F^{3}ARIoT: A Framework for Autonomic Resilience of IoT Applications in the Fog

Umar Ozeer[a], Loïc Letondeur[a], Gwen Salaün[b], François-Gaël Ottogalli[a], Jean-Marc Vincent[b]

[a] *Orange Labs, Meylan, France*
[b] *University Grenoble Alpes, INRIA, LIG, Grenoble, France*

Abstract

Fog Computing is especially appealing to the Internet of Things (IoT) because it provides computing, storage, and communication resources at the edge of the network, near the physical world (PW). Thus, IoT located in the PW can have interesting properties such as low latencies, real-time operations, and data privacy. The Fog, however, is unstable because it is constituted of billions of devices in a dynamic environment. Moreover, the Fog is cyber-physical and devices are thus subjected to external PW conditions which increase the occurrence of failures. When failures occur in such an environment, the resulting consequences on the PW can be hazardous and costly. This paper presents F^{3}ARIoT, a framework for autonomic resilience of IoT applications in the Fog. This framework recovers from failures as well as maintains consistency and safety with respect to the PW during the recovery procedure. F^{3}ARIoT was implemented and evaluated on a smart home application. A performance evaluation showed that it has a negligible overhead and recovers from failures in a very short delay with respect to end-users.

Key words: Fog computing, Internet of Things, Resilience, Fault Tolerance, Consistent Recovery.

1. Introduction

The Internet of Things (IoT) can be described as a network of devices (or *objects*) having a cyber-physical existence as they lie at the frontier between the digital and the physical world. The physical world refers to spaces with 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 an existence 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).

One of the major drawbacks of IoT devices 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. Fog Computing [1, 2, 3] is a continuum ranging from the IoT devices in the Physical World (PW), to the datacenters of the cloud, that provides computing, storage and communication resources. Telcos infrastructures are typical examples of this continuum. At the edge and near the IoT devices, millions of ISP boxes serve as a gateway for IoT devices and can host local tasks as described in [4]. Thousands of base stations that provide mobile connectivity, can run Fog tasks in a neighborhood perimeter as can do thousands of street hubs that provide wired connectivity (xDSL, optical fiber). In addition, some local micro-datacenters can be deployed as illustrated in [4]. Further, hundreds of Points of Presence (PoPs) interconnect networks of hubs, base stations and other PoPs at a regionnal level [5]. Therefore, the Fog does not only provide necessary resources for IoT applications, but it also meets the latencies, privacy, QoS, location awareness which the cloud fails to resolve. For this reason, the deployment of IoT applications in the Fog has become increasingly popular and finds application in many fields such as smart homes and cities, agriculture, healthcare, transportation, and many more [6].

The Fog-IoT ecosystem, however, suffers from a low stability and is prone to failures because of bulk production of devices and cheap design. Devices in the Fog are prone to many types of failures due to power (e.g., accidental unplugging, battery drain), to hardware (e.g., hardware malfunction, overheating), or to software (e.g., bugs). 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 [7]. Motion of wireless devices causes fluctuations and attenuations in the level of received signals [8]. 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.

When a software or an IoT device fails, it loses its volatile memory (i.e., its state of execution). If it is restarted/rebooted, its state of execution becomes inconsistent with the rest of the application because it processes events differently. It has consequently an unexpected and inconsistent behaviour. This can cause 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 PW by inducing unsafe and 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. In a crop irrigation application, the reboot of the application or negligent recovery procedures 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.*

The Fog-IoT environment brings new challenges regarding resilience since applications are stateful and cyber-physical. Stateful applications have, at any given time instant, internal states characterised by the execution conditions, input parameters, environment variables, stored data, etc. Since applications are cyber-physical, devices can interact with the PW and thus they can change the state of the PW. The state of the PW relates to what we can perceive around us (temperature, humidity, light intensity) and is given by sensors and actuators. Therefore, after a failure, the recovery procedure should repair the application and restore its state taking into account the changes in the context of the PW as well as the impact of the state restoration procedure on the PW. These considerations ensure that the state of the application is consistent with the state of the PW so that cyber-physical inconsistencies are avoided. However, maintaining cyber-physical consistency (CP-consistency) is complex because cyber-physical events, unlike applicative events, are non-deterministic and depend on the PW time, space, and context. For instance, the events provoking the turning off of a lamp in an office (e.g., because of an increase in light intensity at noon) may no longer be valid a few hours later. Furthermore, the output devices that interact with the PW can act on the latter in a definitive way and the state of the PW cannot be rolled back. For example, the delivery of pesticides on crops by a smart irrigation system cannot be reverted. For these reasons, care should be taken when restoring the states of devices. The technique for restoring the state (e.g., a basic replay of events) of a device may introduce intermediary states that can have undesirable or even dangerous consequences on the PW and thus break CP-consistency. For instance, replaying messages on an injection device of a patient may inject already administered doses of a drug.

In this work, we propose F^{3}ARIoT (**F**ramework **F**or **A**utonomic **R**esilience of the **F**og and **IoT**), an autonomic failure management approach for IoT applications in the Fog which maintains CP-consistency when recovering from failures. The approach focuses on Smart Home / Building environments. F^{3}ARIoT is based on four functional steps: (i) state saving, (ii) monitoring, (iii) failure notification, and (iv) recovery. Each step implements a collection of similar roles taking into account the specificities of the ecosystem (e.g., heterogeneity, resource limitations, cyber-physical interactions). *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. This is done through uncoordinated checkpoint, message logging, and function call recording techniques [9]. *Monitoring* aims at observing and reporting information on the lifecycle (e.g., restarted, stopped, failed, recovered) of the application. It is particularly useful for failure detection. 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 (e.g., drug injection, elderly people surveillance, crop irrigation, ...). F^{3}ARIoT was implemented and 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. Our evaluation showed that F^{3}ARIoT is able to recover from failures in less than one second.

To sum up, the contributions of this paper are as follows:

- A model of the Fog-IoT ecosystem is proposed. This model defines the devices constituting the Fog infrastructure and the applicative entities that run on this infrastructure.
- The design of an autonomic end-to-end failure management approach, which is composed of the following four functional steps: state saving mechanisms, monitoring for detection of failures, failure notifications for limiting the impact of failures, and recovery which involves the reconfiguration and restoration of a consistent state of the application.
- State saving policies which describe various combinations of state saving techniques in order to cope with the specificities of the Fog-IoT ecosystem.
- A recovery procedure that is consistent within the application and with respect to the physical world (i.e., cyber-physical consistency).
- The implementation of the failure management approach ($\texttt{F}^3\texttt{ARIoT}$ framework) and its evaluation on a smart home testbed.

The outline of this paper is as follows. Section 2 defines a model of IoT applications in the Fog. Section 3 describes the architecture of F^{3}ARIoT and details the different failure management steps involved in the proposed approach. Section 4 reports the implementation of F^{3}ARIoT and its evaluation on a smart home application. Section 5 discusses related work. Finally, Section 6 concludes and explores future work.

2. Models

F^{3}ARIoT relies on a model of IoT applications which is presented in this section. The model describes the characteristics of the different entities involved in the Fog-IoT ecosystem. The failure management mechanisms involved in F^{3}ARIoT are based on this proposed model in order to manage the entities of the application with respect to their characteristics. 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.

2.1. Behavioural Model

The behavioural model (BM) 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 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 Section 3.5). 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 (e.g., Bosch [10]), or be extracted from runtime contextual observations of an entity [11].

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 the reception and emission of events as well as internal computations which are independent of received/emitted events (e.g., an action executed periodically such as a lamp switching on every day at 6 p.m.). A change in state is modelled by a transition function which takes as input a state, a set of parameters and defines the invariants for each parameter.

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, a transition is denoted as $\{ms\} \xrightarrow{e,act,v_e,inv_v} \{ms\} \in T_{dbm}$.

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 states 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, a transition is denoted as $s \xrightarrow{e,act,v_e,inv_v} s' \in T_{ebm}$.

The two types of BM are illustrated in Figure 1. Figure 1 (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.

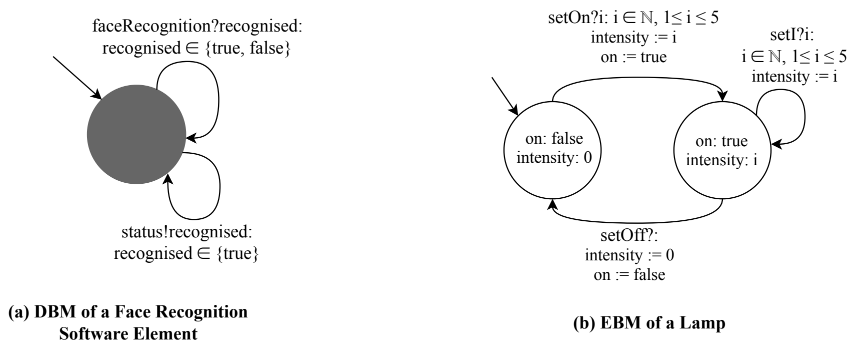

Figure 1: EBM and DBM Models

Therefore, the invariant defined for this transition is *recognised* $\in \{true\}$. Figure 1 (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 $false$.

Note that one could make use of these behavioural models for verification purposes as it is done for instance in [12, 13], but this is not the goal of this work.

2.2. Application Model

The PW plays an important role in cyber-physical IoT applications because devices such as sensors and actuators interact with the PW. A model of the PW which describes how the application interacts with PW and how it changes the state of the PW is useful 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. In the following, we first define the PW and then we define the entities involved in an IoT application.

The physical world is defined with respect to geographical spaces. These spaces are zones delimited by physical boundaries and defining 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) [14]. 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.*

An IoT application is composed of the following entities: appliances, software elements, fog nodes, and logical bindings. 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 achieved 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,* $\forall v \in V_{app},\ deg(v) \neq 0$.

2.3. 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 [15] that maps G_{app} to G_{infra}. Figure 2 illustrates the placement and deployment of an application onto an infrastructure.

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

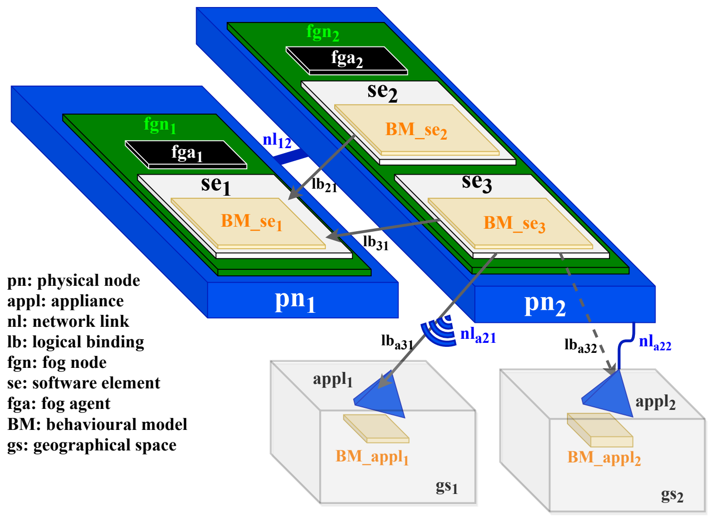

Figure 2: Mapping of the Application on the Infrastructure

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. 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} provides 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.

2.4. Failure Model

F^{3}ARIoT aims at repairing crash failures that affect both infrastructure and applicative entities. A crash failure occurs when an entity which was executing its correct behaviour deviates from its expected behaviour by halting unexpectedly. The failure model is motivated by real case failures that can be observed in the Fog-IoT environment. Devices are usually connected via wireless networks which are unstable and volatile. Physical nodes and appliances can fail because of a power failure (e.g., temporary outage, accidental unplugging, battery drain). When this happens, the hosted fog node and software elements also fail. Software elements can crash due to lack of resources for their execution, unhandled exceptions, design/development errors, etc. Figure 3 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).

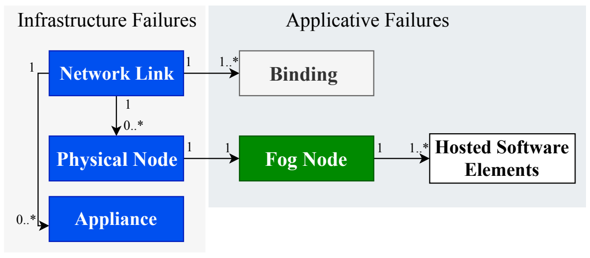

Figure 3: Causality Between Failures

3. Failure Management Approach and Architecture

The goal of failure management is to detect the occurrence of a failure and to implement a recovery procedure which suits the type of failure that occurred. This end-to-end failure management is rather complex and should be completely automated. It should not require any manual intervention in order to eliminate any human errors and increase efficiency in terms of time and cost. F^{3}ARIoT makes use of the models presented in Section 2 to provide autonomic failure management. In the following, Section 3.1 gives an overview of the steps involved in the approach. Section 3.2 details the failure managers implementing these steps as well as their architecture. Section 3.3, 3.4, and 3.5 successively presents the state saving, failure detection and notification, and recovery steps of the failure management approach.

3.1. Autonomic Failure Management

Our failure management approach is based on the principle of autonomic computing which advocates for self-management [16] of applications. The approach consists of four functional steps. Each functional step implements a set of similar roles ensured by the failure managers involved in F^{3}ARIoT. These failure managers are presented in the next subsection.

(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 [9, 17, 18, 19]. 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 constructed by the events given by sensors and actuators (see Section 3.3).

(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. Monitoring is carried out by using the following techniques: heartbeat, ping-acks, local system observation, and applicative messages observation. 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 (see Section 3.4.1).

(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 that are impacted by the 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. 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. Degraded mode 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 entity provides only part or none of its services (see Section 3.4.2).

(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 (see Section 3.5).

Figure 4 illustrates how these failure management steps are performed on an applicative entity and how it affects the application over time. When an applicative entity is running and is fully operational, two failure management steps are performed: state saving and monitoring. A failure of the applicative entity at time t_1 is detected by the monitoring step. The state saving step of the entity subsequently stops. A 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 to a degraded mode 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 re-establish 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.

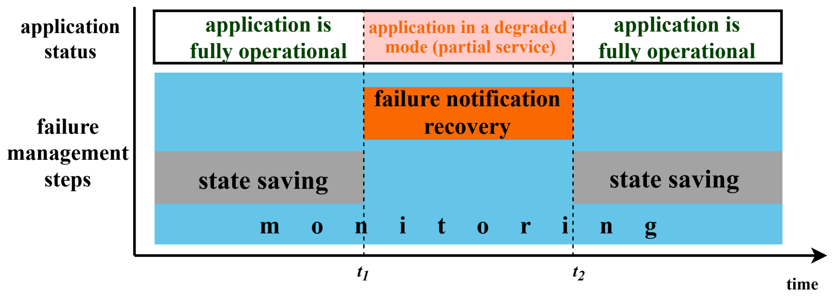
application status
application is fully operational
application in a degraded mode (partial service)
application is fully operational
failure management steps
failure notification recovery
state saving
state saving
m o n i t o r i n g
t_1
t_2
time

Figure 4: Failure Management Steps of an Applicative Entity

F^{3}ARIoT implements autonomic failure managers, which continuously retrieve information about the application for state saving and monitoring, and execute reconfigurations and recovery procedures based on the interpretation of the information retrieved. These failure managers and their architecture are presented in the next subsection.

3.2. F^{3}ARIoT Architecture

This section highlights the failure managers (FMs) involved in F^{3}ARIoT, their roles, and how they interact with each other to implement the failure management steps. They are placed and interact with each other as illustrated in Figure 5. The FMs are composed of local and global managers. Local FMs are (i) *Wrappers* which encapsulate software elements/appliances, and (ii) *Fog Agents* which set up wrappers and monitor fog nodes. Local FMs are deployed on each fog node. The global FMs are composed of the *Global Decision Manager (GDM)* which analyses the information reported by the local FMs when failures occur and plans the recovery actions to perform. F^{3}ARIoT also relies on the following global managers: the *Application Lifecycle Manager (ALM)*, a *Stable Storage (SSG)*, and *Thing'in* which is an object registry.

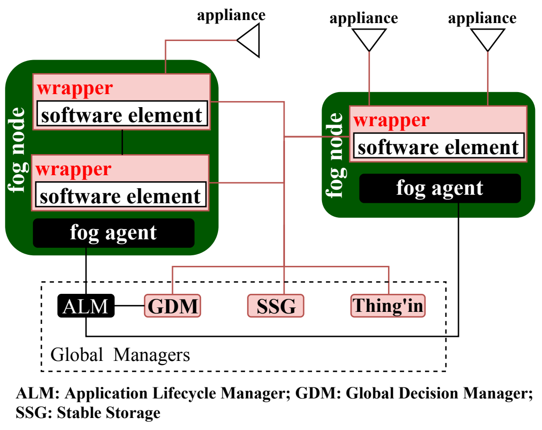

Figure 5: F^{3}ARIoT Architecture

3.2.1. Local Failure Managers

A wrapper acts as a membrane [20] by encapsulating software elements and appliances. A wrapper intercepts incoming/outgoing operations of software elements and appliances. It can also control their behaviour and perform reconfiguration operations on them. A wrapper encapsulates one local software element and its neighbouring appliances because the latter have no accessible hosting capabilities. Each entity encapsulated by the wrapper is referred to as an *encapsulation.*

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 wrap-*

per; 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 $\{se\} \bigcup APL$.

A wrapper has multiple roles involving the four failure management steps described in Section 3.1. For each encapsulation, the wrapper defines the policies for these steps:

- State saving: The wrapper defines and applies the policies for saving and storing the state of its encapsulations. These policies are based on uncoordinated checkpoint, message log, and function call record. To do so, the wrapper intercepts incoming/outgoing operations as well as piggybacks additional information on these operations as part of its state saving role. The wrapper also saves all variables and corresponding values that characterise the state of an entity as defined by its behavioural model. The state saving step is further discussed in Section 3.3.
- Monitoring: The wrapper defines and applies the monitoring policies for failure and recovery detection of its encapsulations. To do so, the wrapper monitors local software elements through local system observation to avoid influence on the network traffic. Remote encapsulations are monitored by observing applicative messages and implementing ping-acks. The monitoring step is further described in Section 3.4.1.
- 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 mode. 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. The propagation of failure notification is further discussed in Section 3.4.2.

- Recovery: The wrapper defines and applies the local rules for recovery of its encapsulations. Recovery policies describe these rules and are detailed in Section 3.5. The wrapper reconfigures (architecture and placement) its encapsulations during the recovery procedure. It also retrieves the saved data for restoring the state of its encapsulations.

Fog agents are involved in the initialisation phase of the application. They work together with the ALM for lifecycle management (setup, installing runtime, reconfigurations) of local software elements and neighbouring appliances. A fog agent has in addition two roles with respect to failure management:

- Monitoring: 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.

- 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.

3.2.2. 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. Global FMs can be deployed on the infrastructure dedicated to the application or in the Telcos' infrastructure. F^{3}ARIoT relies on the following global managers: a Stable Storage (SSG), the ALM Thing'in, and the GDM.

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 [21] such as a replicated file system on fog nodes or a RAID storage system [22]. In this work, we assume a reliable storage medium on the Telco's Fog infrastructure which is dedicated to storage of state data.

Thing'in [23] 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) [14] which gives a virtual representation of geographical spaces (e.g., a home or building) with semantic and topological 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.

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. When a failure occurs, the GDM and the wrappers propagate failure notifications to dependent entities impacted by the failure. The GDM also decides the steps for recovery (see Section 3.5). For instance, this usually involves sending requests to Thing'in to find replacement appliances or a new placement for software elements.

The following sections give a more detailed presentation of each failure management step and explain how they are carried out by the local and global FMs.

3.3. State Saving

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 data characterising the state of an entity participating in an application are given by its behavioural model. The state data are composed of (i) the variables and corresponding values characterising the state of each applicative entity, as well as (ii) the transitions the entity 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, and 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 single 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 section discusses the state saving approach, the chosen techniques for saving the state of an application, and how they are implemented in order to cope with the specificities of the Fog-IoT ecosystem.

The state saving step 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. 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 are thus given by the corresponding EBM model of the encapsulation. Message logging and function call logging consist in saving the transitions that are executed by the entity upon emission/reception of events and execution of internal/remote computations/functions. These transitions are given by the EBM or DBM model of an encapsulation.

3.3.1. State Saving Policy

A wrapper assigns a state saving policy for each of its encapsulations. This policy describes the strategy that is most suited to save the state of an encapsulation with respect to its properties and those of its runtime environment. 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. Figure 6 illustrates the structure of the description file generated for each encapsulation. It gives an overview of the different parameters of the state saving policy before describing them in more details in Section 3.3.2.

EncapsulationId gives the identifier of a software element or appliance encapsulated by the wrapper. The state saving policy defined in this description file applies to the identified encapsulation. As 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 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. In the case of software elements, **localData** indicates the files that are needed for the initialisation

```
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: Description of the State Saving Policy of an Encapsulation

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 adopted state saving technique 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 performed at reception only or at both emission and reception. *OP* indicates whether a pessimistic or an optimistic approach is adopted for logging. In the pessimistic approach, each event is logged synchronously on the stable storage before delivering it to the encapsulation. In the optimistic approach, a set of events are temporarily stored locally before sending them all at once on the stable storage. 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. **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. **VT** defines the validity time *vtValue* for each type of event *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.

A wrapper automatically generates this description file for each of its encapsulations. 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 subsection. The filesystem paths to access the local storage as well as the access to the stable storage have to be specified by the developer and/or operator (DevOps) of the application. The *Weights* and *VT* of events have also to be specified 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 paper. The following subsection further describes how the state saving techniques are chosen.

3.3.2. State Saving Techniques

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 be included 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.

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 1 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.

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 received messages, the wrapper intercepts incoming messages before delivering them to the encapsulation. Emitted messages 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.

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

		Communication Model			
		Function Call	Messaging		
Local Storage	Persistent	Optimistic Function Call Logging at Reception	Optimistic Message Logging at Reception	Software Element	**Destination Entity**
		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 1: State Saving Techniques

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.

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

Checkpoint Parameters. 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 with respect to 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 shows how checkpoints are triggered. 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 ($l.12 - l.13$).

Validity Time (VT). A wrapper also associates contextual information upon saving messages. More precisely, a wrapper associates a *validity time (vt)* for each message saved as described by the state saving policy (see Figure 6). 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

Algorithm 1 Frequency of Checkpoint based on Events Processed

```
1: Event reception by wrapper: $(encap, evt)$        ▷ $evt$ 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:     $N_{evt} \leftarrow 0$        ▷ reset $N_{evt}$
7: else if $evt.weight == 1 \mid\mid evt.weight == unknown$ then
8:     $encap.N_{evt} ++$        ▷ increment $N_{evt}$ if weight of $evt$ is 1 or is unknown
9:     if $encap.N_{evt} \geq encap.N_{ckpt}$ then
10:        $sendCheckpointRequest(encap)$        ▷ request checkpoint if $N_{evt} \geq N_{ckpt}$
11:        $N_{evt} \leftarrow 0$
12:    end if
13: else if $evt.weight == 0$ then
14:     $continue$
15: end if
```

no longer valid and should not be used in the computation of the target consistent state to restore the application. In the case of 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.

3.4. Monitoring and Failure Notification Propagation

The monitoring step of the failure management approach reports information on entities that have failed. When this happens, failure notifications are propagated to entities having dependencies on the failed entity so that they can move to a degraded mode.

3.4.1. Failure Detection

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.

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.

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 2: Monitoring Techniques by Wrappers and Fog Agents

Software Element. 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 a 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 duration the wrapper waits for a response from the encapsulated software element before suspecting its failure.

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 only on applicative message observation is not possible. In this case, the wrapper uses the same technique as 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 exchanged on the network link for monitoring purposes. Such data packets are exchanged regularly between appliances connected on the same network even when there are no applicative messages 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.

Physical/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. If T_{hbr} expires before the reception of a heartbeat message, the fog agent suspects the failure of the neighbouring fog/physical node.

When the fog agent or wrapper suspects the failure of an entity, it adds the entity to its local list of failed encapsulations. It then pushes a failure notification to the GDM. Then, if the failure is confirmed by the GDM, failure notifications should be propagated to the entities that have dependencies on the failed entity so that they can move to a degraded mode. It is worth noting that in this paper we sometimes refer to failures as failure suspicions. The GDM may confirm a failure suspicion which is actually not a failure. This false positive may be due to a transient failure. In such a case, the device will be added to the pool of available devices, and integrated again to the running application if no equivalent replacement has been found.

3.4.2. Failure Notification Propagation

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 mode. 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. In a degraded mode, an encapsulation provides only part or none of its services. Thus, failure notification avoids making dependent entities unresponsive, generating the re-emission of events that cannot be delivered because of the failure, and creating a cascading failure where the whole application fails [24].

Failure notifications are propagated as illustrated in Figure 7. The failure notification received by a wrapper wrp_1 can come from another wrapper wrp_2 (i.e., if one of its encapsulations has failed or one of its encapsulations is affected by a failure) or from the GDM in the case of a physical node failure (i.e., because in this case, the wrapper also has failed). Therefore, when a wrapper receives a failure notification that impacts one of its encapsulations, it behaves as follows:

- If there are no other entities that are dependent on the impacted encapsulation, then the failure notification is not further propagated. 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 impacted encapsulation has dependent neighbours, 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 Section 3.5.

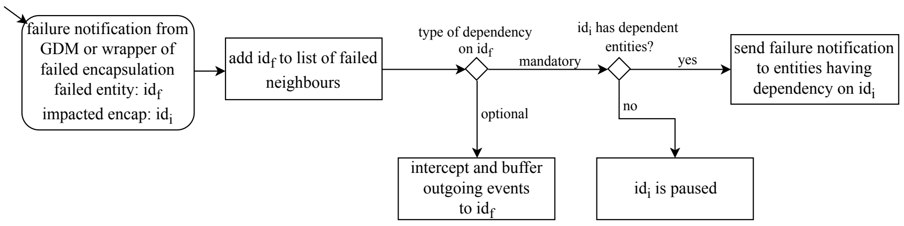
failure notification from GDM or wrapper of failed encapsulation failed entity: id_f impacted encap: id_i
add id_f to list of failed neighbours
type of dependency on id_f
mandatory
optional
id_i has dependent entities?
yes
no
send failure notification to entities having dependency on id_i
intercept and buffer outgoing events to id_f
id_i is paused

Figure 7: Reception of Failure Notification from GDM or Wrapper of Failed Encapsulation

3.5. Recovery

The recovery procedure repairs the failed entity and restores the state of the application. As soon as an entity has failed ①, the GDM plans the recovery actions to implement as illustrated in Figure 8. Failure notifications ② are first sent to dependent applicative entities as discussed in the previous section. The recovery procedure relies on two main steps: (i) Architectural Reconfiguration where the architecture of the application is rebuilt ⑤, and, (ii) State Restoration where a CP-consistent state of the application is restored ⑥. CP-consistency aims at restoring a consistent state of the application with respect to the PW so that dangerous and costly situations are avoided. To reestablish consistency, we reuse the state saved by state saving mechanisms, we reset the failed entity with that state, and we finally start the application from the same state where it was before failure. In the worst case (e.g., failed applicance that cannot be replaced), the application keeps working in degraded mode. Note that the mechanisms implemented during these two steps depend on the type of entity that has failed.

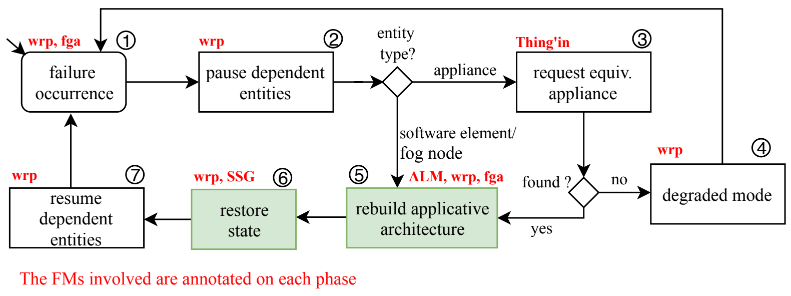

Figure 8: Global Decision Manager Recovery Workflow

3.5.1. Architectural Reconfiguration

Architectural reconfiguration ⑤ involves the rebuilding of the applicative architecture. In the case of a software element failure, architectural reconfiguration may be as simple as a restart of the software element by the wrapper. Architectural reconfiguration is more complex in the case of a fog node failure. In this case, the ALM determines a failure-free fog node to host the set of failed software elements. When an appliance fails, it is replaced by another functionally equivalent one that meets the geographical constraints of the failed one. 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 functionally equivalent appliance is given by Thing'in ③ and can be a *temporary* or a *permanent* replacement according to the set of services provided by the replacement compared to the failed one. If no replacement appliance is found, architectural reconfiguration fails and the application has to continue its execution with less features in a degraded mode ④.

3.5.2. State Restoration and CP-Consistent Recovery

Once architectural reconfiguration is completed, the saved states of the failed entities are used to restore a CP-consistent state of the application. 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 15. *(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.*

The technique for state restoration is chosen according to the data saved during the state saving phase. For instance, state restoration technique usually involves replaying a set of saved messages after a checkpoint. However, during the state restoration process, the entity transits through a set of intermediary states before reaching the target state. When transiting through intermediary states, a software element can re-emit messages already delivered. In the case of an actuator, the device will act on the PW and therefore change its state through each intermediary state. This result in a violation of CP-consistency.

In order to ensure CP-consistency, a wrapper defines a recovery policy for each of its encapsulations. The recovery policy (a) makes use of the EBM of the encapsulated appliances and validity times of operations for CP-consistent state restoration, (b) makes use of the EBM or DBM of software elements for state restoration and indicates the outgoing events that should be intercepted and discarded during this procedure, (c) specifies the *number of restart trials (nrt)* of a software element on the local fog node before requesting a new placement and redeployment (default value is set to 3), (d) defines an *external administrator (extAdmin)* where a notification is pushed when an appliance or a physical node fails (this is particularly useful to notify service providers or care-givers in smart homes for medicated people).

In order to illustrate CP-consistent recovery, let us consider the saved state of an entity 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 procedure based on an EBM is illustrated in Figure 9 (a). E is first processed by a filter function so that the events having an expired validity time are discarded and the remaining sequence of non-expired transitions T_{replay} is returned as output. The target state s_t is determined by the EBM by executing the transitions in T_{replay}. The entity can then be set to this target state avoiding intermediary states. In the case of appliances, this procedure maintains CP-consistency by avoiding the impacts of intermediary states that may make the PW unsafe. Since software elements do not interact with the PW, this procedure provides an optimisation of state restoration because it avoids the replay of the set of operations on the software element that can be computation intensive as well as avoid the re-emission of already delivered events.

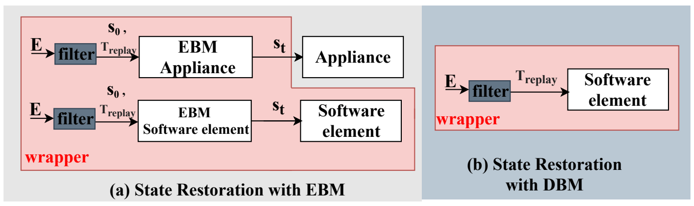

Figure 9: State Restoration using BM

In the case of failures of software elements characterised by a DBM, the details of the state space are unknown. The set of events are thus replayed directly on the software element, after filtering expired events if they are known, as illustrated in Figure 9 (b). The outgoing events emitted during the state restoration are considered as obsolete and are not delivered again. They are intercepted by wrappers and discarded according to each recovery policy.

3.6. Ensuring Service Continuity

The EBM of appliances aims at achieving CP-consistency, but also to preserve the continuity of the service provided by the replacement appliance after state restoration. The data available to restore the state of a replacement appliance may have been produced by another appliance, albeit functionally equivalent, having different API implementations or data formats. This is the case of appliances produced by different manufacturers (e.g., Awox and Philips Hue lamps). In this case, the target state, s_t, is first computed by the EBM of the failed appliance. Then, the corresponding equivalent state, s'_t, in the EBM of the replacement appliance is determined according to the ontological equivalences computed by Thing'in. Once s'_t is determined, the set of transitions to reach this target state from an initial state can be computed from the EBM model of the replacement appliance and therefore it can be restored in the state s'_t. After state restoration, incoming events intended for the failed appliance can be dynamically adapted before being transferred to the replacement appliance. Algorithm 2 illustrates the part of the behaviour of the wrapper to ensure the continuation of the services of a replacement appliance.

Algorithm 2 Ensuring Service Continuity of 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
 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:             end if
16:         else                              ▷ encapsulated by different wrapper
17:             $send(aplRepl.wrp, e_{apl})$          ▷ redirect event to wrapper of aplRepl
18:         end if
19:     else                                  ▷ event deleted if no replacement apl found
20:         $delete(e_{apl})$
21:     end if
22: end if
```

Several cases are identified depending on the types of APIs implemented by the appliances 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 keeps working in a degraded mode and offers fewer features.

4. Implementation and Evaluation 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 the previous section: 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) [4] and the Object Registry (Thing'in) [23] are provided by other platforms at Orange Labs.

This section aims at presenting the implementation of F^{3}ARIoT and its evaluation on a smart home application. To this end, Section 4.1 details how the framework is implemented and configured. Section 4.2 reports the experimental environment. Section 4.3 describes the results of this evaluation.

4.1. Configuration of the F^{3}ARIoT

Each failure manager (GDM, fog agent, wrapper) 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 (e.g., MQTT or Socket.IO [25]) with the other failure managers. The configuration file of the fog agent specifies the configuration for monitoring neighbouring physical nodes. For instance, it defines the frequency of emitting and receiving heartbeat messages to/from its neighbours. These values are 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. The configuration file of wrappers defines the relevant information (e.g., state saving policy, recovery policy, BM) about its encapsulations. For example, the configuration file of a wrapper is composed of four components as illustrated in Figure 10. *Architecture* defines the local and neighbouring architecture, that is, 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. The wrapper generates a configuration file for each encapsulation which details these four components.

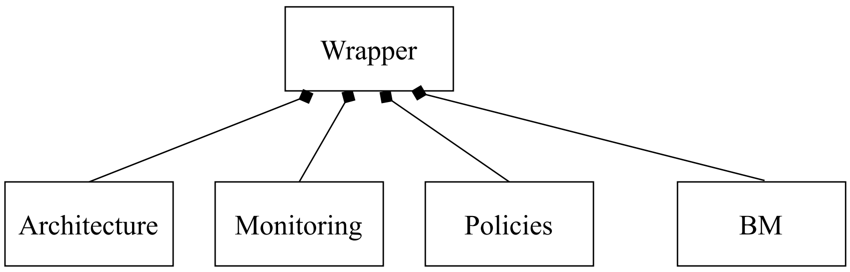

Figure 10: Components of Configuration Files of Wrappers

4.2. Experimental Environment

The target experimental environment for evaluating F^{3}ARIoT is a smart home application inspired from an industrial project at Orange Labs [4]. The testbed is composed of infrastructure and applicative entities that can be found in real-life smart homes. For instance, the *Connected Home Service* [26, 27] by Orange includes the appliances involved in this testbed. The smart home use case application is light automation and physical intrusion detection.

Figure 11 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.

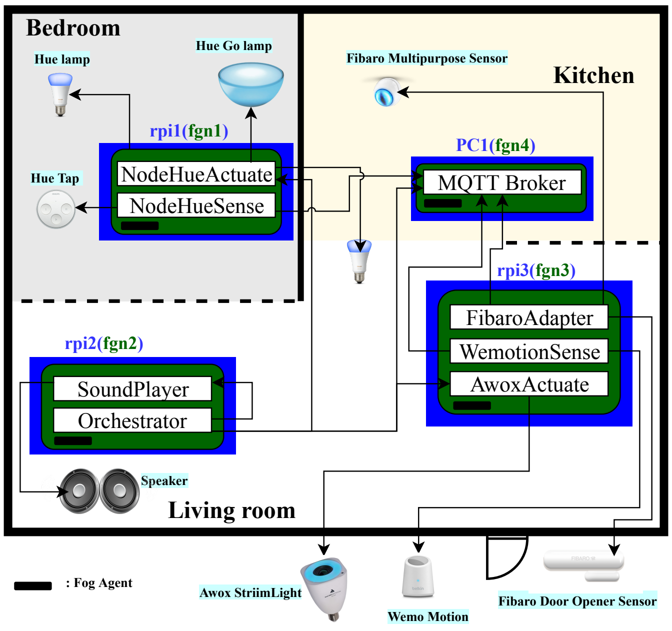

Figure 11: Smart Home Testbed

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 (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 [28].
- *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 pressed button number 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 Striim-Light 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. A set of user stories corresponding to this use case are identified. 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 set of user stories was 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 based on indoor light automation where the Hue Tap buttons are used to control the Hue lamps. The Hue Tap device is composed of four buttons to turn on/off the lamps with different intensities. This 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.

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 wrapper encapsulates a software element and the neighbouring appliances. Table 3 summarises the wrappers and their encapsulated software element and appliances. For instance the wrapper *wrp1* encapsulates the software element NodeHueActuate as well as both the 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 SE**	**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 3: 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.

4.3. Performance Evaluation

This section presents 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.

To do so, we developed a *Random Failure Injector (RFI)* which randomly injects failures on the application at a specified frequency. A *Measurement Logger (ML)* tool was also developed for logging information during the experiments. The first experimental campaign aims at evaluating F^{3}ARIoT for single software element failure at a given frequency. The RFI is thus 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 12 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, SpeakerActuate, 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.

Complementarily to these data, we have also collected minimum and maximum times during our experiments. The software element which restarted and was functional with a minimum delay is se1 (422ms). The software element with the observed maximum time to restart is se7 (1141ms). The minimum times to restore the state of the software elements range from 15ms (for se2) to 23ms (for se6). The observed maximum times to restore range from 118ms (for se2) to 271ms (for se6). These ranges of values are

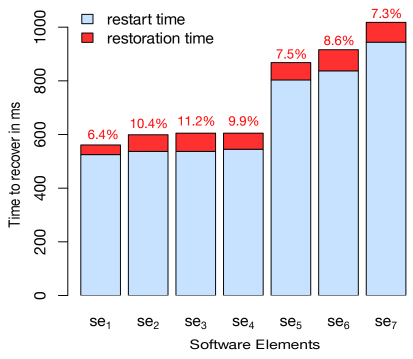

Figure 12: Time to Recover Software Elements Failures

mainly due to the instability of wireless networks involved in such environments.

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 work.

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 13. 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 12. This increase accounts for the repeated solicitation of the stable storage within small time intervals.

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

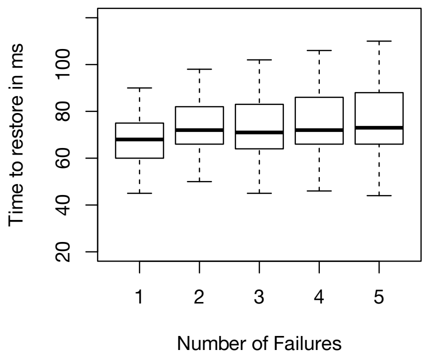

Figure 13: Time to Restore State of Software Elements for Multiple Failures

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 14 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.

The performance evaluation presented in this section aimed at evaluating whether the recovery procedure is done in an acceptable delay with respect to end users. According to [29, 30], one second is the limit in response time for the user's flow to stay uninterrupted. Moreover, [30] estimates that in case of failures, recovery should be done in less than 15 seconds 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 one second. The time for restoring a CP-consistent state of appliances is achieved in less than 200ms. To sum up,

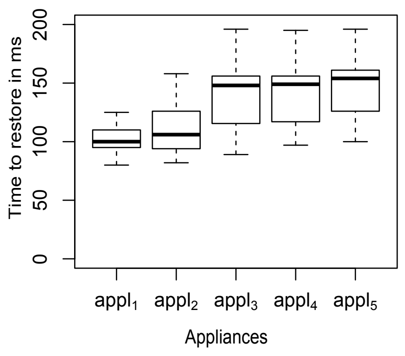

Figure 14: Time to Restore State of Appliances

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.

5. Related Work

This section focuses on recent failure management solutions for the fog and IoT applications.

Rivulet [31] 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 [32] 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 sometimes 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.

[33] 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 caregivers/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.

[34] 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 execution 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.

[35] 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 [36]. 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.

Other fault tolerance studies have been focussing on the reliability of data reported by devices. For instance, in [37] the authors designed a framework for fog devices in healthcare to ensure reliable data transmission between storage nodes and processing nodes through diffusion algorithms and routing on multiple network links. The reliable transmission mechanism recollect lost or inaccurate data automatically. [38] focuses on incorrect data reported. The authors propose mechanisms for identifying whether a sensor is transmitting faulty data for example because of hardware malfunction by cutting its power off and analysing its voltage response. Tolerating only data faults and reliable transmission are not enough to provide reliable services in the Fog-IoT context and does not address application resiliency and CP-consistency. [39] studies fault prediction which provides the means to ensure safe operations of IoT applications as well as to anticipate maintenance. [40] proposes replicated services for fault recovery in IoT. This approach requires multiple devices that have identical services. On the other hand, [41] introduces virtual service composition in which data from multiple sensors of different modalities are used as fault tolerant backups for each other. [42] 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 [43, 44, 45, 46, 47, 48, 49, 50]. 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 [51]. 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.

A preliminary version of the current paper was published in [blind] and is extended here with a detailed presentation of the models, of the fours steps of the failure management approach, of the architecture, and of the safe recovery approach that maintains cyber-physical consistency.

To summarize, our failure management approach resolves the following key challenges of the Fog-IoT ecosystem:

- Cyber-Physical Consistency: Since the Fog-IoT ecosystem is cyber-physical, our failure management approach recovers the application in a consistent state with respect to the PW. This means that the approach takes into account the changes in the state of the PW upon implementing a recovery procedure. In this way, a recovery procedure which is safe for the PW is implemented and costly situations in the PW are avoided.

- Disruptions: Our approach limits the disruptions caused by the approach itself during monitoring of the application and during the actions implemented for recovery. The monitoring step limits the interference with the execution of the application. Moreover, the recovery procedure implemented avoids impacting the failure-free part of the application.

- Heterogeneity: The Fog-IoT ecosystem is highly heterogeneous. Thus our approach relies on the integration of multiple techniques that are suited to the properties of the infrastructure and applicative entities participating in the application.
- Cost: The Fog-IoT ecosystem is limited in terms of processing, storage, and communication capabilities. Therefore, our failure management approach avoids techniques based on replication and makes use of infrastructure entities with higher capabilities for the deployment failure managers.

6. Conclusion

This paper has presented F^{3}ARIoT, 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, and maintains a consistent behaviour of the application after recovery. The failure management approach is based on four functional steps. State saving aims at saving the data from which the state of the application can be resumed. The approach monitors both infrastructure and applicative entities for failure detection. When a failure is detected, failure notifications are propagated to the impacted part of the application so that they can move into a degraded mode. In order to recover, an architectural reconfiguration is performed and the data stored during the state saving phase are used to restore a CP-consistent state of the application. Maintaining CP-consistency is important as it avoids dangerous and costly situations upon recovery. It is achieved by defining recovery policies and exploiting the behavioural models of entities during state restoration. This ensures that intermediary states are avoided and that the state of the physical world after recovery is consistent with the pre-failure state. The approach was implemented as a framework called F^{3}ARIoT and was deployed on a smart home application based on an industrial project at Orange Labs. The results of this evaluation showed that the framework recovers from failures in a CP-consistent way and the recovery time is bounded by one second.

Future work includes the description of how to make the global failure managers in F^{3}ARIoT reliable. This can be based on a recursive design where the failure managers monitor each other and implement the recovery procedures. The approach can also be made pro-active by implementing predictive maintenance so that the occurrence of a failure and its impact may be avoided. Some parameters (such as validity time of events and *weights* of events) have to be manually filled in by the DevOps. These parameters could be automatically computed. For instance, the validity times of events could be inferred from the type of the event and from the frequency at which the event is sent. Finally, in order to keep good performance 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.

References

[1] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog Computing and its Role in the Internet of Things, in: Proc. of MCC'12, Proc. of MCC'12, ACM, 2012, pp. 13–16.

[2] D. Roca, R. Milito, M. Nemirovsky, M. Valero, Tackling IoT Ultra Large Scale Systems: Fog Computing in Support of Hierarchical Emergent Behaviors, Springer, 2018, pp. 33–48.

[3] L. M. Vaquero, L. Rodero-Merino, Finding your Way in the Fog, ACM SIGCOMM Computer Communication Review 44 (5) (2014) 27–32.

[4] L. Letondeur, F.-G. Ottogalli, T. Coupaye, A Demo of Application Lifecycle Management for IoT Collaborative Neighborhood in the Fog, in: IEEE Fog World Congress, IEEE Fog World Congress, IEEE, 2017, pp. 1–6.

[5] Y. Xia, X. Etchevers, L. Letondeur, T. Coupaye, F. Desprez, Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed IoT applications in the fog, in: Proc. of SAC'18, ACM, 2018, pp. 751–760.

[6] P. Hu, S. Dhelim, H. Ning, T. Qiu, Survey on Fog Computing: Architecture, Key Technologies, Applications and Open Issues, Journal of Network and Computer Applications 98 (2017) 27–42.

[7] T. Watteyne, S. Lanzisera, A. Mehta, K. S. J. Pister, Mitigating Multipath Fading through Channel Hopping in Wireless Sensor Networks, in: 2010 IEEE International Conference on Communications, Proc. of the International Conference on Communications, IEEE, 2010, pp. 1–5.

[8] K. Woyach, D. Puccinelli, M. Haenggi, Sensorless Sensing in Wireless Networks: Implementation and Measurements, in: 4th Int. Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks, 4th Int. Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks, 2006, pp. 1–8.

[9] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, A Survey of Rollback-recovery Protocols in Message-passing Systems, ACM Comput. Surv. 34 (3) (2002) 375–408.

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

[11] J. W. Nimmer, M. D. Ernst, Automatic Generation of Program Specifications, SIGSOFT Softw. Eng. Notes 27 (4) (2002) 229–239.

[12] A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, G. Salaün, Rigorous Design and Deployment of IoT Applications, in: Proc. of FormaliSE'19, 2019, pp. 21–30.

[13] A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, G. Salaün, IoT Composer: Composition and Deployment of IoT Applications, in: Proc. of ICSE'19, IEEE / ACM, 2019, pp. 19–22.

[14] R. Volk, J. Stengel, F. Schultmann, Building Information Modeling (BIM) for Existing Buildings - Literature Review and Future Needs, Automation in Construction 38 (2014) 109 – 127.

[15] A. Brogi, S. Forti, C. Guerrero, I. Lera, How to Place Your Apps in the Fog - State of the Art and Open Challenges (2019).

[16] J. O. Kephart, D. M. Chess, The Vision of Autonomic Computing, Computer 36 (1) (2003) 41–50.

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

[18] K. M. Chandy, L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, ACM Trans. Comput. Syst. 3 (1) (1985) 63–75.

[19] R. Strom, S. Yemini, Optimistic Recovery in Distributed Systems, ACM Trans. Comput. Syst. 3 (3) (1985) 204–226.

[20] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The FRACTAL Component Model and Its Support in Java: Experiences with Auto-adaptive and Reconfigurable Systems, Softw. Pract. Exper. 36 (11-12) (2006) 1257–1284.

[21] B. Lampson, H. E. Sturgis, Crash Recovery in a Distributed Data Storage System, Tech. rep., Xerox Palo Alto Research Center (1979).

[22] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, D. A. Patterson, RAID: High-performance, Reliable Secondary Storage, ACM Comput. Surv. 26 (2) (1994) 145–185.

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

[24] F. Montesi, J. Weber, Circuit Breakers, Discovery, and API Gateways in Microservices, ArXiv.

[25] Socket.IO Website, `https://www.npmjs.com/package/socket.io`.

[26] Orange Maison Connectée, `https://boutique.orange.fr/maison/domotique/`.

[27] Orange Maison Protégée, `https://boutique.orange.fr/telesurveillance`.

[28] Mosquitto Website, `https://mosquitto.org/`.

[29] S. K. Card, G. G. Robertson, J. D. Mackinlay, The Information Visualizer, an Information Workspace, in: Proc. of CHI'91, Proc. of CHI'91, ACM, 1991, pp. 181–186.

[30] R. B. Miller, Response Time in Man-Computer Conversational Transactions, in: Proc. of AFIPS '68 (Fall, part I), Proc. of AFIPS '68 (Fall, part I), ACM, 1968, pp. 267–277.

[31] M. S. Ardekani, R. P. Singh, N. Agrawal, D. B. Terry, R. O. Suminto, Rivulet: A Fault-tolerant Platform for Smart-home Applications, in: Proc. of Middleware'17, Proc. of Middleware'17, ACM, 2017, pp. 41–54.

[32] N. Mohamed, J. Al-Jaroodi, I. Jawhar, Towards Fault Tolerant Fog Computing for IoT-Based Smart City Applications, in: 9th Annual Computing and Communication Workshop and Conference

(CCWC'19), 9th Annual Computing and Communication Workshop and Conference (CCWC'19), IEEE, 2019, pp. 752–757.

[33] T. N. Gia, A.-M. Rahmani, T. Westerlund, P. Liljeberg, H. Tenhunen, Fault Tolerant and Scalable IoT-Based Architecture for Health Monitoring, in: IEEE SAS, IEEE SAS, IEEE, 2015, pp. 1–6.

[34] S. Guillet, B. Bouchard, A. Bouzouane, Safe and Automatic Addition of Fault Tolerance for Smart Homes Dedicated to People with Disabilities, in: Trends in Ambient Intelligent Systems, Springer, 2016, pp. 87–116.

[35] J. P. G. Sterbenz, Smart City and IoT Resilience, Survivability, and Disruption Tolerance: Challenges, Modelling, and a Survey of Research Opportunities, in: 9th International Workshop on Resilient Networks Design and Modeling (RNDM), 9th International Workshop on Resilient Networks Design and Modeling (RNDM), 2017, pp. 1–6.

[36] D. P. Abreu, K. Velasquez, M. Curado, E. Monteiro, A resilient Internet of Things architecture for smart cities, Annals of Telecommunications 72 (1).

[37] K. Wang, Y. Shao, L. Xie, J. Wu, S. Guo, Adaptive and Fault-tolerant Data Processing in Healthcare IoT Based on Fog Computing, IEEE Transactions on Network Science and Engineering (2018) 1–11.

[38] T. Chakraborty, A. U. Nambi, R. Chandra, R. Sharma, M. Swaminathan, Z. Kapetanovic, J. Appavoo, Fall-curve: A Novel Primitive for IoT Fault Detection and Isolation, in: Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys'18), Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys'18), ACM, 2018, pp. 95–107.

[39] X. Xu, T. Chen, M. Minami, Intelligent Fault Prediction System based on Internet of Things, Computers & Mathematics with Applications 64 (5) (2012) 833 – 839, advanced Technologies in Computer, Consumer and Control.

[40] P. H. Su, C. Shih, J. Y. Hsu, K. Lin, Y. Wang, Decentralized Fault Tolerance Mechanism for Intelligent IoT/M2M Middleware, in: IEEE World Forum on Internet of Things (WF-IoT'14), IEEE World Forum on Internet of Things (WF-IoT'14), 2014, pp. 45–50.

[41] S. Zhou, K.-J. Lin, J. Na, C.-C. Chuang, C.-S. Shih, Supporting Service Adaptation in Fault Tolerant Internet of Things, in: Proc. of SOCA'15, Proc. of SOCA'15, IEEE, 2015, pp. 65–72.

[42] Z. Zieliski, J. Chudzikiewicz, J. Furtak, An Approach to Integrating Security and Fault Tolerance Mechanisms into the Military IoT, Springer, 2019, pp. 111–128.

[43] C. H. Tseng, Multipath Load Balancing Routing for Internet of Things, Journal of Sensors (2016) 1–8.

[44] Q. Le, T. Ngo-Quynh, T. Magedanz, RPL-based Multipath Routing Protocols for Internet of Things on Wireless Sensor Networks, in: International Conference on Advanced Technologies for Communications (ATC'14), International Conference on Advanced Technologies for Communications (ATC'14), 2014, pp. 424–429.

[45] S. Misra, A. Gupta, P. V. Krishna, H. Agarwal, M. S. Obaidat, An Adaptive Learning Approach for Fault-Tolerant Routing in Internet of Things, in: IEEE Wireless Communications and Networking Conference (WCNC'12), IEEE Wireless Communications and Networking Conference (WCNC'12), 2012, pp. 815–819.

[46] J. Chiu, A. Liu, C. Liao, Design the DNS-Like Smart Switch for Heterogeneous Network Base on SDN Architecture, in: International Computer Symposium (ICS'16), International Computer Symposium (ICS'16), 2016, pp. 187–191.

[47] J. Bai, Y. Sun, C. Phillips, CRRP: A Cooperative Relay Routing Protocol for IoT Networks, in: 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC'16), 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC'16), IEEE, 2016, pp. 1–6.

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

[49] M. Sha, D. Gunatilaka, C. Wu, C. Lu, Empirical Study and Enhancements of Industrial Wireless Sensor-Actuator Network Protocols, IEEE Internet of Things Journal 4 (3) (2017) 696–704.

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

[51] G. Kakamanshadi, S. Gupta, S. Singh, A Survey on Fault Tolerance Techniques in Wireless Sensor Networks, in: International Conference on Green Computing and Internet of Things (ICGCIoT'15), International Conference on Green Computing and Internet of Things (ICGCIoT'15), 2015, pp. 168–173.