance fails, a functionally equivalent appliance should take over the failed one. If there is no functionally equivalent appliance available, the application has to continue its execution with less features in a degraded mode. When reconfiguration is completed, the state of the repaired entity has to be restored in order to re-establish a consistent behaviour of the application.

State restoration makes use of the data saved during the state saving step as well as the behavioural model of the application. The behavioural model allows the computation of a target consistent state from the state data saved. The application is then set to this target state to ensure consistency within the application and also with respect to the physical world. When the failed entity is recovered, notifications are propagated to the dependent entities (i.e., previously notified entities) so that they can resume their execution and the application can be fully operational again.

1.5 Contributions

This work deals with the management of failures of IoT applications in the Fog taking into account the specificities of the environment as discussed in Section 1.2.2. In order to do so, it makes the following contributions:

- A model of the Fog-IoT ecosystem is proposed. More specifically, the model defines the devices constituting the Fog infrastructure and the applicative entities that run on this infrastructure. The behaviour of the different entities involved in an application as well as how they interact with each other and with the physical world are defined.
- The design of an autonomic end-to-end failure management approach. It is composed of the following four functional steps: (i) state saving where the state of the application is saved, (ii) monitoring for detection of failures, (iii) failure notifications which are propagated to limit the impact of failures, and (iv) recovery which involves the reconfiguration and restoration of a consistent state of the application. The mechanisms to implement these steps are defined based on the proposed model. An architecture for implementing these steps is also proposed.
- State saving policies which describe various combinations of state saving techniques in order to cope with the specificities of the Fog-IoT ecosystem. A state saving policy defines the following parameters for each entity participating in the application: the state data that have to be saved as well as the technique, frequency and storage location of these data. A state saving policy is designed and chosen with respect to properties such as the local constraints, resources available and the communication model implemented.
- A recovery procedure that is consistent within the application and with respect to the physical world (i.e., cyber-physical consistency). The part of the application impacted by the failure is notified by failure notification propagation. The recovery procedure repairs the failed entity and restores the state of the application. State restoration makes use of the behavioural models and recovery policies which define the rules for a safe recovery, that is, a consistent state is restored with respect to the physical world. A degraded mode is also allowed where a partial service is provided if recovery is not possible.
- A formal specification of the recovery procedure of the proposed failure management approach using a modern specification language called LNT. The specification of the approach allows the verification and validation of properties of interest using model checking techniques. It was also useful for the clarification of several questions as well as the identification and correction of some issues in the approach.
- An evaluation of the proposed failure management approach on a smart home testbed. To this end, the failure management approach was implemented as a framework called F^{3}ARIoT (**F**ramework **F**or **A**utonomic **R**esilience of the **F**og and **IoT**, pronounced *far-i-o-t*) and deployed on a smart home testbed composed of IoT devices that exhibits

the specificities of the Fog-IoT ecosystem. A functional and a performance evaluation showed that F^{3}ARIoT works well in practice and is able to repair failures of devices and of the application without introducing a significant additional overhead. Moreover, a consistent state of the application with respect to the physical world is restored in a reasonable user time.

The present research work was carried out in the COPS[1] team of Orange Labs Meylan in collaboration with the CONVECS[2] research team of Inria Grenoble and the POLARIS[3] research team of LIG[4]. It led to the following publications in international conferences:

- Umar Ozeer, Loïc Letondeur, François-Gaël Ottogalli, Gwen Salaün, and Jean-Marc Vincent. Designing and Implementing Resilient IoT Applications in the Fog: A Smart Home Use Case. In Proceedings of the 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), pages 230-232. IEEE, 2019.
- Umar Ozeer, Xavier Etchevers, Loïc Letondeur, François-Gaël Ottogalli, Gwen Salaün, and Jean-Marc Vincent. Resilience of Stateful IoT Applications in a Dynamic Fog Environment. In Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous), pages 332-341. ACM, 2018.

1.6 Thesis Structure

The rest of this Thesis is organised into three successive parts:

(i) Part I is composed of Chapter 2 which presents the basic concepts around failure management in distributed applications. Chapter 3 describes the existing approaches for managing failures in distributed applications, including the IoT domain application. The limitations of the proposed approaches are discussed and the requirements to manage failures in the Fog-IoT ecosystem are defined.

(ii) Part II details our proposed failure management approach for IoT applications deployed in the Fog. It is composed of five chapters.

Chapter 4 defines a model for the Fog-IoT ecosystem. It characterises the Fog infrastructure, IoT applications and their behaviour, the physical world, and describes the failure model considered. It also presents a running example of IoT application in order to illustrate the model as well as the concepts introduced in the following chapters.

Chapter 5 gives an overview of the different steps involved in the proposed failure

[1]Connected Objects Platforms and Services
[2]Construction of Verified Concurrent Systems
[3]Performance analysis and Optimization of LARge Infrastructures and Systems
[4]Laboratoire d'Informatique de Grenoble

management approach as well as its architecture. It presents the failure managers, their roles and how they interact with each other and with the application.

Chapter 6 focuses on the proposition of various techniques for saving the state of the application and how they are chosen to cope with the specificities of the Fog-IoT ecosystem.

Chapter 7 reports how failure detection is achieved and the subsequent failure notification propagation in order to limit the impact of failures.

Chapter 8 describes the recovery procedure. It proposes the rules for repairing and restoring the state of the application. It shows how the data retrieved during the state saving procedure are used to restore a consistent state of the application and how consistency with respect to the physical world (i.e., cyber-physical consistency) is maintained.

(iii) Part III presents the evaluation and validation of the proposed failure management approach. It is composed of the following two chapters.

Chapter 9 proposes a formal specification of the recovery approach and the verification of properties of interest using model checking techniques. It aims at validating the functional properties of the approach.

Chapter 10 presents the implementation of the failure management approach through a developed framework called $\texttt{F}^3\texttt{ARIoT}$ and its deployment on a smart home application. A performance evaluation of $\texttt{F}^3\texttt{ARIoT}$ is given.

Finally, Chapter 11 concludes this work by summarising our main contributions, and discusses future work.

Part I

Failure Management in Distributed Applications

Chapter 2

Failure Management Concepts

Contents

A distributed IoT application in the Fog consists of a collection of applicative entities that collaborate by exchanging messages with each other and with the physical world (PW) in order to ensure the application's functions. Applicative entities can take the form of processes, software elements or services which communicate with each other via communication channels. They also communicate with cyber-physical devices, such as sensors and actuators, which interact with the PW.

Figure 2.1 illustrates an instance of a distributed IoT application composed of three applicative entities (e_1, e_2, and e_3) where the horizontal lines represent the progress of their executions and the arrows show the exchange of messages. The message received from the PW, m_{input_pw}, constitutes an input into the application from the PW whereas the message m_{output_pw} is an output from the application to the PW. In this case, e_1 and e_3 interact with the PW and hence act as an interface between the application and the PW. m_{app_1} and

m_{app_2} are applicative messages. Note that *infrastructure entities*, that is devices which provide execution resources to the application, are not depicted here.

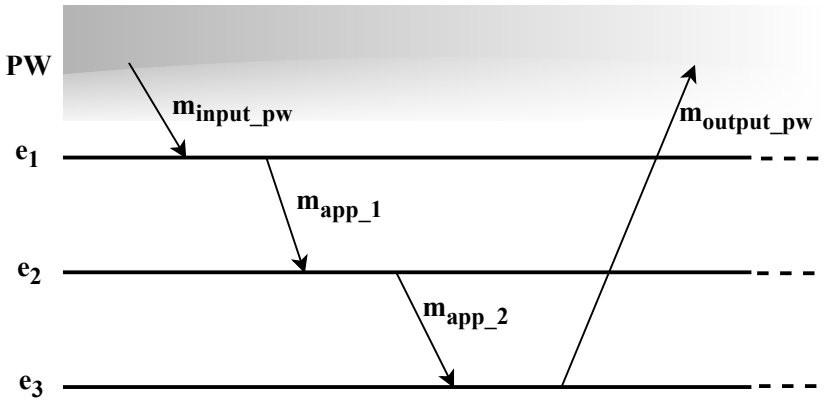


Figure 2.1: An Example of Exchanged Messages in a Distributed Application

This chapter aims at introducing the basic concepts involved in failure management in distributed applications, and more specifically for IoT applications in the Fog. Section 2.1 introduces the terminology used in this Thesis. Section 2.2 describes a general approach to manage failures in distributed applications. Section 2.3 details the characteristics of failures, their consequences, and how they are detected. Section 2.4 summarises the different redundancy techniques for achieving fault tolerance. Section 2.5 reports the reconfiguration operations implemented to recover from failures. Finally, Section 2.6 concludes this chapter.

2.1 Terminology

This section defines the terminology used in the rest of this Thesis. The terminology is taken from [29, 96] on *dependability*.

A *functional specification* defines what the application is intended to achieve. The *behaviour* of the application describes what the application is doing. A *correct service* is provided if the behaviour of the application is in conformity to its functional specification. A *failure* occurs when the service deviates from its specification. An *error* is an incorrect internal state that may potentially lead to a failure. The cause of the error is called a *fault*.

Dependability is defined as the trustworthiness of an application such that reliance can justifiably be placed on the service it delivers (the service corresponds to the application's behaviour) [48]. The management of failures is related to the following dependability attributes. The *reliability* attribute refers to the ability to continuously provide a correct service. The *availability* attribute corresponds to the readiness to provide a correct service. Finally, *safety* refers to the absence of (unintended) catastrophic consequences on the physical world (i.e., on human lives, animals, and plants).

There are four means of achieving dependability:

- Fault prevention: methods for preventing the occurrence or introduction of faults.
- Fault tolerance: methods for providing a service conforming to its specification despite the occurrence of faults.
- Fault removal: methods for reducing the number and impacts of faults.
- Fault forecasting: methods for assessing the presence of faults, their frequency, future occurrence and their consequences.

Resilience is the persistence of dependability when facing changes of different nature. In this Thesis, we focus on resilience in the context of failures and fault tolerance as a means to provide dependability. In other words, we devise mechanisms for the application to continue to provide its services despite the presence of failures. Note that we do not consider or diagnose the origin of failures, but we focus on providing the mechanisms to recover from an identified failure and to re-establish the application's correct behaviour.

2.2 Automating Failure Management

The automation of failure management in the Fog-IoT ecosystem is very important in order to cope with its complexity. The ecosystem is complex because of its heterogeneity, high-scale, and geographical distribution.

Figure 2.2 depicts the general approach to automate the management of failures in distributed applications. It consists of a simple automation loop with three steps. The application is observed so that information about the application during its runtime is reported. The information reported allows decisions to be made on how to repair the application in case of failures. The subsequent repair actions are then performed on the application. This general approach can also be addressed in a finer granularity with the MAPE-K autonomic control loop which is presented in Chapter 5.

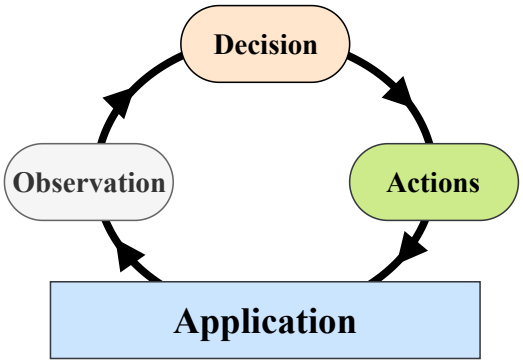


Figure 2.2: Failure Management Automation Loop

In the failure management automation loop depicted in Figure 2.2, the aim of *Observation* is to continuously collect and report information during the runtime of the application. Three types of information are retrieved:

- Failures: The information on failures can be a failure detection or a failure suspicion. A failure detection indicates that a failure has occurred without any uncertainty whereas a failure suspicion indicates that an entity is suspected to have failed. For instance, an entity can be suspected of failure if its response time is higher than usual.

- Recovery: The information on recovery indicates that a previously failed entity has now recovered and is functional.

- Redundant Information: Observation also collects redundant information such as events, messages exchanged, and functions executed which can be used to recover the application and maintain its correct behaviour.

The *Decision* step decides how the data reported during the observation of the application are interpreted and how to react to them. For instance, a decision has to be made when to consider a suspected entity as failed. The decision making step also plans the steps to recover the application based on the entity that has failed and on the type of failure. The *Actions* part of the loop performs these steps which correspond to (i) a *reconfiguration*, that is, a set of operations to be performed on the application, and (ii) the use of the redundant information collected during the observation of the application in order to maintain its correct behaviour. When a recovery is observed, a decision is also needed to determine if any further changes have to be performed on the application (e.g., reconfiguration).

In order to implement such an automation of failure management, the type of failures that affects the infrastructure and the application has to be identified. As illustrated in Figure 2.3, the failure model determines the techniques for failure detection. More specifically, it determines the information from which failures can be inferred or detected during the observation of the application. The definition of the failure model also gives information on the impacts and consequences (e.g., service unavailability, impacts on other parts of the application, loss of data) of the failures. Hence, a redundancy technique to tolerate failures is chosen based on the impacts and consequences of the failures affecting the application. Finally, the recovery procedure implemented by a failure management approach is based on the consequences of the failures and makes use of the chosen redundancy techniques to recover the application.Note that the failure detection mechanisms are independent of the redundancy and recovery techniques implemented.

In the following sections, we first introduce the concepts of failures in distributed applications, and then the different types of redundancy and recovery techniques are described.

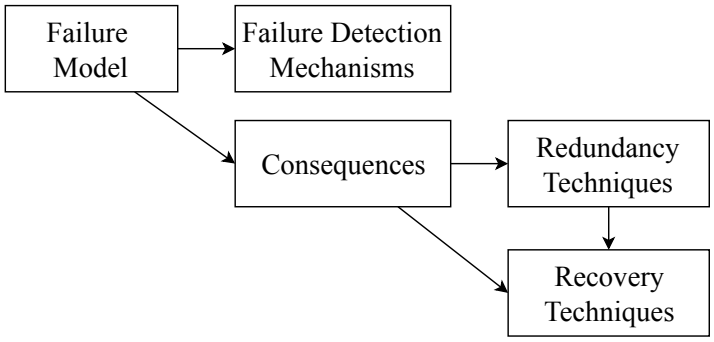


Figure 2.3: Determination of Information for Failure Management

2.3 Failures

The probability of failures in distributed applications increases with respect to the time of execution [140] and the distribution of resources [149]. Thus, large-scale IoT applications have a very low mean time between failures (MTBF) and the probability of failures in such applications is also high because they have long or *infinite* execution time. As discussed in Chapter 1, the additional specificities of the Fog-IoT ecosystem further increase the probability of failures. In order to address the challenges of managing these failures, it is first necessary to identify the types and properties of the failures that affect the application. In the following, we classify failures according to their characteristics. The impacts and consequences of failures are also described.

2.3.1 Failure Models

The failure model describes the characteristics of failures that affect the different entities of the infrastructure and application, as well as their consequences on the application and on the PW. Several classifications of failures can be found in the literature.

Failures can be classified according to their severity [29]:

- Omission failures are used to model communication failures where message exchanges can be lost.
- Timing failures occur when a response lies outside the specified time interval.
- Crash failures occur when an application, which was behaving correctly, halts its activities and does not execute any further operations. It does not send nor receive any messages.
- Arbitrary or Byzantine failures are characterised by the absence of any assumption on how the application can behave. It includes malicious behaviours such as falsifying messages.

Failures can also be classified according to their duration [29]:

- Transient failures are temporary and do not persist (finite duration). They occur once and then disappear.
- Intermittent failures appear and disappear on their own accord. They are recovered without any external repair actions.
- Permanent failures have an infinite duration (until they are explicitly repaired).

In the approach of this Thesis, we are interested in the crash failure model. This type of failure is permanent. It affects both applicative and infrastructure entities that compose a distributed application. This failure model is motivated by real case of failures that can be observed in the Fog-IoT ecosystem. Crash failures occur when there is a power failure due to a temporary outage, accidental unplugging, or battery drain. IoT devices are usually deployed in the PW. This implies that they are subjected to environmental conditions that causes wear out as well as vandalism. Software elements can crash due to a lack of resources for their execution, unhandled exceptions, design/development errors or poor memory management. Therefore, in the rest of this Thesis, we limit our discussion to crash failures.

2.3.2 Consequences of Failures

When an entity of a distributed application fails, there are several consequences within the application as well as on the PW.

(i) The failure of an infrastructure entity (e.g., Raspberry Pi or ISP gateway) is transitive: it causes the failure of the applicative entities hosted on the device.

(ii) The failure of an entity causes the unavailability of the services and resources provided by the affected entity.

(iii) A failed applicative entity loses its volatile memory and thus all the data stored in this memory since the beginning of its execution are lost. These data characterise the state of execution of the entity and determine its behaviour.

(iv) A failure can propagate to other failure-free entities of the application. As Leslie Lamport puts it, "*a distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*" [94]. The propagation of failures can occur because of dependencies between the entities of a distributed application where the correct behaviour of an entity depends on the correct behaviour of another one.

(v) A failure can have an impact on the PW: it can make the PW unsafe for living things including humans, animals and plants. For instance, the failure of a smoke detector in a smart home can make the PW unsafe for the house tenants. The failure of devices which deliver water to crops in a smart irrigation application can affect the crops.

The consequences mentioned above should be taken into account in the management of failures. Both applicative and infrastructure failures should be handled. When a failure is detected, the failure management approach should assess and limit the impact of the failure within the application and on the PW. The application should be repaired to limit the service or resource unavailability, as well as to limit the data lost. In the approach of this Thesis, we aim at recovering from failures of the infrastructure and application so as to limit service unavailability. Information about the application during its runtime (e.g., messages exchanged, the data in the volatile memory of the application) are saved. They are used to limit data lost and to implement a safe recovery.

2.3.3 Detection of Failures

The detection of failures in distributed applications is fundamental because fault tolerance techniques rely on it to react to failures. The primary role of the failure detector is to monitor the application by collecting information from which the occurrence of a failure can be deduced. The mechanisms implemented by the failure detector are usually determined by the type of failures to detect but are independent of the fault tolerance techniques.

Failure detection is hard because of the asynchronous nature of distributed applications. There are no bounds on execution speed nor communication delays. As a consequence, it is impossible to determine precisely if a remote entity has failed or is merely slow or busy. In this case, the failure detector can only suspect a failure and the suspicion may be erroneous. A failure detector can thus be unreliable in the sense that it can make mistakes [53].

The main approach to detect crash failures in distributed applications is based on timeouts. A timeout bounds the response time of an entity. Upon expiration of the timeout, the failure of the entity can be inferred. This gives rise to two failure detection techniques based on a push approach such as *heartbeat* [58] and a pull approach such as *ping-ack* [125].

a . Heartbeat

As illustrated in Figure 2.4, an entity e sends a heartbeat message m_{hb} at a regular interval t_r to a failure detector fd. Each reception of m_{hb} aims at indicating to the failure detector that e is failure-free. The failure detector configures a timeout t_o based on the time interval t_r and the network latency between e and fd. The timeout t_o is reset at each reception of the heartbeat message m_{hb} before its expiration. That is, if $t_{hb} < t_o$ where t_{hb} is the elapsed time between the reset of the timeout and the reception of m_{hb}, then fd deduces that e is alive. If the timeout expires before the reception of m_{hb} (i.e., $t_{hb} = t_o$), then fd suspects the failure of e.

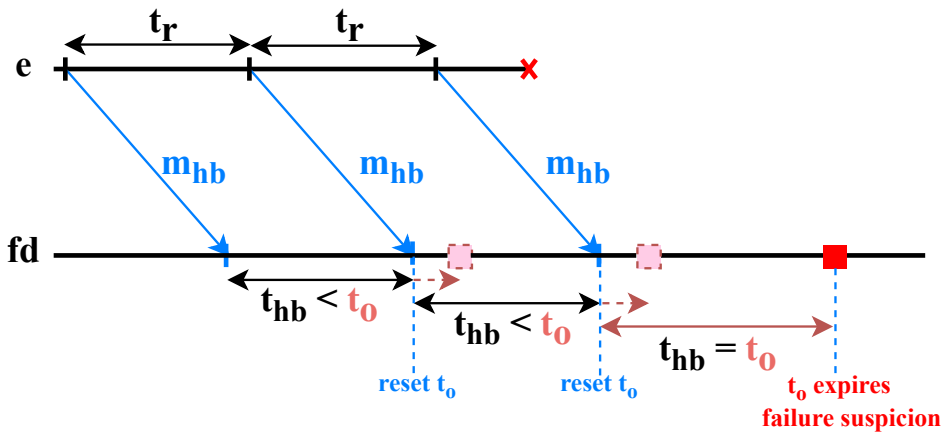


Figure 2.4: Heartbeat Failure Detection

b . Ping-ack

In this approach, as illustrated in Figure 2.5, the failure detector fd requests the status of an entity e by sending a message m_p and starts a timeout t_o. The failure detector waits for an acknowledgement message to infer whether e is failure-free. If an acknowledgement message m_a is received by fd before the expiration of t_o (i.e., $t_{pa} < t_o$ where t_{pa} is the elapsed time between sending m_p and receiving m_a), then the timeout is stopped. A new message m_p is sent after t_r and the timeout t_o is started again. On the other hand, if m_a is not received before the expiration of t_o, then the failure detector suspects the failure of e.

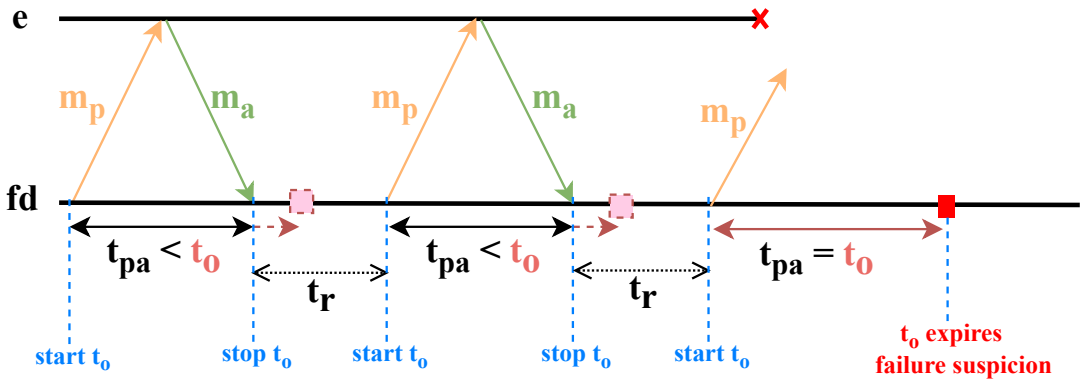


Figure 2.5: Ping-ack Failure Detection

The heartbeat approach only needs half of the messages required by the ping-ack. It is also harder to configure the timeout in ping-ack since the two messages can be subjected to network delays. The advantage of the ping-ack, however, is that both timeouts t_o and t_r are configured by the failure detector. Thus, the observed entity acknowledges the reception of the ping messages and does not have to configure any timeouts.

The configuration of timeouts for failure detection is hard because it determines how much resources are consumed by the failure detector, the level of interference with the execution of the application, and the reliability of the information that it provides. The desired properties of a failure detector are (i) *minimal interference* with the execution of the application in terms of overhead and resource consumption, (ii) *high reliability* such that the failure detector does not make mistakes frequently, and (iii) *high reactivity* such that a failure detector reports a failure as soon as it happens. However, these properties cannot all be achieved simultaneously since they contradict each other. The timeout of the failure detector should be configured with a compromise between these properties. Let us consider a failure detector based on the heartbeat mechanism. In order to ensure reactivity, a short period should be configured for the heartbeat. This means more messages are exchanged on the network and thus more bandwidth and processing power are consumed by the failure detector violating the minimal interference property. Moreover, since the timeout is short, the failure detector is more likely to wrongly report a failure violating the high reliability property. In this Thesis, we choose to provide failure detection with a minimal interference because of the limited resources and capabilities of the Fog-IoT ecosystem. Failure detection is however not part of our core contributions. For more information on this subject, we refer the reader to the seminal work of Chandra and Toueg [53] as well as the work of A. Schiper [138], P. Sens [107, 34], M. Hurfin and M. Raynal [82], and R. Guerraoui and P. Kuznetsov [71].

2.4 Fault Tolerance based on Redundancy

The previous section defined the characteristics of failures, their consequences as well as the need for their detection and how it is achieved. This section focuses on redundancy techniques as a means of achieving fault tolerance. Fault tolerance techniques rely on redundancy which consists in introducing additional resources, aside from those required by the application to behave correctly. In case of a failure, the redundant resources are used to maintain or re-establish the correct behaviour of the application.

2.4.1 Types of Redundancy

Redundancy techniques can take the following forms [87].

(i) Hardware redundancy: This consists in adding extra hardware to tolerate failures. This type of redundancy can be found in different types of systems, from biological to engineering systems. For instance most mammals have two eyes, lungs and kidneys even if they can function with only one of each. Aircraft engineering also uses hardware redundancy: a 4-engine aircraft can fly with only three engines and may have redundant power generators.

(ii) Information redundancy: This consists in adding redundant information to data in order to provide fault tolerance. For instance, extra information can be added when

transmitting data to allow recovery from corrupted data.

(iii) Time redundancy: This type of redundancy involves the repetition of operations. For instance, multiple copies of a message can be sent to make sure that it is delivered. Operations can also be repeated after a failure to restore the volatile memory of a software element. Time redundancy aims at reducing the amount of additional hardware at the expense of using extra time.

(iv) Software redundancy: This can take the form of multiple instances of the same software elements that participate in the application. It can also be programs and data which are not part of the business core of the application such as additional lines of codes to check for errors.

These forms of redundancy are rarely employed separately. Rather, failure management involves the implementation of multiple types of redundancy in order to tolerate failures. The resulting mechanisms implemented from these forms of redundancy give rise to several forms of fault tolerance concepts which are presented in the rest of this section.

2.4.2 Replication

Replication uses hardware, software and time redundancy. It consists in creating multiple replicas of applicative and infrastructure entities to tolerate failures. For instance, in order to tolerate n failures, $n+1$ replicas are introduced. This guarantees that at least one replica is always available. Replication is primarily used to ensure high availability, so that a failure does not cause a service interruption. In order to keep the multiple instances of the same entity consistent, the different replicas should be synchronised. Replication can be classified into three categories with respect to the level of synchronisation between the replicas.

- Active replication: All the replicas have the same deterministic behaviour and execute the same operations. They receive and process the same events in the same order.

- Passive replication: A primary replica executes all the operations and sends update to the backup replicas. When the primary replica fails, one of the backup becomes the primary.

- Semi-active replication: This is a hybrid approach. It extends active replication with the notion of *leader* and *follower* such that non-deterministic decisions are performed by the leader and then sent to the followers.

Due to the high cost of hardware and software redundancy, other approaches have focused on information and time redundancy as described in the next subsection.

2.4.3 Maintaining a Consistent Behaviour of the Application

The loss of the volatile memory upon the failure of an entity implies the loss of its state of execution. The state of execution is characterised by the operations the entity has executed during its runtime. These operations correspond to the emitted/received events/messages as well as functions executed. If an entity is recovered without taking into account its pre-failure state of execution, its behaviour (i.e., the way it will process forthcoming events) upon recovery will be different from its expected behaviour. Thus, during the runtime of an application, the observation process should collect information on the state of execution of the application. This state has to be restored as part of the recovery procedure in order to maintain the correct behaviour of the application after recovery.

State restoration is based on information and time redundancy. It consists in saving the state of the application regularly on a stable storage and whenever there is a failure, the data saved are retrieved and used to restore the state of the application. The main difficulties of this approach is to determine the data to save and which technique to adopt so that a globally consistent state of the distributed application can be restored.

a . Stable Storage

The stable storage ensures that the stored data persists despite the occurrence of the tolerated failures [66]. It is not affected by the failures of the application and thus the data needed for recovery can always be retrieved. The implementation of the stable storage can take various forms according to the assumptions that can be made and the type of failures to tolerate. For instance, to be able to tolerate a single software element failure, the stable storage can correspond to the volatile memory of another software element [37]. In order to tolerate only software element failures, the stable storage can correspond to the persistent local storage of physical devices. Finally, to tolerate multiple failures occurring on both applicative and infrastructure entities, the stable storage can be implemented on a dedicated storage medium external to the infrastructure on which the application runs. In this case, the reliability of the storage medium can be ensured through techniques such as RAID (Redundant Array of Inexpensive Disks) file systems [57].

b . Consistent Global State

The global state of a distributed application is composed of the set of local states of all the applicative entities participating in the application as well as the states of the communication channels through which the applicative entities exchange messages.

Intuitively, a global state of a distributed application is said to be consistent if, for a failure-free execution, for any message m whose receive event is recorded in the global state then the corresponding send event is also recorded in the global state [54]. Figure 2.6 gives an example of a consistent and an inconsistent global state [66]. The distributed application is composed of three software elements (or processes). The horizontal lines represent their execution with time and the arrows illustrate the messages exchanged between them. The hashed diamond represents the state of the software element at a particular instant in its

execution. Figure 2.6(a) shows a global consistent state since the message m_1 which is sent but not yet received represents a message travelling across the network. In Figure 2.6(b), the message m_2 is received by P_2 whereas it is not recorded in the local state of P_1. This situation cannot happen in a correct behaviour of the application. Therefore, the global state composed of the local states of the three software elements is inconsistent. It is worth noting that the collection of the initial local states of the distributed application forms a global consistent state and so does the collection of the current states.

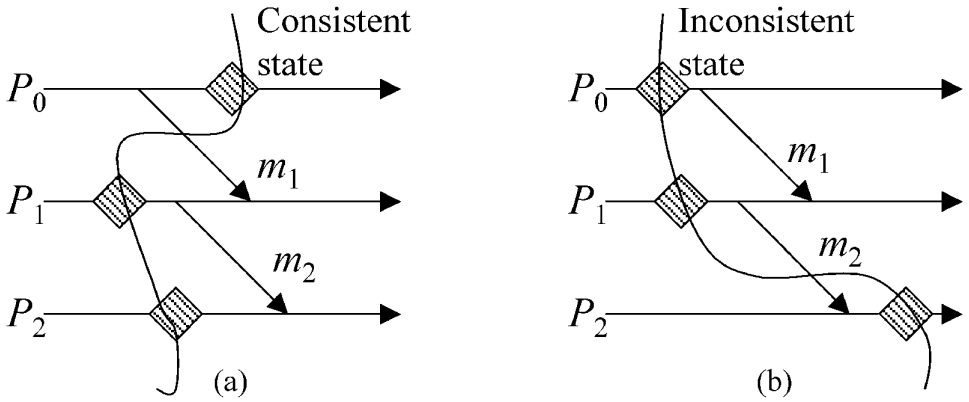


Figure 2.6: An Example of a Consistent and Inconsistent Global State [66]

2.5 Reconfiguration

As discussed in Section 1.2.2, the Fog-IoT ecosystem is dynamic due to multiple reasons such as failure occurrences, mobility, instability of wireless networks and evolution in the architecture of applications. As a result, distributed applications intended to be deployed in the Fog are designed so that entities are loosely coupled. This allows part of the distributed application to be modified while the unaffected part maintains its services. This process of modifying the structure or *configuration* of the application during runtime is called *dynamic reconfiguration*. It prevents a total shut down in order to make changes to the application.

A dynamic reconfiguration of a distributed application may have different objectives such as incremental updates for extending the services of the application, for security or maintenance purposes, for optimisation and performance objectives, or for reacting and adapting the behaviour application when failures occur. Note that the computation of the new configuration may be done with various optimisation criteria which is out of the scope of this Thesis. We refer the reader to the following works in this domain [120, 152, 153, 157]. In the following, we are interested in methods for determining a new configuration of the distributed application when a failure occurs.

2.5.1 Types of Reconfiguration

A configuration models a distributed application in terms of a set of software elements [1] and bindings between them which are mapped onto physical hosts [78]. The software elements are composed of data and program units whereas the bindings connect the interfaces of the software elements via which they communicate. Bindings can be incoming, outgoing or bi-directional. Dynamic reconfiguration concerns the modification of this configuration at runtime and may take various forms including the addition, removal, replacement, and modification of the behaviour and interactions of entities participating in the distributed application [39, 79, 116].

We identify four types of possible reconfiguration in the context of failures: *architecture*, *placement*, *functional*, and *implementation*.

a . Architectural Reconfiguration

The architecture of the distributed application describes its logical structure or topology. An architectural reconfiguration involves a change in this structure such as adding or removing software elements as well as modifying the bindings between them. For instance in Figure 2.7, the change in architecture is a modification in the bindings between the software elements se_1 and se_2 because se_2 has failed. In this case, se_1 is connected to se_3 and thus the events from se_1 to se_2 are redirected towards se_3.

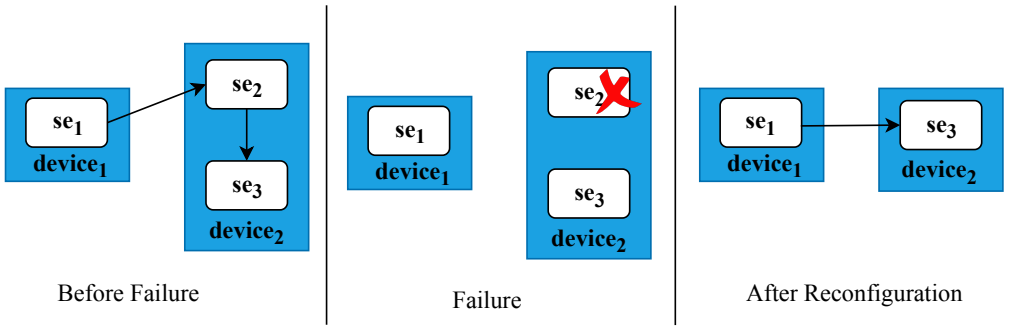


Figure 2.7: Architectural Reconfiguration

b . Placement Reconfiguration

A placement describes how software elements and bindings are mapped onto physical devices. A placement reconfiguration changes only this mapping, that is, it relocates a software element without changing the architectural configuration of the application. An example of placement reconfiguration is given in Figure 2.8 where the software element se_3 originally placed on the $device_3$ is moved onto the $device_2$ because of the failure of $device_3$. In this case, the configuration of the application has to be changed because the failure of

[1] The term software *components* or *modules* are usually used in this context. We keep the term *software element* in order to be consistent with the rest of this document.

the device also implies the failure of se_3 and the latter cannot be placed again on the same device. However, the bindings between the entities have not changed.

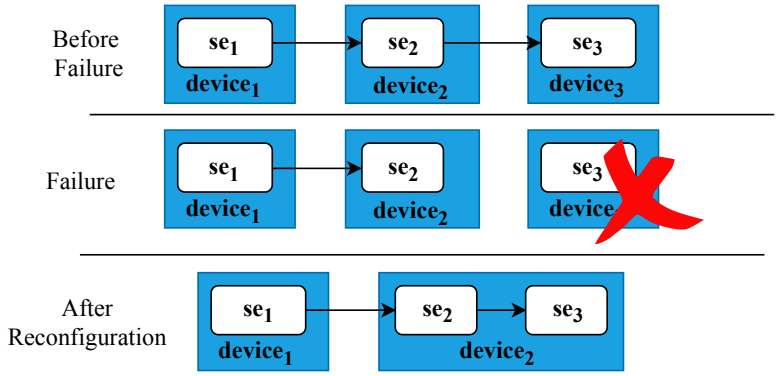


Figure 2.8: Placement Reconfiguration

c . Functional Reconfiguration

Functional reconfiguration (or interface reconfiguration) consists in modifying the access to the functions or services provided by an entity. It consists in restricting the access to part or all of the interfaces an entity exposes. For instance, in Figure 2.9, the software element se_3 originally exposes two interfaces *ItfA* and *ItfB*. The software elements se_1 and se_2 are connected to se_3. se_1 uses the services provided through *ItfA* whereas se_2 uses the services provided through *ItfA* and *ItfB*. se_3 is connected to se_4 and the service provided through $ItfB$ depends on the correct behaviour of se_4. In this case, when se_4 fails, se_3 restricts the access to the interface *ItfB*. This type of behaviour is referred to as a degraded mode.

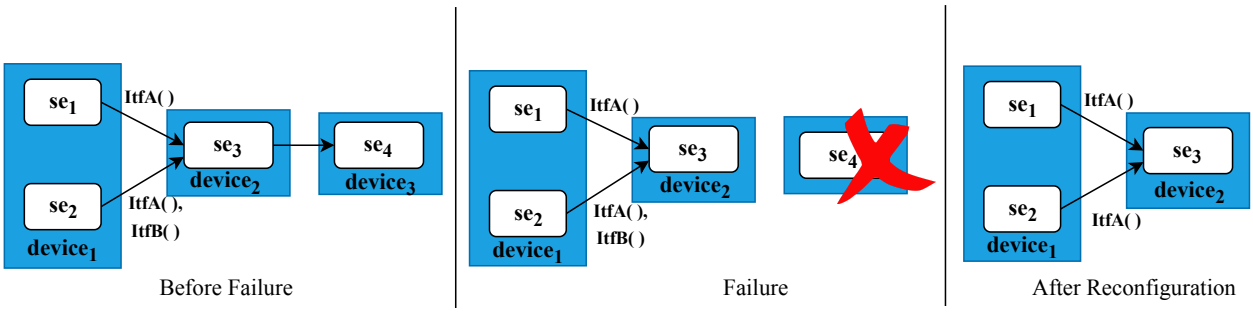


Figure 2.9: Functional Reconfiguration

d . Implementation Reconfiguration

In implementation reconfiguration an entity's internal code instructions are modified without changing the architecture and placement of the application. This may be done in order to correct a bug that provokes failures. For instance, in industrial applications, hot patches are applied during the runtime of the application without downtime.

2.5.2 Challenges of Implementing Dynamic Reconfiguration

The implementation of dynamic reconfiguration is hard because it poses the following challenges [98, 123]:

- The reconfiguration should keep the architecture and structure consistent with the intended behaviour of the application. This means that the reconfiguration should not invalidate the architectural invariants of the application such as absence of cycles or disconnected entities.
- The implementation of dynamic reconfiguration should take into consideration the execution of other entities which are not under reconfiguration. For instance, since the services provided by a software element are unavailable during its reconfiguration other entities should not request its services. This can be done by temporarily stopping the impacted entities or replying with a notification of the unavailability of the requested services.
- Upon reconfiguration, if the entity is re-initialised, its state of execution has to be restored so that its state and behaviour is consistent with the rest of the application.
- Dynamic reconfiguration may involve the combination of different types of reconfigurations as well as different entities. This should be performed in a coordinated way to avoid inconsistent or incorrect execution of the reconfiguration procedure [129]. In this case, architectural changes should be propagated so that the rest of the application has an updated view of the architecture of the application.

Several approaches to implement dynamic reconfiguration have been proposed to address these challenges. They are presented in the next chapter.

2.6 Summary

This section introduced the basic concepts involved in failure management of distributed applications. The automation of failure management is achieved by continuously observing the application during its runtime. The information reported during observation allows a decision making process to determine the operations to perform on the application for recovery. The information collected during observation is also used to maintain a consistent behaviour of the application after recovery. The next chapter presents the fault tolerance approaches based on the concepts presented in this chapter.

Chapter 3

Existing Fault Tolerance Approaches

Contents

The concepts presented in the previous chapter lead to several failure management approaches with respect to the application domain considered and its related constraints. This chapter presents and compares the solutions that are closest to our approach. In order to do so, Section 3.1 of this chapter discusses the desirable properties and evaluation criteria for a fault tolerance approach in the Fog-IoT ecosystem. Then, the following sections presents the existing solutions based on replication (Section 3.2), state restoration (Section 3.3), and dynamic reconfiguration (Section 3.4). Section 3.5 details the existing fault tolerance solutions in the Fog-IoT ecosystem. We discuss and assess each of these solutions with respect to the evaluation criteria. Section 3.6 summarises the related work.

3.1 Comparison and Evaluation Criteria

In order to compare the existing fault tolerance solutions in distributed applications, we propose four global evaluation criteria which are chosen with respect to the properties of the Fog-IoT ecosystem as discussed in Section 1.2.2: *cyber-physical consistency*, *disruptions*, *heterogeneity*, and *cost*.

- Cyber-Physical Consistency: Since the Fog-IoT ecosystem is cyber-physical, a fault tolerance approach for this ecosystem should recover the application in a consistent state with respect to the PW. This means that the approach should take into account the changes in the state of the PW upon implementing a recovery procedure. For instance, when a thermostat in a smart home is recovered, the changes in temperature of the rooms should be taken into account before restoring the state of the device. The consequences of a failure on the PW (see Section 2.3.2) should also be assessed so that a recovery procedure which is safe for the PW is implemented and costly situations in the PW are avoided. For instance, in a patient's drug injection device, a failure may be fatal for the patient. The recovery procedure implemented can have the same consequence (e.g., more drug doses may injected because of operations performed on the device during the recovery procedure) if the procedure does not restore a state of the device that is consistent with its pre-failure state and with respect to the PW.

- Disruptions: A fault tolerance approach should limit the disruptions caused by the approach itself during the observation of the application and during the actions implemented for recovery (see Figure 2.2). Observation should limit the interference with the execution of the application. For instance, a synchronisation of all the entities participating in the application in order to compute a globally consistent state interferes with the nominal behaviour of the whole application. Also, the application may have to be paused to compute the global checkpoint. The approach should also limit the disruptions caused by a failure. In order to do so, the recovery procedure implemented should not impact the failure-free part of the application. For instance, rebooting all the software elements or performing a global rollback of an application because of a single failure disrupts the whole application.

- Heterogeneity: The Fog-IoT ecosystem is highly heterogeneous. Thus, relying on a uniform technique for fault tolerance is not feasible because of the diverse properties and constraints (e.g., functional, geographical) of the different entities of an application. A fault tolerance approach should therefore cope with this heterogeneity by integrating multiple techniques that are suited to the properties of the infrastructure and applicative entities participating in the application. For instance, one way of recovering from software element failures is to perform a placement reconfiguration. However, since the software element deployed on a smart refrigerator is tied to the device, a placement reconfiguration may not be possible. Devices may also expose different types of APIs for their monitoring (e.g., heartbeat/ping-acks). The mecha-

nisms provided for fault tolerance should therefore be adapted to the constraints and properties of the entities.

- Cost: The Fog-IoT ecosystem is limited in terms of processing, storage, and communication capabilities. Therefore, a fault tolerance approach should consume a minimum of resources and introduce a minimum overhead. This is because these resources should rather be dedicated to the execution of the application. For example implementing a ping-ack mechanism at a very high frequency for failure detection may consume too much bandwidth. The cost of the approach can also be computed based on the additional resources that are added solely for tolerating failures. For instance, this is the case of approaches based on replication. If an application is replicated once, then half of the resources consumed are exclusively dedicated for fault tolerance.

These properties are important in order to design a fault tolerance solution that is suited for the Fog-IoT ecosystem. In the following, we discuss the existing fault tolerance approaches with respect to these criteria.

3.2 Fault Tolerance based on Replication

The Triple Modular Redundancy (TMR) and voting technique [103] is a well known active replication technique where computations are performed three times on three instances of an entity (for example on three software elements hosted on separate devices) and a voting mechanism is called if their outputs differ. This type of fault tolerant technique is usually adopted in critical and embedded applications. Replication is also commonly used in the aviation industry.

Fault tolerance in Delta-4 [60] is achieved by replication of software components on distinct host computers interconnected by a local area network. The activities of the replicas are coordinated in order to process errors and give the illusion of a single failure-free software component. Delta-4 implements active, semi-active and passive replication. In the active replication strategy, all replicas process concurrently all input messages so that their states are closely synchronised. In the absence of failures, the replicas produce the same output messages in the same order, and outputs can be taken from any replica. Delta-4 uses a special protocol called inter replica protocol for detecting failures of replicas and host processors. This technique can achieve a quasi-instantaneous recovery if all the correct replicas produce the same output messages in the same order. In the semi-active strategy, the leader replica processes all the input messages and provides output messages. In the absence of failures, the other replicas do not produce output messages. Their states are updated either by direct processing of messages of by means of notifications from the leader replica. In the passive strategy, a primary replica processes the input messages and provides output messages. In the absence of failures, the states of the other replicas are regularly updated by means of checkpoints from the primary replica.

The cost induced by replication is very high since to tolerate k failures, $k/(k+1)$ of the available resources are used exclusively to ensure fault tolerance. These resources are wasted during failure-free executions since they could be used for other purposes such as increasing performances.

In the Fog-IoT ecosystem, the main drawbacks of fault tolerance techniques based on replication are the following.

- Cyber-Physical Consistency: Replication of some IoT devices may violate cyber-physical consistency. For instance, active replication of actuators is not always feasible because the actions performed by the device on the PW would be performed multiple times.

- Disruptions: Fault tolerance techniques based on replication cause a minimum disruption in the application when a failure occurs. This is because the recovery procedure is simple as the application can switch to a failure-free replica when one replica fails. Such a recovery procedure does not affect the failure-free entities of the application.

- Heterogeneity and Diversity of Constraints: Replication may not be possible because of the space that devices occupy in the PW. For instance, a patient cannot carry multiple pacemakers. Another amusing but illustrative example is the following. Having two connected freezers in a smart home to tolerate failures of one of them is not realistic, not only because of the room it occupies but also because the goods inside one replica are tied to it and cannot be automatically transferred to the another replica when failures occur. Some devices and software elements cannot be replicated because their behaviour does not allow it. For instance, some devices can only connect to one gateway (e.g., due to the use of certificates for authentication) and thus the gateway cannot be replicated.

- Cost: The cost associated with replication can be excessive since at least half of the available resources would be dedicated to fault tolerance. Additional overhead is also introduced to synchronise the replicas.

Replication has the advantage of minimising the disruptions in the application when a failure occurs. However, in most cases, replication is not a suitable means for tolerating failures in the Fog-IoT ecosystem. Replication does not guarantee cyber-physical consistency. Replication has a very high cost and the limited resources in terms of processing, storage, and communication makes it an unsuitable fault tolerance technique. Replication may also be unsuitable because of the heterogeneous constraints of the ecosystem. Our proposed approach therefore avoids hardware and software redundancy because of these reasons. In order to reduce the cost, cope with heterogeneity, and overcome the cyber-physical inconsistencies, we focus more on techniques based on information and time redundancy. This is discussed in the next section.

3.3 Fault Tolerance based on State Restoration

There are two main approaches for saving information from which the state of an application can be restored: *checkpoint*, and *event log*. They are detailed in the following sections.

3.3.1 Checkpoint

Checkpoint techniques consist in saving regularly, for each entity participating in the distributed application, a set of data that represents its execution state. A checkpoint of an entity can be composed of data in its volatile memory, runtime variables, input parameters, and configuration files. Following a failure, a set of stored checkpoints is used to restore the most recent global consistent state of the application.

Checkpoint strategies can be classified into 3 categories with respect to how the global consistent state is created: *coordinated checkpoint*, *uncoordinated checkpoint* and *communication induced checkpoint*.

a . Coordinated Checkpoint

Coordinated checkpoint, also called global or synchronised checkpoint, as it name suggests, is done by synchronising all the entities participating in the distributed application so that the set of states saved is globally consistent. The first algorithm to record a global consistent state was devised by Chandy and Lamport [54]. The algorithm uses a control message called a marker. Assuming that the messages on the communication channels are delivered using a FIFO (First In First Out) strategy, the marker is able to separate the messages that constitute the state of the communication channel and those which should be recorded in the local checkpoint of an entity. Other algorithms of coordinated checkpoint have been proposed by T. Lai [93], F. Mattern [110, 109] and C. J. Fidge [69].

Cost. In the Fog-IoT ecosystem, the cost of synchronisation to implement coordinated checkpoint becomes too large because all the entities participating in the application have to synchronise to save the state of the application. Coordinated checkpoint thus has a high cost in order to guarantee a globally consistent checkpoint. **Disruptions.** The whole distributed application is impacted by a single failure since potentially all entities have to rollback to a previous state to ensure global consistency. Thus, the whole application is disrupted for a single failure. Also, if the mean time between failures (MTBF) is lower than the mean time to repair (MTBR), the application cannot recover. Such an outcome is predictable in the Fog-IoT ecosystem because of the high scale of applications and high frequency of failures. This suggests that a state saving scheme in which only the failed entities are impacted by the repair and state restoration procedures should be adopted. **Heterogeneity.** Coordinated checkpoint, similarly to the other checkpoint techniques, does not support highly heterogeneous IoT applications since some devices (e.g., sensors and actuators) are functionally constrained and cannot be expected to perform a checkpoint. **CP-Consistency.** Coordinated checkpoint does not capture the events emitted

and received after a checkpoint is performed. These can include events coming from the PW. Thus, the recovery implemented in this case is not consistent with respect to the PW since the changes in the PW is not taken into account when restoring the state of the application.

In order to overcome the synchronisation cost and global disruptions upon recovery induced by coordinated checkpoint, our approach integrates uncoordinated checkpoint.

b . Uncoordinated Checkpoint

Uncoordinated checkpoint [66, 162], on the other hand, removes the synchronisation overhead by letting applicative entities save their state individually.

Cost. One of the main advantages of uncoordinated checkpoint is that each applicative entity can save its state at will, for instance when the size of the checkpoint is minimal and thus reducing the synchronisation and storage overhead [158]. Uncoordinated checkpoint thus has a lower cost than coordinated checkpoint. **Disruptions.** However, the elimination of the synchronisation comes at the expense of global consistency. Upon recovery, the individual checkpoints have to be analysed in order to compute the latest global consistent state and this may lead to a *domino effect* [131] where a global consistent checkpoint cannot be extracted from the set of available checkpoints (see Figure 2.6). This result in the whole application restarting in its initial state because the stored checkpoint cannot be used for state restoration. Since, in this case, uncoordinated checkpoint needs a global rollback which can potentially lead to a domino effect, it is not suited for the Fog-IoT ecosystem. A global rollback or a domino effect disrupts the whole application. **CP-Consistency.** Both a global rollback and a domino effect restore a state that may be inconsistent with respect to the PW. If a domino effect occurs, the whole application restarts in its initial state without taking into account the changes in the state of the application and the changes in the PW. In the case of a global rollback, the PW events generated after the checkpoint are not computed in the global state restored. Thus, CP-consistency is violated during recovery.

However, uncoordinated checkpoint can be coupled with logging techniques to prevent a global rollback as well as the domino effect. In this case, the PW events after a checkpoint can be recorded. Our approach thus makes use of the combination of these two techniques as discussed in Section 3.3.2.

c . Communication Induced Checkpoint

Communication induced checkpoint (CIC) [22, 31] is a compromise between coordinated and uncoordinated checkpoint. It avoids the synchronisation overhead between applicative entities by piggybacking[1] control information on applicative messages. The applicative entities take two types of checkpoints called *local* and *forced* checkpoints. Local checkpoints

[1]That is incorporating additional information on applicative messages

are taken independently of each other and forced checkpoint may be triggered by the information piggybacked upon reception of a message.

Cost. The properties of CIC are interesting because of the lower cost compared to coordinated checkpoint in synchronisation between entities participating in the application. The size of exchanged messages can however become large depending on the amount of information piggybacked. **Disruptions.** CIC bounds the rollback propagation during recovery in order to avoid the domino effect. However, it does not avoid a rollback to the last globally consistent checkpoint. The global rollback disrupts the whole application since it involves the whole application. **CP-Consistency.** CIC, similarly to the previous checkpoint techniques, does not capture the PW events generated after a checkpoint is performed. Thus, the recovery implemented in this case is not consistent with respect to the PW.

Checkpoint techniques based on rollback recovery are not suitable for the Fog-IoT ecosystem. This is because the recovery procedure disrupts the whole application for a single failure. Moreover, PW events that occur after a checkpoint are not taken into account in the recovery procedure. Thus, consistent with respect to the PW is violated. Uncoordinated checkpoint shows some good properties in terms of cost because it does not require any synchronisation scheme. Checkpoint techniques may also be unsuitable because some IoT devices are functionally constrained and cannot be checkpointed. Thus, state saving cannot rely on solely checkpoint techniques. Uncoordinated checkpoint can be coupled with event logging techniques in order to avoid a global rollback recovery as well as the domino effect. Moreover, event logging can be carried out without checkpoint for entities that cannot be checkpointed. This is discussed in the next subsection.

3.3.2 Event logging

Event logging (also called message logging) consists in recording events such as messages emitted and received as well as function calls[2] on a stable storage.

Event logging relies on a piece-wise deterministic (PWD) assumption [145]. This assumption states that all non-deterministic events (such as reception of an event or events coming from the PW) can be identified and are used for replaying during state restoration such that the applicative entity can deterministically reach its pre-failure state [66]. When an entity fails, the replaying of the event logs recreates its pre-failure state. Event logging can also be mixed with checkpoint techniques so that only the events following the last checkpoint are replayed.

The techniques used for event logging can be classified into three categories, with respect to the type of synchronisation and how the events are recorded, namely *pessimistic*, *optimistic*, and *causal* message logging.

a . Pessimistic Message Logging

[2]That is local or remote functions that are executed

Pessimistic message logging [24] records each message synchronously. It makes a "pessimistic" assumption: it assumes that a failure can occur at any given time and thus records an event on the stable storage before the event is processed. This guarantees that all the messages that have been processed are recorded on the stable storage.

Cost. Checkpoints combined with pessimistic logging can be done in an uncoordinated way. Thus, there is no need for synchronisation between the entities of the application for performing checkpoints. When a checkpoint is performed, the set of logged messages prior to the checkpoint can be purged. This can be used to control and limit the amount of information stored. A drawback of this technique is performance degradation during failure-free execution in applications that communicate a lot. In this case, this technique consumes the bandwidth that should be dedicated to the execution of the application. **Disruptions.** Pessimistic message logging has the advantage of being able to restore the state of the application after a failure without affecting failure-free entities. Hence, it causes a minimum disruption in the application during recovery. **CP-Consistency.** Pessimistic message logging has also the advantage of being able to capture the events received and sent to the PW after a checkpoint is performed. However, replaying messages on IoT devices for state restoration may have undesirable or dangerous consequences on the PW. This is because the device goes through an intermediary state for each message replayed before reaching the target state. For instance, in a drug injection device, replaying messages would mean that the already administered doses of a drug will be injected again. Even if a global state within the application is reached after restoring the target state, the state with respect to the physical world may be inconsistent because the restoration procedure has an impact on the physical world and has made it unsafe. We refer to this phenomenon as *cyber-physical inconsistency*. This is further discussed in Chapter 8.

Uncoordinated checkpoint combined with pessimistic logging presents some interesting properties in terms of cost and in limiting the disruptions of failure-free entities during the recovery procedure. Moreover, if an entity cannot be checkpointed because of its functional constraints, pessimistic logging can be used without performing checkpoints. Our approach thus makes use of the concepts of uncoordinated checkpoint and pessimistic logging.

b . Optimistic Message Logging

In contrast to the pessimistic technique, optimistic message logging [145] operates asynchronously. In this case, events are grouped and flushed onto the stable storage in a single operation. The recovery procedure is similar to the pessimistic technique if the "optimistic assumption" is met, that is, a failure does not occur before the events stored locally are sent to the stable storage.

Cost. The advantage of this technique is that it reduces the synchronisation delays and overhead drastically compared to pessimistic logging. Thus, it consumes less resources than the pessimistic approach. It is a good technique to adopt in applications where the frequency of failures is not high and failure-free performance is of concern. **Disruptions.** If the optimistic assumption is met, the state of the application can be restored without

affecting failure-free entities. However, if this assumption does not hold, then the recovery of a failed applicative entity is more costly and impacts other entities of the distributed application. A global rollback that disrupts the whole application may be needed. **Heterogeneity.** Optimistic logging, similarly to pessimistic logging, is also interesting because it can be implemented without checkpoint if an entity cannot be checkpointed. Thus, entities that cannot be checkpointed because of their functional constraints (e.g., blackboxes) can be restored using the messages logged. **CP-Consistency.** Optimistic logging, similarly to the pessimistic approach, can also capture PW events and requires the replay of messages during the recovery procedure. Therefore, both pessimistic and optimistic logging induce cyber-physical inconsistencies because of the replay of messages.

Uncoordinated checkpoint combined with optimistic logging also presents some interesting properties. It has a lower cost than pessimistic logging if the optimistic assumption is met. The recovery procedure can be implemented on a failed entity without affecting other failure-free entities if the optimistic assumption is met. Thus, our approach also makes use of the concepts of optimistic logging to save events.

c . Causal Message Logging

Causal logging [23] combines the advantages of pessimistic and optimistic logging. In this technique, the causal relationships between the exchanged messages are tracked and piggybacked on applicative messages. In this way the applicative messages carry the information to reconstruct the state of a failed entity.

Cost. Causal logging limits the synchronous access to the stable storage but is costly in bandwidth because the amount of causal information piggybacked on applicative messages can become large. Its cost in bandwidth can be larger than the optimistic and pessimistic approaches. Thus, causal logging can be costly in resource consumption. **Disruptions.** Causal logging only limits the rollback of any failed entity to the most recent checkpoint on stable storage [66]. This means that it may cause disruptions of failure-free entities during the recovery procedure. **CP-Consistency.** This global rollback can thus cause inconsistencies with respect to the PW as discussed in the previous sections.

Table 3.1 recaps the existing state saving techniques and indicates whether each technique is adapted with respect to the identified properties of the Fog-IoT ecosystem. Green coloured cells indicate that the state saving technique is suitable with respect to a property of the ecosystem whereas red coloured cells indicate the contrary. Yellow coloured cells indicate that the state saving technique is partially suitable. The Fog-IoT ecosystem requires state saving strategies that have a low consumption of resources, limit the disruption of the application, support high heterogeneity, and take into account the cyber-physical nature of IoT applications in order to recover in a state which consistent with the PW. Approaches based on a global rollback for recovery are unsuitable because the recovery procedure impacts all the failure-free entities of the application. Since the Fog-IoT ecosystem includes cyber-physical interactions, after a rollback to a previous state there is no guarantee that the pre-failure execution will be regenerated. The action that generated an input may not

be repeated again in the physical world. In addition, some devices cannot rollback to a previous state because they act on the physical world in a definitive way. Uncoordinated checkpoint can be coupled with logging techniques so as to eliminate the synchronisation overhead of coordinated checkpoint. This combination also allows the recovery from failures without impacting failure-free entities. Moreover, PW events can be captured. These events, however, should not be replayed during the recovery procedure because it creates intermediary state which may have undesirable or dangerous consequences on the physical world. In order to support heterogeneity, a uniform technique for the whole distributed application is not suitable. Instead, multiple techniques should be implemented within an application with respect to the properties of the individual entities.

State Saving Technique	Cost		Disruptions	Heterogeneity		Cyber-Physical Interaction (CPI)		
	Synchronisation Overhead	Resource Consumption	Recovery Impact	Heterogeneous applicative and infrastructure entities	Functional Constraints	Capture CPI	Avoid Intermediary State	Cyber-Physical Inconsistency
Coordinated Checkpoint	Synchronisation overhead becomes too large	Additional resources consumed by the state saving technique are not significant if the data to save is well defined and the frequency of checkpoint is well parametrised	All failure-free entities are impacted Computations are lost (global rollback)	Multiple techniques are needed in order to cope with heterogeneity	Not all entities can implement a checkpoint strategy. For instance, sensors and actuators with restricted API cannot be checkpointed	Checkpoint strategies cannot capture cyber-physical events when they occur after a checkpoint	A target state can be set without going through intermediary states	Global rollback implies that the restored state is consistent within the application. However, consistency with respect to the PW is not maintained.
Uncoordinated Checkpoint	No synchronisation between entities		Global rollback necessary Domino effect possible, affecting all entities					
Communication Induced Checkpoint	Synchronisation overhead minimal		Global rollback affecting all entities					
UC + Pessimistic Logging	Absence of synchronisation for UC Synchronisation for logging can become large if frequent communication		State saved is globally consistent Only failed entity is impacted by the recovery procedure		State saving can rely only on logging without uncoordinated checkpoint if an entity cannot be checkpointed	Events from the physical world can be captured	Events have to be replayed, thus creating intermediary states during the state restoration procedure	Intermediary states violate CP-consistency
UC + Optimistic Logging	Absence of synchronisation for UC Minimal synchronisation for logging		Only failed entity is recovered if optimistic assumption is maintained					
UC + Causal Logging	Absence of synchronisation for UC Overhead of piggyback increases rapidly	Bandwidth consumption becomes high	Failure-free entities may have to rollback					

Table 3.1: Comparison Between Existing State Saving Techniques

3.3.3 Tools For Fault Tolerance based on State Restoration

From an implementation point of view, checkpointing can be done at the system or application level.

System level checkpoint [127] consists in saving the state of an application in a completely transparent way with respect to the application. This approach is commonly used in high performance computing (HPC) where a system or core dump (register values, program, counters, execution parameters, address space, etc.) of the computational state of an applicative entity is done. Coordinated checkpoint implemented at the system level for HPC applications has received the most attention because it is based on concepts that are simple to implement and do not require any change in the source code of software elements. Such implementations include Condor [101], CoCheck [142], KAAPI [86], LAM-MPI [137], and OPEN-MPI [83]. Coordinated checkpoint is also available on production machine such as the supercomputer IBM Blue Gene/P [141]. Such tools are usually operating system specific. Thus, they cannot be used to checkpoint IoT devices that do not run Linux based operating systems. Distributed Multi-Threaded CheckPointing (DMTCP) [26] was studied in the IoT domain and showed good performances [28]. It provides a transparent system level checkpoint mechanism which does not require modification of the original application code or of the operating system. DMTCP implements a coordinator which accepts checkpoint command and notifies the managed processes. The DMTCP library then performs a checkpoint of each process to save its state. The checkpoint operation freezes the processes in their current computation and saves the checkpoint in a file. The inverse operation is done to restart the processes in their checkpointed state. However, one of the main drawbacks of DMCTP is that it can only run on some Linux distributions. Therefore, it can only be used to checkpoint processes running on the operating system and cannot be used to checkpoint IoT end devices.

Application level checkpoint [43] consists in saving only a few data structures that can be enough to recover the state of the application. An administrator (i.e., operator or developer) of the distributed application defines the data that should be included in the checkpoint as well as the frequency at which these data are saved. This approach is interesting because it limits the size of the checkpoint to only the data from which a state can be recovered. It is thus less costly than system level checkpoint. Moreover it eliminates the dependence of the checkpoint on the device or operating system. Application level checkpoint can also be used to perform a checkpoint of sensors and actuators by requesting the data that make up the state of the device. Likewise, this type of implementation of checkpoints can be used to determine the state of firmwares for which system level checkpoint cannot be performed. Application level checkpoint thus has a low cost in terms of resource consumption and can cope with the functional constraints of IoT devices. For these reasons, our approach makes use of application level checkpoint which is more suited for the Fog-IoT ecosystem.

3.4 Fault Tolerance based on Dynamic Reconfiguration

We focus in this section on some existing dynamic reconfiguration frameworks for achieving fault tolerance in distributed applications which are the closest to our proposed solution. We evaluate these approaches with respect to the criteria discussed in Section 3.1. Note that in this case, the *disruption* criteria is not applicable.

3.4.1 Solutions based on Dynamic Reconfiguration

In [50, 129] a lightweight infrastructure for reconfiguring applications based on Lira is proposed. The framework provides fault tolerance of component-based applications by detecting failures and recovering through dynamic reconfiguration at component and application level. A suitable new configuration is decided at runtime following a set of prespecified reconfiguration policies. The Lira architecture is composed of a set of agents which acts on the managed components of the application, implements the reconfiguration logic and communicates asynchronously with other agents using a management protocol. The component parameters that can be reconfigured are exported by an agent as variables or functions. A management information base contains the list of variables and functions exported by the agents. Lira implements different types of agents. The component agents are associated to software components and they monitor their states, manage their lifecycles and execute the reconfiguration procedure. Host agents provide the runtime support for installation and activation of agents and components. They allow the reconfiguration of the environment where the application is deployed. The application agent is a higher level agent that controls a set of components and hosts. **Cost.** The approach is designed to be lightweight. To this end, it makes use of the resources already available and uses a simple communication protocol based on SNMP [49]. **Heterogeneity.** The approach takes into account heterogeneity by establishing a reconfiguration strategy which decides the most rewarding new reconfiguration for the managed application, according to some predefined optimisation criteria. Thus, the agents are programmed on a component-by-component basis to respond to reconfiguration requests appropriate for that component. Our approach for reconfiguration is similar because we deploy managers that have a knowledge of the software elements (components) of the application. However, in this approach based on Lira, the host agent must run on the device where components and agents are deployed. In our approach, we also consider devices such as sensors and actuators that have no hosting capabilities. In this case, the agents are capable of managing remote entities. **CP-Consistency.** This approach does not focus on cyber-physical applications. Thus, it does not save the state of the application and does not record events received from or sent to the PW. Consistency with respect to the PW is not considered upon recovery.

Jade [38] is a middleware for repair management of distributed software environments and more specifically J2EE application clusters. The approach is based on wrapping legacy applications with Java objects in order to provide a uniform management interface to ab-

stract the heterogeneity of the legacy management capabilities. Thus it does not require the modification of the application it manages. It allows the autonomous management and reconfiguration of the managed system. Jade is based on a FRACTAL [44] component model which is a general component model implementing introspection capabilities to monitor the managed application as well as explicitly controlling the execution of components including their lifecycle (e.g., start and stop). It also implements reconfiguration capabilities at runtime. Jade is based on autonomic managers that implement control loops which capture information on the managed distributed application (such as QoS violations, CPU/memory usage, failure occurrences). It implements reactors that receive notifications for analysis and decision (repairing, resizing clusters). It also uses actuating managers to implement the necessary reconfiguration operations (adding and removing replicated components, updating connections). Active replication is used so that Jade provides a fault-tolerant repair service. Jade is interesting because, similarly to our approach, it implements an autonomous repair management based on the principles of autonomic computing. **Cost.** Jade targets repair management of J2EE application servers. Such applications usually run on cloud platforms which do not suffer from a lack of resources and capabilities. In order to run Java virtual machines in the Fog-IoT ecosystem, powerful machines are needed. This implies a high cost in terms of resource consumption. In our approach, we take into account the available resources of the environment in order to deploy managers that implement dynamic reconfiguration. **Heterogeneity.** The J2EE environment does not exhibit the level of heterogeneity of IoT applications as well as the diversity in functional constraints of devices. For instance, in IoT applications, architectural reconfiguration may involve not only software elements but also appliances such as sensors and actuators. Placement reconfiguration is not feasible for all devices, since some of them do not have accessible hosting capabilities. Jade does not take into account these heterogeneous constraints since the J2EE environment is different from the Fog-IoT ecosystem. In order to support heterogeneity, our approach implements multiple mechanisms that fit the properties and characteristics of the entities of the application. **CP-Consistency.** Since Jade was designed for the J2EE environment which does not include cyber-physical interaction, it does not recover a cyber-physical consistent state of the application. In order to guarantee CP-consistency, our approach restores the state of the application so that global consistency is preserved as well as a consistent behaviour with respect to the PW is maintained.

VAMP [68] is a self-deployment protocol that was designed to automatically configure cloud applications consisting of a set of software elements to be deployed on different virtual machines (VM). In addition to the deployment of VMs, the protocol starts the hosted software elements in a precise order according to their functional dependencies so that important architectural invariants are respected. This protocol works in a decentralised way and does not require a centralised server. To do so, each VM is embedded with a local configurator which interacts with configurators on remote VMs to solve dependencies and to determine when all the dependent entities of a software element are satisfied so that the latter can be started. The protocol implements a heartbeat mechanism for failure

detection of virtual machines. When a failure is detected, a notification is sent to the remaining virtual machines and a new VM instance is created. The protocol succeeds in deploying an application when faced with a finite number of failures. VAMP is also based on autonomic computing to automate repair management. **Cost.** VAMP was designed for virtual machines that run on cloud platforms and was implemented in Java. Thus, the lack of resources for the execution of configurators was not a priority in its design. **Heterogeneity.** VAMP focuses on the reconfiguration of virtual machines and does not take into account the heterogeneity of the underlying infrastructure. This is because it targets homogeneous applications running on cloud platforms. Thus, it does not take implement reconfiguration mechanisms that take into account the heterogeneous characteristics of entities. **CP-Consistency.** VAMP targets stateless applications that run in the cloud. Such applications do not involve cyber-physical interactions. VAMP, therefore, does not save the state of the application and does not provide mechanisms for maintaining a consistent behaviour of the application by restoring a CP-consistent state.

In [40], the authors present a reconfiguration protocol which applies changes to a set of connected components for transforming a current component assembly to a target one given as input. Reconfiguration steps aim at (dis)connecting ports and changing component states (started/stopped). The protocol is robust since it preserves consistency of the architecture of the application and is able to resist multiple number of failures that occur during the reconfiguration procedure. This assembly of components can be safely introspected and further reconfigured. Two recovery policies are proposed: a roll-backward policy that rolls back a failed reconfiguration and a roll-forward policy which pursue the reconfiguration as far as possible towards a desired target. **Heterogeneity.** This work focuses on the reconfiguration of web applications. These applications are not highly heterogeneous as the Fog-IoT ecosystem. Moreover, the protocol assumes that the components of an application are hosted on the same VM and a unique centralised manager is in charge of the reconfiguration steps. However, the Fog-IoT ecosystem is geographically distributed with software elements hosted on multiple devices. In this case, a distributed repair management protocol is needed. **CP-Consistency.** This work does not take into account cyber-physical interactions because its application domain is different from that of the Fog-IoT. Therefore, cyber-physical consistency upon recovery was not a concern in the design of the protocol.

In the context of wireless sensor networks (WSN), [113] presents the use of MinTax, a high-level language which is independent from the underlying hardware node. It is designed for energy-aware applications, to perform reconfiguration in WSN which have very strong constraints in terms of energy, processing, memory, and communication capabilities. The code is compiled dynamically on the node after its deployment. It takes into consideration the strong constraints in terms of energy as well as the size of code to perform the reconfiguration because of hardware and memory constraints. [32] proposes a reconfiguration mechanism to tolerate node and link failures. It establishes an alternate path to the root node in a hierarchical aggregation tree when a node or a link fails. The approach exploits the inherent node and link redundancy in WSN for reconnection and establishing new paths. **Cost.** One of the major concerns of reconfiguration in WSN is

the cost of the operations performed in terms of energy consumed. This is because devices usually run on batteries having limited capacities. In our approach, we focus on the cost in terms of overhead and resources (e.g., processing and storage). **Heterogeneity.** WSN, as its name suggests, is composed of sensors. In our Fog-IoT context, we consider various devices including sensors, actuators, and nodes having hosting capabilities (e.g., Raspberry Pi). Hence, the Fog-IoT ecosystem is much more heterogeneous in terms of physical architecture, functions, and network protocols. Our approach focus on the recovery of these different types of devices and software elements. **CP-Consistency.** WSN reports information about the state of the PW. They do not include actuators (e.g., lamps, heaters) which are capable of changing the state of the PW. Moreover, these sensors are usually stateless, that is they can be recovered by simply rebooting or replacing a failed sensor. Consequently, they do not take into account the restoration of the state of the application with respect to the PW.

3.4.2 Dynamic Reconfiguration in the Fog-IoT Ecosystem

Dynamic reconfiguration which aims at adapting distributed applications to specific conditions of the runtime environment have been studied for more than a couple of decades and resulted in many techniques and tools. Many other dynamic reconfiguration approaches have been proposed such as [157, 120, 72, 59]. These approaches focus on other application domains that present different constraints such as energy cost, throughput, and real-time constraints. Most of these solutions focus on preserving architectural invariants as well as ensuring functional and non-functional aspects of the application under reconfiguration. However, the Fog-IoT ecosystem presents additional specificities that should be taken into account for implementing dynamic reconfiguration.

- Cost: Most of the dynamic reconfiguration approaches have been proposed for applications that run in Cloud platforms. Since the Cloud is considered to abundant resources, these approaches do not focus on minimising the overhead and the resources consumed by the managers deployed. However, the Fog-IoT ecosystem is constrained in terms of capabilities. The resources available should be dedicated to the execution of the application rather than to the managers involved in reconfiguring the application.
- Heterogeneity and Diversity of Constraints: The existing techniques for dynamic reconfiguration do not support applications that exhibit the level of heterogeneity of IoT applications as well as the diversity in functional constraints of devices and applicative entities. For instance, in IoT applications, architectural reconfiguration may involve not only software elements but also sensors and actuators. The reconfiguration of sensors and actuators should take into consideration the geographical requirements as well as the APIs exposed by these devices. For instance, an architectural reconfiguration of a temperature sensor cannot involve another sensor located in a different room since the data reported would be incorrect. Placement reconfiguration is not feasible for all devices, since some of them do not have accessible

hosting capabilities. Moreover, failures of devices which provide specific services such as sensors may not always be repaired. These cases should be considered so that the application is reconfigured to provide a partial service.

- Cyber-Physical Consistency: Performing a reconfiguration is not enough to ensure a consistent behaviour of the application. This is because the state of a reconfigured entity has to be restored so that it can resume its behaviour in its pre-failure state. Reconfiguration should therefore be accompanied by state restoration so that global consistency in the application is maintained. State restoration also aims at ensuring CP-consistency such that the application has a consistent behaviour with respect to the PW. In this way, unsafe situations in the PW are avoided.

Our approach proposes reconfiguration operations that take into account the properties of devices upon reconfiguration. In order to limit the consumption of resources dedicated to the application, the managers performing reconfigurations are deployed only on nodes that have hosting capabilities and have enough resources to run the manager. Properties such as geographical constraints are taken into account when performing an architectural reconfiguration of an actuator such that the replacement actuator has the same scope of action as a failed one in the PW. Our approach also makes use of state restoration techniques, as discussed in the previous sections, to guarantee CP-consistency. In the following sections, we focus more on existing fault tolerance approaches for IoT applications.

3.5 Fault Tolerance in the Fog-IoT Ecosystem

This section discusses some existing approaches for tolerating failures of IoT applications and of the Fog. We first present the failure management solutions for Smart Home and Smart City applications because it is the target domain application of this Thesis. Then, we present existing solutions in healthcare since tolerating failures is critical in such an application domain and some recent research works have addressed this issue. Next, we briefly present solutions in miscellaneous IoT application domains before concluding.

3.5.1 Smart Home / City

Rivulet [27] is a fault-tolerant distributed platform for running smart home applications. It focuses on delivery of events and tolerates failures such as link losses, network partitions, and sensor failures. It relies on a model where software elements are executed on heterogeneous smart consumer devices within the home. Delivery of data is ensured by a delivery service with two types of guarantees: a best-effort guarantee for delivery of sensor events where a few values can be missed and a stronger delivery guarantee that ensures delivery despite failures for cases that cannot afford to miss delivery of events because it would be undesirable and potentially catastrophic. The latter comes with a more costly bandwidth and battery life. Rivulet is focused on delivery of events and targets stateless smart home applications. It does not provide any mechanisms for restoring the state of the application

upon recovery. Thus, cyber-physical consistency is not taken into account.

The authors in [117] discuss the different building blocks to provide resilient services for IoT applications in the Fog for smart city applications. They propose that IoT devices should be connected to redundant fog nodes so as to survive fog node failures. They suggest that better fault tolerance can be achieved if the communication coverage of fog nodes overlap across multiple IoT distribution areas and fog nodes should monitor each other based on proximity for faster failure recovery. The authors also identify the need for maintaining state data of stateful fog services in smart city applications. The discussed solution to tackle failures in this context is based on replication of stateful fog services such that the replicas of a fog service are updated with each service invocation and each change in state data so that a replica can replace a failed service. Replication of services in a Fog-IoT environment, however, is not always feasible because a service can be tied to the IoT device and thus cannot be replicated without replicating the device too. Replication of stateful IoT devices is also not realistic or sometime even infeasible. For instance, updating the state of the replica of actuators violates consistency with respect to the PW since actions would be performed multiple times in the PW. Geographical constraints may prevent the use of replication: some IoT devices such as smart window blinds can only operate in a unique space and thus cannot support physical replicas. Replication, when possible, may also be too costly.

[143] discusses the challenges, modelling, and research opportunities for achieving resilience of IoT in the context of smart cities. The authors discuss an architecture based on the concept of *islands* and *corridors* of resilience. The idea is to group clusters of connectivity corresponding to individual smart cities containing IoT devices and infrastructures for mobile telephone, internet as well as cloud and data-centers. This grouping is done so that the clusters (or *islands*) can continue to operate when the links to the core infrastructure are disconnected. In order to tolerate this kind of failure, network services such as web caches, DNS servers, edge infrastructure as well as 4G and 5G services should be replicated within each island. These islands should be connected to the core network and thus to one another through *corridors* capable of surviving large-scale disasters and attacks by implementing multipath routing and transport as well as path optimisations. A similar approach is discussed in [20]. The authors propose a resilient IoT architecture for smart cities composed of multiple layers. A first layer of the architecture is composed of *IoT islands* which the authors define as a group of physical devices such as sensors and actuators deployed in the city. An IoT island sends data to IoT services where they are processed. For the purpose of data transmission, an IoT gateway provides access to multiple IoT islands and acts as a bridge to the upper layer of the architecture called IoT middleware. The IoT gateway is responsible for guaranteeing data delivery. The routing protocol for lossy networks (RPL) is proposed to this end because it supports dynamic networks and allow the use of multipath techniques which enhance resilience of data transmission. The IoT middleware layer handles the following functions: heterogeneity by acting as an interpreter between the communication protocols used in the IoT islands, routing of data, and discovery of devices and services in the IoT infrastructure layer. A resilience manager in

the same layer has the role of supervision and defines the recovery actions that have to be applied in case of failures. These studies focus on very high level constructs. Albeit interesting, the proposed approaches are very difficult to test or implement. At this stage in the development of the IoT, failure management should rather focus on more practical aspects. Also, a unifying fault-tolerant architecture for IoT application could be an infeasible approach because the concepts and technologies in IoT continues to evolve rapidly. It imposes too many constraints in the design and development of the application. In addition, such an approach is not supported by legacy applications. On the contrary, there should be a clear decoupling between the architecture of the application and that of the proposed fault tolerance solution. Therefore, a failure management solution should adapt to the architecture and requirements of the application rather than vice versa.

3.5.2 Healthcare

[76] proposes an architecture for supporting network fault tolerance for the healthcare environment. It is based on the wireless protocol 6LoWPAN for an energy efficient communication infrastructure. The architecture consists of a customised star-based 6LoWPAN sensor nodes, which retrieve patients bio-signals, connected to a gateway composed of multiple sink nodes with backup routing and access to the internet. In order to provide fault tolerance, the inactivity of a sensor node over a predefined period of time triggers a discovery protocol which initiates actions to determine if a failure has occurred. The protocol begins by requesting the status of the sensor node. If the latter fails to reply, a warning message is broadcasted through another sink node to eliminate the possibility of a faulty sink node. The set of sensor nodes reacts to the warning message such that the failure can be identified. The use of backup routing between sink nodes allow one to maintain connectivity in case of failing connections and prevent traffic bottleneck due to high receiving data rate. The proposed architecture also implements a notification mechanism for care-givers/doctors. This approach makes use of star-based architecture as well as customised hardware. This type of approach is use case specific and cannot be extended to more generic IoT domain applications. Moreover, the approach relies heavily on notification and intervention of care-givers/doctors to confirm and correct a failure. In our approach we aspire at having a complete automation of detection of failures and recovery.

[77] proposes a fault tolerance mechanism in smart homes dedicated to people with disabilities. In this work, the authors focus on fault tolerance as a safety property and provide methodologies for the design and control of such smart homes. To do so, a smart home is modelled as a hierarchy of hardware and software elements, sensors, and actuators that are distributed in the rooms in order to help a person with impairment to perform activities of daily living. The components of the smart home and their properties are specified by the means of a synchronous framework. Formal synthesis techniques are then used to create a controller designed to keep the smart home in a correct state taking into account dynamicity, controllability, and temporal constraints. Security constraints are defined to guarantee a safe behaviour and improve security of the smart home environment under different ex-

ecution conditions. This approach focuses on security properties and more particularly on how to provide protection and adequate assistance with respect to the person's disabilities when a failure occurs. These security properties are verified for different execution modes of the application. Our approach is different because we aim at recovering the application and limiting service unavailability. Our recovery procedure implements mechanisms that avoid unsafe situations in the PW by (i) restoring a consistent state of the application by taking into account the changes in the PW during the recovery procedure, (ii) ensuring a consistent behaviour of the application after the recovery procedure is implemented, thus avoiding harmful impacts on the PW.

[75] discusses the technical requirements for reliable mobile health services. The authors identify a set of reliability requirements including monitoring of operations of hardware and software elements as well as provision for redundant hardware so that in case of outage a stand-by device with replicated application and service takes over. Power-fallback is proposed so that in the case of the failure of the main power supply there is an automatic handover to the alternative power source (e.g., battery). The authors recommend the implementation of techniques to allow reconnection to alternative cellular or WLAN networks. This ensures transmission of data when patients are moving as well as automatic adaptation of processing and storage capacities of centralised infrastructures according to load of mobile health users (patients and care experts). This approach relying on replication can become very costly, may not be suited because of geographical constraints. It may also be infeasible because it violates CP-consistency in the case of actuators that perform physical actions on the PW. Reconnection to alternative networks is interesting in some cases. However, many IoT devices implement only one network interface and they cannot switch to other types of network. In our approach, in order to solve this issue, if the failure of a network equipment that cannot be repaired implies that an IoT device can no longer communicate, then the IoT device is considered as failed. A recovery procedure is then performed to replace the IoT device. Our approach thus focuses on replacing a failed IoT device with another one which have similar features.

3.5.3 Miscellaneous IoT Application Domains

[163] studies fault prediction which provides the means to ensure safe operations of IoT applications as well as to anticipate maintenance. [146] proposes replicated services for fault recovery in IoT. This approach requires multiple devices that have identical services. On the other hand, [167] introduces virtual service composition in which data from multiple sensors of different modalities are used as fault tolerant backups for each other. [168] proposes fault tolerance techniques for IoT in the military domain. Fault Diagnosis is done at the sensor level by sending the same input to pairs of processing nodes and comparing their response, and at system level by distributed agents based on the simple network management protocol (SNMP). Cryptographic mechanisms are used for authentication and data transmission protection. The recovery phase involves the removal of the faulty node from the architecture. Other works have focused on reliability of the delivery of events from

IoT devices. Particularly, these works propose mechanisms based on the communication infrastructure to provide alternative and simultaneous routing paths [150, 97, 115, 61, 30, 132, 139, 104]. Wireless sensor networks (WSN) is also a very active research area. This application field usually consists in the deployment of low power sensors with limited transmission range for data collection and task monitoring. Various mechanisms have been proposed based on replication (e.g., hardware, paths), data aggregation, and the adaptation of the network topology in order to tolerate failures [88]. Even if fault-tolerant WSN has received much attention, this application domain exhibits a lower heterogeneity than the Fog-IoT ecosystem and has other characteristics (e.g., energy awareness) that should be addressed for failure management. Hence, the fault tolerance solution proposed in WSN does not address the issues inherent to Fog-IoT ecosystem.

3.6 Summary

This section has presented the existing approaches to tolerate failures in distributed application, including in the IoT domain application.

In the first part of this related work we discussed the desired properties of a fault tolerance approach for the Fog-IoT ecosystem. Such an approach to tolerate failures should take into account the cost in terms of overhead and resource consumption because the ecosystem is limited in capabilities. The proposed approach should also cause a minimum disruption in the application during the recovery procedures. For instance, it should not impact failure-free entities. The approach to tolerate failures should also integrate multiple mechanisms in order to cope with the diverse constraints and properties of different part of the application. The restoration of the state of the application after a failure is very important to ensure a consistent state of the application with respect to the PW and a consistent behaviour after recovery. This avoid any harmful impacts on the PW.

In a second part we discussed replication techniques for tolerating failures. This approach is very costly because at least half of the resources are dedicated for fault tolerance. Moreover, due to functional and geographical constraints, replication of some entities may not be feasible. Replication of IoT end devices may not always be feasible because actions performed by actuators may be executed multiple times in the PW.

Then, we discussed fault tolerance techniques based on state restoration. Table 3.1 recaps the properties of interest of each technique and indicate whether they are suited for the Fog-IoT ecosystem. Approaches based on global rollback for recovery are unsuitable because they disrupt the whole application for a single failure. Some devices (such as actuators) cannot rollback to a previous state because they act on the PW in a definitive way. Moreover, in order to support heterogeneity, a uniform state saving technique is not suitable. Multiple techniques should be integrated so as to fit the properties of individual entities within the application. Cyber-physical consistency is not guaranteed by these approaches because the replay of events may have harmful impacts on the PW.

Next, we presented existing approaches for dynamic reconfiguration. The existing approaches do not support hight level of heterogeneity. Thus, they do not take into account the diverse constraints in the Fog-IoT ecosystem. Moreover, the implementation of reconfiguration is not enough to ensure a consistent behaviour of the application. The existing approaches do not recover the application in a CP-consistent state.

The existing fault tolerance approaches in the IoT domain application were then presented. These approaches are usually use case specific and cannot be extended to more generic IoT domain applications. Other propositions are based on very high level constructs and are difficult to asses or implement at this current immature stage of the IoT. Also, these approaches do not assess the impact of failures and of the recovery procedures with respect to the physical world so as to maintain its safety.

Our approach addresses the issue of cyber-physical consistency by saving the state of the application and capturing a state of the PW during the runtime of the application. During the recovery procedure, a target state which is consistent within the application and also consistent with respect to the PW is computed using the saved data. Our approach causes a minimum disruption in the execution of the application since only the failed entity is recovered without rollback of failure-free entities. Heterogeneity is addressed by providing multiple mechanisms (i.e., for saving the state of the application, failure detection, reconfiguration, and recovery) based on the properties of the entities of the application. The cost induced by the approach is minimised by taking into account the available resources for the execution of the application as well as deploying managers only on nodes that have hosting capabilities and an excess of resources.

In the following, Part II of this Thesis details our contribution in designing an autonomic failure management approach that takes into account the specificities of the Fog-IoT ecosystem.

Part II

Autonomic Failure Management Approach For IoT Applications in the Fog

Chapter 4

Fog-IoT Model

Contents

This chapter aims at modelling IoT applications in the Fog. The model describes the characteristics of the different entities involved in the Fog-IoT ecosystem. Our failure management approach is based on this proposed model in order to manage the application. This implies that the approach implements the different mechanisms for failure management based on the characteristics of the managed entities. For instance, the monitoring mechanisms are determined by the type of entity (i.e., infrastructure or applicative) and the behavioural model of an entity is used for consistent state restoration during the recovery procedure. To this end, the application is defined by different types of applicative entities hosted on infrastructure entities. Applicative entities participate in the execution of the application through their functions and services which are described by their corresponding behavioural models. The behavioural model describes how the state of an applicative entity changes upon interaction with other entities and with the physical world. Infrastructure entities provide execution resources to run the application. In the following, we give a formal definition of the Fog-IoT ecosystem.

4.1 Behavioural Model

The behavioural model describes the behaviour of an application as a set of states and how these states change with respect to the operations that are executed by the entities of the

application. The behavioural model is important in order to characterise the state of the application at a given time instant. The information provided by the behavioural model is used in failure management to compute a consistent target state in order to restore the application during a recovery procedure (see Chapter 8). The information available to construct the behavioural model of an entity may be given by the manufacturer of a device or the developer of a service. For instance, a description of the behavioural model of the Ambient Light Sensor (ALS) [6] device is given by its manufacturer Texas Instruments. The behavioural model of the humidity and pressure sensor BME280 is also provided by its manufacturer Bosch [2]. Figure 4.1 illustrates part of the behavioural model of the BME280 device. The devices offers three sensor modes: sleep mode (default mode, no measurements are performed), forced mode (perform one measurement, store results and return to sleep mode), and normal mode (perpetual cycling of measurements and inactive periods). These modes can be selected by setting the value of $Mode[1:0]$.

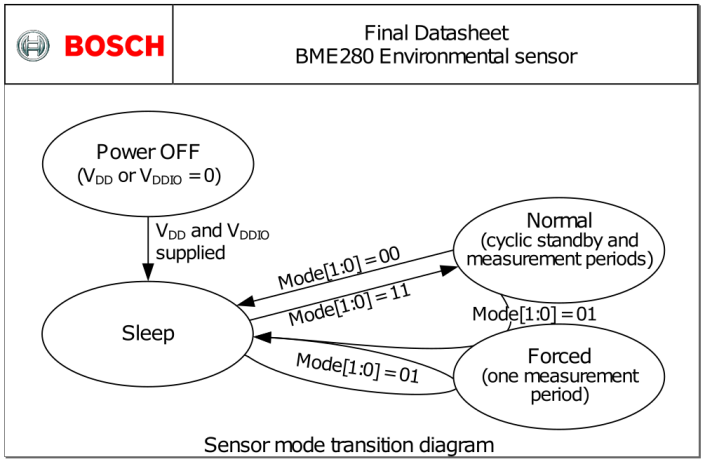


Figure 4.1: Part of the Behvaioural Model of BME280 Device [2]

The behavioural model may also be extracted from runtime contextual observations of an entity [122]. For instance the tool Learnlib [130] provides active automata learning [25] algorithms which are used for inferring models from observations. To this end, it interacts with a target application via queries (e.g., API calls) and analyses the output trace to construct the behavioural model of the application. In the following, we define the behavioural model of an application.

The execution state, or state for short, of an entity is characterised by a set of variables such as input parameters, execution conditions, environment variables, and stored data.

Definition 1 *(State) The State of an entity is defined as a set of couples* $(var_i, value_i)$ *where* var_i *is a variable identifier and* $value_i$ *is its corresponding value.*

The state of an entity changes upon reception and emission of events. The execution of internal computations also changes the state of an entity. An internal computation is

independent of received or emitted events. For instance, an action executed periodically such as a lamp switching on every day at 6 p.m. represents an internal computation.

The behavioural model (BM) of an entity describes how its state changes during its runtime. It is described by a set of states and transitions connecting these states. A transition represents the change of state upon reception/emission of events or the execution of an internal computation. Two types of BM are defined according to the information available on the behaviour of an entity: a *Default Behavioural Model (DBM)* and an *Enriched Behavioural Model (EBM).*

The DBM characterises entities for which building a complete BM is not feasible because the necessary information is lacking or impossible to retrieve (e.g., hidden APIs or obfuscated proprietary APIs). Therefore, a DBM defines the behaviour of an entity for which only the incoming/outgoing events of the entity can be observed and therefore can be defined. However, the subsequent changes in state induced by these events cannot be characterised. In this case, we refer to the set of states as a single *macro-state.*

Definition 2 *(DBM) A Default Behavioural Model is defined by the following tuple,* $DBM = (\{ms\}, ms, \Sigma, T_{dbm})$ *where the set of states is a singleton containing one macro-state* $\{ms\}$*;* Σ *is a finite set of events;* $T_{dbm} \subseteq \{ms\} \times \Sigma \times Act \times V_\Sigma \times Inv_V \times \{ms\}$ *is a finite set of transitions such that:*

- $Act \in \{?, !\}$ *where* ? *characterises the reception of an event and* ! *characterises the emission of an event.*
- V_Σ *corresponds to a set of expressions (e.g., a set of values) received or emitted.*
- Inv_V *defines the invariants associated to the values* V_Σ*. The invariants characterise the range of correct values of the variables in* V_Σ*. A transition is executed if the invariants are satisfied.*

In this case, we denote $\{ms\} \xrightarrow{e,act,v_e,inv_v} \{ms\} \in T_{dbm}$.

An EBM characterises applicative entities for which the behaviour specification is provided by the administrator of the entity (e.g., manufacturer of a device, the developer or provider of a service) or can be built from the observation of the entity. In this case, the set of state can be made explicit, as well as the events that cause the changes of states.

Definition 3 *(EBM) An Enriched Behavioural Model is defined by the following tuple,* $EBM = (S, s_0, \Sigma, T_{ebm})$ *where* S *is a finite set of states;* $s_0 \in S$ *is the initial state;* Σ *is a finite set of events;* $T_{ebm} \subseteq S \times \Sigma \times Act \times V_\Sigma \times Inv_V \times S$ *is a finite set of transitions such that:*

- $Act \in \{?, !, \sigma\}$ *where* ? *characterises the reception of an event,* ! *characterises the emission of an event, and* σ *characterises an internal computation.*
- V_Σ *corresponds to a set of expressions (e.g., a set of values) emitted or a set of variables allocated in the case of a received event.*

- Inv_V *defines the invariants associated to the values* V_Σ.

In this case, we denote $s \xrightarrow{e,act,v_e,inv_v} s' \in T_{ebm}$.

The two types of BM are illustrated in Figure 4.2. Figure 4.2 (a) illustrates the DBM of a face recognition software element. It is represented with a macro-state. In this case, the inputs and outputs based on events received and emitted are known. However, the variables describing the state of the software element and how these events affect its state are unknown (for e.g., images stored and how they are processed). The software element receives the event $faceRecognition?recognised$ such that the variable *recognised* indicates whether the person is recognised or not. Thus, the invariant of the parameter *recognised* is as follows, $recognised \in \{true, false\}$. The software element emits the event *status*! to indicate that a person is recognised. Therefore, the invariant defined for this transition is $recognised \in \{true\}$.

Figure 4.2 (b) illustrates the EBM of a connected lamp. The lamp has two states that are described by the variables *on* and *intensity*. The lamp receives events for its actuation. The transition $setOn?i$ represents the reception of an event to turn on the lamp at a given brightness intensity i. The variable *on* takes the value *true* and the variable *intensity* takes the value of the parameter i in the received event. The transition $setI?i$ modifies the intensity of the lamp. Both transitions define the invariants for the value of the parameter i which can take the values in the set $\{1, 2, 3, 4, 5\}$. Thus, the lamp can be turned on with five levels of intensity. Note that if the invariants are not satisfied, the transition is not executed. Executing the transition $setOff?$ turns off the lamp. In this case, the variable *intensity* is set to 0 and the variable *on* to $false$.

4.2 Physical World Model

The PW plays an important role in cyber-physical IoT applications because devices such as sensors and actuators interact with the PW. Thus, when a failure occurs in an IoT

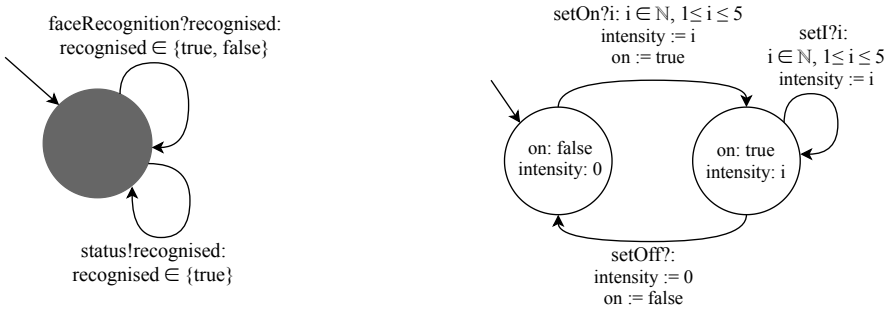


Figure 4.2: EBM and DBM Models

application, the unavailability of the service provided by the failed entity may impact the PW. For instance, the failure of a thermostat in a smart home will impact the ambient temperature and will make the house tenants uncomfortable. In this case, the temperature set by the thermostat can characterise the state of the smart home. Moreover, it is also important to characterise the scope of action of devices which interact with the PW. For example, the heater which is controlled by the thermostat may be located in a specific room of the smart home and thus only this specific room may be impacted by the failure. The state of the PW can be characterised by events given by sensors and actuators such as humidity, light intensity, motion, noise, smoke/gas, soil moisture, etc. A model of the PW which describes how the application interacts with PW and how it changes the state of the PW is useful. This is because it allows the observation of the state of the PW and thus a recovery procedure that is consistent with respect to the PW can be implemented.

The physical world is defined with respect to geographical spaces which are zones delimited by physical boundaries and define the scope of action of sensors and actuators. For instance, in a smart home, the geographical spaces represent the different rooms in the home and the external spaces such as the porch or the garden. These geographical spaces can be given by the BIM (Building Information Modelling) [155]. Each geographical space is characterised by a state consisting of a set of sensor and actuator events.

Definition 4 *(Geographical Space) A Geographical Space is defined as* $gs = (id, G, S)$ *where id identifies gs uniquely; G is an Euclidean space; the set* $S = \{(e_1, vt_1), ..., (e_n, vt_n)\}$ *characterises the state of gs where* e_i *is a sensed or actuated event and* vt_i *is the time duration for which the event is maintained and valid.*

Definition 5 *(Physical World) The Physical World is modelled as a finite set of n geographical spaces such that* $PW = \bigcup_{i=1}^{n} gs_i$ *where* gs_i *is a geographical space.*

Let us consider a smart home composed of three rooms. The physical world is given by $PW = \{room_1, room_2, room_3\}$. Each room is a geographical space. Consider that $room_1$ is equipped with a connected lamp, an air-conditioner, and a door opener sensor. The lamp should always remain on, the air-conditioner is set to 18°C, and the door should be kept open no more than 20s. In this case, the state of the room is given by these devices and is denoted $room_1(S) = ((lightOn, \infty), (temp : 18°, \infty), (doorClose, \infty))$. When the door is opened, the door opener sensor reports an event and thus $room_1(S) = ((lightOn, \infty), (temp : 18°, \infty), (doorClose, \infty), (doorOpen, 20s))$. Note that the timestamps of the events are also included when they are saved. The validity time of the events gives an indication of the context of the PW events. They are used to compute a consistent state with respect to the PW when a failure occurs. This is further discussed in Chapter 8.

4.3 Application Model

This section defines the entities involved in an IoT application. It is composed of the following applicative entities: appliances, software elements, fog nodes, and logical bindings. Below are the definitions of these entities. In the following, all identifiers are unique.

An appliance is an entity that provides a fixed set of services that are only operable through its exposed APIs. They have no accessible hosting capabilities and are usually cyber-physical entities that provide sensing and actuating services over a geographical space. An appliance has a behaviour and a corresponding current state at any given point in its execution.

Definition 6 *(Appliance) An appliance is defined by the tuple* $apl = (id, gs, I, bm, cs)$ *where id is an identifier; gs is a geographical space; I is a finite non-extensible set of interfaces through which its services are accessible; bm is the behavioural model of the appliance, and cs is its current state.*

A software element is a unit of software to be executed. It participates in the execution of the application through its corresponding functions and services. A software element has a behaviour and a corresponding current state at any given point in its execution.

Definition 7 *(Software Element) A Software Element is defined by* $se = (id, I, bm, cs)$ *where id is an identifier; I is a finite set of interfaces exposed by the software element through which its services are accessible; bm is the behavioural model of the software element, and cs is its current state.*

Software elements can interact with each other and with appliances. These interactions are done thanks to communications through logical bindings. Logical bindings are directed according to the dependencies between interacting entities. An entity e_1 is said to be functionally dependent on another entity e_2 (denoted $e_1 \rightarrow e_2$) if e_1 implements its services by using other services that are implemented by e_2. The dependency between e_1 and e_2 can be mandatory or optional. A mandatory dependency means e_1 requires e_2 to be functionally operable whereas e_1 can be functionally operable even if an optional dependency is not satisfied.

Definition 8 *(Logical Binding) A Logical Binding is defined by* $lb = (id, se, e, dir, dep)$ *where id is an identifier; se is a software element; e is a software element or an appliance;* $dir \in \{\rightarrow, \leftarrow\}$ *gives the direction of the dependency, that is,* $se \rightarrow e$ *or* $se \leftarrow e$*, and dep indicates whether the dependency is mandatory or optional.*

Note that there are no bindings between appliances. An appliance, therefore, always has at least one binding to a software element. If a logical binding exists between se and e, they are said to be neighbours, that is, $se \in neigh(e)$ and therefore $e \in neigh(se)$. Also, the number of neighbours of an applicative entity e is given by $deg(e) = |neigh(e)|$.

Software elements are hosted by fog nodes. The fog node provides access to the underlying physical resources and the runtime environment for the execution of software elements.

A fog node also hosts a special software element called *Fog Agent* which is responsible for lifecycle management (setup, installing runtime, reconfigurations) of local software elements and neighbouring appliances.

Definition 9 *(Fog Node) A Fog Node is denoted by* $fgn = (id, SE, fga)$ *where* id *is an identifier;* SE *is a finite set of software elements hosted on the fog node, and* fga *is a fog agent.*

Definition 10 *(Application) An IoT Application, or Application for short, is modelled as a directed acyclic graph,* $G_{app} = (V_{app}, E_{app})$*. Each vertex represents a software element* $se \in SE$ *or an appliance* $apl \in APL$ *such that* $V_{app} = SE \bigcup APL$*. Logical bindings are represented by the set of edges* E_{app}*. The application graph is connected, that is for an application with more than one software element/appliance,* $\forall v \in V_{app},\ deg(v) \neq 0$*.*

Figure 4.3 depicts the architecture of an IoT application. In this example, the application is composed of two fog nodes fgn_1 and fgn_2. The fog node fgn_1 hosts one software element se_1 and the fog agent fga_1. Both software elements se_2 and se_3, and the fog agent fga_2 are hosted on fgn_2. The behaviour of the respective software elements are given by BM_se_1, BM_se_2 and BM_se_3. The software elements se_2 and se_3 depend on se_1. They communicate with se_1 through the logical bindings lb_{21} and lb_{31}, respectively. The software element se_3 is connected to and depends on the appliances $appl_1$ and $appl_2$. This is illustrated by the bindings lb_{a31} and lb_{a32}. se_3 has an optional dependency on $appl_2$ which is depicted by a dashed arrow. All the other dependencies are mandatory. The behaviour of the two appliances are given by BM_appl_1 and BM_appl_2. The appliances are located in two distinct geographical spaces gs_1 and gs_2.

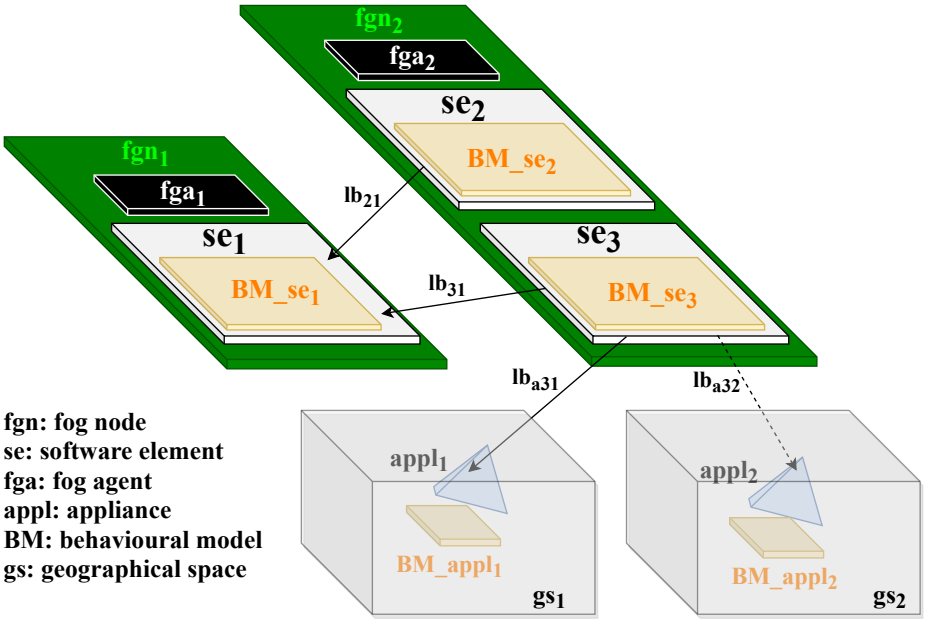


Figure 4.3: IoT Application Model Architecture

4.4 Infrastructure Model

The Fog infrastructure is composed of the following three types of physical entities: physical nodes, appliances, and network links. Since the services (software) provided by the appliance are tied to the device (hardware), we refer to an appliance as both an infrastructure and applicative entity. In the following, all identifiers are unique.

A physical node is a device that provides physical resources (computing, storage, and communication) and is fully administrable (resource allocation, deployment, reconfiguration, lifecycle management, etc.).

Definition 11 *(Physical Node) A Physical Node is denoted by* $pn = (id, fgn)$ *where id is an identifier and* fgn *is a unique fog node hosted by the physical node.*

A network link provides the transmission medium between a couple of physical nodes or a physical node and an appliance.

Definition 12 *(Network Link) A Network Link is denoted by* $nl = (id, pn, e)$ *where id is an identifier; pn is a physical node, and e is a physical node or an appliance.*

If a network link exists between two infrastructure entities pn and e, they are said to be neighbours, that is, $pn \in neigh(e)$ and $e \in neigh(pn)$. The number of neighbours of an infrastructure entity e is given by $deg(e) = |neigh(e)|$.

Definition 13 *(Infrastructure) A Fog Infrastructure, or Infrastructure for short, is modelled as an undirected graph,* $G_{infra} = (V_{infra}, E_{infra})$*. Each vertex represents a physical node* $pn \in PN$ *or an appliance* $apl \in APL$*. Thus,* $V_{infra} = PN \bigcup APL$*. Each edge represents a network link. The infrastructure graph is connected, that is for an infrastructure with more than one physical node/appliance,* $\forall v \in V_{infra}, deg(v) \neq 0$*.*

In order to run an application, it has to be deployed on the infrastructure according to a placement [42] that maps G_{app} to G_{infra}. Figure 4.4 illustrates the placement and deployment of an application onto an infrastructure.

The infrastructure is composed of two physical nodes, two appliances, and three network links. The physical nodes pn_1 and pn_2 host and provide the execution resources to the fog nodes fgn_1 and fgn_2, respectively. The network link nl_{12} provide the transmission medium between the two physical nodes. It provides the resources for the implementation of the bindings between the remote software elements. nl_{a21} is a wireless network link between pn_2 and $appl_1$ whereas nl_{a22} is a cabled network link between pn_2 and $appl_2$. These two network links, nl_{a21} and nl_{a22}, allow the interactions between the software element se_3 and the appliances $appl_1$ and $appl_2$, respectively.

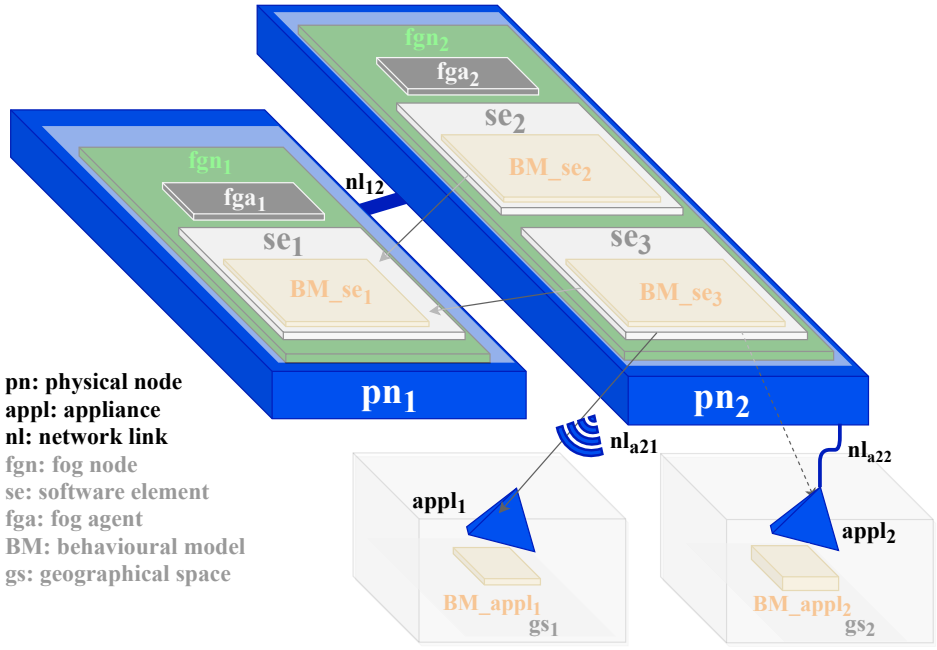


Figure 4.4: Mapping of the Application on the Infrastructure

4.5 Failure Model

The crash failure model is considered at the application and infrastructure levels. A crash failure occurs when an entity which was executing its correct behaviour deviates from its expected behaviour by halting unexpectedly.

At the application level, when a software element fails, it halts and stops its behaviour. It does not send or receive any messages and does not execute any further internal computations. A fog node fails when it can no longer give access to the underlying physical resources. The failure of a fog node induces the failure of all the hosted software elements since the local physical resources are no longer exploitable. A logical binding fails when the two entities connected by the binding can no longer interact via this binding because it can no longer transmit any messages or events.

Infrastructure failures affect appliances, physical nodes, and network links. When an appliance fails, it stops executing its behaviour. A physical node fails when it can no longer provide its resources to a fog node and will thus induce the failure of the hosted fog node and of its software elements. A network link fails when it can no longer transmit data. The failure of a network link can disconnect a physical node or an appliance from the infrastructure. In this case the physical node or appliance is unreachable and is therefore considered as failed. We, however, assume that failures do not create a network partition which results in a decomposition of the infrastructure or application graph in sub-graphs.

Figure 4.5 shows the causality between infrastructure and applicative failures: the failure

of a network link induces the failure of the binding(s) implemented over that network link. The failure of a physical node results in the failure of the hosted fog node and software element(s).

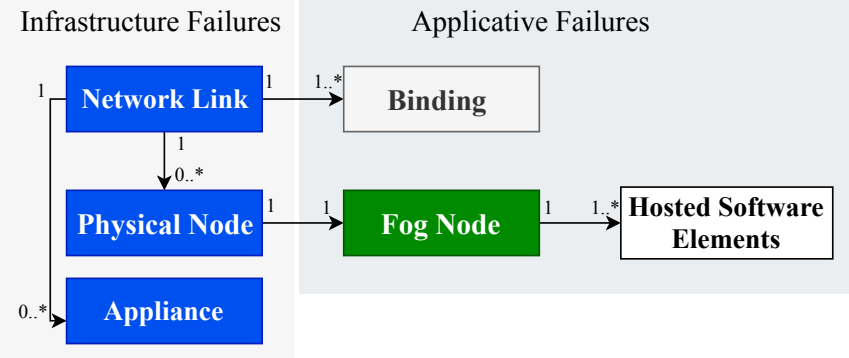


Figure 4.5: Causality Between Failures

The proposed failure model is motivated by real case of failures that can be observed in the Fog-IoT ecosystem. A physical node can fail because of a power failure (e.g., temporary outage, accidental unplugging) or the overheating/malfunction of the hardware. The failure of an appliance can occur due to hardware problems because sensors and actuators are placed in the PW and are therefore subjected to wear out caused by the PW environmental conditions. Appliances can also fail due to power failure because of a temporary outage, an accidental unplugging, or a battery drain. Moreover, infrastructure entities may be connected via wireless network links which are volatile. Infrastructure entities can, thus, temporarily lose synchronisation. Software elements can crash due to a lack of resources for their execution, unhandled exceptions, design/development errors, or poor memory management.

4.6 Running Example

A Fog-IoT application based on the model defined above is presented in this section. Figure 4.6 depicts an IoT application placed and deployed on a target Fog infrastructure in a smart home context. This application is a simplified version of the smart home testbed presented in Chapter 10 which automates the lights and detects intrusion in the house.

The smart home is composed of three geographical spaces where the appliances are located: the *bedroom*, the *living room*, and the *porch*.

The infrastructure is composed of three physical nodes on which fog nodes are hosted: $pn1(fgn1), pn2(fgn2)$ and $pn3(fgn3)$. The physical nodes are connected to each other through a cabled network link. $pn1$ is connected to the *Hue Lamp* and *Tap* device via a Zigbee network link. $pn2$ is connected to the *Wemo Motion Sensor* and *Awox Striimlight Lamp/Audio* via a Wi-Fi network link.

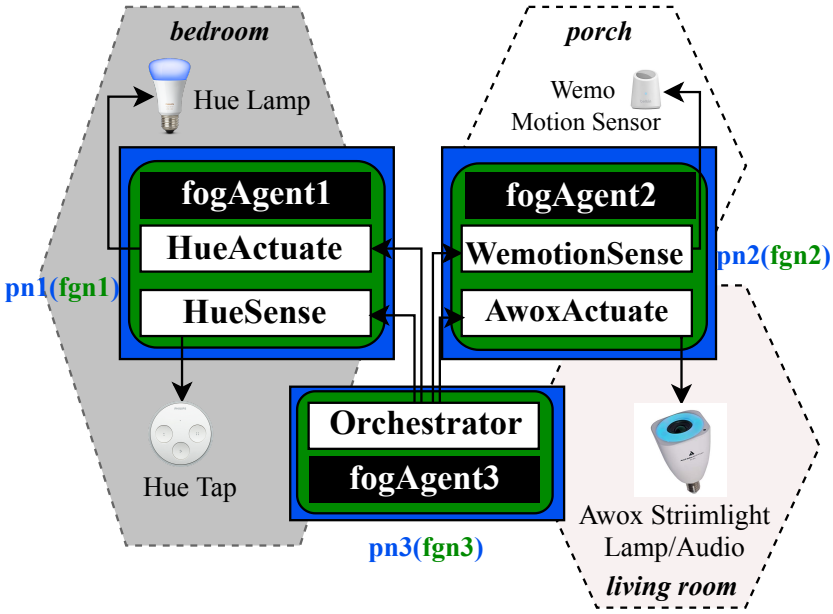
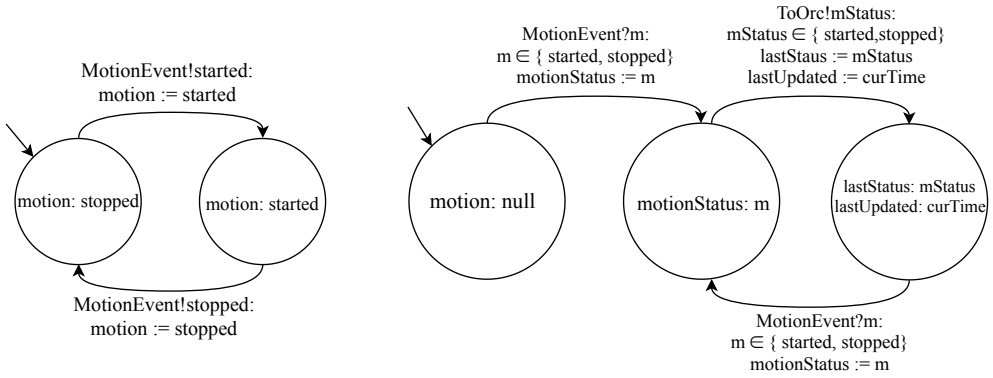


Figure 4.6: Running Fog-IoT Application

Each fog node hosts software element(s) and a fog agent. Each appliance is connected to a corresponding software element which retrieves events from the appliance concerning the geographical space in which it is located or pushes events to the appliance to perform actions on its geographical space. The software element *Orchestrator* determines the actions that should be triggered based on the sensed and actuated events. For instance, when one of the buttons of the *Hue Tap* device is pressed, the event is retrieved by the *HueSense* and forwarded to the *Orchestrator*. Depending on the button pressed and the number of times it is pressed, the *Orchestrator* determines the colour/intensity of the lamp in the bedroom or if it should be turned off. It then sends an event to the *HueActuate* to trigger the chosen action on the *Hue Lamp*. The *Wemo Motion Sensor* senses motion in the *porch*. If motion is sensed after bedtime, the integrated speaker in the awox device is triggered by the *Orchestrator* to warn the house tenants.

Figure 4.7 depicts the EBM models of the appliance *Wemo Motion Sensor* and of the software element *WemotionSense*. When a motion is detected, an event is sent by the motion sensor and the variable *motion* that characterises its state takes the value *started*. The variable *motion* is set to *stopped* when it detects that the motion has stopped. The *WemotionSense* retrieves the events from the motion sensor. It sets the variable *motionStatus* to *started* or *stopped* with respect to the event reported by the motion sensor. It then sends the corresponding event to the *Orchestrator* and updates the value of its *lastStatus* as well as the value of the timestamp given by *lastUpdated*.

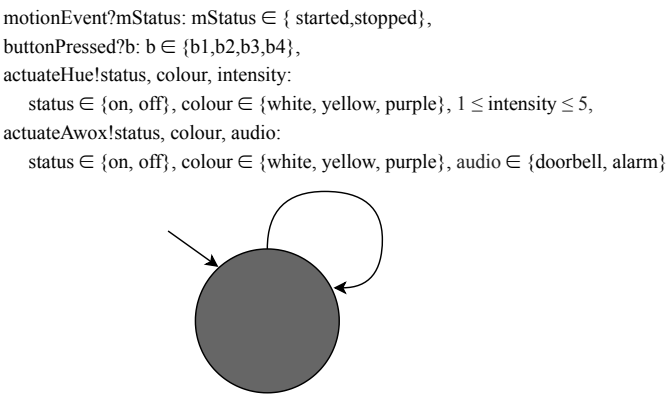
Figure 4.8 illustrates the DBM of the *Orchestrator*. This software element is considered to be a black-box. Thus, the incoming and outgoing events are known (i.e., the events received/sent from/to the other software elements) whereas its set of states and the way



(a) EBM of Wemo Motion Sensor **(b) EBM of WemotionSense**

Figure 4.7: EBM Behavioural Models

its states change with respect to these events are not known. It is thus represented by a macro-state.



DBM of Orchestrator

Figure 4.8: DBM Behavioural Model

4.7 Summary

This chapter has defined a model of the Fog-IoT ecosystem which describes the infrastructure, the application and its behaviour as well as the relation between the application and the physical world. This model is used by our failure management approach in order to implement the different mechanisms for managing the infrastructure and the application with respect to their characteristics. A failure model motivated by failures that can be observed in the Fog-IoT ecosystem is proposed. Finally, a running example of an IoT application deployed on a Fog infrastructure is presented.

Chapter 5

Resilience Approach Overview

Contents

The lifecycle of the application and of the infrastructure is presented in this chapter. It describes the different phases of the application and infrastructure. When the infrastructure is in a running phase, it provides the execution resources to the application. The infrastructure is continuously monitored so that the status of its lifecycle can be updated when a failure occurs. When the application is in a running phase, it executes its nominal behaviour. During its running phase two types of information are retrieved. First, the application is monitored to retrieve information about its lifecycle status. This allows the detection of failures. Second, information about the state of the application is retrieved. This consists in saving the variables and corresponding values characterising the state of the application as well as the transitions executed, according to the behavioural model of the application. When an entity of the application fails, failure notifications are propagated to entities impacted by the failure. The application is in a degraded mode where only a partial service is provided. When the failed entity recovers, then the application becomes fully operational again. This chapter presents the different steps (i.e., state saving, monitoring, failure notifications, and recovery) involved in our approach with respect to the lifecycle of the application. An architecture composed of a set of failure managers to realise these different steps and to provide autonomic failure management is proposed.

5.1 Application Lifecycle

The lifecycle of the application [67] includes all the phases [1] for building (design, development, testing) and executing an application. We focus in here on the execution phases which include initialisation, run, reconfiguration, and failures until the application is terminated.

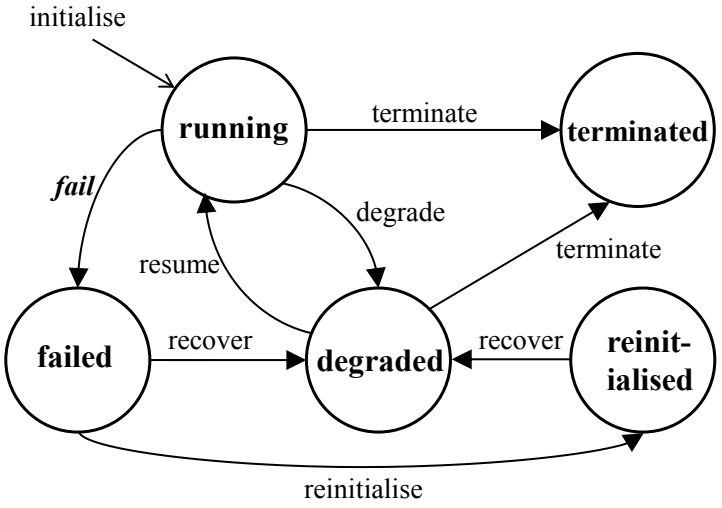


Figure 5.1: Lifecycle of an Applicative Entity

Figure 5.1 illustrates the different phases and transitions of the lifecyle of software elements and appliances. The labels, except *fail*, represent an action or a set of actions performed on a managed application.

The *initialise* transition includes all the initial lifecycle phases such as placement, deployment, download of all the necessary resources as well as setting up and configuring the runtime (e.g., downloading and installing libraries and packages, configuring the services provided by the operating system) until the applicative entity is in the *running* phase. An applicative entity executes its nominal behaviour when it is in the *running* phase. In this phase, it can emit messages, compute received messages, and execute internal computations. The initialisation phase is supervised and ensured by the *Application Lifecycle Manager (ALM)* which has a knowledge of the rules and constraints to be able to run an applicative entity such as placement location, resources needed, dependencies that should be satisfied, and runtime configuration.

When an applicative entity is in a *running* phase:

- It can move to the *terminated* phase where its behaviour is deliberately stopped and it is permanently excluded from the application. An applicative entity can be terminated upon the decision of the ALM because it has reach the end of its

[1] We use the term *phase* instead of *state* to differentiate between the lifecycle state and the state of execution.

execution. An entity can be terminated for failure management purposes, for example if a dependent entity has failed and cannot be repaired.

- It can move to a *degraded* phase for failure management purposes. In this phase, the entity temporarily provides none or only part of its behaviour. This is particularly important in order to notify the dependent entities that a failure has occurred. It allows a temporary pause to limit the failure from cascading to failure-free *running* entities. From this phase, it can move back into the *running* phase to *resume* its nominal behaviour or terminate.
- It can *fail* where its behaviour is unexpectedly stopped. When this happens, the *recover* transition aims at repairing the failed entity. The entity moves in a *degraded* phase after the recovery procedure. Its behaviour is only *resumed* when all its dependent entities are in their *running* phase. When an entity fails, it can also be *initialised* again (i.e., *reinitialised*) before recovery. This may happen when a new placement is required or the runtime needs to be configured again.

5.2 Infrastructure Lifecycle

The lifecycle of the infrastructure describes the lifecycle phases of physical nodes. It is simpler than the lifecycle of the application. These phases are illustrated in Figure 5.2.

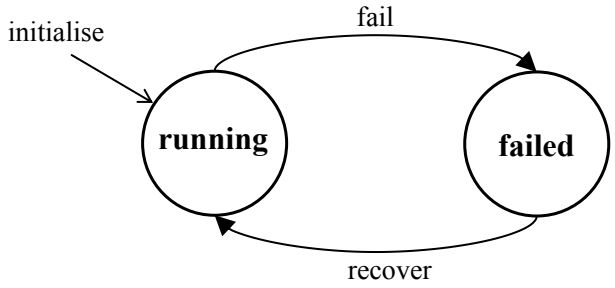


Figure 5.2: Lifecycle of a Physical Node

A physical node has two phases in its lifecycle: *running* or *failed*. In the running phase, the physical node provides its available resources for the execution of the application. When it fails, these resources are no longer accessible. It can *recover*, for example by rebooting, so that it moves back in its *running* phase. It is worth noting that the lifecycle of a fog node is tied to that of its associated physical node.

The next section gives an overview of the steps in managing failures.

5.3 Autonomic Failure Management

This section gives a global overview of the different steps involved in the proposed approach and how autonomic failure management is achieved.

The proposed failure management approach consists of four functional steps which are triggered according to the lifecycle phases of the applicative and infrastructure entities:

(i) *State saving*: This step aims at saving the state of the application in an uncoordinated way through techniques of checkpoint, message logging, and function call logging. The current state of each applicative entity is continuously saved and stored during the running phase of its lifecycle. A state of the physical world is also captured and is given by sensors and actuators.

(ii) *Monitoring*: This step aims at observing and reporting information on infrastructure and applicative entities relative to the different phases of their lifecycle. Monitoring allows the detection of failures of both infrastructure and applicative entities. It also gives information on the recovery of these entities.

(iii) *Failure notification*: When a failure is detected and confirmed, the state saving step of the failed entity stops. Failure notifications are propagated to notify the failure to dependent entities. Upon the reception of failure notifications, the dependent entities move to the degraded phase of their lifecycle to adapt their behaviour with respect to the failed entity.

(iv) *Recovery*: This is the final step which aims at repairing and restoring the state of the application. It is based on reconfiguration and state restoration. Reconfiguration aims at performing operations on the architecture of the application so as to change or re-establish its structure with respect to the failed entity. Reconfiguration is followed by state restoration where the pre-failure state of the application is restored by retrieving the data stored during its state saving step. When recovery is achieved, the dependent entities are notified through the propagation of recovery notifications so that they can resume their nominal behaviour.

Figure 5.3 illustrates how these failure management steps are performed on an applicative entity and how it affects the application over time. During the running phase of an applicative entity, two failure management steps are performed: state saving and monitoring. In this phase, the application is fully operational. When the applicative entity fails at time t_1, the monitoring step detects that failure. The state saving step of the entity stops. When a failure occurs, the unavailability of the service provided by the failed entity impacts the services provided by the application. Thus, the application is no longer fully operational. The subsequent steps, failure notifications and recovery are engaged. Failure notifications are propagated to the part of the application impacted by that failure. These entities move into a degraded mode of their lifecycle so that a partial service is provided. The recovery step, composed of a reconfiguration and state restoration of the application, is then performed in order to recover the functions of the application. The monitoring step detects that the recovery step is completed (i.e., at t_2). The application is fully operational again and the state saving step of the recovered entity can then be resumed.

A more detailed presentation of each step is given in the next chapters. These four failure management steps are complex and should be completely automated, that is, they

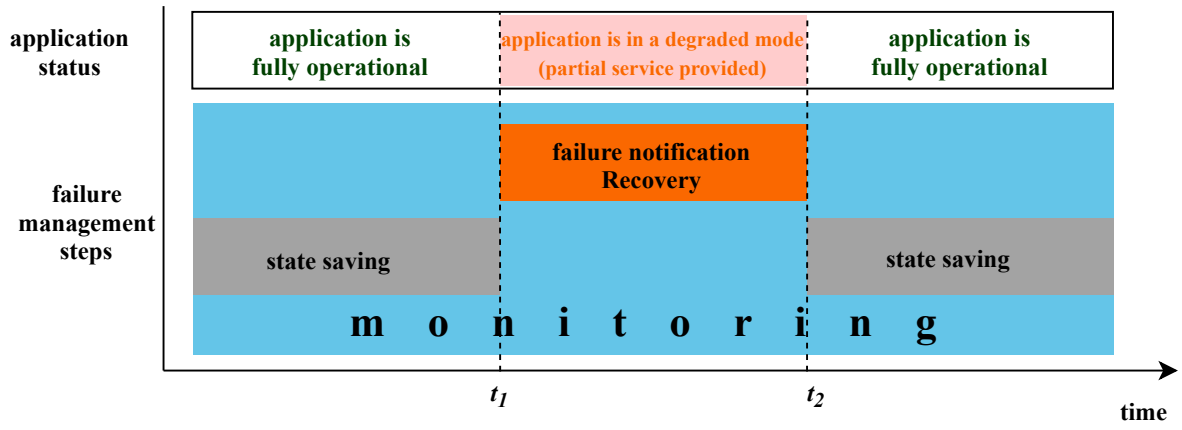


Figure 5.3: Failure Management Steps

should not require any manual intervention in order to eliminate any human errors and increase efficiency in terms of time and cost. In order to do so, the failure management approach is based on autonomic computing [80, 16] which advocates for *self-healing* [91] of applications. In this case, information about the application is continuously retrieved to automatically detect, diagnose, and repair problems so that humans are no longer relevant to take decisions.

Autonomic computing can be modelled by the MAPE-K autonomic control loop that defines a composition of *Monitor-Analyse-Plan-Execute* phases and a *Knowledge Base* which represents the data shared by all the phases. This is illustrated in Figure 5.4.

- *Monitor* collects, aggregates and reports data about the managed entities of the application through *probes*. In the context of our autonomic failure management approach, this corresponds to the provision of mechanisms that retrieves information on the application during the state saving and monitoring steps. The state saving step saves and stores data about the state of the different entities involved in the application. The monitoring step reports data about the lifecycle phases of these entities. These data are then exposed to the *Analyse* phase.
- *Analyse* is in charge of interpreting the data exposed by the previous phase. For instance, it should find out if there are any issues in the managed application by analysing the data reported. This phase is especially used to infer the occurrences of failures as well as to deduce the recovery of an entity.
- *Plan* defines the actions required to achieve a certain goal according to the defined strategies and policies. In this case, it defines the actions in terms of reconfiguration and state restoration that should be implemented based on the failure or recovery reported.
- *Execute* receives the actions decided by the *Plan* phase. These actions are performed by *effectors* that can act on the managed application in order to implement the

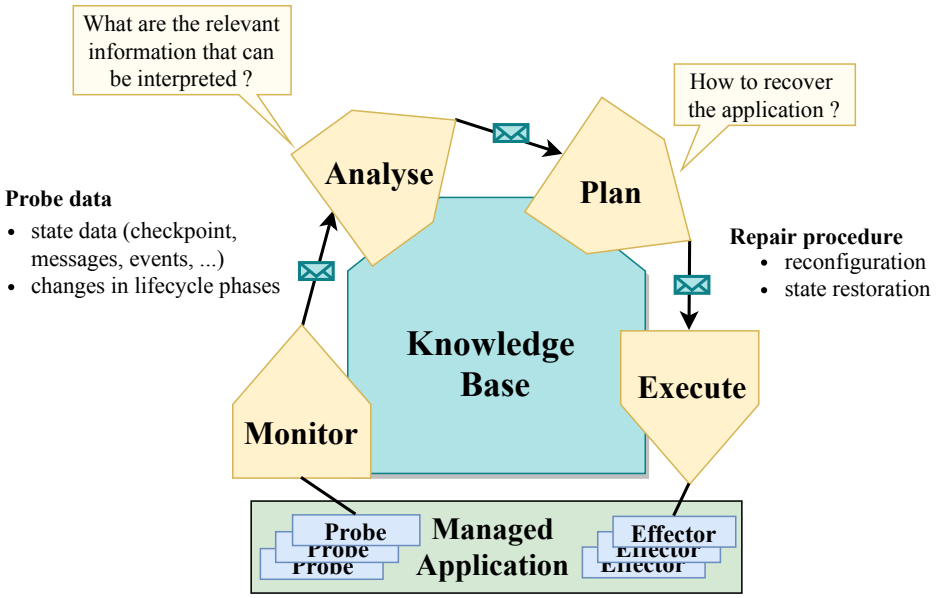


Figure 5.4: MAPE-K Autonomic Loop for Failure Management

necessary reconfiguration and state restoration.

These steps of the autonomic control loop are executed continuously during which the *Knowledge Base* is enriched with information on the different steps that are executed. In our approach, the probes and effectors correspond to local failure managers that are deployed on the fog nodes of the application. Global failure managers executes the *Monitor-Analyse-Plan-Execute* phases. These failure managers and their roles are described in the next section.

5.4 Failure Management Architecture

This section proposes an architecture of the failure managers (FMs) and an overview of their roles.

The FMs are placed and interact with each other as illustrated in Figure 5.5. They are composed of a set of local and global managers. The local managers are composed of *Wrappers* and *Fog Agents* which are deployed on fog nodes. They correspond to the probes and effectors that retrieve information and execute actions on the managed application. The wrappers act as membranes [44]. A wrapper intercepts incoming/outgoing operations of software elements and appliances, is able to control their behaviour, and perform reconfiguration operations on them. Fog agents set up wrappers on the local fog nodes and monitor neighbouring fog nodes. The *Global Decision Manager (GDM)* is a global manager which analyses the information provided by the probes (i.e., wrappers and fog agents) when failures occur and plans the recovery actions to perform. The failure management approach also relies on the following global managers: the ALM for lifecycle management,

a *Stable Storage (SSG)* to store state data, and *Thing'in* which is an object registry.

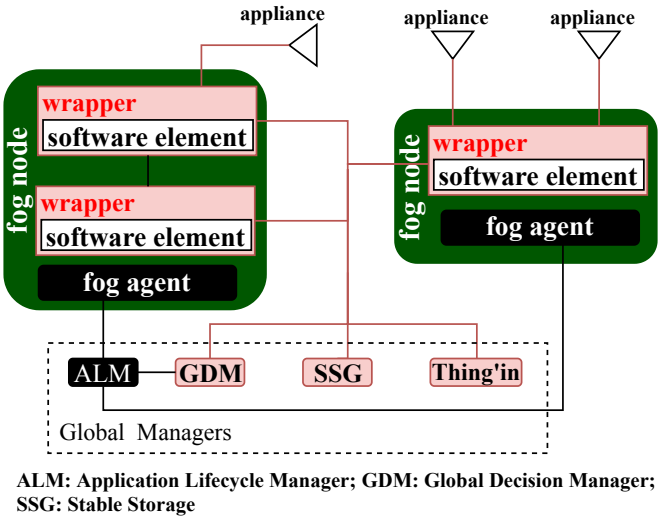


Figure 5.5: Failure Management Architecture

In the following, Section 5.4.1 and Section 5.4.2 detail the functions and roles of the local and global FMs, respectively.

5.4.1 Local Failure Managers

The local FMs are deployed on each fog node. They have only a local or neighbouring view of the entities to which they are associated.

a . Wrappers

A wrapper has multiple roles involving the four failure management steps described in Section 5.3. In order to ensure these roles, a wrapper acts as a membrane [44] to software elements and appliances, that is, it encapsulates software elements and appliances. Each software element or appliance associated to a wrapper is referred to as an *encapsulation*. A wrapper has the role of a probe to its encapsulations. It saves their states and monitors them for failure detection. The wrapper has also the role an effector to its encapsulations. It propagates failure notifications, control their behaviours (e.g., pause, terminate), and perform recovery actions on them.

Definition 14 *(Wrapper) A wrapper is defined by the tuple* $wrp = (id, fgn, se, APL, Dep, Req, Policy)$ *where* id *is the identifier of the wrapper;* fgn *is the fog node on which the wrapper is deployed;* se *is the software element encapsulated by the wrapper;* APL *is a finite set of encapsulated appliances such that* $\forall apl \in APL,\ apl \in Neigh(se)$*; Dep defines the set of dependencies of each encapsulation; Req defines a set of entities which depends on each encapsulation of the wrapper;*

Policy describes the failure management strategies for state saving, monitoring, failure notification and recovery of each encapsulation in $\{se\} \bigcup APL$.

The wrapper associated to a software and/or appliance(s) can be deployed locally on the same fog node or remotely. The local or remote deployment of the wrapper is determined by the nature of the encapsulation and the local resources available. In the case of software elements, the wrapper is deployed locally on the same fog node if the additional resources consumed by the wrapper does not exceed the available unconsumed resources by the application. Otherwise, the wrapper has to be deployed remotely on another fog node to avoid the consumption of resources dedicated to the application. A wrapper cannot be deployed on an appliance because it has no accessible hosting capabilities. Therefore, the wrapper associated to an appliance is always deployed remotely. In this case, the wrapper deployed for a software element is also associated to the set of neighbouring appliances if there are any. When multiple wrappers encapsulate an appliance, one of the wrappers is designated as primary. The primary wrapper of an appliance is the one chosen by the GDM for performing the reconfiguration and state restoration procedures when required. Table 5.1 recaps the deployment of the wrapper and its encapsulations with respect to the discussed constraints.

Entity	**Excess resources available locally?**	**Wrapper deployment**
Appliance		Remote
Software element	Yes	Local
	No	Remote

Table 5.1: Association and Deployment of Wrappers

As a consequence of the rules for association and deployment of wrappers, each wrapper has a one-to-one association with respect to software elements, that is, a wrapper encapsulates exactly one software element. The association of wrappers to appliances can be many-to-many since multiple software elements can interact with an appliance. Thus, the wrapper of each software element that interact with an appliance also encapsulates the appliance. Figure 5.6 depicts this relation between the wrapper and its encapsulations. A wrapper encapsulates one software element and an appliance is encapsulated by one or multiple wrappers depending on the number of neighbouring software element(s) of the appliance.

The roles of the wrapper with respect to its encapsulations are:

- State saving: The wrapper defines and applies the strategies for saving and storing the state of its encapsulations. These strategies are described by the means of state saving policies which are discussed in Chapter 6. To this end, it intercepts incoming/outgoing operations as well as piggybacks additional information on these operations as part of its state saving role. It also requests and stores their checkpoints according to the defined strategy.

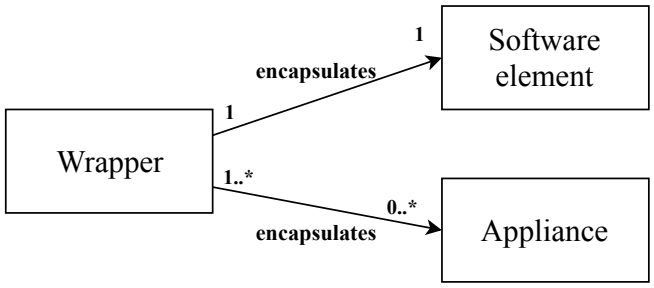


Figure 5.6: Entities Encapsulated by a Wrapper

- Monitoring: The wrapper defines and applies the monitoring strategy for failure and recovery detection of its encapsulations.
- Failure notification: The wrapper notifies the GDM of the failure of its encapsulations. It also receives failure notifications concerning failures that impact the execution of its encapsulations. When it receives such notifications, it moves the impacted encapsulations into a degraded phase. For instance, this can take the form of filtering of events and restrictions to the access of the encapsulation's APIs so it can operate in a degraded mode. The wrapper also propagates the failure notifications to the neighbours which have dependencies on the impacted encapsulations.
- Recovery: The wrapper defines and applies the local rules for recovery of its encapsulations. Recovery policies describe these rules and are detailed in Chapter 8. The wrapper reconfigures (architectural and placement) its encapsulations during the recovery procedure. It also retrieves the saved data for restoring the state of its encapsulations.

The details of the different strategies for state saving, monitoring, failure notification, and recovery are further discussed in Chapters 6, 7, and 8, respectively.

b . Fog Agents

Beside the lifecycle initialisation phase as discussed in Section 5.1, the fog agent has two more roles in failure management.

(i) Monitoring: The fog agent is also responsible for the monitoring step of failure management. A fog agent monitors each of its neighbouring physical nodes for failure detection by implementing a heartbeat mechanism. It detects the recovery of a physical node using the same mechanism.

(ii) Wrapper management: The fog agent manages local wrappers, that is, it sets up the wrappers and assigns the software elements and appliances to be encapsulated by the wrappers.

The local failure managers, therefore, report information about the application to the global failure managers, and subsequently receive instructions to perform operations on

the managed application. The global managers also provide the necessary information in order to perform these operations correctly.

5.4.2 Global Failure Managers

The global FMs have a global view of a subset of applicative and infrastructure entities, and subsequently participate in their failure management. This subset of entities is generally grouped with respect to geographical, functional, applicative, performance or other constraints. In the following, it is assumed that the global FMs are involved in a single application. Each global FM is subsequently treated as a single functional unit.

The most important global FM is the GDM which is a decision making manager which determines the actions to perform when a failure occurs. To this end, the GDM relies on the following global managers: the Stable Storage (SSG), Thing'in as well as the ALM.

a . Stable Storage (SSG)

The role of the SSG is to provide a reliable storage service that is unaffected by applicative failures so that wrappers can store and retrieve state data of their encapsulations. During the state saving step, the wrapper stores the state data on the SSG. When the encapsulation fails, it retrieves these data to restore a pre-failure state of the entity. The implementation of a stable storage may take various forms [95] such as a replicated file system on fog nodes or a RAID storage system [57]. It can also be based on data placement strategies [119] which exploit the Telco's Fog infrastructure. The type of implementation of the stable storage does not affect the proposed failure management approach. In this work, we assume a reliable storage medium on the Telco's Fog infrastructure which is dedicated to storage of state data.

b . The Object Registry Thing'in

Thing'in [14, 15] is a registry of the entities involved in an IoT application. It is implemented as a graph database. It defines ontologies that describe the properties of these entities as well as their relationships with each other and with the physical world. Thing'in also integrates a *building information model* (BIM) [155] which gives a virtual representation of geographical spaces (e.g., a home or building) with semantic and topologic information such as appliances' location and their scope of action on a geographical space. Thing'in exposes a set of APIs for querying the database. It can be queried for appliances having equivalent features to those of another appliance in a given geographical location. For instance, it can be queried for sensors capable of detecting the same changes in the environment (e.g., temperature, motion) and actuators capable of acting on the environment in similar ways (e.g., lamps, heaters). Thing'in is therefore particularly useful to find the replacement of a failed appliance.

c . Global Decision Manager (GDM)

The GDM is a decision making entity. It receives failure notification messages from fog agents and wrappers. It keeps a record of the failed/recovered entities and updates Thing'in accordingly. Upon the reception of a failure suspicion notification, the GDM decides when to consider the suspected entity as failed. When a failure is confirmed, it retrieves the dependent entities from the ALM and propagates failure notifications to them. It also decides the strategy and steps for recovery. For instance, this usually involves deciding if an entity is considered as failed based on the failure suspicion messages received, sending requests to Thing'in to find replacement appliances, and sending requests to the ALM to find a new placement for software elements. This is further discussed in Chapter 8.

5.5 Summary

This chapter presented a global overview of the autonomic failure management approach. The lifecycle of the application and of the infrastructure are described. The approach is composed of four functional steps: state saving, monitoring, failure notification and recovery. These steps are triggered according to the lifecycle of the application.

These different steps are ensured by local and global FMs. The local FMs are composed of wrappers and fog agents. A wrappers is involved in the four failure management steps concerning its encapsulations. The fog agent is involved in application lifecycle management, monitoring of neighbouring physical node, and the management of local wrappers. The global FMs are composed of the ALM which gives instructions to the fog agent for lifecycle management, a SSG where state data are stored, an object registry that keeps a record of entities involved in an application, and a GDM which takes decisions to recover from failures.

Chapter 6

State Saving Approach

Contents

The objective of the state saving step of the failure management approach is to save information on the state of the running application. State saving is done in an uncoordinated way. The state data are given by the behavioural models of the entities participating in the application. The state data are composed of the variables and corresponding values characterising the state of each applicative entity, as well as the transitions it executes. The state data are important in order to restore the state of the application so that a consistent behaviour can be maintained after recovery. The Fog-IoT ecosystem is highly heterogeneous in terms of communication model, functional constraints, resource capacities (e.g., storage, processing). In order to cope with the heterogeneous nature of the Fog-IoT ecosystem, the state saving approach cannot rely on a sole technique. For instance, the Fog-IoT implements multiple communication model based on messaging (e.g., MQTT, message buffers) and function calls (REST and SOAP APIs). In this case, state saving should implement techniques based on both message logging and function call logging. This chapter discusses the state saving approach, the chosen techniques for saving the state of an application, and how they are implemented in order cope with the specificities of the Fog-IoT ecosystem.

In the following, Section 6.1 motivates the state saving step of the failure management approach. Section 6.2 defines the state saving policies which detail the techniques and strategies for saving the state of the application. Section 6.3 concludes this chapter.

6.1 Motivation

An application executes its behaviour during the running phase of its lifecycle. Thus, during this phase, the state of the entities participating in the application changes upon reception and emission of events, as well as when carrying out internal computations. As discussed in Section 2.3.2, when a failure occurs, the failed entity loses the data which characterise its current state. These data include the variables and their corresponding values in its memory. They also include the events processed by the entity which determine its current behaviour. This implies that if the failed entity is merely replaced or restarted/rebooted, it restarts its behaviour in its initial state. In this state, the entity processes events in a different way than in its pre-failure state. Its behaviour is thus inconsistent with the rest of the application.

Figure 6.1 illustrates an example of inconsistent behaviour in a patient's morphine injection application. The application consists of a control button and a subcutaneous morphine injection device connected to the patient. When the patient presses the control button, a message $m_{injectdose}$ is sent to the injection device and a morphine dose is injected in the patient's bloodstream. Let us consider that a maximum of three doses can be administered per day for the safety of the patient. To this end, the injection device keeps a record of the number of doses administered and does not administer more than three doses even if it receives more messages from the control button. However, if the injection device fails and is rebooted (or is replaced with a new one), the device loses its current state. The number of already injected doses of morphine becomes unknown. The replaced/rebooted injection device executes its behaviour in its initial state in which the value of the variable $dose = 0$. In this case, the injection device is in an inconsistent state. When the patient presses the control button again, extra doses of morphine are injected. When this happens, the PW becomes unsafe since it can be dangerous to the patient. Thus, cyber-physical consistency is violated, that is, the application has an inconsistent behaviour with respect to the physical world.

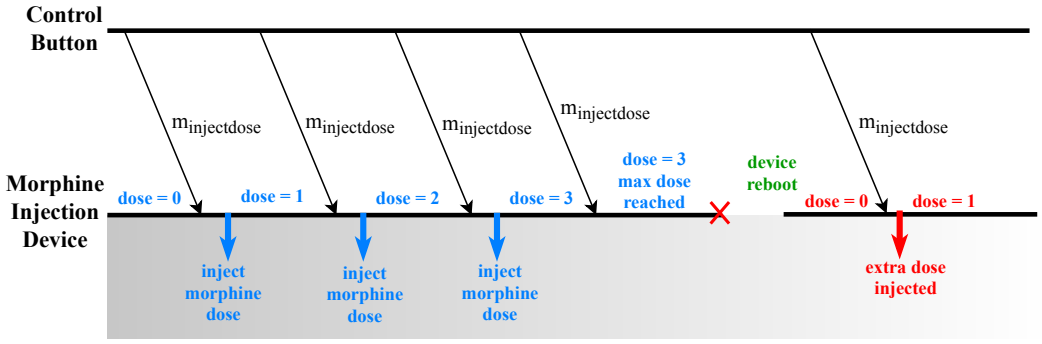


Figure 6.1: An Example of Inconsistent Behaviour

State saving aims at saving data from which a consistent pre-failure state of an entity can be restored. The execution of the entity is then resumed in this state so that it maintains

a correct behaviour. In the example illustrated in Figure 6.1, these data correspond to the the value of the variable *dose* of the injection device as well as the messages sent by the control button device and received by the injection device. After reboot, these messages can be used to deduce the number of morphine doses that have already been administered to the patient. Thus, the variable *dose* can be set accordingly so that the injection device does not have an inconsistent behaviour and the application is in a consistent state with respect to the PW.

The state of an entity is characterised by the data in its volatile memory as well as the data stored on the local storage (e.g., libraries, packages, configuration files). The identification of the type of entity that has failed is important since it determines which data are lost and should therefore be saved during failure-free execution. The failure of a software element implies the loss of the data in its volatile memory whereas its stored data on the local storage of the physical node may still be available. However, when a physical node fails, in addition to the data in their volatile memory, the locally stored data by the hosted software elements are also lost. This is because the local storage of the physical node is no longer available. When an appliance fails, the data in its volatile memory and local storage are lost.

Therefore, the state saving phase should save the data which characterise the current state of the application. These data are then used to restore the state of the application when a failure occurs. The state restoration procedure avoids the restart of the application in its initial state in case of failures and maintains a consistent behaviour of the application.

6.2 State Saving Policy

The state saving phase of software elements and appliances is performed by wrappers. A wrapper saves the data from which the current state of its encapsulations can be resumed after a failure. These data are referred to as the state data of the encapsulation and is composed of checkpoints, message logs, and function call logs.

Definition 15 *(State Saving) State saving consists in saving a set of data during the running phase of an application from which its current state can be resumed after a failure.*

The composition of the state data saved of an applicative entity depends on the state saving technique used. The state saving step is based on multiple techniques including checkpoint, message logging, and function call logging. This is because the Fog-IoT ecosystem is highly heterogeneous in nature and imposing one uniform technique for state saving is not feasible. A checkpoint consists in saving and storing the set of couples of variables and corresponding values, $s = \bigcup(var_i, value_i)$ that characterise the state of an encapsulation at a given time in its execution. The data to save in a checkpoint is thus given by the corresponding EBM model of the encapsulation. Message logging (resp. function call logging) consists in saving the transitions that are executed by the entity upon emission and reception of events (resp. execution of internal computations and remote function calls). The transitions are given

by the EBM or DBM model of an encapsulation.

We propose state saving policy in order to choose a state saving strategy which is suitable and adapted to the properties of an entity. A state saving policy describes all the details associated to the state saving step of an applicative entity. To this end, a wrapper assigns a state saving policy for each of its encapsulations. The state saving policy describes a state saving strategy that is suited to the encapsulation with respect to its properties and that of its runtime environment. In order to do so, the wrapper embeds a description file of the state saving policy for each of its encapsulation which describes the locally stored data to save on the stable storage, the state saving technique, the frequency of checkpoints, the storage location of the state data, and the additional data to include when saving events. This is illustrated in Figure 6.2. A description file is generated for each encapsulation of a wrapper. Figure 6.2 gives an overview of the different parameters of the state saving policy before describing them in more details in the following sections.

EncapsulationId gives the identifier of a software element or appliance encapsulated by the wrapper. The state saving policy defined in this description file applies for the identified encapsulation.

For appliances, **LocalData** indicates the data that are stored on the local storage as well as the filesystem path to access these data. It corresponds to configuration files of

```
EncapsulationId: <id>                                  /*id of encapsulation*/

LocalData:  {(<name>, <path>)}                  /*set of couples (name, path)*/

StateSavingApproach:
             Technique: Checkpoint | MsgLog | FctCallLog
             ER:  Reception ⊕ Emission&Reception
             OP:  Optimistic ⊕ Pessimistic

CkptParams: /*a weight of 0-2 is associated to each event to determine the ckpt frequency*/
          Weight: { (<evtType>, (0 ⊕ 1 ⊕ 2))}
          Freq: (T_ckpt, <ckptInterval>) | (<N_ckpt>)
          Storage: (stableStorage ⊕ stableStorage&Local)

Storage:
        Local: <path>    /*local filesystem path to store state data*/
        StableStorage: <path>    /*access to stable storage*/

VT: {(<evtType>, <vtValue>)} /*a validity time is associated to each event*/
```

⊕ : exclusive or | : inclusive or MsgLog: message logging FctCallLog: function call logging

Figure 6.2: Description of the State Saving Policy of an Encapsulation

appliances. For instance, for a temperature sensor it can be the configured frequency of sending temperature data. It can also be local configuration files such as information on passwords or ports so that the appliance can connect to a service or a network. For software elements, *localData* indicates the files that are needed for the initialisation and running phase of the software element, such as binaries, libraries and scripts. These are needed if the software element has to be recovered on a different fog node.

StateSavingApproach details how the state of the encapsulation is saved. *Technique* specifies the state saving technique adopted based on checkpoint, message logging, and function call logging. The chosen technique for an entity can be one of these techniques or a combination of multiple techniques. *ER* specifies whether logging is done at reception only or both emission and reception. *OP* indicates whether an optimistic or pessimistic approach is adopted for logging. Sections 6.2.3 and 6.2.4 detail the different techniques and strategies for state saving.

The information defined in **CkptParams** are used to compute the frequency of checkpoints. A maximum time elapsed between two checkpoints is defined by $ckptInterval$. The checkpoint frequency is also based on the cost of processing events. To this end, *Weight* gives information on the cost of processing an event (e.g., time to compute) by the encapsulation. When the total cost exceeds N_{ckpt}, a checkpoint is performed. A checkpoint can be stored on the stable storage or both locally and on the stable storage. Section 6.2.2 describes in more details how the frequency of checkpoints are computed and how checkpoints are stored.

Storage specifies the access to the local and stable storage. *Local* gives the local filesystem path to temporarily store state data. *StableStorage* details how to access the stable storage. For instance, state data can be pushed to the stable storage via REST API or by publishing events on a communication bus. The choice for storing state data on the local or stable storage depends on the state saving approach. This is further discussed in the following subsections.

VT defines the validity time *vtValue* for each type of events *evtType*. It indicates the time duration for which the event is valid and therefore should be used to compute a consistent state of the application during recovery. Section 6.2.5 further describes the validity time of events.

The state of an encapsulation is therefore saved and stored by the wrapper according to its state saving policy. The parameter *StateSavingApproach* of the description file is automatically assigned based on the characteristics of an encapsulation and of its runtime environment as discussed in the following sections. The filesystem paths to access the local storage as well as the access to the stable storage has to be input by the developer and/or operator (DevOps) of the application. The *Weights* and VT of events have also to be input by the DevOps of the application. Note that values for the *weight* ($w = 1$) and VT ($vtValue = \infty$) are automatically assigned by default. This avoids the DevOps to manually report the values for each event. The *Weights* and VT can also be automatically

inferred as discussed in the future work of this Thesis (see Chapter 11.2). The details of the state saving policies and how they are chosen are further described in the following sections.

6.2.1 State Saving Approach

The state saving approach is built upon three strategies: uncoordinated checkpoint, message logging, and function call logging.

The wrapper performs uncoordinated checkpoint for encapsulations which are characterised by an EBM. This is because the variables and their corresponding values to include in a checkpoint are given by the EBM model. The checkpoint technique can further be coupled with message logging and/or function call logging in order to save the operations that are executed after a checkpoint. In this way, the current state of the application can be restored based on the transitions executed after a checkpoint. An encapsulation which is characterised by a DBM cannot rely on the checkpoint technique since the details of its set of states are unknown. In this case, the state saving technique is based on message and/or function call logging depending on the communication model implemented. Table 6.1 recaps the choice of the state saving techniques based on the type of behavioural model of an entity.

Behavioural Model	State Saving Technique based on
EBM	Checkpoint and Logging
DBM	Logging

Table 6.1: BM and State Saving Techniques

Message and function call logging can further be based on an optimistic or a pessimistic approach. Logging can also take place at the reception or emission. Table 6.2 illustrates how these different strategies are chosen with respect to the following three criteria: the communication model, the properties of the local storage, and the type of the destination entity.

a . Communication Model

The communication model implemented between two interacting entities can be based on messaging (e.g., message buffers, message oriented middleware) or function calls (e.g., REST and SOAP APIs). The communication model thus determines whether the state saving technique is based on message logging or function call logging. Function call logging can also be used to save the internal computations executed by an encapsulation. An entity can interact with multiple other entities and can thus implement both communication models. In this case, the technique for saving its state is based on both message logging and function call logging.

In order to log messages received, the wrapper intercepts incoming messages before delivering them to the encapsulation. Messages emitted are also intercepted before forwarding them to the destination entity. Likewise, function calls are intercepted before calling the original function. Function call logging can also be achieved at the caller or at the callee (we refer to the former as emission and to the latter as reception, respectively). The intercepted messages and function calls are saved at emission and/or at reception depending on the type of destination entity.

b . Destination Entity

A message can be logged either by the emitting or by the receiving wrapper. The choice of which wrapper saves the message or function call is done according to the nature of the destination entity. If the destination is a software element, logging is done at the receiving wrapper independently of whether an appliance or a software element has emitted the event. However, the wrapper associated to an appliance is deployed on the neighbouring fog node because of the lack of hosting capabilities on the appliance itself. In this case, it means that the wrapper has to save the emitting events intended for an appliance because logging cannot be done at the receiving end. Therefore, a wrapper that encapsulates only a software element will save receiving events only. On the other hand, a wrapper that encapsulates both a software element and an appliance has to log events that are received for the software element and events that are emitted to the encapsulated appliance.

c . Local Storage Properties

In order to determine if the state data can be stored locally on the fog node, the nature

Local Storage		Communication Model: Function Call	Communication Model: Messaging	Destination Entity
Persistent		Optimistic Function Call Logging at Reception	Optimistic Message Logging at Reception	Software Element
		Optimistic Function Call Logging at Emission & Reception	Optimistic Message Logging at Emission & Reception	Appliance
Unstable		Pessimistic Function Call Logging at Reception	Pessimistic Message Log at Reception	Software Element
		Pessimistic Function Call Logging at Emission & Reception	Pessimistic Message Logging at Emission & Reception	Appliance

Table 6.2: State Saving Techniques

of the local storage infrastructure is characterised as persistent or unstable. An unstable local storage means that data stored locally can be lost or become inaccessible at any time instant because of failures. Thus, state data cannot be stored locally, even temporarily, because if a failure occurs, the state data can no longer be retrieved. Appliances have an unstable storage and do not have additional disk storage capacities. This limitation implies that their state data cannot be stored on the appliance itself. The storage of physical nodes can be classified as unstable or persistent depending on the capacities and stability of the device. A physical node has an unstable local storage if it cannot be rebooted after a failure and the data stored locally become inaccessible. For instance, a physical node powered by a battery cannot be automatically rebooted if it fails because of a battery drain. In this case, the local storage of such a physical node is classified as unstable. A physical node having a persistent storage implies that state data such as checkpoint and logs can be stored locally. The logs can be grouped locally and later be flushed all at once on the stable storage. This gives rise to the logging techniques based on an optimistic or a pessimistic approach as described in Section 6.2.3. Therefore, optimistic logging is suitable for a persistent local storage whereas pessimistic logging is done for encapsulations which are deployed on an infrastructure entity which has an unstable local storage.

6.2.2 Frequency of Checkpoint

The wrapper defines the frequency at which an encapsulation is checkpointed. The frequency of checkpoint is based on the execution time since the last checkpoint as well as the number and types of events processed by the encapsulation.

A maximum time interval between two checkpoints is first set. It is denoted T_{ckpt}. If $T_{ckpt} \geq (T_{current} - T_{lastCkpt})$, where $T_{current}$ is the current local timestamp of the encapsulation and $T_{lastCkpt}$ is the timestamp of the last checkpoint performed, then the wrapper performs a checkpoint of the encapsulation.

A checkpoint can be performed before the expiration of T_{ckpt} based on the events that are processed by the encapsulation. N_{ckpt} denotes the maximum number of events processed after which a checkpoint has to be performed. To this end, events are classified into three weights $w \in \{0, 1, 2\}$ according to how the event is processed by the encapsulation. An event that does not affect the state of the encapsulation has a weight $w = 0$ and is therefore not taken into account in the calculation of N_{ckpt}. For instance, events that are received by a stateless software element are assigned a weight $w = 0$. In this particular case, the software element is never checkpointed. Events that are costly in terms of resources and induce a long processing by the encapsulation are assigned a weight $w = 2$. For example, an event that induces a hot patching (e.g., version update) of a software element is assigned a weight $w = 2$. A checkpoint is immediately performed after the encapsulation has finished processing the event so as to optimise the state restoration time. It is not processed again during the recovery procedure. All other events are assigned, by default, a weight $w = 1$. When the total weight of the processed events exceeds N_{ckpt}, that is, $\sum w_i \geq N_{ckpt}$, a checkpoint of the encapsulation is performed by the wrapper.

Algorithm 1 illustrates the computation of the checkpoint frequency. Upon reception of an event ($l.1$), the wrapper saves the event according to the state saving policy and then delivers the event to the corresponding encapsulation ($l.2 - l.3$). If the weight of the event is equal to two ($l.4$), the wrapper requests the checkpoint of the encapsulation and resets the number of processed events N_{evt} ($l.5 - l.6$). The checkpoint request is processed by the encapsulation after processing evt. If the weight of the event is equal to one ($l.7$), N_{evt} is incremented by one ($l.8$). Since the default value of w is one, N_{evt} is also incremented by one if the weight of the event is unknown ($l.7$). In this way, only particular events having weights equal to zero or two may be defined. If the resulting N_{evt} after processing an event is greater than or equal to the maximum total weights of events processed before checkpoint (i.e., N_{ckpt}), then the wrapper request the checkpoint of the encapsulation. N_{evt} is subsequently reset ($l.10 - l.11$). Finally, if the weight of the event is zero, N_{evt} is not incremented ($l.12 - l.13$).

Algorithm 1 Frequency of Checkpoint based on Events Processed

```
1:  Reception of event by wrapper: (encap, evt)        ▷ event evt to be delivered to encap
2:  save(wrp.encap.policy, evt)                        ▷ save evt according to the policy of encap
3:  deliver(encap, evt)                                ▷ delivery of evt to encap
4:  if evt.weight == 2 then
5:      sendCheckpointRequest(encap)                   ▷ request checkpoint if weight is 2
6:      N_evt ← 0                                      ▷ reset N_evt
7:  else if evt.weight == 1 || evt.weight == unknown then
8:      encap.N_evt++                                  ▷ increment N_evt if weight of evt is 1 or is unknown
9:      if encap.N_evt ≥ encap.N_ckpt then
10:         sendCheckpointRequest(encap)               ▷ request checkpoint if N_evt ≥ N_ckpt
11:         N_evt ← 0
12: else if evt.weight == 0 then
13:     continue
```

6.2.3 Optimistic and Pessimistic Logging Strategies

Optimistic and pessimistic logging are the two main strategies adopted for logging. The choice between optimistic or pessimistic logging is done based on the properties of the local storage as discussed in Section 6.2.1. Figure 6.3 depicts the difference between optimistic and pessimistic message logging at reception.

In the case of optimistic logging, upon each reception of the messages m_1, m_2, and m_3, the wrapper intercepts and saves the message on the local storage of the physical node. The local storage location is given by the value of *Storage.Local* which is defined in the description of the state saving policy of the encapsulation (see Figure 6.2). Each message is delivered to the corresponding encapsulation for processing in the same order they are received by the wrapper. Multiple messages are aggregated and sent at once to the stable storage at a frequency based on the number of messages received and processed by the

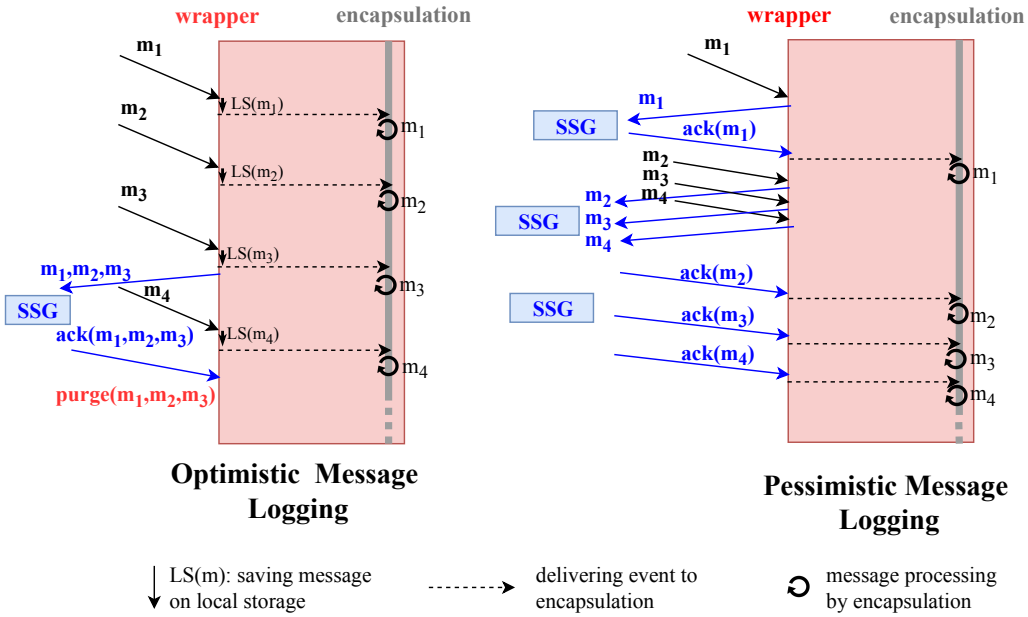


Figure 6.3: Optimistic and Pessimistic Logging by Wrappers

encapsulation. For this example, the frequency is set to three messages. Thus, upon reception of the third message m_3, the three messages m_1, m_2, and m_3 are sent to the stable storage. When the wrapper receives an acknowledgement which indicates that the messages have been stored on the stable storage, the local copies of the messages are purged to avoid saturation of the local storage.

In the pessimistic strategy, upon each reception of a message, the wrapper intercepts and saves the message on the stable storage before delivering it to the encapsulation for processing. In this case, the messages are not aggregated locally before sending to the stable storage because the local storage is unstable. To this end, upon reception of m_1, the wrapper buffers the message, sends it to the stable storage, and waits for an acknowledgement. m_1 is delivered to the encapsulation only after receiving the acknowledgement $ack(m_1)$ which indicates that the messages has been successfully stored on the stable storage. The subsequent messages m_2, m_3, and m_4 are logged in the same way. The wrapper then delivers the messages in the same order that they were received.

6.2.4 Uncoordinated Checkpoint Combined with Logging

Uncoordinated checkpoint is combined with optimistic or pessimistic logging for entities that are characterised by an EBM. The checkpoint of an encapsulation is temporarily stored on the fog node on which the wrapper is deployed before sending it to the stable storage. When a checkpoint is performed, the previous logs and checkpoint become obsolete. This is because the current checkpoint is enough to restore the current state of the encapsulation. They should therefore be purged. When uncoordinated checkpoint is

combined with logging, the logging strategy also determines how the logs and checkpoint preceding the current checkpoint are purged.

Uncoordinated checkpoint combined with the two logging strategies are illustrated in Figure 6.4. The frequency of checkpoint is determined as discussed in Section 6.2.2. Let us consider that each of the messages m_1, m_2 and m_3 has a weight of $w = 1$. The frequency of checkpoint based on the number of events processed is set as follows, $N_{ckpt} = 3$. Thus, a checkpoint should be performed after processing these three messages.

Figure 6.4 (a) depicts the technique for state saving based on uncoordinated checkpoint combined with optimistic message logging at reception. Upon reception of the messages m_1, m_2 and m_3, each message is saved locally and then delivered to the encapsulation. In this case, the saved messages are not sent to the stable storage since the local storage is persistent. Upon performing a checkpoint $ckpt_1$, the local copies of the three messages as well as the previous checkpoint $ckpt_0$ are purged. $ckpt_1$ is stored locally and then a copy is sent to the stable storage. This checkpoint overwrites the previous one (i.e., $ckpt_0$) stored on the stable storage. The local copy of the checkpoint can then be kept or purged depending on its size and the capacity of the local storage.

The combination of uncoordinated checkpoint and pessimistic logging is illustrated in Figure 6.4 (b). In the pessimistic logging strategy messages are not stored locally. Upon performing a checkpoint, the local copy of any previous checkpoint is first replaced by the current checkpoint. The current checkpoint is then immediately sent to the stable storage

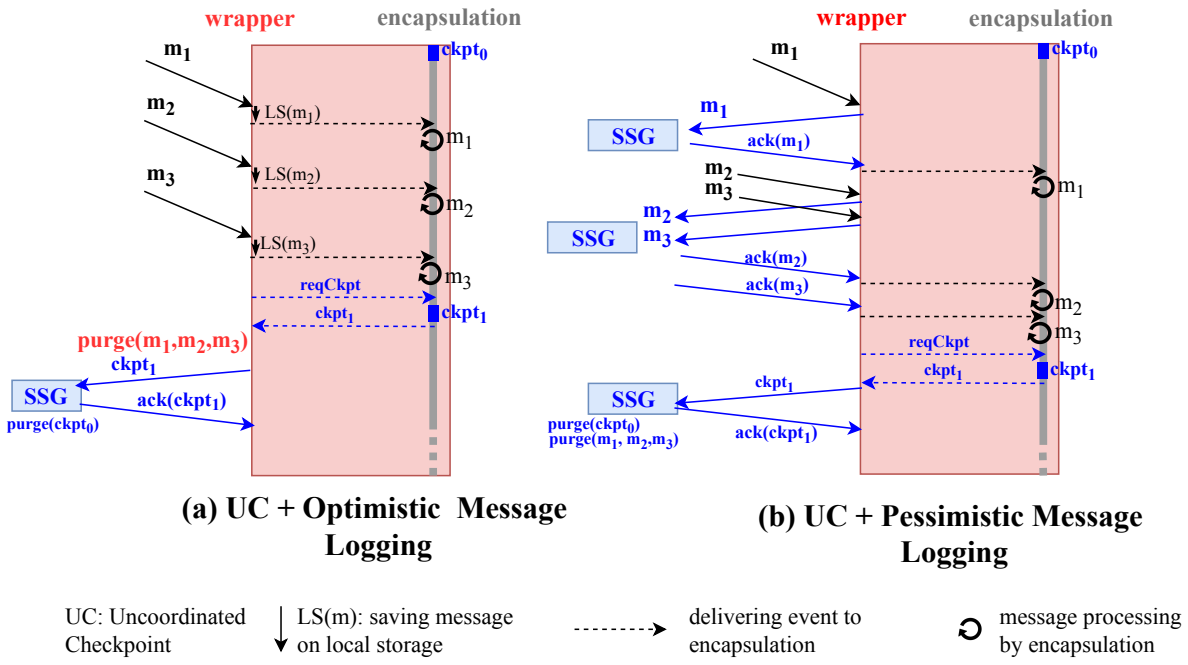


Figure 6.4: Uncoordinated Checkpoint Combined with Logging

because of the unstable nature of the local storage. After reception on the stable storage, the previous checkpoint and all the messages saved on the stable storage can be purged since they are no longer needed to restore the state of the encapsulation. The local copy of the checkpoint can then be kept or purged depending on its size and the capacity of the local storage.

6.2.5 Validity Time

A wrapper also associates contextual information the messages saved. To this end, a wrapper associates a *validity time (vt)* for each message saved as described by the state saving policy (see Figure 6.2). The *vt* indicates whether a message should be taken into account for the state restoration procedure. The expiration of the *vt* implies that the message is no longer valid and should not be used in the computation of the target consistent state to restore the application. For cyber-physical events from appliances, the *vt* indicates the time duration for which the event is valid and should be maintained in the PW. For instance, the event sent to open a connected door lock may have a validity time of one minute. A $vt = 0$ indicates an immediate expiration, that is, the event should never be taken into account for state restoration. A $vt = \infty$ indicates that the event does not expire and should always be taken into account during the state restoration procedure. By default, events are assigned a $vt = \infty$. In this way, only events that have a temporary validity time have to be defined by the DevOps.

6.3 Summary

This section has presented the state saving step of the failure management approach. This step is important as it saves the state of the application so that a consistent state of the application with respect to the PW can be restored during the recovery procedure. State saving policies determine the techniques and strategies for the wrapper to save the state of its encapsulations. The techniques for state saving is based on uncoordinated checkpoint, message logging and function call logging. Logging techniques are used for encapsulations that are characterised by a DBM. It saves the transitions that are executed by the encapsulation. Uncoordinated checkpoint combined with logging techniques are performed for encapsulations that are characterised by an EBM. The frequency of checkpoint is determined according to the time interval since the last checkpoint as well as the number of events processed and the cost of processing these events by the encapsulation. The state saving policy also defines the storage location (i.e, local or stable storage) of the state data as well as the validity time of events.

Chapter 7

Failure Detection and Notification Propagation

Contents

The monitoring step of the failure management approach reports information on failures. Monitoring is very important since it identifies which entity and the type of the entity that has failed. It thus reports information so that failure notification can be propagated and the subsequent recovery procedure can be performed. Therefore, the recovery procedure and the correct behaviour of the application rely on the information reported by the monitoring step. Failure notification aims at propagating notifications to the entities that have dependencies on a failed entity so that they can move to a degraded mode of their lifecycle. For instance, in the case of the failure of a mandatory dependency, the entities affected by the failure should pause their execution. Failure notifications also allows the wrappers to discard messages that are intended for a failed entity. This chapter describes how an application is monitored and how failure notifications are propagated when a failure is detected. To this end, Section 7.1 and Section 7.2 detail, respectively, the monitoring step and the failure notification step of the failure management approach.

7.1 Monitoring

Monitoring is an important step of failure management because it gives information about the lifecycle phases of the infrastructure and applicative entities. More specifically, the information reported by the monitoring step allows the detection of failures as well as the detection of the entities that have recovered from a failure. The failure management approach thus relies on this information to notify dependent entities that a failure has occurred or that a failed entity has recovered. The reconfiguration and recovery actions with respect to the identified failure can then be planned and carried out.

The monitoring step is carried out by the local failure managers: fog agents and wrappers. A wrapper monitors each of its encapsulations. A fog agent monitors the neighbouring physical/fog nodes. In order to ensure their monitoring roles, the wrappers and fog agents implement multiple monitoring techniques based on local system observation, applicative message observation, heartbeats, and ping-acks. Multiple techniques are required because of the heterogeneous nature of the ecosystem. In order to cope with the limited capacities of the Fog infrastructure, a monitoring technique is chosen with respect to the overhead and interference induced by the technique. Thus, a monitoring technique is selected with the objective of having a minimum overhead on the network and avoiding interference with the application.

When a fog agent or a wrapper suspects the failure of an entity that it monitors, it adds the entity in its local list of suspected entities and then notify the failure manager GDM. The notification indicates the identifier of the entity that is suspected of failure and the type of the entity (i.e., software element, appliance, or physical node).

The following subsections successively describe the techniques for monitoring software elements, appliances, and physical nodes.

7.1.1 Software Elements

A wrapper and its encapsulated software element can be deployed on the same fog node or on a remote fog node. The location of the wrapper and software element determines which technique is adopted to monitor the software element. If both wrapper and the software element is local, the monitoring technique chosen is local system observation. If they are remote, the monitoring technique relies on a combination of applicative message observation and ping-ack.

A wrapper monitors its locally encapsulated software element by periodically requesting to the fog node the list of processes running on the local node. The wrapper is thus able to determine if a software element has failed by inspecting this list and mapping it to the corresponding processes of a software element. An important property of this local monitoring technique is that it avoids the wrong suspicion of a software element because there are no uncertainties due to message delays on the network. Local monitoring also avoids influence on the network traffic. Finally, it avoids interfering with the execution of

the application by interrogating software elements on a regular basis.

If the wrapper and its encapsulated software element are located on remote fog nodes, local monitoring is no longer possible. In this case, the wrapper implements a combination of applicative message observation and ping-ack to monitor the remote software element. The combination of these two techniques aims at avoiding, as much as possible, additional network traffic due to monitoring. To this end, the wrapper configures two timeouts: T_{period} and $T_{reponse}$. T_{period} corresponds to the frequency of monitoring. More specifically, it is the frequency at which the wrapper sends ping-ack messages to the encapsulated software element. At each expiration of the timeout, a ping request is sent to the software element. $T_{response}$ is the maximum lapse of time the wrapper waits for a response from the encapsulated software element before suspecting its failure. This monitoring procedure is described in Algorithm 2. The wrapper first starts the timeout T_{period} ($l.1$). When T_{period} expires ($l.2$), the wrapper starts the timeout $T_{response}$ and sends a ping message to the software element ($l.3 - l.4$). If $T_{response}$ expires before receiving an acknowledgement, then the wrapper suspects the failure of the software element. The wrapper adds the software element in its local list of suspected encapsulations and notifies the GDM of the suspicion ($l.5 - l.7$). However, if the software element emits an event that is intercepted by the wrapper before the expiration of T_{period} or $T_{response}$, then T_{period} is restarted and $T_{respose}$ is stopped if it was started ($l.8 - l.10$). This is because the wrapper interprets the emission of an event as the software element being failure-free. Therefore, if an emitted event is intercepted before the expiration of T_{period}, the timeout is reset and the ping message is not sent until the next expiration of T_{period}.

Algorithm 2 Monitoring of Remotely Encapsulated Software Elements

1: *start* T_{period} ▷ first timer is started
2: **if** T_{period} *expires* **then**
3: *start* $T_{response}$ ▷ second timer is started
4: $sendPing(se)$ ▷ ping software element
5: **if** $T_{response}$ *expires* **then** ▷ second timer expires
6: $SuspectedSE.add(se)$
7: $notifyGDM(se, SoftwareElement)$ ▷ a notification is pushed to GDM indicating the id of the failed entity and its type
8: **if** *outgoingEvtIntercepted* **then**
9: *reset* T_{period} ▷ first timer starts again
10: *stop* $T_{response}$ ▷ second timer is stopped if it was started

7.1.2 Appliance

The monitoring technique preferred for appliances is applicative message observation because it limits the influence on the network traffic. No additional messages for monitoring the appliance are emitted by the wrapper. However, this technique can only be chosen if

the appliance communicates at regular interval (e.g., a light intensity sensor that reports values every five seconds). To do so, the wrapper observes the applicative messages reported by the appliance. The wrapper sets a timeout $T_{fd} = T_{msg} + t_{\varepsilon}$ where T_{msg} is the time interval at which messages are emitted by the appliance and t_{ε} is an additional time to account for variability in the network delay. The timer is reset at each message reception. If the timer expires before the interception of a message emitted by the appliance, the wrapper suspects the failure of the appliance.

If the appliance does not communicate at regular interval, the technique based solely on applicative message observation is not possible. In this case, the wrapper uses the same technique as described in Section 7.1.1 for the monitoring of a software element when it is remote to its wrapper

Some appliances have constrained functionalities and do not implement any monitoring APIs. The monitoring of such appliances if they do not communicate at regular interval is complicated since they can not be monitored using ping-acks or heartbeats mechanisms. In this case, the wrapper has to rely on external tools (e.g., wireshark and Zniffer) to observe the data packet exchanges on the network link for monitoring purposes. Such data packets are exchanged regularly between appliances connected on the same network even if applicative messages are not exchanged. The timeout to suspect the failure of the appliance is then based on the frequency of the control packets exchanged on the network link. When such packets are no longer observed from an appliance, the wrapper suspects a failure.

When the wrapper suspects the failure of an encapsulated appliance, it adds the appliance to its local list of failed encapsulations. It then pushes a failure notification to the GDM.

7.1.3 Physical Node and Fog Node

The fog agents monitor their neighbouring physical nodes. Since the lifecycle of a fog node is tied to that of the physical node on which it is hosted, the information reported about physical nodes is also valid for fog nodes. For this purpose, the fog agents implement a heartbeat mechanism to monitor each other.

Each fog agent receives and sends heartbeats towards their neighbouring fog agents. A fog agent associates two timeouts to each neighbouring fog node T_{hbe} and T_{hbr}. The timeout T_{hbe} is used to send heartbeat messages to the neighbouring fog agent at a given frequency so as to indicate to the neighbouring fog agent that the local physical node is failure-free. T_{hbr} is used to monitor the neighbouring physical node by analysing the reception of heartbeat messages. The value of the timeouts are configured according to the network delays between the two physical nodes on which the fog agents are hosted. Algorithm 3 describes the monitoring of a neighbouring physical node by a fog agent. At each expiration of the timeout T_{hbe}, the fog agent emits a heartbeat to the neighbouring fog agent. T_{hbe} is then reset ($l.3-l.5$). At the reception of a heartbeat from a neighbour before the expiration of T_{hbr}, the receiving fog agent resets the timeout ($l.6 - l.7$). If T_{hbr} expires before the

reception of a heartbeat message, the fog agent suspects the failure of the neighbouring fog/physical node. However, if a fog agent fails whereas the fog/physical node is failure-free, the timeout associated to the reception of the heartbeat expires and the neighbouring fog agent will interpret this as a failure of the physical node. In order to avoid this wrong suspicion, when a heartbeat is missed, the neighbouring fog agents implement a ping-ack towards the physical node (i.e., on its IP address or DNS domain name). If the reply is timed out ($l.9$), the fog agent deduces the failure of the fog node. It adds the fog/physical node to its list of suspected entity and sends a failure notification to the GDM ($l.10-l.11$). On the other hand, if a reply is received, the fog agent deduces that the physical node is failure-free. It sends a message to the GDM to request the reinitialisation of a fog agent on the fog node ($l.12-l.13$). This request is forwarded to the ALM which established a remote connection to the fog node in order to reinitialise a new fog agent on the fog node. In this way, the failure of the fog agent is not interpreted as a failure of the physical node.

Algorithm 3 Monitoring of a Physical Node by a Neighbouring Fog Agent

```
1:  start T_hbe                                   ▷ timeout for heartbeat emission
2:  start T_hbr                                   ▷ timeout for heartbeat reception from a neighbour
3:  if T_hbe expires then
4:      sendHeartbeatToNeighbour()
5:      reset T_hbe                               ▷ timeout is reset
6:  if heartbeat is received then
7:      reset T_hbr
8:  if T_hbr expires then                         ▷ timeout for reception of heartbeat from neighbour expires
9:      if ping timed out then
10:         SuspectedPN.add(neighPN)
11:         notifyGDM(neighPN, PhysicalNode)      ▷ a notification is pushed to GDM indicating the id
    and type of the failed entity
12:     else
13:         resquestReinitFGAtoGDM(PhysicalNode)
```

7.1.4 Summary

This section has presented the monitoring step of the failure management approach. Monitoring is achieved through multiple techniques. A monitoring technique is chosen so as to consume the minimum resources and to have a minimum interference with the execution of the application. Table 7.1 recaps the monitoring techniques. Fog agents monitor each other with the heartbeat technique. Local system observation is used to monitor software elements if they are deployed on the same fog node as their wrappers. If a software element and its wrapper is remote, then applicative message observation combined with ping-acks are used. The monitoring of an appliance that communicates at regular interval is done through the observation of the messages it sends. If the appliance does not communicate at regular intervals, then ping-acks is combined with applicative message observation. Ap-

pliances that do not provide a monitoring API is monitored by observing the data packets exchanged on the network link.

Entity	Properties		Entity Monitored by	Monitoring Technique
Physical node	Physical nodes are neighbours		Fog agent	Heartbeats
Software element	Wrapper and software element are on local fog node		Wrapper	Local system observation
	Wrapper and software element are remote		Wrapper	Applicative message observation and ping-acks
	Comm at regular interval	**Monitoring API**		
Appliance	✓	✖	Wrapper	Applicative message observation
	✖	✓	Wrapper	Applicative message observation and ping-acks
	✖	✖	Wrapper by means of external tools	Observation of packets on network layer

Table 7.1: Monitoring Techniques by Wrappers

When the fog agent or wrapper suspects the failure of an entity, it notifies the GDM. Then, failure notifications should be propagated to the entities that have dependencies on the failed entity so that they can move into a degraded mode. The next section describes how failure notifications are propagated.

7.2 Failure Notification Propagation

The GDM receives failure suspicion messages concerning software elements and appliances from the wrappers. It also receives failure suspicion messages concerning physical nodes from fog agents. Upon the reception of failure suspicion messages, the GDM processes these messages and decides whether the entity should be considered as definitely failed. If the failure is confirmed by the GDM, failure notifications are propagated to the wrappers of entities having dependencies on the failed entity. Propagating failure notifications aims at notifying the impacted failure-free entities of the failure. The impacted entities can subsequently move to a degraded mode.

The following sections describe how the failure decision is computed by the GDM and how failure notifications are propagated by wrappers.

7.2.1 Failure Decision

Upon the reception of failure suspicion messages from wrappers and fog agents, the GDM processes these messages and decide whether the entity should be considered as failed. The objective of this decision is to avoid engaging recovery procedure for an entity which is

wrongly suspected of failure. Such procedures only cause unnecessary disruptions of the services provided by the application.

The failure decision is taken according to Algorithm 4. When the GDM receives a failure suspicion message concerning a software element ($l.2$), it de facto considers the software element as failed. This is because when the wrapper and the software element are both deployed on the same fog node, there are no communication uncertainties involved in failure detection. Thus, the GDM can immediately consider the software element as failed. If the encapsulated software element is remote to the wrapper, a wrong suspicion is possible because of the communication uncertainties when implementing failure detection using the ping-ack technique. However, the wrapper cannot distinguish whether it is a wrong suspicion. When this happens, the wrapper has to send a signal to shut down the software element before notifying the GDM. This eliminates the possibility of having multiple instances of the same software element if a recovery procedure is engaged for a wrong suspicion. When the GDM considers a software element as failed, it adds the software element to the global list of failed software element ($l.3$), and then sends a confirmation message to the wrapper of the failed software element ($l.4$).

Algorithm 4 Failure Decision by GDM

1: **Reception of Failure Suspicion:** $(id_f, type)$ ▷ identifier and type of failed entity
2: **if** $type == software\ element$ **then**
3: $FailedSE.add(id_f)$ ▷ add software element to failed list
4: $sendFailureConf(getWrp(id_f))$ ▷ send confirmation to wrapper
5: **else if** $type == physical\ node$ **then**
6: $SuspectedPN.add(id_f)$ ▷ physical node is suspected of failure
7: **if** $getNumSuspicion(id_f) == getNumNeighFga(id_f)$ **then**
8: $SuspectedPN.remove(id_f)$
9: $FailedPN.add(id_f)$ ▷ add to failed list if all neigh fga suspect its failure
10: $sendFailureConf(getNeighFga(id_f))$ ▷ send failure confirmation to all neigh fga
11: **else if** $type == appliance$ **then**
12: $SuspectedAPL.add(id_f)$ ▷ add appliance to suspected list and the associated wrp
13: **if** $getNumSuspicion(id_f) == getNumNeigh(id_f)$ **then**
14: $SuspectedAPL.remove(id_f)$
15: $FailedAPL.add(id_f)$ ▷ add to failed list if all its wrp suspect its failure
16: $sendFailureConf(getWrp(id_f))$ ▷ send failure confirmation to all wrappers of appliance

Fog agents send failure suspicion messages concerning neighbouring physical nodes. Since the lifecycle of a fog node is tied to that of the physical node on which it is hosted, a failed physical node implies that the fog node has failed. In order to avoid wrong suspicions, when a failure suspicion message concerning a neighbouring physical node is received by the GDM, it does not consider the physical node as failed immediately. It first adds the physical node in a list of physical node suspected of failure ($l.5 - l.6$). A wrong suspicion may for example be due to delays on one of the network interfaces of the physical node

(e.g., temporarily saturated bandwidth, interference in the wireless network link). In this case, the heartbeat timeout expires and the neighbouring fog agent suspects the failure of the physical node. Therefore, the GDM inspects the list of suspected physical node before considering the physical node has failed. If it has received failure suspicion messages from all its neighbouring fog agents ($l.7$), the GDM removes the physical node from the suspected list and adds it in a failed list ($l.8 - l.9$). A failure confirmation message is then sent to all the neighbouring fog agents of the failed physical node ($l.10$).

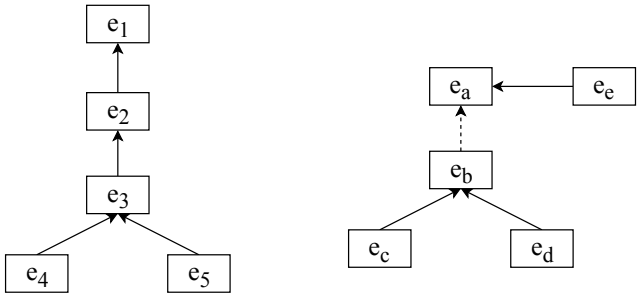
Failure suspicion messages concerning appliances are processed in the same way as those of physical nodes because the wrapper and the encapsulated appliance are remote. Upon reception of a failure suspicion message, the GDM adds the appliance in a suspected list of failed appliances ($l.11 - l.12$). If a failure suspicion message is received from each wrapper that encapsulates the appliance, then the GDM considers the appliance has failed ($l.13 - l.15$), and sends a failure confirmation message to the the wrappers of the failed appliance. It is worth noting that if only one fog agent (respectively wrapper) monitors a physical node (respectively appliance), the GDM confirms the failure after the reception of the first suspicion.

After a failure is confirmed by the GDM, the entities that are impacted by that failure should be notified. The next section describes how these notifications are propagated.

7.2.2 Propagation of Failure Notifications

The propagation of failure notifications aims at notifying the failure to the part of the application impacted by that failure. The impacted entities are those that have a mandatory dependency on the failed entity, as well as, recursively, the entities that have a mandatory dependency on the impacted entities. All the entities impacted by a failure should move to a degraded phase of their lifecycle. The degraded mode of an encapsulation is implemented by its wrapper. It can take different forms depending on the type of dependency on the failed entity such as restricting access to the APIs of the encapsulation, discarding incoming/outgoing operations or transferring them to another failure-free entity, and pausing the behaviour of the encapsulation in the case of a mandatory dependency. Thus, in a degraded mode, an encapsulation provides only part or none of its services. Moreover, failure notifications should be propagated such that depending encapsulations move to a degraded mode by respecting the functional dependencies between the encapsulations. For instance, this means that the encapsulations should move to a degraded mode in a certain order such that a service is not called by a depending entity after the service has been paused. This can make the depending entities unresponsive, generate the re-emission of events that cannot be delivered because of the failure, and create a cascading failure where the whole application fails [118]. Figure 7.1 depicts the software architecture of two applications which will be used in order to illustrate the impact of a failure and how failure notifications should be propagated.

Figure 7.1 (a) depicts an architecture composed of only mandatory dependencies between



(a) Architecture with only mandatory dependencies

(b) Architecture with mandatory and optional dependencies

Figure 7.1: Examples of Architecture with Mandatory and Optional Dependencies

the entities of an application. Let us consider the failure of e_1. By propagation, all the other entities are impacted by that failure and thus their execution should all be paused by propagating failure notifications to them. In order to respect the functional dependencies between the entities, e_4 and e_5 should be the first ones to be paused. This is because they do not have any entities depending on them and thus they can be paused without affecting other entities of the application. They can both be paused at the same time since they do not have any dependencies on each other. e_3 can only be paused if both e_4 and e_5 have been paused. e_2 can then be paused. When e_1 recovers, recovery notifications should also be propagated such that the functional dependencies are respected. The execution of an entity should be resumed only if all its dependencies are satisfied. Therefore, the execution of e_2 and e_3 are first resumed sequentially, and then the execution of e_4 and e_5 can be resumed at the same time. Let us now consider the failure of e_2. The entities impacted by this failure are: e_3, e_4, and e_5. In this case, e_4 and e_5 should be paused before e_3. e_1 is not impacted by the failure and therefore continues its execution. However, e_1 could send messages to e_2. These messages should be intercepted by the wrapper of e_2 because they cannot be delivered. Note that the failure of e_4 and e_5 do not impact any other entities of the application.

Figure 7.1 (b) depicts an architecture composed of both mandatory and optional dependencies between the entities of the application. Consider the failure of e_a. e_b is impacted by this failure. However, since the dependency between e_a and e_b is optional, e_b can still provide its services even if this dependency is not satisfied. Therefore, e_c and e_d are not impacted by the failure of e_a. A failure notification is sent only to the wrapper of e_b. The subsequent messages sent by e_b (i.e., to e_a) is intercepted by its wrapper and those sent by e_e is intercepted by the wrapper of e_a since they cannot be delivered. When e_a recovers, a recovery notification is sent to e_b. The messages sent to e_a can then be delivered.

The failure management approach makes use of failure notifications and paused messages to notify a failure and move the application into a degraded mode. Failure notifications are first propagated to notify the entities that are impacted by a failure. Then, paused messages are propagated so as to respect the order in which the entities impacted by a

failure are paused.

When the failure of a software element or an appliance is confirmed to the wrapper by the GDM, the wrapper propagates failure notifications to the neighbours having a dependency on the failed encapsulation. In the case of a physical node failure, since the wrapper hosted on the corresponding fog node also fails, the GDM has to first retrieve the dependencies of the failed software elements from the ALM in order to notify the neighbouring wrappers. Thus a wrapper wrp_1 receives failure notification messages:

(a) To confirm the failure of one of its encapsulations. This failure notification is sent by the GDM to wrp_1.

(b) To notify the failure of an neighbouring dependency. This happens when the encapsulations of wrp_1 have dependencies on a failed encapsulation of the neighbouring wrapper wrp_2. The failure notification is sent by wrp_2 to wrp_1. This failure notification is sent by the GDM if the neighbouring fog node has failed because in this case wrp_2 has failed too.

(c) To notify the failure of an entity which impacts the functions of an encapsulation of wrp_1. This failure notification is sent by a neighbouring wrapper wrp_2 to wrp_1 because the encapsulation of wrp_2 is impacted by a failure and by propagation an encapsulation of wrp_1 is also affected too.

In the following, we detail how these notifications are propagated and processed by the wrappers so as to respect the functional dependencies of the application.

a . Failure Confirmation from GDM

Failure confirmations are received by wrappers concerning their encapsulated software element and appliances. Upon reception of the failure confirmation from the GDM, the wrapper sends a failure notification to the entities having a dependency on the failed encapsulation. This is illustrated in Figure 7.2:

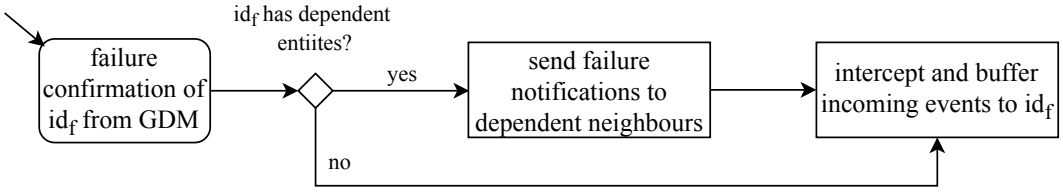


Figure 7.2: Reception of Failure Confirmation from GDM

- If there are no other entities that are dependent on the failed encapsulation, then failure notifications are not sent (e.g., e_4 and e_5 in Figure 7.1 (a)). This is because the failed encapsulation does not impact other failure-free entities. The wrapper intercepts and buffers all the forthcoming incoming events to the failed encapsulation.

- If the failed encapsulation has dependent neighbours (e.g., e_b in Figure 7.1 (b)), a failure notification is sent to each one of them (i.e., to their wrappers). This is done for both mandatory and optional dependencies. The wrapper of the failed encapsulation intercepts and buffers all the forthcoming incoming events to the failed encapsulation. These events are used during the recovery procedure. For instance, in the case of a failed software element, the events can be delivered after the software element has been restarted. In the case of a failed appliance, these events can be forwarded to a replacement appliance. This is discussed in more details in Chapter 8.

A wrapper wrp_1 can thus receive failure notifications from a wrapper wrp_2 if an encapsulation of wrp_1 has a dependency on a failed encapsulation of wrp_2. This is discussed in the following.

b . Failure Notification From GDM or Wrapper of a Failed Encapsulation

A wrapper may receive a failure notification from the GDM[1] or from the wrapper of the failed encapsulation, this implies that one of the encapsulation of the receiving wrapper has a direct dependency on a failed entity id_f. This dependency may be of two types: optional or mandatory. Figure 7.3 illustrates how failure notifications are processed and propagated in each case.

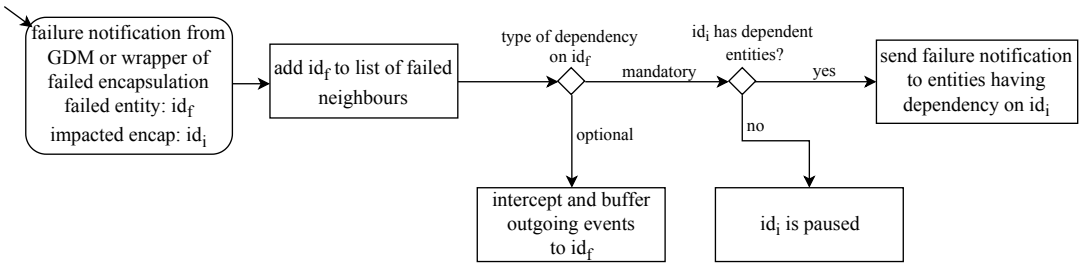


Figure 7.3: Reception of Failure Notification from GDM or Wrapper of Failed Encapsulation

When a wrapper receives a failure notification which impacts one of its encapsulations id_i, it adds the failed entity id_f to its list of failed neighbours. If id_i has an optional dependency on id_f, then the wrapper moves id_i in a degraded mode by intercepting and buffering all the outgoing events emitted to id_f.

If id_i has a mandatory dependency on id_f, then id_i has to be paused. However, id_i and its dependent entities have to be paused in a certain order. id_i can be paused immediately only if it has no other dependent entities. If this is not the case, a failure notification is sent to the entities having a dependency on id_i.

In this case, a wrapper wrp_1 receives a failure notification from another wrapper wrp_2

[1]In case of the failure of a physical node which implies that the wrappers of the failed entities have failed too.

whose encapsulation is impacted by a failure. The encapsulations of wrp_1 are also impacted because they have dependencies on the encapsulations of wrp_2.

c . Failure Notification from Wrapper of Impacted Encapsulation

A wrapper receives failure notification from other wrappers whose encapsulations are impacted by a failure. Figure 7.4 illustrates how such failure notifications are processed.

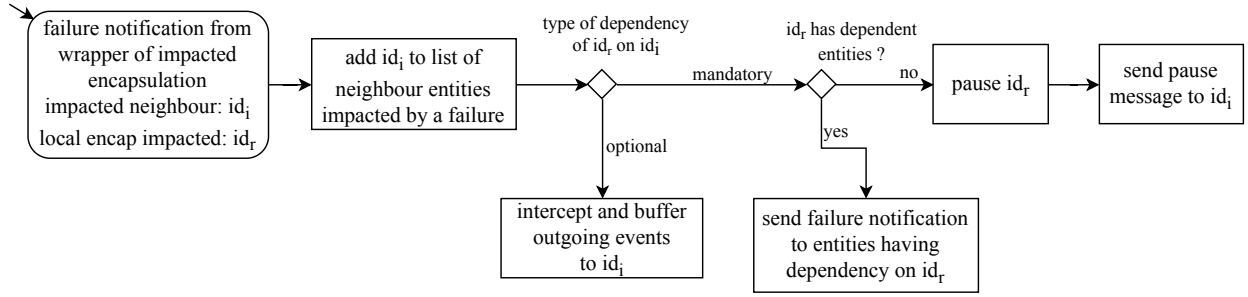


Figure 7.4: Reception of Failure Notification from Wrapper of Impacted Encapsulation

Let us consider a failure notification which is received by the wrapper wrp_r from a neighbouring wrapper wrp_i to indicate the failure of id_f. The encapsulation of wrp_i is impacted by the failure of id_f. The encapsulation id_r of wrp_r is also impacted by the failure because its encapsulations have dependencies on the encapsulations of wrp_i.

At the reception of the failure notification, the wrapper adds id_i in the list of neighbouring entities impacted by the failure of id_f. If id_r has an optional dependency on id_i, then id_r moves to a degraded mode where its wrapper intercepts and buffers the outgoing events to id_i. If id_r has mandatory dependency on id_i, and there exist other dependencies on id_r, then a failure notification is sent to them. Otherwise, if there are no other entities that are dependent on id_r to propagate the failure notification, then id_r moves into a degraded mode where its execution is paused. The wrapper of id_r then sends a pause message to the wrapper of id_i to confirm that id_r has been paused. Upon reception of the pause message, id_i can be paused by its wrapper if all the other entities having a mandatory dependency on id_i are also paused. Thus the wrapper of id_i waits for a pause message from all its neighbours having a mandatory dependency on id_i before pausing its execution.

In this way, the entities that have a mandatory dependency on the failed entity are paused and so are the entities that have a mandatory dependency on the entities that are impacted by the failure. The order in which the entities are paused is respected. Entities having optional dependencies can continue their execution but do not interact with other entities impacted by a failure.

7.2.3 Recovery Notifications

Recovery notifications aims at notifying that a failed entity has recovered. In this way, after recovery, the application can be move into its running phase and be fully operational

again. To this end, when an entity has recovered from a failure, recovery notifications are propagated so that the entities which were impacted by the failure and is currently in a degraded mode can resume their executions. The order in which these entities resume their execution should be respected so that the execution of an entity is not resumed without first resuming its dependencies. Thus recovery notifications and resume messages are used such that the execution of an entity is resumed only if all its mandatory dependencies are satisfied.

When an entity recovers, its wrapper sends recovery notifications to the neighbouring entities (i.e., to their wrappers) that have dependencies on the recovered entity. Upon reception of a recovery notification, the wrapper removes the recovered entity from its list of failed entities and sends this recovery notification to neighbouring wrappers to propagate the notification. In this way all the entities impacted by this recovery are notified. Following a recovery notification, if the receiving wrapper has an encapsulation for which all its mandatory dependencies have recovered and are running, the execution of the encapsulation is resumed. When an encapsulation is resumed, its wrapper sends a resume message to its dependent entities. Upon reception of the resume message, if all other mandatory dependencies of an entity are satisfied (i.e, they are not failed or in a degraded mode), the execution of the encapsulation is also resumed.

7.3 Summary

This chapter has presented the monitoring step of failure management and how failure notifications are propagated when a failure occurs. Monitoring is achieved through multiple techniques. A monitoring technique is chosen so as to consume the minimum resources and to avoid interference with the execution of the application.

Failure notification propagation aims at moving the entities impacted by a failure into a degraded phase. To this end, when a failure is detected, failure notifications are propagated to entities having dependencies on the failed entity. Failure notifications are also propagated to entities that have dependencies on the entities impacted by the failure. Degraded mode is achieved by either intercepting and buffering events, or by pausing the execution of the impacted encapsulations. When the application has recovered from a failure, recovery notifications are propagated so that the application is fully operational again.

Chapter 8

Recovery Approach

Contents

As discussed in the previous chapter, when a failure occurs, failure notifications are propagated to the entities that are impacted by that failure. At the same time, a recovery procedure is planned and performed by the failure managers. The recovery procedure aims at recovering the application from the failure and restoring it in a consistent state, including consistency with respect to the PW. The recovery procedure is done in two successive steps: reconfiguration and state restoration. The type of entity that has failed determines the type of reconfiguration (i.e., architecture, placement) that is implemented. The objective of reconfiguration is to perform operations on the application (e.g., restarting software elements, connecting to new appliances) so as to repair its architecture. Then, the data saved during the state saving phase is used to restore a consistent state of the application, including consistency with respect to the PW (i.e., cyber-physical consistency). The state restored is CP-consistent if the states of the geographical spaces in the PW are not different from their pre-failure states, taking into account expired events and the events generated during the recovery procedure. Moreover, in order to maintain CP-consistency, the state restoration procedure implemented should be safe for the PW, that is, it should not have harmful consequences on the PW.

This chapter presents the reconfiguration and state restoration procedures that are performed in order to recover from infrastructure and applicative failures. Section 8.1 introduces recovery policies which describe the local strategy for recovery. Section 8.2 reports how reconfiguration is performed when a failure occurs. Section 8.3 describes the state restoration procedure so that a CP-consistent state of the application is restored. Section 8.4 details the procedures to ensure that the application has a consistent behaviour after recovery. Finally, Section 8.5 summarises this chapter.

8.1 Recovery Policy

A recovery procedure is performed when the GDM confirms the failure of an entity after receiving failure suspicion messages from wrappers and fog agents. This failure decision and failure notification propagation are described in Chapter 7. The subsequent recovery procedure is composed of two successive steps. The application is first reconfigured. This involves performing operations on the architecture of the application, such as restarting/reinitialising software elements and connecting to new appliances, so as to recover from failures. When reconfiguration is completed, the wrappers retrieve the data saved during the state saving step and restore the application in a CP-consistent state. CP-consistency aims at restoring a consistent state within the application and with respect to the PW.

The recovery procedure makes use of recovery policies which describe the local strategies and actions to perform in order to recover a failed entity. To this end, each wrapper embeds a description of the recovery policy of each of its encapsulations. Figure 8.1 gives an overview of the different parameters of a recovery policy.

```
EncapsulationId: <id>                                   /*id of encapsulation*/

Type: SoftwareElement ⊕ Appliance

SoftwareElement:(restart, <nrt>) | reinitialisation /*encapsulation is a software element*/

Appliance:(reboot, <nrt>) | replacement          /*if encapsulation is an appliance */

StateRestoration:
        BM: DBM ⊕  EBM
        Checkpoint: (local, <path>), (stableStorage, <path>)
        Events: (local, <path>), (stableStorage, <path>)

ExternalAdmin: <path>
```

⊕ : exclusive or | : inclusive or

Figure 8.1: Description of the Recovery Policy of an Encapsulation

EncapsulationId gives the identifier of the software element or appliance encapsulated by

the wrapper. The recovery policy defined in this description file applies for the identified encapsulation.

Type indicates whether the identified encapsulation is a software element or an appliance.

SoftwareElement indicates the approach for the reconfiguration of the software element encapsulated by the wrapper. *restart* specifies that the software element should be restarted on the same fog node after confirmation of the failure by the GDM. The *number of restart trials (nrt)* defines the maximum number of times that the wrapper successively tries to restart the software element on the same fog node. If the number of restart trials exceed *nrt*, then the wrapper notifies the GDM of the impossibility to restart the software element on the same fog node and asks for the *reinitialisation* of the software element on another fog node. Likewise, **Appliance** indicates the approach for reconfiguration of the appliance. The *number of reboot trials (nrt)* defines the maximum number of times that the wrapper successively tries to reboot the appliance. The wrapper then notifies the GDM so that a *replacement* appliance can take over the failed one.

StateRestoration describes the information in order to restore the state of an encapsulation. *BM* indicates the type of behavioural model available for the encapsulation. The state restoration procedure is based on the type of behavioural model available for the encapsulation as discussed later in Section 8.3. The local filesystem path and stable storage to retrieve the state data (checkpoints and events) are specified. This information is extracted from the state saving policy of the encapsulation.

ExternalAdmin defines an external administrator where a notification is pushed when an appliance has failed and cannot be recovered. This is particularly useful to notify service providers so that a new appliance can be automatically shipped or to notify care-givers in smart homes for medicated people.

The recovery procedure of an encapsulation is performed according to its local recovery policy as well as according to the decisions taken by the GDM. The recovery procedure involves the wrappers as well as the global failure managers SSG, ALM, and Thing'in. The SSG is used to retrieve the state data of the application. The ALM and fog agents are involved in the reinitialisation of software elements. Thing'in is used to retrieve information about the available appliances so that a failed appliance can automatically be replaced. The reconfiguration and state restoration techniques performed during the recovery procedure are further discussed in the following sections.

8.2 Reconfiguration

Reconfiguration is the first step of the recovery procedure. Reconfiguration is performed by wrappers and may also involve the global failure managers Thing'in and ALM depending on the type of reconfiguration to be performed. The type of entity that has failed determines the reconfiguration operations that are performed. In the case of a software

element element failure, the reconfiguration first involves the restart on the local fog node. If this is unsuccessful, a placement reconfiguration is performed to reinitialise the software element on a new fog node. A placement reconfiguration of the software elements hosted on a fog node is also performed if the fog node fails. In the case of an appliance failure, the reconfiguration first involves the reboot of the appliance. If this is unsuccessful, an architectural reconfiguration is needed to replace the failed appliance. If there is an absence of replacement appliance, a functional reconfiguration (i.e., degraded mode) is performed.

In the following, the reconfiguration of the application which is performed before restoring the state of the application is described in more details. To this end, the reconfiguration operations upon the failure of software elements and appliances are successively presented.

a . Reconfiguration Upon Software Element Failures

A wrapper performs reconfiguration operations on its encapsulated software element. This procedure, as well as the failure managers involved, is illustrated in Figure 8.2. When the failure of a software element is confirmed by the GDM, its wrapper restarts a new instance of the software element. The wrapper may fail to restart the software element if, for instance, there are not enough resources on the local fog node. A failure to restart the software element may also happen when the wrapper and the software element are remote because the wrapper does not have full administration capabilities on the software element. Therefore, the wrapper tries to restart the software element according to the maximum number of restart trials nrt as defined in the recovery policy of the software element. If the number of trials $tr \geq nrt$, then the wrapper stops trying to restart the software element on the same fog node. It notifies the GDM that the software element cannot be restarted. This implies that a placement reconfiguration is needed. A new placement should be determined for the software element and it has to be subsequently redeployed. To this end, the GDM requests a reinitialisation of the software element to the ALM which determines a new placement for the software element. The ALM, with the help of fog agents, deploys the software element on the new fog node, sets up the new wrapper for the software element, and configures the runtime for its execution. Finally, the wrapper restarts the software element. At this stage, the software element is not yet functional. It is in a degraded mode where the wrapper intercepts any incoming/outgoing operations of the software element. The wrapper restores the state of the software element by retrieving its state data from the local storage of its previous physical node and/or from the SSG. The wrapper then interrogates its neighbours to know if the mandatory dependencies of the software element are satisfied. A neighbouring dependency is satisfied if the neighbouring entity is in its running phase. If this is the case, the behaviour of the software element is resumed. Otherwise, the software element stays in a degraded mode until its mandatory dependencies are satisfied.

The failure of software elements is also provoked by the failure of the physical/fog node on which they are hosted. When a fog node fails, the reconfiguration procedure is more complex since multiple software elements fail at the same time. When this happens, the

GDM requests the reinitialisation of the set of software elements hosted on the failed fog node. The ALM determines a new placement for each of the failed software element. These software elements can be placed and reinitialised on different fog nodes depending on the location of the available resources for the execution of the application. Once the ALM determines the placement of software elements, it makes use of the fog agents which run on the corresponding fog nodes to deploy the software elements, set up their wrappers, and configure their runtime environment. The behaviour of each software element is resumed after its state has been restored and its mandatory dependencies are satisfied. Note that if a wrapper cannot restart the software element on the new fog node, it notifies the GDM so that another placement can be determined.

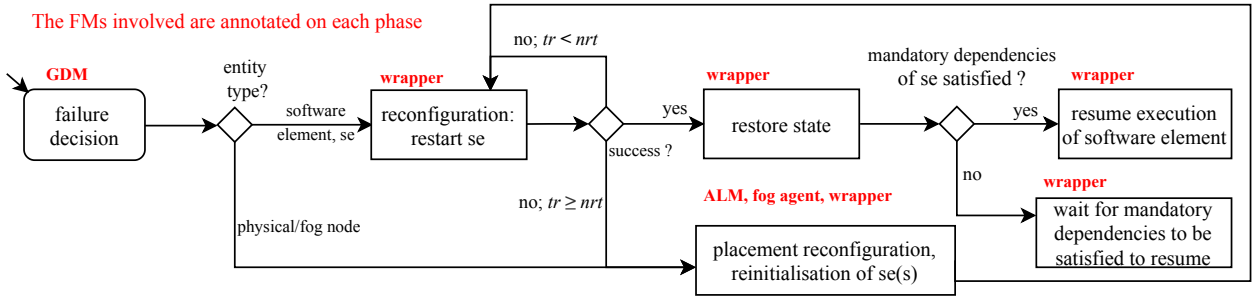


Figure 8.2: Recovery of Software Elements

b . Reconfiguration Upon Appliance Failures

When an appliance fails, if rebooting is unsuccessful, the recovery procedure depends on the availability of a replacement appliance that has similar features and is available on the same geographical space as the failed one. Figure 8.3 depicts the steps to recover from the failure of an appliance as well as the failure managers involved. When an appliance fails, its primary wrapper tries to reboot the appliance according to the maximum number of reboot trials as described in its recovery policy. If the appliance cannot be rebooted, the wrapper notifies the GDM which requests an *equivalent appliance* to the object registry Thing'in. An appliance apl_1 is said to be functionally equivalent to another appliance apl_2 if apl_1 can provide at least a subset of the functionalities of apl_2. This replacement appliance should also be located on the same geographical space as the failed one so that its scope of action covers the geographical space of the failed appliance. This functionally equivalent appliance which is given by Thing'in can be a *temporary* or a *permanent* replacement according to the set of services provided by the replacement appliance compared to the failed one. A permanent replacement appliance is one which offers the full range of services of the failed appliance. It takes over the failed appliance permanently. A temporary replacement appliance is one which offers only a subset of the services of a failed appliance and replaces the failed appliance until a permanent replacement joins the application. For instance, a bedside lamp can be used as a temporary replacement when the primary lamp in a bedroom has failed. When the failed lamp is replaced, the application is reconfigured so that the

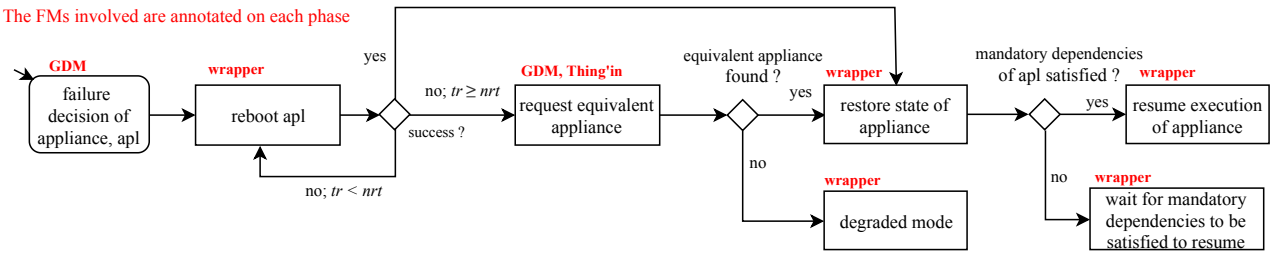


Figure 8.3: Recovery of Appliances

primary lamp takes over the bedside lamp and resumes its nominal behaviour. On the other hand, if two speakers are located in a room and one of them is a backup, then they can be permanent replacements for each other. When reconfiguration is completed, the state data of the failed appliance is used to restore the state of the replacement appliance.

If no alternate temporary or permanent replacement appliance is found, architectural reconfiguration fails and the application has to continue its execution with fewer features in a degraded mode. In this case, the generated events that have to be sent to the failed appliance is intercepted by the wrappers and discarded. Note that, if the appliance is essential to the functions of the application, for instance if all other entities depend on the appliance, the application cannot recover.

In order to restore the state of the application after reconfiguration of software elements and appliances, the respective wrappers retrieve the state data saved during the state saving phase. These data are used to restore the state of the application. The subsequent state restoration procedure performed by wrappers makes use of the behavioural models of their encapsulations in order to guarantee that a CP-consistent state is restored. The next section reports the importance of CP-consistency and how it is achieved.

8.3 Cyber-Physical Consistency

IoT applications are cyber-physical. IoT devices can interact with the PW. This implies that appliances such as actuators can change the state of the PW by performing physical actions. The state of the PW corresponds to what we can perceive around us (e.g., temperature, humidity, light intensity) and is given by sensors and actuators.

During a recovery procedure, restoring the state of an appliance may involve replaying events after a checkpoint. Upon replaying events on an actuator to restore its state, each event processed by the appliance can change the state of the actuator and thus perform actions to change the state of the PW. Even if at the end of the state restoration procedure the application is in a consistent state, consistency with respect to the PW is violated. This is because the replay of the events changes the state of the appliance and thus creates intermediary states before reaching the target state. These intermediary states can create hazardous and costly consequences in the PW. Hence, the state restoration procedure is

inconsistent with respect to the PW. Our approach makes use of the behavioural model of the application and validity times of events in order to compute a target CP-consistent state to restore the application.

The next subsection describes some of the consequences of cyber-physical inconsistency. Then, we explain how the behavioural model of the application is used to restore a CP-consistent state of the application and avoid these undesirable consequences.

8.3.1 Issues in CP-Inconsistency

This section aims at illustrating the issues that arise during the replay of events for state restoration in the Fog-IoT ecosystem because of its cyber-physical properties. In the following, three IoT use cases in different application domains are presented. In each case, the issues and consequences of CP-inconsistency are discussed.

a . Patient Morphine Injection Application

The importance of state saving in a patient morphine injection application was discussed in Chapter 6 and illustrated in Figure 6.1. It showed that if the state of the morphine injection device is not restored after a reboot/replacement of the failed device, then the number of already administered doses of morphine is lost. Therefore, extra doses of morphine can be injected which is dangerous for the health of the patient. A common technique to restore the state of an entity is to replay the events it has processed before its failure. At the end of the replay, the entity reaches its pre-failure state. In the case of the morphine injection device, this means that replaying the messages $m_{injectdose}$ would restore the volatile memory of the device and thus the number of administered doses before the failure of the device would be recovered, as illustrated in Figure 8.4.

At the end of the state restoration procedure, the state of the device has been restored in a consistent state within the application because the value of $dose = 3$ is consistent. However, consistency with respect to the PW has been violated. This is because upon the replay of each message, the device changes state and inject a morphine dose to the

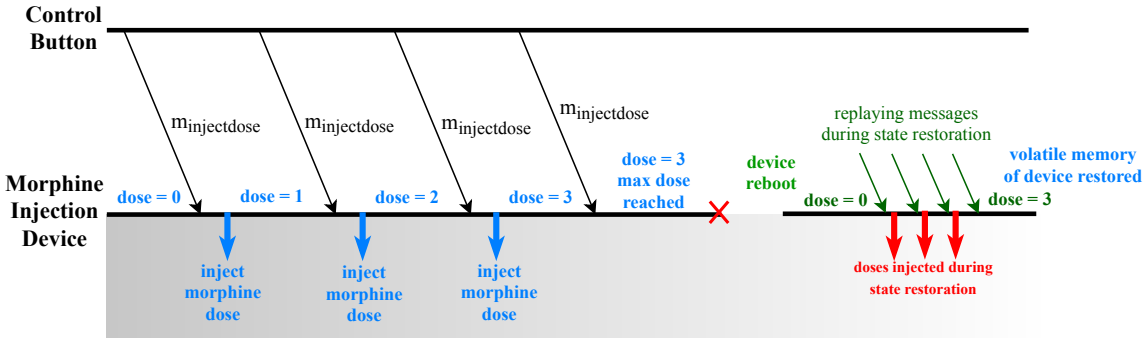


Figure 8.4: An Example of CP-Inconsistency during State Restoration

patient. These changes in state affect the state of the PW and make the PW unsafe to the patient. Thus, CP-consistency is violated because the state restoration procedure has made the PW unsafe.

b . Smart Crop Irrigation

Let us now consider a smart crop irrigation application in which appliances which deliver water and pesticides are deployed on an agricultural field. A server analyses the environmental data to plan and send messages to the appliances for the delivery of water and pesticides to the crops. If the physical server fails, all the data it has collected and stored locally become inaccessible. If the server was checkpointed and the messages sent to the appliances were saved on a stable storage before the failure, a new instance of the server could be deployed. The checkpoint and saved messages can be retrieved to restore the state of the server. However, the messages replayed after a checkpoint will deliver an additional amount of water and pesticides already delivered before the failure. This will result in the damage of the crops which can be very costly for the farmers. Thus, CP-consistency is violated because the consequences of the state restoration procedure have made the PW unsafe (i.e., with respect to the crops) and has caused unnecessary expenses.

c . Smart Home

A recovery procedure in a smart home application that does not account for consistency with respect to the PW may also be hazardous and costly. For instance, in a smart home application for elderly and medicated persons, even the failure of a lamp, which may at first seem unimportant, may be dangerous. The state of the lamps should be restored to avoid disrupting the activities of the house tenants. Moreover, restoring the state of a lamp by replaying the set of saved events will make the lamp blink multiple times. This may annoy the house tenants suffering from epilepsy. Repeated blinking may also provoke the failure of the lamp. Moreover, an inconsistent state with respect to the PW may be restored if the validity of events is not taken into account during the state restoration procedure. The validity of an event gives information about the time duration for which the event is maintained and valid in the PW. For instance, an event to unlock the door of the house may have a validity time of 30s after which the door should be locked for the safety of the house tenants (e.g., to prevent physical intrusion). This means that the event to unlock the door should not be replayed during a state restoration procedure if its validity has expired. Likewise, the temperature reported by a sensor at an instant in time may no longer be valid during a state restoration procedure because the PW has evolved (e.g., day and night).

Therefore, CP-consistency is essential in order to avoid dangerous and costly situations in the PW. CP-Inconsistency arises when:

- The state of the application is not restored upon recovery. In this case, the application has an inconsistent behaviour because it processes events in a different way than expected. This inconsistent behaviour may impact the PW by provoking hazardous

situations (This is discussed in Section 6.1).

- The state of the application transits through a set of intermediary states during the state restoration procedure before reaching the target state. This happens, for example, when events are replayed during the state restoration. In this case, the replay of each event causes a change in the state of the PW which can be unsafe and provoke costly situations.
- The validity time of events are not taken into account upon state restoration. This result in the replay of expired events and thus the state restored is inconsistent with respect to the PW.

In order to avoid CP-inconsistency, the failure management approach takes into account the validity times of events during state restoration. The replay of events is avoided by making use of behavioural models to compute a target consistent state of the application. The application can then be set in this state without replaying events. In this way, CP-inconsistencies are avoided. In the following, the use of behavioural models and validity time of events to guarantee CP-consistency are described.

8.3.2 State Restoration and CP-Consistent Recovery

CP-consistency aims at restoring a consistent state of the application, including consistency with respect to the PW. CP-consistency avoids harmful situations and guarantees that the recovery is safe for the PW. The state restored upon recovery is said to be CP-consistent if the states of the geographical spaces in the PW are not different from their pre-failure states, taking into account expired events and the generated events during the recovery procedure.

Definition 16 *(CP-Consistency) Let us denote $gs(S)$ the state of a geographical space gs where $S = \{(e_1, vt_1), ..., (e_n, vt_n)\}$ (see Definition 4). A state restoration procedure is said to be CP-consistent if $\forall gs \in PW$, $gs(S_{rec}) = (gs(S_{pf}) \backslash gs(S_{exp})) \bigcup gs(S_{afbr})$ where gs is a geographical space in the PW; $gs(S_{rec})$ is the state of gs after recovery; $gs(S_{pf})$ is the pre-failure state of gs; $gs(S_{exp})$ is the expired state of gs which is composed of a set couples (evt_i, vt_i) where the values of vt_i have expired, and $gs(S_{afbr})$ is a set of couples (evt_j, vt_j) representing the set of events that occurred over gs after the failure but before recovery.*

In order to restore the application in a CP-consistent state, the failure management approach makes use of the the validity time of events as well as the behavioural model of the entities of an application. The state saving policy determines the state data that are available for the state restoration procedure. This state restoration procedure is performed by wrappers for their respective encapsulations by retrieving their state data from the local storage and/or the stable storage. In the following, state restoration for entities which are characterised by an EBM and a DBM are successively described.

a . State Restoration based on an EBM

Let us consider the behavioural model of a software element or an appliance which is denoted EBM and a state saving policy of the entity which is based on uncoordinated checkpoint combined with pessimistic message logging. The saved state of the entity is thus composed of a checkpoint s_{ckpt} as well as a sequence of transitions executed during the runtime of the entity with their associated validity time, $E = \{(t_0, vt_0), .., (t_m, vt_m)\}$. The state restoration using the EBM of an entity is illustrated in Figure 8.5. E is first processed by a filter function. This function aims at discarding all the events that have an expired vt at the time that the state restoration procedure is being performed. These events are discarded as they should not be taken into account in the state restoration procedure. The output of the filter function is the remaining sequence of non-expired events. Note that the order of occurrence of the events are kept. Thus, $filter(E) = T_{replay}$ where T_{replay} is the sequence of transitions which has a vt that has not expired. The current state of the entity is thus composed of the couple (s_{ckpt}, T_{replay}).

However, as discussed in Section 8.3.1, replaying the events in T_{replay} on an appliance may cause CP-inconsistencies. In order to avoid violating CP-consistency, for restoring the state of the entity, the wrapper first feeds the checkpoint and then the sequence of transitions T_{replay} as input in the EBM of the entity. The EBM executes the sequence of transitions in T_{replay}, respecting the order in which they were executed before the failure, by starting at the state s_{ckpt}. Thus, the target state s_t is computed as follows,

$$s_{ckpt} \xrightarrow{e_{replay_1}, act, v_e, inv_v} s_1 \xrightarrow{e_{replay_2}, act, v_e, inv_v} \dots \xrightarrow{e_{replay_n}, act, v_e, inv_v} s_t$$

Once s_t is computed, the appliance or software element can be set to this state. In this way, the intermediary states during the state restoration procedure are avoided and CP-consistency is maintained.

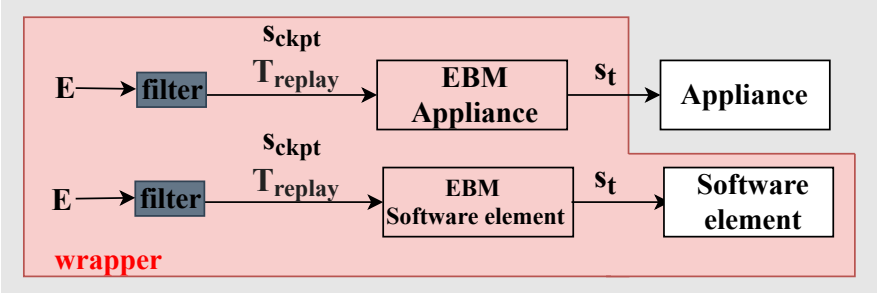


Figure 8.5: State Restoration using EBM

Using the EBM of an entity during the state restoration procedure has the following advantages. In the case of appliances, this procedure maintains CP-consistency by avoiding the harmful impacts of replaying events and executing intermediary states on the PW. Software elements do not interact with the PW and therefore replaying events on software elements does not violate CP-consistency. The events re-emitted by the software element during the replay should however be discarded. The advantage of using an EBM to compute the target state before restoring software elements is to avoid the replay of events that

can take time to be processed by the software element. The re-emission of events by the software element during the state restoration procedure is also avoided. Hence, performing state restoration by making use of the EBM of a software element optimises the procedure.

b . State Restoration based on a DBM

The available behavioural model of a software element may be a DBM. The set of states of a software element characterised by a DBM is unknown. This means that the target state of the software element cannot be computed beforehand from its state data. In this case, a state saving policy based on optimistic or pessimistic message logging is considered. The saved state of the software element is composed of a sequence of transitions executed during the runtime of the entity with their associated validity time, $E = \{(t_0, vt_0), .., (t_m, vt_m)\}$. In order to restore the state of the software element, E is first processed by the filter function to discard the expired events. Thus, $filter(E) = T_{replay}$ where T_{replay} is the sequence of transitions which has a vt that has not expired. The sequence of events are then replayed directly on the software element, in the same order they were received, so that it can recover its pre-failure state. Note that if the validity time of the events are also unknown, all the logged events are replayed on the software element by the wrapper. When these events are replayed on the software element, it transits through a set of intermediary states before reaching the target state. During this process, the software element may re-emit events that were already delivered during its pre-failure runtime. These outgoing events emitted are considered as obsolete and should not be delivered again. Thus, the wrapper intercepts and discards all the outgoing events during the state restoration process.

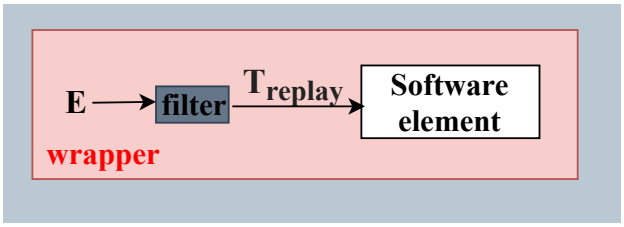


Figure 8.6: State Restoration using DBM

When the application has recovered from a failure and its behaviour is resumed, the wrappers deliver the forthcoming events to the recovered software elements. However, a replacement appliance, albeit functionally equivalent, may use different network protocols and data formats. Thus, the state data saved during the runtime of the failed appliance have to be adapted into a format that can be processed by the replacement appliance. The forthcoming events intended for the failed appliance also have to be adapted and redirected to the replacement appliance if they are encapsulated by different wrappers. The adaptation and redirection of events ensure that the application maintains a consistent behaviour after recovery. This is discussed in more details in the next section.

8.4 Ensuring Service Continuity of Replacement Appliances

The aim of a replacement appliance is to take over a failed one by ensuring the functions and services previously provided by the failed appliance. To this end, the state data of the failed appliance is used to set a CP-consistent state of the replacement appliance. Moreover, after state restoration, the events that are intended to the failed appliance should be redirected to the replacement appliance. Likewise, the events generated by a replacement appliance should be redirected to the corresponding software element that was connected to the failed appliance. This ensures a consistent behaviour of the application after the recovery procedure.

8.4.1 Motivation

A failed appliance and its replacement may be of the same family and be functionally equivalent but, however, have different manufacturers, network protocols, data formats. In this case they expose interfaces that are functionally equivalent but syntactically different.

Let us consider two functionally equivalent actuators (lamps): Awox StriimLight lamp and Philips Hue lamp (see Chapter 10). They can be turned on/off with different intensities and colours. However, the Awox StriimLight lamp connects with a Wi-Fi network and exposes a SOAP API. On the contrary, the Philips Hue lamp connects with a Zigbee network and exposes a REST API. They use different data formats to perform the same operation (e.g., turn on the lamp). This implies that the state data of one of the lamp cannot be used as is to restore the state of the other lamp. Moreover, when the failed Awox StriimLight lamp is replaced by the Philips Hue lamp, the events generated for the Awox StriimLight have to be adapted in format which can be processed by the Philips Hue lamp and then redirected to the Hue lamp.

Let us consider now consider two functionally equivalent sensors (motion): Fibaro Multipurpose sensor and the Wemo Motion sensor (see Chapter 10). They are connected with a Z-wave and Wi-Fi network, respectively. The events they send when a motion is detected have different formats. Thus, if the Wemo Motion is used to replace the Fibaro Multipurpose sensor, then the events generated by the Wemo Motion should be adapted to correspond to the format of the event generated by the Fibaro Multipurpose sensor. This ensures that the event can be processed by the corresponding software element.

To this end, Thing'in provides an interface which allows the adaptation of events and state data for replacement appliances having the same functionalities but having different data formats and APIs. Thus, our failure management approach relies on Thing'in in order to ensure the continuity of the services provided by a replacement appliance. The next subsections detail the mechanisms implemented by the failure management approach to interact with Thing'in for this procedure.

8.4.2 Adaptation of State Data

The data available to restore the state of a replacement appliance may have been produced by a currently failed appliance having different API implementations and data formats. In this case, Thing'in adapts the state data based on the EBMs of the failed and replacement appliances. This procedure is illustrated in Figure 8.7.

The target state to be restored s_t is first computed by the EBM of the failed appliance. The process is carried out by the wrapper of the failed appliance and is similar to the process illustrated in Figure 8.5. The validity time of events are taken into account so that s_t is CP-consistent. Then, the state s_t has to be adapted so that the replacement appliance can be restored. Thing'in computes the corresponding equivalent state s'_t by making use of the EBM of the replacement appliance and the ontological equivalences between the appliances. When the replacement appliance is in the state s'_t, it exhibits the same behaviour as the failed appliance in the state s'_t. Note that the computation of the equivalent state, given by Thing'in, is not in the scope of this Thesis. When s'_t is determined by Thing'in, this state is sent to the wrapper of the replacement appliance. The wrapper sets the state of the replacement appliance to s'_t so that the application is restored in a CP-consistent state.

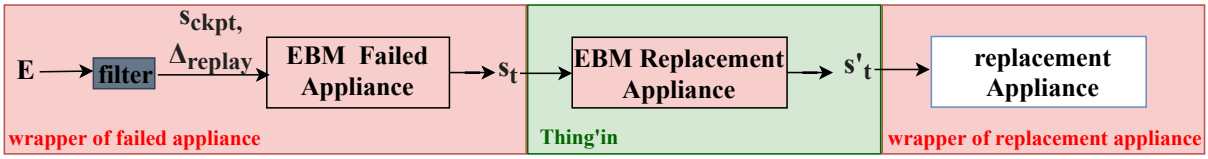


Figure 8.7: Adaptation of State Data for Restoring Replacement Appliance

8.4.3 Adaptation and Redirection of Events

After the state of the appliance has been restored in a CP-consistent state, the incoming events originally intended for the failed appliance should be redirected to the replacement appliance. Likewise, events produced by the replacement appliance should be redirected to the software element that was connected to the failed appliance. This allows a complete take over by the replacement appliances and ensures that the application maintains a consistent behaviour after recovery.

When replacement appliances have different API implementations and data formats, the redirected events to the replacement appliance have to be dynamically adapted before they can be processed by the replacement appliance. Events that are produced by replacement appliances should also be adapted so that they can be processed the destination software element.

Algorithm 5 illustrates the part of the behaviour of the wrapper with respect to the adaptation and redirection of events to replacement appliances.

Several cases are identified depending on the types of APIs implemented by the appliances

Algorithm 5 Delivering Events to Replacement Appliances

1: **Reception of event intended for a failed appliance:** (apl, e_{apl})
2: **if** $apl \notin FailedAPL$ **then**
3: $save(wrp.apl.policy, e_{apl})$
4: $deliver(apl, e_{apl})$
5: **else**
6: **if** $aplRepl \neq null$ **then** ▷ replacement appliance aplRepl exists
7: **if** $aplRepl.wrp == apl.wrp$ **then** ▷ encapsulated by same wrapper as aplRepl
8: **if** $aplRepl.I == apl.I$ **then** ▷ same APIs
9: $save(wrp.aplRepl.policy, e_{apl})$
10: $deliver(aplRepl, e_{apl})$
11: **else** ▷ different APIs
12: $e_{adp} \leftarrow Thingin.adaptEvt(e_{apl}, apl.ebm, aplRepl.ebm)$
13: $save(wrp.aplRepl.policy, e_{adp})$
14: $deliver(aplRepl, e_{adp})$
15: **else** ▷ encapsulated by different wrapper
16: $send(aplRepl.wrp, e_{apl})$ ▷ redirect event to wrapper of aplRepl
17: **else** ▷ event deleted if no replacement apl found
18: $delete(e_{apl})$

and whether they are encapsulated by the same wrapper ($l.7 - l.8$). If the failed and replacement appliances are encapsulated by the same wrapper ($l.7$) and implement the same APIs ($l.8$), then the wrapper saves the event according to the state saving policy and delivers the event to the replacement appliance ($l.9 - l.10$). The event does not have to be adapted. However, if the failed and replacement appliances have different API implementations ($l.11$), then the format of the events have to be adapted before sending it to the replacement appliance. In this case, Thing'in provides a transformation function ($l.12$) for adapting events, which gives for each event intended to the failed appliance, e_{apl}, the corresponding event to actuate the replacement appliance, e_{adp}. The adapted event is computed based on the EBM models of the two appliances and the ontologies defined in Thing'in. The wrapper then sends e_{adp} to the replacement appliance after saving the event according to its policy ($l.13 - l.14$). Otherwise, if the replacement appliance is encapsulated by another wrapper, the event is forwarded to the wrapper that encapsulates the replacement appliance which also handles its state saving ($l.15 - l.16$). If no replacement appliance is found, then the event is discarded ($l.17 - l.18$). The application is in a degraded mode and offers fewer features.

Algorithm 6 illustrates the part of the behaviour of the wrapper with respect to the adaptation and redirection of events generated by replacement appliances.

If both the replacement appliance and the failed appliance is connected to the same software element se, this implies that they are encapsulated by the same wrapper ($l.2$). In this case,

Algorithm 6 Delivering Events Generated by Replacement Appliances

1: **Reception of event generated by a replacement appliance:** $(aplRepl, e_{aplRepl})$
▷ event $e_{aplRepl}$ generated by replacement appliance $aplRepl$
2: **if** $aplRepl.se == aplFailed.se$ **then** ▷ connected to the same software element se
3: **if** $aplRepl.I == aplFailed.I$ **then** ▷ have same APIs
4: $save(wrp.aplRepl.policy, e_{aplRepl})$
5: $deliver(se, e_{aplRepl})$
6: **else** ▷ different APIs
7: $e_{adp} \leftarrow Thingin.adaptEvt(e_{aplRepl}, aplRepl.ebm, aplFailed.ebm)$
8: $save(wrp.aplRepl.policy, e_{adp})$
9: $deliver(se, e_{adp})$
10: **else** ▷ connected to a different software element
11: $send(aplRepl.wrp, e_{apl})$ ▷ redirect event to wrp of software element

if both appliances have the same API implementations, then the event is saved according to its state saving policy and delivered to the local software element se ($l.3-l.5$). However, if they have different API implementations, then the equivalent event e_{adp} is computed by Thing'in based on the EBM of the appliances ($l.6-l.9$) before delivery. If the replacement appliance and the failed appliance are not encapsulated by the same wrapper (i.e., the event is intended for a software element encapsulated by another wrapper), then the wrapper of the replacement appliance sends the event to the wrapper of the destination software element ($l.10-l.11$). The event is saved by the wrapper of the software element.

Note that for a temporary replacement, e_{adp} does not always exist because the replacement appliance offers only a subset of the behaviour of the failed appliance. This is why the application should be reconfigured as soon as a permanent replacement is available.

8.5 Summary

This chapter has presented the final step of the failure management approach. The recovery procedure is based on recovery policies which describe local recovery strategies. A recovery is composed on two successive steps: reconfiguration and state restoration. Reconfiguration aims at performing operations on the architecture of the application. In the case of software element failures, the reconfiguration involves the restart of the software element or its reinitialisation on another fog node according to its recovery policy. In the case of appliance failures, a replacement appliance which has the same features as the failed one takes over the functions of the failed appliance. When reconfiguration is completed, the application is restored in a CP-consistent state. This is performed by computing a target state using the behavioural model of the application, thus avoiding the replay of events on appliances and its impacts on the PW. The events generated by replacement appliances as well as the events redirected to replacement appliances are adapted according to the APIs they

expose. This ensures the continuity of the services provided by the replacement appliance and maintains a consistent behaviour of the application after recovery.

Part III

Validation of the Failure Management Approach

Chapter 9

Model Checking

Contents

The design of the proposed distributed failure management approach is a difficult and an error-prone task. This is mainly due to the highly distributed nature of the Fog-IoT ecosystem which requires a distributed architecture for failure management. Thus, the behaviours of the different failure managers are executed concurrently. This makes the design of the approach complicated and consequently the validation of the correctness of the failure management approach is of prime importance. To this end, formal techniques and tools were used to ensure that the approach respect important correctness properties.

This chapter reports the formal specification of the proposed failure management approach. The corresponding formal model is used to carry out an extensive analysis to ensure that the proposed approach works as expected and respects important correctness properties such as architectural invariants (e.g., *there are no disconnected entities in the application*) and functional properties (e.g., *when a software element fails, it eventually recovers and resumes its behaviour in its pre-failure state*). This verification is performed using model checking techniques [74]. Since the most important part of the approach is the recovery step, the proposed verification is focussed on this step.

9.1 Specification

This section introduces the formal specification of the failure notification and recovery steps of the failure management approach. LNT [52] was chosen as specification language because it is expressive enough and adequate for formally describing the behaviour of our failure management approach. Moreover, it is equipped with CADP [74], a rich toolbox for analysing LNT specifications using model checking techniques.

9.1.1 LNT

LNT is an extension of LOTOS [85], an ISO standardised process algebra, which allows the definition of data types, functions, and processes. Table 9.1 provides an overview of the behavioural fragment of LNT syntax. B stands for a LNT term, A for an action, E for a Boolean expression, x for a variable, T for a type, and P for a process name. The syntax fragment presented in this table contains the termination construct (**stop**) and actions (A). LNT processes are then built using several operators: sequential composition (**;**), conditional statement (**if**), non-deterministic choice (**select**), parallel composition (**par**) where the communication between the involved processes is carried out by rendezvous on a list of synchronised actions, looping behaviours described using process calls or explicit operators (**while**), and assignment (**:=**) where the variable should be defined beforehand (**var**). LNT is formally defined using operational semantics based on Labelled Transition Systems (LTS).

B	::=	**stop**	$A\ (!E, ?x)$	$B_1; B_2$
	|	**if** E **then** B_1 **else** B_2 **end if**	|	**select** B_1 `[]` ... `[]` B_n **end select**
	|	**par** $A_1, ..., A_m$ **in** $B_1 \| \| ... \| \| B_n$ **end par**	|	$P[A_1, ..., A_m](E_1, ..., E_n)$
	|	**while** E **loop** B **end loop**	|	**var** $x{:}T$ **in** $x := E; B$ **end var**

Table 9.1: LNT Syntax (Behaviour Part)

9.1.2 Specification of the Failure Management Approach

The specification for the failure management protocol consists of three parts: *data types* (∼100 lines), *functions* (∼600 lines), and *processes* (> 800 lines). A large part of the specification depends on the input application model (involved applicative entities and their dependencies), and is therefore automatically generated from a program written in Python (∼1,500 lines). For instance, the running application presented in Section 4.6 (consisting of nine software elements / appliances) results in about 2,500 lines of LNT specification that are generated automatically from the Python program.

Data types are used to describe mainly the application model (fog nodes, software elements, appliances, dependencies, behavioural models) as defined in Chapter 4. Figure 9.1 shows some of the datatypes used to define an application. An application consists of a set of Fog Nodes, a set of appliances, and a set of logical bindings. A fog node is defined as a

set of software elements. Each software element consists of an identifier, a set of interfaces (operations) as well as an LTS describing the order in which these operations must be triggered (behavioural model).

```
1  type Application is
2        application ( fgns: FGNSet, apps: ApplianceSet, bindings: BDSet )
3  end type
4
5  -- a set of fog nodes
6  type FGNSet is
7      set of FogNode
8  end type
9
10 type FogNode is
11     fognode ( ident: ID, ses: SESet )
12 end type
13
14 -- a set of software elements
15 type SESet is
16     set of SE
17     with "==", "!="
18 end type
19
20 type SE is
21     se ( ident: ID, interface: IDSet, model: LTS )
22 end type
```

Figure 9.1: Excerpt of the LNT Data Types Defined for Modelling an Application

Functions apply to data expressions and are necessary for several kinds of computations: extracting information from the application such as dependent entities, checking if two appliances have equivalent behavioural models, checking whether the application respects some specific invariants (absence of cycles, no disconnected entity, etc.), computing the target state in case of recovery of an entity, etc.

Processes are used to specify the behaviour of the global and local failure managers. Only the behaviour of the managers which are important for the recovery step are specified, namely the SSG for retrieving state data and the GDM to decide recover step. The behaviour of the wrappers and fog agents on local fog nodes are combined into one process called *Local Agent*. Another process, called *Simulator*, is used to make the application execute functional operations and inject failures to the system. This simulator process is parametrised by the length of the execution (the maximum number of functional operations) and the maximum number of failures to inject during an experiment.

Figure 9.2 shows the four LNT modules used to specify the approach where boxes correspond to LNT modules (generated code with dashed boxes). When a module is inside another one, it means that the external one includes the inner one. The DATATYPES

module defines data types and functions. The APPLI module describes the application model. The GDM module defines the behaviour of the Global Decision Manager, which is independent of the application, so written once and for all. Finally, the MAIN module defines all other processes for the local and global failure managers.

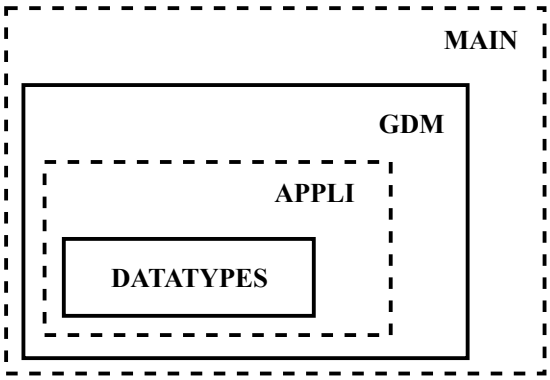


Figure 9.2: Overview of the Specification Structure

For illustration purposes, let us give one example of processes. Figure 9.3 shows an instance of the simulator process. One can see that this process can either make the application evolved (top part). It also shows why a part of the specification depends on the application, and also how different types of failures can be injected in the application. The process keeps track of the number of occurred operations ($l.3$: functional or $l.4$: failure) in order to terminate (correctly) when the simulation has reached the indicated maximum number of execution steps. As illustrated in the same figure, we use actions for modelling functional operations (e.g., $l.7 - l.11$: initiate, on, off) or for identifying the occurrence of specific events in the prococol (e.g., $l.15$: failureSE, $l.17$: failureAppliance, $l.19$: failureFogNode).

Finally, the MAIN process is generated and represents all processes (simulator, global managers, local agents associated to software elements and appliances) executed concurrently as well as the way they interact with each other.

9.2 Properties to Verify

We identified 12 key properties that must be respected by the protocol. These properties can be organised in three different groups: (i) final objective of the approach (prop. 1 below) (ii) architectural invariants (prop. 2, 3, 4 below), and (iii) additional functional properties (prop. 5-12 below). For some of these properties, we also give their formulation in the MCL language [108], the temporal logic used in CADP toolbox. Note that, since some of these properties depend on the functional actions used in the application (prop. 6, 7, 9), they are generated automatically using our Python program at the same time as the LNT specification. Note that we could verify additional properties, but the following 12 properties are the most important ones.

```
1  process simulator [ initiate :any, on:any, off:any, failureSE: any, failureAppliance: any,
       failureFogNode: any, finish: any] (nbFuncOperationsMax: Nat, nbFailureMax: Nat) is
2  var x, y: ID, sender: ID, receiver: ID, nbFuncOperations: Nat, senderTargetState: ID,
       receiverTargetState: ID, nbFailure: Nat in
3    nbFuncOperations := nbFuncOperationsMax ; -- to count the size of the trace
4    nbFailure := 0 ; -- to count the number of effective failures
5    while (nbFuncOperations > 0) loop
6      select -- functional behaviour
7          initiate (?sender of ID, ?receiver of ID, ?senderTargetState of ID, ?receiverTargetState of ID)
8      [ ]
9          on (?sender of ID, ?receiver of ID, ?senderTargetState of ID, ?receiverTargetState of ID)
10     [ ]
11         off (?sender of ID, ?receiver of ID, ?senderTargetState of ID, ?receiverTargetState of ID)
12     [ ] -- failure injector (3 kinds of failures )
13         if (nbFailure < nbFailureMax) then
14           select
15             failureSE (?x of ID, ?y of ID)
16           [ ]
17             failureAppliance (?x of ID, ?y of ID)
18           [ ]
19             failureFogNode (?x of ID)
20           end select;
21           nbFailure := nbFailure + 1 -- updating the number of failures
22         end if
23     end select;
24     nbFuncOperations := nbFuncOperations - 1 -- updating the number of operations
25   end loop;
26   finish -- correct termination
27 end var
28 end process
```

Figure 9.3: Example of Simulator Process

(i) Final Objective:

1. The state of a failed software element (or appliance if there is an equivalent appliance available) after restoration is identical to its pre-failure state.

(ii) Architectural invariants:

2. There are no cycles of dependencies in the architecture of the application.
3. There are no disconnected entities in the application.
4. Appliances have no dependencies on other appliances.

(iii) Functional properties:

5. When a failure occurs, the failed entity eventually recovers. This is true for appliance only if there is an equivalent one available.

```
library actl.mcl end library

[ true* . '{FAILURESE ?se:String ?state:String}' ]
        AU_A_A(true, not '{FAILURESE !se !.*}',
                '{RESTORESECOMPLETED !se !.*}', true)
```

This property is formalised by making use of action CTL patterns [64].

6. When a software element is paused, it eventually starts again. This is not always true for appliances because they cannot always be replaced.
7. When a failure occurs, every mandatory dependent software element and appliance (by propagation) is paused.
8. A software element or appliance cannot execute its functional behaviour when paused or failed.
9. The managers implementing the failure management (GDM, SSG, Local Agents) can always terminate correctly.

   mu X . (< **true** > **true** **and** [**not** FINISH] X)

10. The application is fully operational except when operating in degraded mode.
11. There is no sequence of two failures without a restore in between (illustrated on software elements below).

```
(
   [ true* . ' FAILURESE ?se:String ?state:String ' .
    not (' RESTORESECOMPLETED !se !.* ') .
    ' FAILURESE !se !.* '
   ] false
)
```

12. A sequence exists resulting in the application execution with no failure.

All the properties verified on the running application illustrated in Figure 4.6 consists of about 600 lines of MCL. Half of the code corresponds to property 8, since in that case we generate one property for each possible functional operation for each entity in the application. Property 8 consists of the conjunction of all these atomic properties.

The next section presents the experiments carried out for the evaluation of the failure management approach.

9.3 Evaluation

To verify that our failure management approach satisfies all the twelve properties of interest, we use as input a set of applications. This set of applications are extracted from the running application illustrated in Figure 4.6. The structure of the application as well as the number of fog nodes, software elements, appliances, and bindings are modified. This allows the input of applications with different architectural configurations. For each resulting application, we generate the part of the LNT specification depending on the application, and then we call CADP exploration tools for generating an LTS describing all the possible executions for that application. In this LTS, transitions are labelled with the actions introduced previously in the specification, and we use these actions in the properties to check that the protocol works as expected.

The analysis of the protocol was run on a macOS Mojave machine with a 2.8 GHz Intel Core i7 processor, 16GB of DDR3 RAM and 256GB PCIe-based flash storage. An experiment takes as input an application, the maximum number of functional operations executed by the application, and the maximum number of failures. It is worth noting that since we use enumerative techniques here, there is no need to experiment with large applications, long executions or a high number of failures in order to find issues in the approach. In contrast, most problems are usually detected on small applications and scenarios.

Table 9.2 summarises some of the experiments we carried out during the verification phase. The first column identifies the example. The next four columns characterise the size of the application (number of fog nodes $|FGN|$, of software elements $|SE|$, of appliances $|APL|$ and of dependencies $|DEP|$). The following two columns characterise the scenario in terms of execution length ($|E|$) and number of failures ($|F|$). Finally, we give the size of the raw LTS (number of states $|S|$ and transitions $|T|$), the time in seconds for generating the LTS (Gen.), and the time for verifying all temporal properties (Verif.). Those properties are analysed on a minimised version of the LTS (reduced with respect to strong bisimulation). It takes a few seconds to obtain that minimised version, which is computed using CADP reduction tools. The minimised version is about the half in average in terms of number of states and transitions compared to the raw version.

The row with identifier 4 in Table 9.2 corresponds to the running application illustrated in Figure 4.6. These results show that the larger the application is (in terms of number of entities), the longer it takes to generate the corresponding LTS model (see row with identifier 6 for example). Second, the main factor impacting the size of the LTS is the simulation parameters. Since we rely on enumerative techniques, one more functional action or one more failure generate many more executions since this action/failure may occur at any moment during the application execution and this results in additional interleavings of actions (see rows with identifiers 1 and 2 in the table). The time for model checking all properties is generally much longer than the generation time. More generally, the verification time linearly increases with the size of the LTS whereas the generation time tends to explode when the application size increases. We use rather small applications for verifica-

Ident	Appli. model				Simul.		LTS (raw)		Time (sec.)	
	|FGN|	|SE|	|APL|	|DEP|	|E|	|F|	|S|	|T|	Gen.	Verif.
1a	3	3	2	4	10	1	56,416	193,025	11	179
1b	3	3	2	4	10	2	156,432	517,758	12	535
1c	3	3	2	4	10	3	236,871	713,535	12	783
1d	3	3	2	4	10	4	281,549	756,433	12	1,285
1e	3	3	2	4	10	5	296,946	716,387	12	4,469
2a	3	4	3	7	5	1	3,815	13,045	15	64
2b	3	4	3	7	5	2	16,081	52,582	16	163
2c	3	4	3	7	7	2	52,207	198,247	16	338
2d	3	4	3	7	7	3	105,550	358,924	17	397
2e	3	4	3	7	10	3	489,778	2,096,701	25	1,012
3a	4	5	3	11	5	2	29,779	112,377	102	365
3b	4	5	3	11	10	3	314,349	2,077,252	109	992
4	3	5	4	8	5	2	22,709	58,177	2,403	111
5	5	6	4	11	5	2	33,444	121,254	327	371
6	5	8	4	15	5	2	53,973	212,742	9,392	16,764

Table 9.2: Experimental Results

tion purposes (up to 12 entities for example with identifier 6 in the table) because this is not necessary to use large applications for finding issues, and contrarily, most problems are usually found on small yet pathological applications. Moreover, verification in this case is done once before deployment of the application and long verification time can therefore be afforded.

9.4 Detected Issues

The specification and verification helped to refine our understanding of the finer points of the recovery procedure. This section focuses on three points of interest which were identified or confirmed using model checking.

First, it is worth reminding that, although the failure management approach is always able to work and terminate correctly, this is not the case of the application. Indeed, in case of appliance failure, if there is no functionally equivalent appliance available, the state of the application cannot be repaired and restored. In this case, the application has to keep working in degraded mode. This was confirmed using verification techniques. During our experiments, when there was no additional equivalent appliance for replacing a failed one, the first part of property 10, " *the application is fully operational* " was violated. This is because an appliance has failed and cannot be replaced. In this case, the application is not fully operational since it offers a partial service in a degraded mode.

Another interesting problem comes from the propagation of failure notifications when there is a failure in an application with multiple dependencies. Let us consider an application with four entities and dependencies among them looking like a diamond. For instance e_2 depends on e_1, e_3 depends on e_1 and e_4 depends on both e_2 and e_3. If e_1 fails, by propagation two failure notifications will be received at the local agent of e_4. If those messages are not consumed, this can induce a deadlock (correct termination of the protocol is not possible, property 9) because one of the local agents of e_2 and e_3 is not able to propagate its notification, resulting in an erroneous situation. This can be corrected by either receiving as many notifications as supposed with respect to the failed entity and the structure of the application, or by accepting the first failure notification message and discarding the forthcoming similar ones.

As far as multiple simultaneous failures are concerned, the protocol was originally supposed to support such failures. However, the design of the approach was more complicated and several properties were violated. This is due to the fact than when trying to handle several failures at the same time, some contradictory messages can be exchanged because of delays in asynchronous message exchanges (e.g., one message saying to resume the execution an entity whereas another failure has occurred so a contradictory message asking to pause is also received). As a consequence, in this first version of the protocol, the GDM treats failures one after the other, which is reasonable since the occurrence of multiple failures is scarce and the implementation of the failure management approach on a realistic testbed shows that the time taken to recover from failures is less than one second (see Section 10).

9.5 Summary

This chapter has described the formal specification and verification of our failure management approach, with a focus on the recovery step, using model checking techniques. The verification of the approach is particularly important because it is complex and thus prone to design errors. The aim of verification using model checking techniques was to ensure that the approach works as expected and respects important correctness properties. It was also useful for clarifying several questions about the approach as well as for identifying and correcting some issues in the approach (e.g., failure notification propagation and degraded mode). It is worth noting that all key properties were satisfied for all the examples of our dataset of applications used for verification purposes. In addition, the issues and feedback of the specification and verification of the approach helped in its implementation as a framework called F^{3}ARIoT for IoT DevOps. This framework and its deployment on a smart home testbed is presented in the next chapter.

Chapter 10

F^{3}ARIoT - Implementation and Evaluation

Contents

In the previous chapter, a formal verification of the failure management approach was presented. It validates the correctness of the approach. However, this evaluation cannot reproduce the complexity of the Fog-IoT ecosystem and its runtime. Moreover, it does not give any indication on the time taken to perform the recovery procedures. Hence, this chapter is complementary to the previous one. This chapter reports the implementation of the failure management approach presented in Part II of this Thesis. The approach is implemented as a framework called F^{3}ARoT, intended for IoT DevOps[1]. In order to evaluate F^{3}ARoT, it is deployed on a smart home application. A functional evaluation is performed to confirm the results presented in Chapter 9. A performance evaluation is carried out to analyse whether the time taken to recover from failures is done in an acceptable delay with respect to end users, which according to [47, 114], is **one second** for the user's flow to stay uninterrupted.

In the following, Section 10.1 describes the implementation of F^{3}ARoT and how it is

[1]Development and Operations team

configured for deployment. Section 10.2 describes the smart home application and the deployment of F^{3}ARoT on this application for its evaluation. Sections 10.3 and 10.4 discuss the methodologies and the tools, respectively, for performing this evaluation. Sections 10.5 and 10.6 detail the result of the evaluation of F^{3}ARoT. Finally, Section 10.7 concludes this chapter.

10.1 Implementation of F^{3}ARIoT

F^{3}ARIoT is a framework designed for DevOps of IoT applications. The framework provides the following failure managers as described in Part II of this Thesis: the SSG, the Global Decision Manager (GDM), the fog agents, and the wrappers. The SSG was implemented as a *MongoDB* database. The other failure managers were developed in *Node.js* because it is lightweight (i.e, low memory footprint) and its packet manager, *npm*, handles effectively the management of runtime dependencies. Moreover, the failure managers are completely portable and can run on heterogeneous physical nodes independently of the underlying operating system. This makes its integration simple in the Fog-IoT environment. Note that the Application Lifecycle Manager (ALM) [99] and the Object Registry (Thing'in) [15, 14] are provided by other platforms at Orange Labs.

The failure managers GDM, fog agent, and wrappers each embeds a configuration file based on a *json* format. The GDM is generic and independent of the use case application. Its configuration file indicates how it communicates with the other failure managers. The configuration file of the fog agent specifies the configuration for monitoring neighbouring physical nodes. The configuration file of wrappers defines the relevant information (e.g., state saving policy, recovery policy, BM) about its encapsulations. In the following, we explain in more details these configuration files.

a . Configuration of the GDM

Figure 10.1 shows the structure of the configuration file of the GDM. This configuration file indicates how the GDM communicates with the other failure managers. The GDM sends requests to the ALM ($l.2$) for reinitialisation of software elements and Thing'in for requesting replacement appliances ($l.12$). Two communication methods are supported: MQTT and event-based sockets. *MQTT* [4, 3] is a standard messaging protocol based on a publish/subscribe mechanism ($l.3, l.13$). MQTT is useful because it consumes very low resources and is therefore suitable for IoT applications. *DNSname* gives the domain name of the device on which the MQTT broker is deployed and *MqttPort* gives the logical port to connect in order to publish and subscribe to events. *Socket.IO* [11] ($l.7, l.17$) is an event-based communication library of Node.js. In this case, the *DNSname* gives the domain name of the device on which the ALM/Thing'in is deployed. Note that the domain names can be replaced by an IP address to reach the device.

```
{
  "ALM": {
    "Mqtt": {
      "DNSname": "<broker-dns-name or IP-adr>",
      "MqttPort": "<mqtt-port>"
    },
    "Socket": {
      "DNSname": "<alm-dns-name or IP-adr>",
      "socketPort": "<socket-port>"
    }
  },
  "Thingin": {
    "Mqtt": {
      "DNSname": "<broker-dns-name or IP-adr>",
      "MqttPort": "<mqtt-port>"
    },
    "Socket": {
      "DNSname": "<thingin-dns-name or IP-adr>",
      "socketPort": "<socket-port>"
    }
  }
}
```

Figure 10.1: Configuration File of the GDM

b . Configuration of Fog Agents

The structure of the configuration file of fog agents is illustrated in Figure 10.2. The configuration part for communication with the ALM and GDM is similar to that presented in Figure 10.1. Thus, we show only the configuration part for monitoring of neighbouring physical nodes. Figure 10.2 shows the configuration file of the fog agent *rpi-x.fga*. The configuration file lists two sets fog agents. The set *PhysicalNodeMonitoredBy* gives the list of fog agents that monitors the physical node on which *rpi-x.fga* is deployed ($l.2$). Thus, *rpi-x.fga* sends heartbeat messages to these fog agents. The frequency of emission of the heartbeat is based on the average network latency between the two fog agents. The default value of *hb-period-emission* is calculated as follows, $period = 1.5 * (RTT/2)$ where RTT is the network round-trip time between the two fog agents. The period is 1.5 times higher than time to send a heartbeat message to account for network variations. This value can be adjusted by the IoT DevOps so that the detection of a failure of a physical node is more/less reactive with respect to their use case applications. For example, $l.12 - l.15$ shows the configuration to send heartbeats to the fog agent *rpi-0001.fga* on port 3001. The period of emission of the heartbeat is 50ms. In this case, *rpi-x.fga* sends a heartbeat message to *rpi-0001.fga* every 50ms. The set *NeighPhysicalNodeToMonitor* gives the list of the neighbouring physical node that *rpi-x.fga* should monitor. Thus, *rpi-x.fga* receives heartbeat messages from these neighbouring fog agents. For example, $l.25 - l.28$ shows the configuration for monitoring a neighbouring physical node. The fog agent *rpi-x.fga* receives hearbeat messages from the neighbouring fog agent deployed on the physical node *rpi-0002* at a period of 60ms. Therefore, if a heartbeat message is not received in the 60ms time interval, *rpi-x.fga* suspects the failure of *rpi-0002*.

```
{
  "fog-agent-id": "rpi-x.fga",
  "physical-node-id": "rpi-x",
  "PhysicalNodeMonitoredBy": [
    {
      "id": "<fog-agent-id>",
      "DNSname": "<fga-physical-node-dns-name-or-IP>",
      "hb-port": "<heartbeat-port-for-monitoring>",
      "hb-period-emission": "<period-of-sending-heartbeat>"
    },
    {
      "id": "rpi-0001.fga",
      "DNSname": "pn.far-iot.rpi-0001",
      "hb-port": "3001",
      "hb-period-emission": "50"
    }
  ],
  "NeighPhysicalNodeToMonitor": [
    {
      "id": "<physical-node-id>",
      "hb-port": "<heartbeat-port-for-monitoring>",
      "hb-period-reception": "<period-of-reception-heartbeat>"
    },
    {
      "id": "rpi-0002",
      "DNSname": "pn.far-iot.rpi-0002",
      "hb-port": "3002",
      "hb-period-reception": "60"
    }
  ]
}
```

Figure 10.2: Configuration File of a Fog Agent

c . Configuration of Wrappers

The configuration file of a wrapper is composed of four components as illustrated in Figure 10.3. *Architecture* defines the local and neighbouring architecture. More specifically, it defines the entities that the wrapper encapsulates and the dependencies of these encapsulations. *Monitoring* defines the monitoring technique for each encapsulation. *Policies* defines the state saving policy and the recovery policy for each encapsulation. *BM* defines the behavioural model of the encapsulations. The monitoring technique as well as the policies for state saving and recovery are automatically assigned based on the properties of the encapsulations as discussed in Part II of this Thesis. The wrapper generates a configuration file for each encapsulation which details these four components.

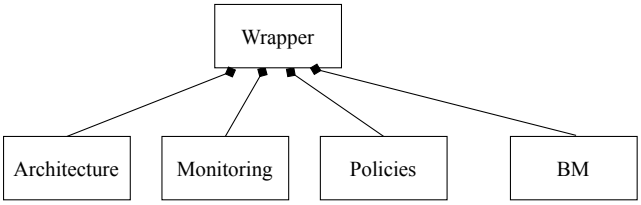


Figure 10.3: Components of Configuration Files of Wrappers

```
{
  "encapsulationId": "se-0001",
  "typeEntity": "software element",
  "Architecture": {
    "fog-node": "fgn-0001",
    "EncapsulationDependsOn": [
      {
        " identifier ": "apl-0002",
        "typeEntity": "appliance",
        "dependency": "optional"
      }
    ],
    "DependsOnEncapsulation": [ ]
  },
  "monitoring": {
    "technique": "local system observation",
    "period": "10"
  },
  "BM": {
    "model": "EBM",
    "path": "./BM-se-0001"
  },
  "Policies": {
    "LocalData": [
      {
        "exec-process-se-0001": "~/appli/se-0001.js"
      }
    ],
    "StateSavingApproach": {
      "Technique": "Checkpoint, MsgLog",
      "ER": "Emission, Reception",
      "OP": "Pessimistic"
    },
    "CkptParams": {
      "Weight": [
        {
          "updateVersion": "2"
        }
      ],
      "Freq": {
        "Tckpt": "12h",
        "Nckpt": "10"
      }
    },
    "Storage": {
      "local": "~/appli/localStorage",
      "StableStorage": {
        "comm": "Mqtt",
        "DNSname": "ssg.far-iot.mongoDB",
        "port": "1883"
      }
    },
    "nrt": "3",
    "ExternalAdmin": {
      "comm": "REST",
      "API": "/failureNotification/permanent/fognode/fgn-0001"
    }
  }
}
```

Figure 10.4: Configuration File of a Wrapper

Figure 10.4 depicts an example of part of the configuration file generated by the wrapper for an encapsulation. The encapsulation is a software element and its identifier is *se-0001* ($l.2 - l.3$). It is hosted on the fog node *fgn-0001*. The software element has an optional dependency on the appliance *apl-0002* ($l.4 - l.10$). There are no entities that has dependencies on the *se-0001* ($l.13$). The monitoring of the software element is based on local system observation with a period of 10ms ($l.15 - l.17$). This is the default monitoring technique for software elements. The behavioural model of the software element is an EBM and is described in the file *BM-se-0001* ($l.19 - l.21$). The state saving and recovery policies of *se-0001* is described as from $l.23$. *LocalData* gives the resources to execute the sotfware element ($l.24 - l.28$). In this case, it is composed of a *Node.js* process. The state saving approach is based on checkpoint and pessimistic message logging at emission and reception ($l.29 - l.32$). This state saving technique is assigned based on the properties discussed in Chapter 6. The event *updateVersion* has a *weight = 2*. This implies that a checkpoint is performed immediately after the reception of this event. By default, all other events received/emitted by the software element have a *weight* = 1. A checkpoint is performed after processing ten events. The maximum time interval between two checkpoints is 12h ($l.34 - l.42$). *Storage* ($l.45 - l.50$) specifies the access to the local and stable storage. By default, the number of reboot trials (*nrt*) of software elements is 3 ($l.53$). *ExternalAdmin* defines a REST API to push a notification in case the fog node *fgn-0001* has failed and cannot be recovered.

10.2 Experimental Environment

The target experimental environment for evaluating F^{3}ARIoT is a smart home application inspired from an industrial project at Orange Labs [99]. The testbed is composed of infrastructure and applicative entities that can be found in real-life smart homes. For instance, the *Connected Home Service* [7, 8] by Orange include the appliances involved in this testbed.

10.2.1 Smart Home Testbed

Figure 10.5 depicts the smart home testbed with the location of the appliances in the home as well as the placement of the software elements hosted on fog nodes.

The physical nodes of the testbed are PC1, rpi1, rpi2, and rpi3 which are respectively a PC (x86_64, 4GB RAM, Windows 7), and three Raspberry Pi Model 3 Type B (64-bit, 1.2Ghz, quad-core ARM Cortex-A53 processor, 1 GB RAM, 16GB microSD storage, Raspbian GNU/Linux 8.0 jessie). These devices are representative of the physical resources and capacities that are available at the edge of the network and more specifically in a smart home.

The appliances are as follows: two Philips Hue lamps, a Hue Go lamp (bedside lamp), a Hue Tap (a set of four connected push-release buttons), a Fibaro Multipurpose Sensor

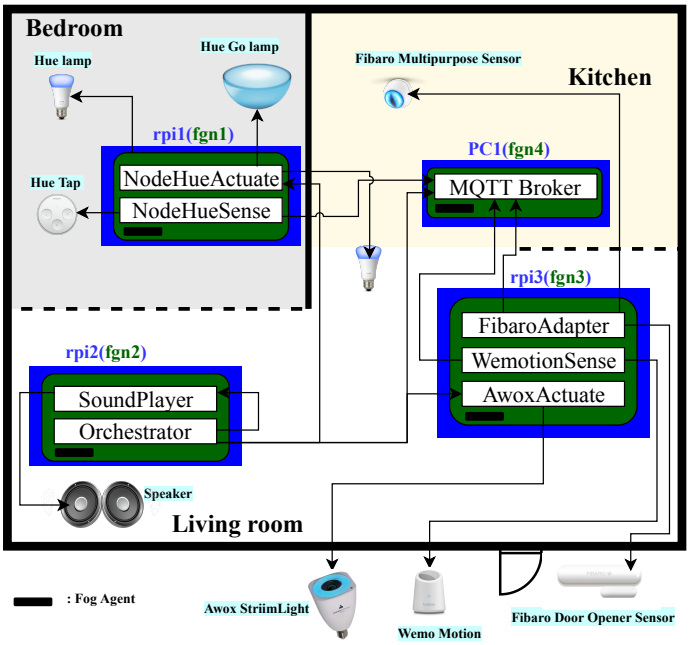


Figure 10.5: Smart Home Testbed

(motion, light, temperature and vibration sensors), an analog wired Speaker, an Awox StriimLight lamp (lamp with integrated speaker), a Wemo Motion Sensor, and a Fibaro Door Opener Sensor. The Philips Hue devices use the wireless protocol Zigbee. The Fibaro devices uses Z-wave protocol. Awox StriimLight and Wemo Motion are connected through Wi-Fi.

The application consists of four fog nodes *fgn1, fgn2, fgn3* and *fgn4* which are hosted on the physical nodes. The software elements running on these fog nodes are:

- *MQTT Broker*: a Message Oriented Middleware (MOM) based on a publish-subscribe communication pattern. It is an implementation of a MQTT broker based on Mosquitto [3].
- *Orchestrator*: it subscribes to all the events published on the MQTT Broker. It defines the corresponding scenarios and user stories (set of actions) that should be triggered based on the patterns of events reported by sensors. It sends messages to other software elements according to the scenarios defined.
- *NodeHueSense*: it retrieves the button events from the Hue Tap Buttons device and publishes them onto the MQTT Broker. The event published contains the button number pressed and the local time it was pressed.
- *FibaroAdapter*: it retrieves events sensed by the Fibaro Multipurpose Sensor and Door Opener Sensor and publishes them on the MQTT Broker. It also configures the frequency of reported events from the devices. For instance, an event is sent each

time the door is opened and closed.

- *WemotionSense*: it reports motion events sensed by the Wemo Motion device and published them onto the MQTT Broker. A motion event is sent when motion is sensed and when the motion subsequently stops.
- *NodeHueActuate*: it accepts messages for the control of the Hue lamps (e.g., turning on/off, changing colour/intensity). The lamps are controlled via the REST API they expose.
- *AwoxActuate*: it accepts messages for the control of the Awox StriimLight lamp and its integrated speaker. The lamp is controlled via its SOAP API.
- *SoundPlayer*: it accepts messages for the actuation of the Speaker. It is based on the open-source audio player *mpg123*.

The application was developed in *Node.js* and *Go* as they consume low resources. They are therefore suitable for running on Raspberry Pis.

10.2.2 Use Case Application

The smart home use case application is light automation and physical intrusion detection. This type of smart home application is increasingly popular because it aims at the comfort and convenience of the house tenants. To this end, a set of user stories corresponding to this use case is identified and enumerated below. These user stories are defined at the software element *Orchestrator* which recognises patterns of events and triggers the corresponding events for actuation of the IoT devices.

a . User Story: Indoor Light Automation

The Hue Tap buttons are used to control the Hue lamps. The device has four buttons *b1, b2, b3* and *b4*. The software element *NodeHueSense* retrieves the buttons pressed as well as the number of times they are pressed. In this way, the bedroom lamp can be turned on with different intensity. For instance, pressing the button *b3* turns on both the bedroom lamp and kitchen lamp at a very low intensity. This is particularly useful to prevent itching of the eyes when the house tenants wakes up at night to go to the bathroom.

b . User Story: Welcome Home

In the second user story, the house tenant comes back home after dark. The lamp in the porch (i.e., Awox StriimLight) is turned on when motion is detected by the Wemo Motion Sensor. The lamp is kept on for 30 seconds, the time for the person to unlock the front door. It is then automatically turned off. Upon entering the house, the Hue lamp in the living room is turned on and a welcome music is played on the Speaker.

c . User story: Bedtime

In the bedtime user story, the house tenant indicates that he is going to bed by pressing the button *b4* of the Hue Tap device. In doing so, all the lamps of the house are turned off and an alarm is set to notify physical intrusion into the house. If any motion is detected inside the house by the Fibaro Multipurpose Sensor or if the door is opened (detected by the Fibaro Door Opener Sensor), then an alarm is triggered on the Speaker and all the lamps of the house are turned on in a red colour. It is worth noting that the alarm is automatically unset when the house tenant presses any other button upon waking up at night to prevent the activation of the alarm by himself.

These user stories were chosen so as to illustrate the different types of state restoration and recovery that F^{3}ARIoT implements. For instance, the first user story is useful because it allows the software elements and the lamps to move to different states. This allows the design of experiments that can be repeated which involves the same software elements and appliances but with different states to be restored for each experiment. In this way, any bias can be eliminated when evaluating consistent state restoration since the experiment involving the same entities is repeated with different inputs and outputs. The user stories b and c also illustrate the validity of events that should be taken into account for computing the target state to be restored in order to achieve CP-consistency.

10.2.3 Deployment of F^{3}ARIoT

In order to carry out the evaluation of F^{3}ARIoT on this smart home application, a fog agent was deployed on each fog node of the application. Eight wrappers were deployed. Each wrappers encapsulates a software element and the neighbouring appliances. Table 10.1 summarises the wrappers and their encapsulated software element and appliances. For instance the wrapper *wrp1* encapsulates the software element NodeHueActuate as well as the both Hue lamps and the Hue Go lamp. Each wrapper embeds a DBM of its encapsulated software element and an EBM of its encapsulated appliances.

Wrapper	**Encapsulated Software Element**	**Encapsulated Appliances**
wrp1	NodeHueActuate	Hue Lamps Hue Go lamp
wrp2	NodeHueSense	Hue Buttons
wrp3	SoundPlayer	Speaker
wrp4	Orchestrator	–
wrp5	FibaroAdapter	Fibaro Door Opener Sensor Fibaro Multipurpose Sensor
wrp6	WemotionSense	Wemo Motion
wrp7	AwoxActuate	Awox StriimLight
wrp8	MQTT Broker	–

Table 10.1: Wrappers and their Encapsulations

For the purpose of the evaluation, the global failure managers GDM, SSG, ALM, and

Thing'in are deployed on a dedicated high capability computer *HCC* (x86_64, Intel core i7, 4 cores, 2.90Ghz, 16GB RAM and 250GB storage). This computer is considered to be reliable and part of the neighbourhood Telco's infrastructure. A simplified version of the ALM and Thing'in are used in this evaluation.

10.3 Evaluation Methodology

This section explores the methodology for carrying out experiments on a Fog-IoT testbed and how this methodology is applied in the evaluation of F[3]ARIoT on the smart home testbed presented in Figure 10.5. Such an evaluation is complex because of the issues and challenges related to: a) constrained capabilities, b) heterogeneity, c) cyber-physical interactions, d) failure injections, and e) the complexity of reproducing some scenarios. These challenges and how they are tackled are highlighted in this section.

a . Constrained Capabilities

Devices in the Fog-IoT environment, like Arduinos and Raspberry Pi, are usually constrained in terms of functions, processing, storage and communication capabilities. Therefore, the evaluation techniques for diagnosis, auditing and measurements logging should be chosen so as to have a minimal impact on the application. The evaluation tools, for instance, should consume negligible physical resources (CPU, bandwidth, storage) that should rather be dedicated to the application. Therefore, storage of measurement data should be externalised to a dedicated storage infrastructure. Moreover, to avoid consumption of applicative bandwidth and computing resources, auditing of logs is done a posteriori to experiments rather than during the experiments.

b . Heterogeneity

Heterogeneity should be taken into account when mounting a Fog-IoT testbed so that it exhibits the properties of a real deployment. If this is not the case, the evaluation would lead to conclusions that cannot be extrapolated to real world application. Heterogeneity was taken into account in the proposed testbed. The network of devices vary from Wi-Fi, Zigbee and Z-wave. For instance, the Hue lamps communicate with a Zigbee network and implement a REST API whereas the Awox StriimLight lamp has a Wi-Fi network and implements a SOAP API. Multiple technologies are involved like MQTT and message buffers for communication, as well as *Node.js* and *Go* for programming languages. The capacities and physical architecture of the physical nodes also differ (CPU, RAM, physical architecture, OS).

c . PW Isolation and Event Injection

When carrying out experiments on cyber-physical IoT applications, there may be unexpected or unwanted interference from the PW. A change of state in the PW captured by devices will trigger a change in the state of the application. For example, during an experimental scenario where the Wemo Motion Sensor and Awox StriimLight lamp are involved, an accidental motion detected before the start of the experiment may lead to an unexpected state saved and subsequently an unexpected state restored. This may lead to an erroneous evaluation. In order to prevent PW interference, sensor devices should be either physically isolated, emulated or put in a controlled PW environment.

Physical isolation consists in creating a barrier around an appliance such that changes in the PW are not sensed and/or changes in the state of the device do not change the state of the PW. In some cases, physical isolation may be simple to implement and enough to prevent interferences. For instance, the Wemo Motion Sensor can be physically isolated using an opaque box (e.g., carton box). However, inevitable changes in the PW, like changes in temperature or humidity, will be sensed by the Fibaro Multipurpose Sensor. In this case, the state of the surrounding PW environment should be controlled and maintained constant by making use of devices like thermostats. In other cases, it becomes harder and costly to isolate these interferences. For instance, physical isolation of the Speaker would require an anechoic chamber which is an unrealistic solution. A more convenient solution is to replace a sensor that cannot (or is costly to) be physically isolated with emulation. An emulated module is a program that is designed based on the EBM behavioural model of the appliance. The program reproduces the behaviour of the real appliance. Emulation provides an easy and cheap way of isolating a device without decreasing the features of the application.

Isolating an appliance from the PW implies that the events previously generated by the appliance have to be injected. For instance, if a temperature sensor is replaced with an emulated module, changes in temperature of the PW will not be sensed. Two techniques for injecting events are adopted: (i) *Physical Event Injection* where the change of state of an isolated sensor is forced by an electronic/mechanical actuator. For instance, a motion can be injected on the Wemo Motion Sensor (and a press on the Hue Tap buttons) by automating an electronic circuit like Servo [10] or a solenoid-based motor by making use of an Arduino; (ii) *Virtual Event Injection* consists in using a program to inject an event which the application sees as coming from the isolated appliance. This type of injection is most appropriate for the emulated appliance since virtual injection is not always possible on a real device. For example, a virtual event cannot be injected on the Fibaro Door Sensor. It has to be physically brought apart to generate the corresponding door opened event. Virtual event injection can also be used to inject events on software elements. These events should, however, be injected at the producer of that particular event. Injecting directly at the consumer may create inconsistencies in the global state of the application. For example, injecting an event directly on the Speaker to play music may result in its state being inconsistent with the state of the software elements SoundPlayer and Orchestrator.

Figure 10.6 recaps the procedure for deciding the type of isolation and event injection.

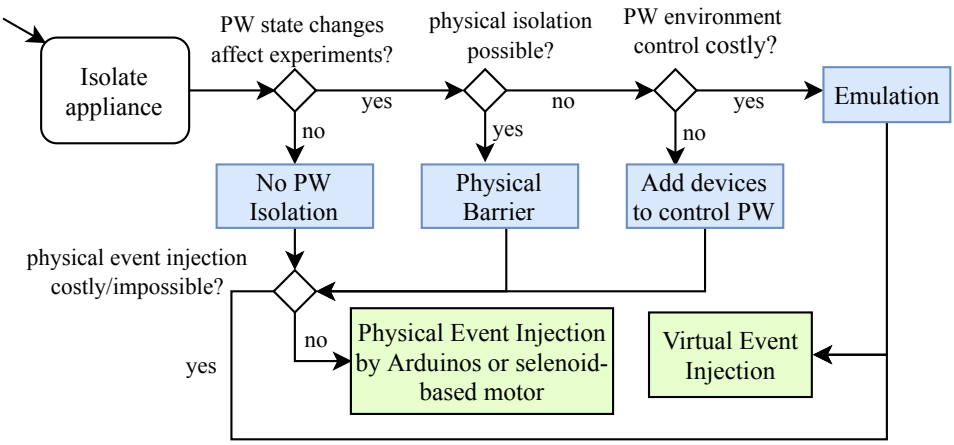


Figure 10.6: PW Isolation and Event Injection

d . Failure Injection

Failures on software elements and emulated modules can be injected by shutting down their corresponding processes or stopping their services through system calls. It can be automated and repeated any number of times without consequences on the software element (software resources like configuration files and databases should be reinitialised) or on the underlying infrastructure. Failures on infrastructure entities (appliances and physical nodes) can be provoked by cutting the power source of the device using a smart plug for devices that use the main power supply from wall sockets or using an external power source modulator circuit for devices powered by a battery. However, repeated failure injections on these devices for evaluation purposes can be harmful and end up in a definite failure. Cutting the power source of a Raspberry Pi or the Hue lamps repeatedly will ultimately cause their break down. In such cases, the failure of the device can be *simulated* such that its failure can be inferred. For instance, a Faraday cage can be used to prevent communication of a wireless appliance. The network interface of a physical node can be shut down whereby the failure detection mechanism will deduce its failure. This technique is used as failure injection on Raspberry Pis. Finally, failure notifications can also be injected to simulate failures. Figure 10.7 recaps the decision for the type of failure injection. Randomisation of failure injections (i.e., randomly choosing entities to inject failures) is also important in order to eliminate bias. The implementation of randomisation and replication of failure injections for our evaluations are described in Section 10.4.

e . Reproducing Miscellaneous Conditions

Some experimental conditions cannot be repeatedly reproduced. This means that repeating an experiment with pre-defined conditions to evaluate the behaviour of the approach may be impossible. For instance, once the firmware of an appliance is updated, rolling back to

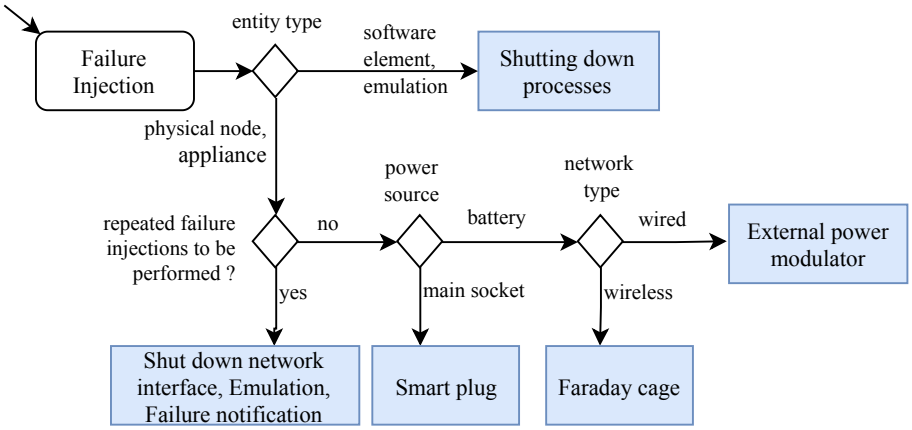


Figure 10.7: Failure Injection

the previous version may be blocked, preventing experimentation during a live update of the device. Likewise, mobility of wireless devices, like Hue Go lamp, are hard to reproduce and the use of robots [21] for this purpose is expensive. Emulation may be adapted for reproducing some of the cases that cannot be reproduced on real devices since it offers the possibility of extending the behaviour of an appliance. Furthermore, emulation coupled with event injection were used to make the application execute the different user stories. This allows the automatic generation of state data of software elements and appliances. This procedure avoids manually running the user stories for data generation. Emulation is also a way of easing monitoring and failure injections since operations can be performed easily on an emulated program.

10.4 Evaluation Tools

In order to implement the methodology discussed in the previous section and perform the evaluation of F^{3}ARIoT, five additional evaluation tools were developed:

a . Random Failure Injector, RFI

The aim of the RFI is to replicate (i.e., repeat on different entities) and randomise failure injections on software elements, appliances and physical nodes. The failure of software elements are provoked by shutting down their corresponding processes or services. The failure of appliances is *simulated* by injecting a failure notification at the corresponding wrappers or by provoking the failure of their emulated modules. Failure notifications are used because, unlike software elements, repeatedly injecting failures on a physical device may cause its break down. The failure of physical nodes are provoked by shutting down their network interfaces. The RFI consists of two components: a *Failure Injection Selector, FIS* and a set of *Failure Injection Operators, FIOs*. A FIO is deployed on each fog node and they synchronise with a global FIS. The FIS is centralised for better control and aftermath

audit. It executes the following function, *Array<Id> injectFailures(Array<Type> E, Int nf, Int tf, Date st, Date et)*, where E contains the type(s) of entities on which to provoke the failures (i.e. physical/fog nodes, appliances, software elements), nf is the number of failures, tf is the number of seconds between failures, st and et are the start and end time of the experiment. The function diffuses nf random entities of type(s) E at a tf rate to the FIOs which inject the corresponding failures. The FIS and FIOs log the failure injections and their timestamps for auditing.

b . Repair Injector, RI

It is used to inject a repair on physical nodes and appliances after their failure have been provoked by the RFI. For instance, the failure of a Raspberry Pi is provoked by the RFI by shutting down its network interfaces. Subsequently, the neighbouring fog agents detect its failure since heartbeat messages are not received. To mock the repair of the Raspberry Pi, the RI reboot its network interface so that the neighbouring fog agents detect the repair. The RI is coupled with the RFI to reproduce the failure and subsequent repair behaviour of physical nodes and appliances.

c . Scenario Injector, SI

The purpose of the SI is to change the state of the application. To do so, it injects an event or a sequence of events corresponding to a given user story. After injection, it returns the resulting state in which the application should be to the Verifier tool. The change of state provoked by the SI allows the generation of state data and the change of state of the application. In this way, the restoration of consistent a state can be evaluated under different states of the application.

d . Verifier

It checks that the application is in a specific state after a scenario injection and checks CP-consistent recovery. To do so, it requests the state of the different entities of the application. It also checks that the application is functional. To do so, it requests a user story injection to the SI and checks that the resulting change in state of the application is consistent with the user story.

e . Measurements Logger, ML

The ML observes the events (failures, reconfiguration, state restoration, recovery, etc.) pushed by the GDM, wrappers and evaluation tools. It logs these events with their timestamps for auditing. For instance, these logs allow the computation of the time taken to repair a failure or the time taken to restore the state of software elements and appliances. All computing and storage tasks related to the experiments are performed on a dedicated machine a posteriori to experiments.

In the following, two types of evaluation are proposed. The functional evaluation focuses on evaluating whether the different failure managers work as expected and that the differ-

ent recovery procedures are successfully achieved. The aim of this evaluation is to confirm the results obtained in Chapter 9 by performing experiments on real application. The performance evaluation focuses on evaluating whether the recovery procedure is implemented in a reasonable user time.

10.5 Functional Evaluation

This section aims at analysing the end-to-end failure management of F^{3}ARIoT under real world conditions and scenarios. In order to do so, the failure of both software elements and appliances are targeted to evaluate failure notification propagation, degraded mode and CP-consistent recovery. Three classes of *Failure-Recovery Scenarios* are implemented for this evaluation. Each scenario describes a set of steps to evaluate the different tasks implemented by the framework. The steps in the scenarios are illustrated in the workflow in Figure 10.8. For each step, it gives the operations, the tools involved, the outputs of the Verifier, and the outputs of the ML. An evaluation starts by first setting up an initial state of the application using the SI. The ML logs the start time T_{start} of the evaluation and the Verifier logs a boolean, B_{succ} when the initial settings are completed. Next, a Failure-Recovery Scenario among *F-R-S1, F-R-S2*, and *F-R-S3* is randomly chosen. The ML logs the timestamps of each step. The Verifier logs a boolean, B_{succ}, representing the success of a recovery procedure.

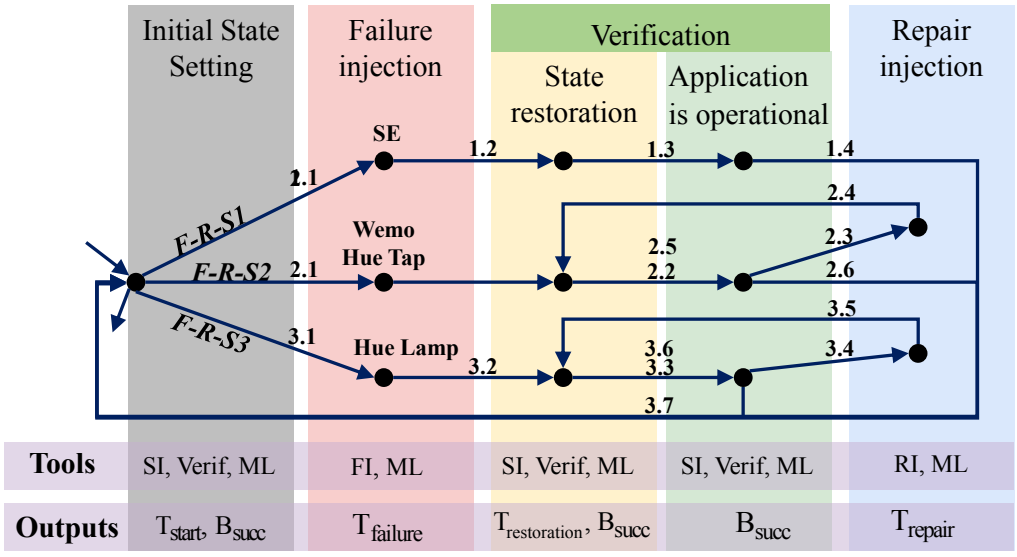


Figure 10.8: Failure-Recovery Scenarios

F-R-S1 aims at evaluating the consistent recovery of software elements. A random software element failure is provoked by the Failure Injector. The Verifier is used to check that the software element has recovered in a consistent state. A set of events is then injected. The correct change of state confirms that the application has been repaired and is fully functional.

F-R-S2 aims at assessing degraded mode, architectural reconfiguration, and CP-consistent recovery. Both the Wemo Motion and Hue Tap buttons are targeted for this experiment because they trigger a degraded mode when they fail since they cannot be replaced in this testbed. A failure is first injected on one of these two devices, randomly chosen. The subsequent failure notification propagation and degraded mode of the application is checked by the Verifier. The RI is then used to inject a repair of the device. This triggers a reconfiguration of the application whereby the previously failed device is reintegrated into the application. Recovery notifications are propagated. The Verifier is used to confirm that the application has been successfully recovered and is fully operational again by requesting a set of motion events and button press injections. It then checks the correct change of state in the application, i.e. the Awox StriimLight/Hue lamps are turned on and off accordingly.

F-R-S3 evaluates architectural reconfiguration and CP-consistent recovery. After the initial settings, a failure is injected onto the bedroom Hue lamp. This results in the state of the failed lamp being restored on the Hue Go lamp (bedside lamp) which is used as a temporary replacement. The Verifier is used to check that the correct CP-consistent state of the bedside lamp is restored. It should correspond to the pre-failure state of the failed Hue lamp. Then, Hue Tap buttons events are injected to check that the forthcoming events are redirected to the replacement appliance by the wrapper and that corresponding changes in the state of the bedside lamp are consistent. Finally, a repair is injected concerning the bedroom Hue lamp. The subsequent CP-consistent recovery on the permanent replacement (i.e. main bedroom lamp) is checked by the Verifier. Hue Tap buttons events are injected. The correct changes of the state of the lamps confirm that the application has recovered and is fully operational.

In order to repeat and randomise the experiments for evaluation, the tools presented in Section 10.4 are synchronised such that two random scenarios from *F-R-S1, F-R-S2* and *F-R-S3* are selected every hour at a random time instant. The experiments lasted for one month for which, 681 software elements and 688 appliances failures were injected. *F-R-S2* was run 393 times involving Wemo Motion Sensor and 183 times for Hue Tap buttons device. *F-R-S3* was run 166 times. The auditing of the logs retrieved during these experiments showed that F^{3}ARIoT works well in practice for failure scenarios that can happen in real-life. F^{3}ARIoT recovers a consistent state of the application upon software element failures. Moreover, a degraded mode is achieved when an appliance, that cannot be replaced, fails. The experiment also shows the success of failure notification propagation and CP-consistent recovery.

Therefore, this functional evaluation confirms the correct behaviour of the F^{3}ARIoT on a real deployment of a smart home application and supports the results presented in Chapter 9.

10.6 Performance Evaluation

This section aims at having a performance evaluation of the recovery process performed by F^{3}ARIoT. More specifically, it aims at showing that the time taken for F^{3}ARIoT to implement PW-consistent recovery is performed within an acceptable time with respect to end users.

a . Software Elements

The first experimental campaign aims at evaluating F^{3}ARIoT for single software element failure at a given frequency. To this end, the RFI is set to provoke one random failure every five seconds. The experiment is stopped after provoking five thousand failures on each software element. The state restoration procedure of the software element consists in replaying events before loading a checkpoint. The mean time to recover a failed software element is given by the ML tool and computed as follows, $T_{recover} = T_{restart} + T_{restore}$, where $T_{restart}$ is the mean time taken to restart the software element and $T_{restore}$ is the mean time taken to retrieve the data and restore its state. Figure 10.9 illustrates the average values of $T_{restart}$ of the software elements where $se_1, se_2, se_3, se_4, se_5, se_6$, and se_7 are respectively AwoxActuate, WemotionSense, FibaroSense, NodeHueActuate, NodeHueSense, Speaker-Actuate, and Orchestrator. It shows that the time taken for a software element to be functional varies, but takes less than one second. The heterogeneous restart times are mainly due to the implementation and the runtime dependencies of the software elements. The figure also illustrates the corresponding mean time taken for state restoration of each software element. It shows that the percentage overhead introduced by F^{3}ARIoT for state restoration is almost negligible compared to the time for the software element itself to restart. Thus, the time for recovery is mostly dependent on the nature of the software element rather than the mechanisms implemented by F^{3}ARIoT.

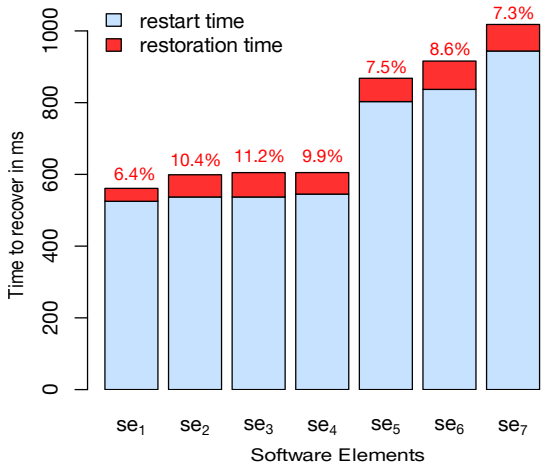


Figure 10.9: Time to Recover Software Elements Failures

Note, however, that the time for state restoration may be higher if there is a costly processing following the replay of an event. This is why the frequency of checkpoints defined

at a wrapper should not only be based on the number of processed events or execution time, but also on the *weights* of events such that a checkpoint can be performed after an event that causes a costly processing. The optimal frequency of checkpoint is a subject of future works.

In a second experimental campaign, the frequency of RFI is modified such that the number of failures in the five-second intervals is increased up to five failures. The aim is to analyse the performance of F^{3}ARIoT under stress. The time to restore the state of individual software elements is computed each time and is depicted in Figure 10.10. The horizontal axis defines the number of failures of software elements in a five-second interval. The vertical axis describes the average time taken to restore the state of each software element. In the case of two failures in the five-second interval, the average time to restore the state of each software element is less than 100ms. In the case of five failures, the average time to restore the state of each software element is less than 120ms. It shows a slight increase in the state restoration overhead as the number of failures increases which remains, however, rather below the restart time of the software elements which can be up to 900ms as illustrated in Figure 10.9. This increase accounts for the repeated solicitation of the stable storage within small time intervals.

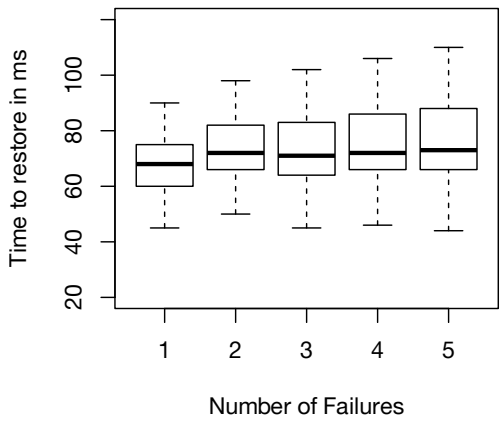


Figure 10.10: Time to Restore State of Software Elements for Multiple Failures

b . Appliances

The objective of the experiments targeting appliances is to measure the time taken for restoring a CP-consistent state of appliances by using their EBM models. The time to restore consists of the time for the wrapper of the appliance to retrieve the state data, feed the set of events into its EBM, restore the resulting target state of the appliance and receive an acknowledgement from the appliance. The experiment is repeated one thousand times for each appliance at five seconds interval. Figure 10.11 depicts the time to restore the state of appliances $appl_1$, $appl_2$, $appl_3$, $appl_4$ and $appl_5$ which are respectively Awox StriimLight, Speaker, livingroom Hue lamp, bedroom Hue Go and bedroom Hue lamps. The time taken to restore a CP-consistent state of the appliance remains close to 100ms for the Awox

StriimLight and is less than 150ms for the Speaker. In the case of the Hue lamps, the state restoration time is always below 200ms. The time for state restoration of appliances are expectedly higher than those of software elements because the appliance and the wrapper are, unlike software elements, delocalised and connected over a wireless network. Thus, a higher time to restore the state of appliances is due to network communication. The time for restoring a CP-consistent state is always below 200ms and is therefore acceptable from a user point of view in a smart home.

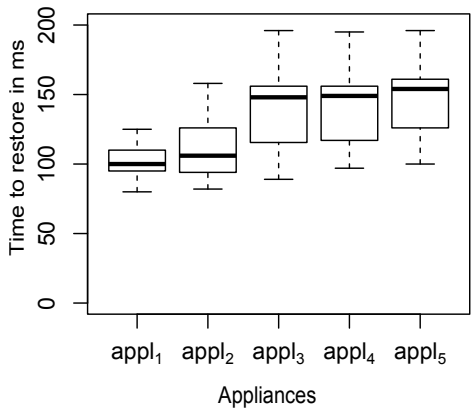


Figure 10.11: Time to Restore State of Appliances

10.7 Conclusion

This chapter has presented an implementation of our failure management approach as a framework called F^{3}ARIoT. It was deployed on a smart home testbed composed of infrastructure and applicative entities that can be found in real-life smart homes.

A functional evaluation of F^{3}ARIoT showed that the failure managers ensure their roles and the recovery procedures are performed successfully for the different types of failures injected. The performance evaluation aimed at evaluating whether the recovery procedure is done in an acceptable delay with respect to end users. According to [47, 114], 1s is the limit in response time for the user's flow to stay uninterrupted. Moreover, [114] estimates that in case of failures, recovery should be done in less than 15s to avoid annoyance and disruption of the user. The performance evaluation showed that the overhead for state restoration is negligible compared to the restart time of software elements. The time to recover, including the time for state restoration, does not exceed 1s. The time for restoring a CP-consistent state of appliances is achieved in less than 200ms. To sum up, these results are satisfactory and show that recovery is done within short delays, even in the case of high frequency of failures, that is completely acceptable from an end user point of view.

Chapter 11

Conclusion

Contents

This Thesis proposed an end-to-end autonomic failure management approach for IoT applications in the Fog that is capable of detecting failures and recovering the application in a consistent state with respect to the PW. The approach avoids costly and unsafe situations in the PW as well as maintains a consistent behaviour of the application after recovery. This chapter concludes this Thesis by summarising our key contributions. Future work to extend the proposed approach is also discussed.

11.1 Summary of Contributions

This section summarises our contributions in designing an autonomic failure management approach for IoT applications in the Fog.

The first contribution of this Thesis is a formal model for IoT applications in the Fog. This model characterises the devices in the Fog infrastructure which provide physical resources as well as applicative entities that run on this infrastructure. The mechanisms performed for failure management such as for observing state changes, detecting failures, reconfiguring the application, as well as for recovery are implemented according to the properties of the entities involved in the application as defined in the proposed model. Thus, the approach relies on this model to implement autonomic failure management.

Then, we proposed an end-to-end failure management approach which is composed of four functional steps. (i) state saving, (ii) monitoring, (iii) failure notifications, and (iv) recovery. State saving aims at saving the state of the application during its runtime. We propose state saving policies which describe several combinations of state saving techniques that suit the specificities of the Fog-IoT ecosystem. Monitoring is a continuous process and aims at reporting information about the failure and recovery of the entities participating in the application. The techniques for monitoring are chosen such that they have a minimal interference on the application. When a failure is detected, the state saving step of the failed entity stops. Failure notifications are propagated to the part of the application impacted by the failure so that the application can move to a degraded mode and a partial service can be provided. To recover from the failure, the application is first reconfigured and then a consistent state of the application is restored, including consistency with respect to the PW. In order to achieve CP-consistency, we propose a state restoration procedure that is based on recovery policies which make use of (i) the data saved during the state saving step, (ii) the behavioural model of the application, and (iii) the validity time of events. The use of the EBM behavioural model avoids the replay of events that have harmful impacts on the PW and ensures consistency between the state of the application and the state of the physical world (i.e., cyber-physical consistency).

We proposed two complementary validations of the approach. First, we described a formal specification and verification of the approach, with a focus on the recovery step, using model checking techniques. This aimed at eliminating design errors and ensuring that the approach works as expected as well as respect important correctness properties. The results of this validation showed that the properties of interest such as architectural invariants (e.g., *there are no disconnected entities in the application*) and functional properties (e.g., *when a software element fails, it eventually recovers and resumes its behaviour in its pre-failure state*) are satisfied. Second, we implemented the failure management approach as a framework called F^{3}ARIoT. This framework is intended for IoT DevOps. The parameters by default can be tuned by IoT DevOps to fit the specificities of their use case applications (e.g., frequency of monitoring, validity time of events, frequency of checkpoints, etc.). F^{3}ARIoT was deployed on a smart home application based on an industrial project at Orange Labs. The results of this evaluation showed that the framework works well in practice and recovers from failures of the infrastructure and application in a CP-consistent way. A performance evaluation of the F^{3}ARIoT showed that the additional overhead introduced by the failure management framework is negligible. Moreover, the recovery time is bounded by one second. The recovery procedure is thus done in a completely acceptable delay in regards to end users.

11.2 Future Work

As far as future works are concerned, we propose some research directions to extend our work. Sections 11.2.1 and 11.2.2 describe short terms perspectives. Sections 11.2.3 and 11.2.4 report medium term perspectives. Section 11.2.5 describes a long term endeavour.

11.2.1 Full Autonomicity

In its current version, F^{3}ARIoT recovers from failures of the infrastructure and application that it manages. One of our priority in terms of future work is to make F^{3}ARIoT fully autonomic. In order to be fully autonomic, F^{3}ARIoT should be able to manage and repair itself. This property is referred to as *self^2-repair* [41]. This can be based on a recursive design where F^{3}ARIoT manages itself in the same way it manages a distributed application so that it can detect failures of the failure managers and recover from these failures. To this end, the global failure manager GDM should be replicated on different physical node. In this case, the replicas monitor each other and an election protocol can determine which replica takes over when a failure occurs. Regarding the wrappers, they can monitor each other by observing the messages exchanged between them and by implementing ping-ack/heartbeat mechanisms. A neighbouring wrapper can be used to implement the recovery procedure of a failed wrapper. When a wrapper recovers from a failure, it has to retrieve the messages (from the local or stable storage) it has logged but are not yet delivered to its encapsulations. If there are multiple messages that have to be delivered to an encapsulation, the messages can be replayed on its EBM to determine the target state to set the encapsulation. If the wrapper fails upon reception of a message but before logging it, the message should be re-emitted by the emitting wrapper after the failed wrapper has recovered. To do so, acknowledgement messages can be used. In this case, a message is acknowledged only after logging the message. This ensures that if the wrapper fails before logging, the message will be re-emitted.

11.2.2 Inferring Parameters

Most of the parameters in the configuration files of F^{3}ARIoT such as monitoring technique and state saving techniques can be automatically assigned based on the properties (e.g., communication model, frequency of emission of messages, entity type, etc.) of the application. However, other parameters such as validity time of events and *weights* of events (for computation of the checkpoint frequency) have to be manually filled in by the DevOps of the IoT application in the configuration file of wrappers. These parameters can be inferred so that the framework could be easily adopted and well appreciated by IoT DevOps. For instance, the validity times of events can be inferred from the type of the event and from the frequency at which the event is sent. The *weights* of events can be inferred by measuring the time an entity takes to process the event. The frequency of checkpoint is then automatically set based on the information inferred. The average time taken to compute an event can also be used to distinguish between a busy and a failed entity. The frequency

of monitoring can then be dynamically set based on the variations in communication delays and the time to compute events.

11.2.3 Predictive Maintenance

In this Thesis, we had a reactive approach to failure management. That is, a recovery procedure is engaged only after a failure has occurred. A pro-active approach can be achieved by coupling the current approach with predictive maintenance. This pro-active approach would recover from the occurrence of failures and would also be able to predict when a failure might occur based on machine learning techniques. Such prediction can be based on the time of execution, the frequency and history of failures, performance data, or heat dissipated by a device. The occurrence of the failure may then be prevented by performing a maintenance (e.g., reboot of the device, temporary shut down, notifications to plug a device before its battery drains out of power). In this way, the failure of a component of the application that may bring down the whole application may be avoided. For instance, IIoT (Industrial IoT) applications may involve engine cylinders or turbines for which it would be preferable to implement predictive maintenance because the failure of such components may damage surrounding equipments and be very costly.

11.2.4 Scalability

The evaluation of F^{3}ARIoT on the smart home application showed that the recovery procedure exhibits good performances with respect to end users even in the case of high frequency of failures. In order to keep these performances for applications that are highly geographically distributed, such as smart cities, the GDM should be geographically distributed. This aims at minimising the network latencies between the entities of the application and the GDM. In this way, each instance of GDM manages a subset of the entities (i.e., a cluster) grouped with respect to geographical constraints. For instance, a GDM instance can be deployed in each smart home or across each neighbourhood in the Telco's infrastructure (e.g., Telecom's towers, PoP of Internet Service Providers, Mini data-centers such as RuggedPod [135]). The GDM instances can share information about the impact of a failure across multiple clusters. They can also share information on the resources available in each cluster to perform recovery procedures.

11.2.5 Extension to Other IoT Application Domains

Although the proposed approach in this Thesis was designed for IoT applications in the smart home/building domains, the presented concepts and ideas can be used in other application domains. It would be interesting to study how this approach can be enriched to manage failures in these cases. For instance, in a drone application domain, the choice between the different state saving policies, in addition to the parameters presented in Table 6.2, may also depend on the mobility of the entities in the application. State data could be stored and replicated on nearby drones so as to optimise the recovery time. The proposed recovery policies can also be extended. For example, when a drone cannot be rebooted, rather than performing an immediate reinitialisation of the hosted software elements, another drone can be used to bring back the failed drone in a network range in which remote maintenance can be performed.

Bibliography

[1] Arduino Website. https://www.arduino.cc/.

[2] BME280 Combined Humidity and Pressure Sensor - Final Data Sheet. https://www.embeddedadventures.com/datasheets/BME280.pdf.

[3] Mosquitto Website. https://mosquitto.org/.

[4] MQTT Website. http://mqtt.org/.

[5] Onion Omega Website. https://docs.onion.io/omega2-docs/.

[6] OPT3001 Ambient Light Sensor (ALS). http://www.ti.com/lit/ds/symlink/opt3001.pdf.

[7] Orange Maison Connectée. https://boutique.orange.fr/maison/domotique/.

[8] Orange Maison Protégée. https://boutique.orange.fr/telesurveillance.

[9] Report: Orvibo Smart Home Devices Leak Billions of User Records (June, 2019). https://www.vpnmentor.com/blog/report-orvibo-leak/.

[10] Servo Circuit. https://www.arduino.cc/en/reference/servo.

[11] Socket.IO Website. https://www.npmjs.com/package/socket.io.

[12] That "Internet of Things" Thing: In the real world, Things Matter More Than Ideas. http://www.rfidjournal.com/articles/view?4986.

[13] The "only" Coke Machine on the Internet. https://www.cs.cmu.edu/~coke/history_long.txt.

[14] Thing'in Graph. https://hellofuture.orange.com/en/thingin-the-things-graph-platform/.

[15] Thing'in Platform and The Web of Things. http://thinginthefuture.com/.

[16] An Architectural Blueprint for Autonomic Computing. *IBM White Paper*, 31:1–6, 2006.

[17] Current Trends in Smart City Initiatives: Some Stylised Facts. *Cities*, 38:25 – 36, 2014.

[18] SmartSantander: IoT Experimentation Over a Smart City Testbed. *Computer Networks*, 61:217 – 238, 2014. Special issue on Future Internet Testbeds – Part I.

[19] The Internet of Things: Mapping the Value Beyond the Hype. https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world, 2015.

[20] David Perez Abreu, Karima Velasquez, Marilia Curado, and Edmundo Monteiro. A resilient Internet of Things architecture for smart cities. *Annals of Telecommunications*, 72(1), 2017.

[21] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. In *IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Dubai, 2015.

[22] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. de Mel. An Analysis of Communication Induced Checkpointing. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pages 242–249, 1999.

[23] Lorenzo Alvisi, Karan Bhatia, and Keith Marzullo. Causality Tracking in Causal Message-logging Protocols. *Distrib. Comput.*, 15(1):1–15, 2002.

[24] Lorenzo Alvisi and Keith Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Trans. on Software Engineering*, 24(2):149–159, 1998.

[25] Dana Angluin, Sarah Eisenstat, and Dana Fisman. Learning regular languages via alternating automata. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3308–3314, 2015.

[26] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, 2009.

[27] Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B. Terry, and Riza O. Suminto. Rivulet: A Fault-tolerant Platform for Smart-home Applications. In *Proc. of Middleware'17*, Middleware '17, pages 41–54. ACM, 2017.

[28] F. Aïssaoui, G. Cooperman, T. Monteil, and S. Tazi. Intelligent Checkpointing Strategies for IoT System Management. In *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 305–312, 2017.

[29] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and Its Threats: A Taxonomy. In Renè Jacquart, editor, *Building the Information Society*, pages 91–120, Boston, MA, 2004. Springer US.

[30] J. Bai, Y. Sun, and C. Phillips. CRRP: A Cooperative Relay Routing Protocol for IoT Networks. In *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–6, 2016.

[31] R. Baldoni, J. . Helary, A. Mostefaoui, and M. Raynal. A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 68–77, 1997.

[32] Beneyaz A. Begum and Satyanarayana V. Nandury. Component-based Self-Healing Algorithm with Dynamic Range Allocation for Fault-Tolerance in WSN. In *Proceedings of the 7th International Conference on Computer and Communication Technology*, ICCCT-2017, pages 58–65, New York, NY, USA, 2017. ACM.

[33] Paolo Bellavista and Alessandro Zanni. Feasibility of Fog Computing Deployment Based on Docker Containerization over RaspberryPi. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN '17, pages 16:1–16:10. ACM, 2017.

[34] Marin Bertier, Olivier Marin, and Pierre Sens. Performance Analysis of a Hierarchical Failure Detector. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 635–644, 2003.

[35] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog Computing: A Platform for Internet of Things and Analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer International Publishing, 2014.

[36] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and its Role in the Internet of Things. In *Proc. of MCC'12*, pages 13–16. ACM, 2012.

[37] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault Tolerance Under UNIX. *ACM Trans. Comput. Syst.*, 7(1):1–24, 1989.

[38] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, N. de Palma, V. Quema, and J. . Stefani. Architecture-Based Autonomous Repair Management: Application to J2EE Clusters. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 369–370, 2005.

[39] Mohamed-Lamine Boukhanoufa. *Adaptability and Reconfiguration for Embedded and Real-Time Systems*. PhD thesis, Université Paris Sud - Paris XI, 2012.

[40] Fabienne Boyer, Olivier Gruber, and Damien Pous. Robust Reconfigurations of Component Assemblies. In *Proc. of ICSE'13*, pages 13–22. IEEE Press, 2013.

[41] Fabienne Boyer, Noel Palma, Olivier Gruber, and Sylvain Sicard. Full Autonomic Repair for Distributed Applications. *Softw. Pract. Exper.*, 44(9):1027–1045, September 2014.

[42] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. How to Place Your Apps in the Fog - State of the Art and Open Challenges, 2019.

[43] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level Checkpointing for Shared Memory Programs. *SIGPLAN Not.*, 39(11):235–247, October 2004.

[44] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL Component Model and Its Support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[45] L. Buechley and M. Eisenberg. The LilyPad Arduino: Toward Wearable Engineering for Everyone. *IEEE Pervasive Computing*, 7(2):12–15, 2008.

[46] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — A Technique for Cheap Recovery. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3. USENIX Association, 2004.

[47] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proc. of CHI '91*, pages 181–186. ACM, 1991.

[48] W. C. Carter. A Time for Reflection. In *Proceedings IEEE International Symposium on Fault Tolerant Computing (FTCS-12).*

[49] Dr. Jeff D. Case, Russ Mundy, David Partain, and Bob Stewart. Introduction and Applicability Statements for Internet-Standard Management Framework. RFC 3410, 2002.

[50] Marco Castaldi, Antonio Carzaniga, Paola Inverardi, and Alexander L. Wolf. A Lightweight Infrastructure for Reconfiguring Applications. In Bernhard Westfechtel and André van der Hoek, editors, *Software Configuration Management*, pages 231–244, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[51] Luca Catarinucci, Danilo de Donno, Luca Mainetti, Luca Palano, Luigi Patrono, Maria Laura Stefanizzi, and Luciano Tarricone. An IoT-Aware Architecture for Smart Healthcare Systems. *IEEE Internet of Things Journal*, 2(6):515–526, 2015.

[52] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator, Version 6.7. Inria, 2018.

[53] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, March 1996.

[54] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[55] Abiy Biru Chebudie, Roberto Minerva, and Domenico Rotondi. Towards a Definition of The Internet of Things (IoT). Technical report, IEEE, 2015.

[56] D. Chen, P. Bovornkeeratiroj, D. Irwin, and P. Shenoy. Private Memoirs of IoT Devices: Safeguarding User Privacy in the IoT Era. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1327–1336, 2018.

[57] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[58] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the Quality of Service of Failure Detectors. *IEEE Trans. Comput.*, 51(5):561–580, May 2002.

[59] Xuejun Chen and Martin Simons. A Component Framework for Dynamic Reconfiguration of Distributed Systems. In Judith Bishop, editor, *Component Deployment*, pages 82–96, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[60] M. Chereque, D. Powell, P. Reynier, J. . Richier, and J. Voiron. Active Replication in Delta-4. In *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 28–37, 1992.

[61] J. Chiu, A. Liu, and C. Liao. Design the DNS-Like Smart Switch for Heterogeneous Network Base on SDN Architecture. In *2016 International Computer Symposium (ICS)*, pages 187–191, 2016.

[62] David C. Chou. Cloud Computing: A Value Creation Model. *Computer Standards & Interfaces*, 38:72 – 77, 2015.

[63] Aakanksha Chowdhery, Marco Levorato, Igor Burago, and Sabur Baidya. *Urban IoT Edge Analytics*. Springer International Publishing, 2018.

[64] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of ICSE'99*, pages 411–420. ACM, 1999.

[65] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Rivière. On Using Micro-Clouds to Deliver the Fog. *IEEE Internet Computing*, 21(2):8–15, 2017.

[66] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[67] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-Configuration of Distributed Applications in the Cloud. In *Proc. of CLOUD'11*, pages 668–675. IEEE Computer Society, 2011.

[68] Xavier Etchevers, Gwen Salaün, Fabienne Boyer, Thierry Coupaye, and Noel De Palma. Reliable Self-deployment of Distributed Cloud Applications. *Softw., Pract. Exper.*, 47(1):3–20, 2017.

[69] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proc. of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[70] Sergio Fortes, José Antonio Santoyo-Ramón, David Palacios, Eduardo Baena, Rocío Mora-García, Miguel Medina, Patricia Mora, and Raquel Barco. The Campus as a Smart City: University of Málaga Environmental, Learning, and Research Approaches. *Sensors*, 19(6), 2019.

[71] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The Failure Detector Abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, February 2011.

[72] Gerhard Friedrich, Mariagrazia Fugini, Enrico Mussi, Barbara Pernici, and Gaston Tagni. Exception Handling for Repair in Service-Based Processes. *IEEE Trans. Software Eng.*, 36(2):198–215, 2010.

[73] Guilherme Galante and Luis Carlos E. de Bona. A Survey on Cloud Computing Elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, pages 263–270, Washington, DC, USA, 2012. IEEE Computer Society.

[74] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 2(15):89–107, 2013.

[75] Martin Gerdes, Yohanes Baptista Dafferianto Trinugroho, Mari Næss, and Rune Fensli. Security, Reliability and Usability of mHealth Environments. In *Mobile Health*, pages 1043–1066. Springer International Publishing, 2015.

[76] Tuan Nguyen Gia, Amir-Mohammad Rahmani, Tomi Westerlund, Pasi Liljeberg, and Hannu Tenhunen. Fault Tolerant and Scalable IoT-Based Architecture for Health Monitoring. In *IEEE SAS*, pages 1–6. IEEE, 2015.

[77] Sébastien Guillet, Bruno Bouchard, and Abdenour Bouzouane. Safe and Automatic Addition of Fault Tolerance for Smart Homes Dedicated to People with Disabilities. In *Trends in Ambient Intelligent Systems*, pages 87–116. Springer International Publishing, 2016.

[78] C. Hofmeister and J. Purtilo. Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement. In *[1993] Proceedings. The 13th International Conference on Distributed Computing Systems*, pages 101–110, May 1993.

[79] Christine Ruth Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, College Park, MD, USA, 1993. UMI Order No. GAX94-07643.

[80] P Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM Corporation (October 15, 2001).

[81] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. Survey on Fog Computing: Architecture, Key Technologies, Applications and Open Issues. *Journal of Network and Computer Applications*, 98:27–42, 2017.

[82] Michel Hurfin and Michel Raynal. A Simple and Fast Asynchronous Consensus Protocol Based on A Weak Failure Detector. *Distributed Computing*, 12(4):209–223, 1999.

[83] Joshua Hursey, Jeffrey Squyres, Timothy Mattox, and Andrew Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. pages 1–8, 01 2007.

[84] J. Isaak and M. J. Hanna. User Data Privacy: Facebook, Cambridge Analytica, and Privacy Protection. *Computer*, 51(8):56–59, 2018.

[85] ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, ISO, 1989.

[86] Samir Jafar, Axel Krings, and Thierry Gautier. Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing. *IEEE Trans. Dependable Secur. Comput.*, 6(1):32–44, January 2009.

[87] Barry W. Johnson. Fault-tolerant Computer System Design. chapter An Introduction to the Design and Analysis of Fault-tolerant Systems, pages 1–87. Prentice-Hall, Inc., 1996.

[88] G. Kakamanshadi, S. Gupta, and S. Singh. A survey on Fault Tolerance Techniques in Wireless Sensor Networks. In *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 168–173, 2015.

[89] Carlos Kamienski, Juha-Pekka Soininen, Markus Taumberger, Stenio Fernandes, Attilio Toscano, Tullio Salmon Cinotti, Rodrigo Filev Maia, and Andre Torre Neto. SWAMP: an IoT-based Smart Water Management Platform for Precision Irrigation in Agriculture. In *2018 Global Internet of Things Summit (GIoTS)*. IEEE, 2018.

[90] Vasileios Karagiannis, Periklis Chatzimisios, Francisco Vazquez-Gallego, and J Alonso-Zarate. A Survey on Application Layer Protocols for the Internet of Things. *Trans. IoT Cloud Comput.*, 3:11–17, 01 2015.

[91] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[92] N Kushalnagar, G Montenegro, and C Schumacher. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. 01 2007.

[93] Ten H. Lai and Tao H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[94] Leslie Lamport. Distribution, May 1987. Email Message Sent to a DEC SRC Bulletin Board at 12:23:29 PDT on 28 May 87.

[95] Butler Lampson and Howard E. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, Xerox Palo Alto Research Center, 1979.

[96] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology.* Springer Vienna, 1992.

[97] Q. Le, T. Ngo-Quynh, and T. Magedanz. RPL-based Multipath Routing Protocols for Internet of Things on Wireless Sensor Networks. In *2014 International Conference on Advanced Technologies for Communications (ATC 2014)*, pages 424–429, 2014.

[98] Marc Léger. *Fiabilité Des Reconfigurations Dynamiques dans les Architectures à Composants.* PhD thesis, École Nationale Supérieure des Mines de Paris, 2009.

[99] Loïc Letondeur, François-Gaël Ottogalli, and Thierry Coupaye. A Demo of Application Lifecycle Management for IoT Collaborative Neighborhood in the Fog. In *IEEE Fog World Congress*, pages 1–6. IEEE, 2017.

[100] Mo Li and Yunhao Liu. Underground Structure Monitoring with Wireless Sensor Networks. In *Proc. of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 69–78. ACM, 2007.

[101] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the COndor Distributed Processing System. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[102] S. Lukasik. Why the Arpanet Was Built. *IEEE Annals of the History of Computing*, 33(3):4–21, 2011.

[103] R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[104] N. Maalel, E. Natalizio, A. Bouabdallah, P. Roux, and M. Kellil. Reliability for Emergency Applications in Internet of Things. In *2013 IEEE International Conference on Distributed Computing in Sensor Systems*, pages 361–366, 2013.

[105] Somayya Madakam and Rajesh M. Holmukhe. Songdo Smart City. In *Advances in Civil and Industrial Engineering*, pages 278–298. IGI Global, 2019.

[106] L. Mainetti, L. Patrono, and A. Vilei. Evolution of Wireless Sensor Networks Towards the Internet of Things: A Survey. In *SoftCOM 2011, 19th International Conference on Software, Telecommunications and Computer Networks*, pages 1–6, Sep. 2011.

[107] O. Marin, M. Bertier, and P. Sens. DARX - A Framework for the Fault-Tolerant Support of Agent Software. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 406–416, 2003.

[108] R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.

[109] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

[110] Friedemann Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[111] K. Mekki, E. Bajic, F. Chaxel, and F. Meyer. Overview of Cellular LPWAN Technologies for IoT Deployment: Sigfox, LoRaWAN, and NB-IoT. In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 197–202, March 2018.

[112] Peter Middleton, Jim Tully, and Peter Kjeldsen. Forecast: The Internet of Things, Worldwide, 2013, November 2013.

[113] Fabien Mieyeville, Mihai Galos, and David Navarro. Dynamic Reconfiguration for Software and Hardware Heterogeneous Real-time WSN. In *SENSORCOMM 2012: The Sixth International Conference on Sensor Technologies and Applications*, 2012.

[114] Robert B. Miller. Response Time in Man-Computer Conversational Transactions. In *Proc. of AFIPS '68 (Fall, part I)*, pages 267–277. ACM, 1968.

[115] S. Misra, A. Gupta, P. V. Krishna, H. Agarwal, and M. S. Obaidat. An Adaptive Learning Approach for Fault-Tolerant Routing in Internet of Things. In *2012 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 815–819, 2012.

[116] Kaveh M. Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, UK, 1999.

[117] N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Towards Fault Tolerant Fog Computing for IoT-Based Smart City Applications. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0752–0757, 2019.

[118] Fabrizio Montesi and Janine Weber. Circuit Breakers, Discovery, and API Gateways in Microservices. *ArXiv*, 2016.

[119] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand. iFogStor: An IoT Data Placement Strategy for Fog Infrastructure. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 97–104, 2017.

[120] A. Nabina and J. L. Nunez-Yanez. Dynamic Reconfiguration Optimisation with Streaming Data Decompression. In *2010 International Conference on Field Programmable Logic and Applications*, pages 602–607, 2010.

[121] Behailu Shiferaw Negash, Amir M. Rahmani, Pasi Liljeberg, and Axel Jantsch. *Fog Computing Fundamentals in The Internet-of-Things*. 04 2017.

[122] Jeremy W. Nimmer and Michael D. Ernst. Automatic Generation of Program Specifications. *SIGSOFT Softw. Eng. Notes*, 27(4):229–239, 2002.

[123] Peyman Oreizy et al. Issues in Modeling and Analyzing Dynamic Software Architectures. In *Proceedings of the international workshop on the role of software architecture in testing and analysis*, pages 54–57, 1998.

[124] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee. A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 117–124, 2016.

[125] M. Pasin, S. Fontaine, and S. Bouchenak. Failure Detection in Large Scale Systems: a Survey. In *NOMS Workshops 2008 - IEEE Network Operations and Management Symposium Workshops*, pages 165–168, April 2008.

[126] Charith Perera, Yongrui Qin, Julio C. Estrella, Stephan Reiff-Marganiec, and Athanasios V. Vasilakos. Fog Computing for Sustainable Smart Cities: A Survey. *ACM Comput. Surv.*, 50(3):32:1–32:43, 2017.

[127] James S Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical report, Technical Report UTCS-97-372, 1997.

[128] Alex Polacco and Kayla Backes. The Amazon Go Concept: Implications, Applications, and Sustainability. *Journal of Business & Management*, 24(1), 2018.

[129] Stefano Porcarelli, Marco Castaldi, Felicita Di Giandomenico, Andrea Bondavalli, and Paola Inverardi. A Framework for Reconfiguration-Based Fault-Tolerance in Distributed Systems. In Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems II*, pages 167–190, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[130] Harald Raffelt and Bernhard Steffen. LearnLib: A Library for Automata Learning and Experimentation. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, pages 377–380. Springer Berlin Heidelberg, 2006.

[131] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[132] S. Rani, S. H. Ahmed, R. Talwar, J. Malhotra, and H. Song. IoMT: A Reliable Cross Layer Protocol for Internet of Multimedia Things. *IEEE Internet of Things Journal*, 4(3):832–839, 2017.

[133] R. Ratasuk, B. Vejlgaard, N. Mangalvedhe, and A. Ghosh. NB-IoT System for M2M Communication. In *2016 IEEE Wireless Communications and Networking Conference*, pages 1–5, 2016.

[134] Damian Roca, Rodolfo Milito, Mario Nemirovsky, and Mateo Valero. *Tackling IoT Ultra Large Scale Systems: Fog Computing in Support of Hierarchical Emergent Behaviors*, pages 33–48. Springer International Publishing, 2018.

[135] RuggedPOD Website. `http://ruggedpod.qyshare.com`.

[136] Stavros Salonikias, Ioannis Mavridis, and Dimitris Gritzalis. Access Control Issues in Utilizing Fog Computing for Transport Infrastructure. In Erich Rome, Marianthi Theocharidou, and Stephen Wolthusen, editors, *Critical Information Infrastructures Security*, pages 15–26. Springer International Publishing, 2016.

[137] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. *The International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[138] André Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distrib. Comput.*, 10(3):149–157, April 1997.

[139] M. Sha, D. Gunatilaka, C. Wu, and C. Lu. Empirical Study and Enhancements of Industrial Wireless Sensor–Actuator Network Protocols. *IEEE Internet of Things Journal*, 4(3):696–704, 2017.

[140] Daniel P. Siewiorek. *Reliable Computer Systems: Design and Evaluation, Third Edition*. A K Peters/CRC Press, 1998.

[141] Carlos Sosa and Brant Knudson. IBM System Blue Gene Solution: Blue Gene/P Application Development. Technical report, International Technical Support Organization, 2009.

[142] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 526–531. IEEE Computer Society, 1996.

[143] J. P. G. Sterbenz. Smart City and IoT Resilience, Survivability, and Disruption Tolerance: Challenges, Modelling, and a Survey of Research Opportunities. In *2017 9th International Workshop on Resilient Networks Design and Modeling (RNDM)*, pages 1–6, 2017.

[144] E. Strickland. Cisco Bets on South Korean Smart City. *IEEE Spectrum*, 48(8):11–12, 2011.

[145] Rob Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[146] P. H. Su, C. Shih, J. Y. Hsu, K. Lin, and Y. Wang. Decentralized Fault Tolerance Mechanism for Intelligent IoT/M2M Middleware. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 45–50, 2014.

[147] A. Subahi and G. Theodorakopoulos. Ensuring Compliance of IoT Devices with Their Privacy Policy Agreement. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 100–107, 2018.

[148] M. Terán, J. Aranda, H. Carrillo, D. Mendez, and C. Parra. IoT-based System for Indoor Location Using Bluetooth Low Energy. In *2017 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6, Aug 2017.

[149] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd., 2nd edition edition, 2002.

[150] Chinyang Henry Tseng. Multipath Load Balancing Routing for Internet of Things. *Journal of Sensors*, 2016:1–8, 2016.

[151] Umit D. Ulusar, Gurkan Celik, Erdinc Turk, Fadi Al-Turjman, and Halil Guvenc. *Practical Performability Assessment for ZigBee-Based Sensors in the IoT Era*, pages 21–31. Springer International Publishing, Cham, 2019.

[152] Sathish S. Vadhiyar and Jack J. Dongarra. Self Adaptivity in Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[153] Marco Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Comput.*, 28(12):1709–1732, December 2002.

[154] Luis M. Vaquero and Luis Rodero-Merino. Finding your Way in the Fog. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[155] Rebekka Volk, Julian Stengel, and Frank Schultmann. Building Information Modeling (BIM) for Existing Buildings — Literature Review and Future Needs. *Automation in Construction*, 38:109 – 127, 2014.

[156] Jian-xiong Wang, Yang Liu, Zhi-bin Lei, Kang-heng Wu, Xiao-yu Zhao, Chao Feng, Hong-wei Liu, Xue-hua Shuai, Zhong-min Tang, Li-yang Wu, Shao-yun Long, and

Jia-rong Wu. Smart Water Lora IoT System. In *Proceedings of the 2018 International Conference on Communication Engineering and Technology*, ICCET '18, pages 48–51, New York, NY, USA, 2018. ACM.

[157] Weixun Wang, Prabhat Mishra, and Sanjay Ranka. *Dynamic Reconfiguration in Real-Time Systems*. Springer New York, 2013.

[158] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W. Kent Fuchs. Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(5):546–554, May 1995.

[159] T. Watteyne, S. Lanzisera, A. Mehta, and K. S. J. Pister. Mitigating Multipath Fading through Channel Hopping in Wireless Sensor Networks. In *2010 IEEE International Conference on Communications*, pages 1–5, 2010.

[160] Christof Weinhardt, Arun Anandasivam, Benjamin Blau, Nikolay Borissov, Thomas Meinl, Wibke Michalk, and Jochen Stößer. Cloud Computing – A Classification, Business Models, and Research Directions. *Business & Information Systems Engineering*, 1(5):391–399, Oct 2009.

[161] K. Woyach, D. Puccinelli, and M. Haenggi. Sensorless Sensing in Wireless Networks: Implementation and Measurements. In *2006 4th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, pages 1–8, Feb 2006.

[162] Jian Xu and Robert H.D. Netzer. Adaptive Independent Checkpointing for Reducing Rollback Propagation. In *Proc. 5^{th} IEEE SPDP*, pages 754–761. IEEE, 1993.

[163] Xiaoli Xu, Tao Chen, and Mamoru Minami. Intelligent Fault Prediction System based on Internet of Things. *Computers & Mathematics with Applications*, 64(5):833 – 839, 2012. Advanced Technologies in Computer, Consumer and Control.

[164] M. B. Yassein, W. Mardini, and A. Khalil. Smart Homes Automation Using Z-Wave Protocol. In *2016 International Conference on Engineering MIS (ICEMIS)*, pages 1–6, Sep. 2016.

[165] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of Things for Smart Cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014.

[166] Ben Zhang, Nitesh Mor, John Kolb, Douglas S. Chan, Nikhil Goyal, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. The Cloud is Not Enough: Saving IoT from the Cloud. In *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'15, pages 21–21. USENIX Association, 2015.

[167] Sen Zhou, Kwei-Jay Lin, Jun Na, Ching-Chi Chuang, and Chi-Sheng Shih. Supporting Service Adaptation in Fault Tolerant Internet of Things. In *Proc. of SOCA '15*, pages 65–72. IEEE, 2015.

[168] Zbigniew Zieliski, Jan Chudzikiewicz, and Janusz Furtak. *An Approach to Integrating Security and Fault Tolerance Mechanisms into the Military IoT*. Springer International Publishing, 2019.