Verification of a Failure Management Protocol for Stateful IoT Applications

Umar Ozeer[1], Gwen Salaün[2], Loïc Letondeur[1], François-Gaël Ottogalli[1], and Jean-Marc Vincent[2]

[1] Orange Labs, Meylan, France
[2] University of Grenoble Alpes, LIG, CNRS, Inria, France

Abstract. Fog computing provides computing, storage and communication resources at the edge of the network, near the physical world. Devices deployed in the Fog have interesting properties such as short delays, responsiveness, optimised communications and privacy. However, these devices have low stability and are prone to failures. Thus, there is a need for management protocols to tolerate failures of IoT applications in the Fog. We propose a failure management protocol which recovers from failures of devices and software elements involved in an IoT application. Designing such highly distributed management protocols is a difficult and error-prone task. Therefore, the main contribution of this paper is the formal specification and verification of this failure management protocol. Formal specification is achieved using a process algebraic language. The corresponding formal model was used to carry out extensive analysis of the protocol to ensure that it preserves important architectural invariants and functional properties. The verification step was performed using model checking techniques. The analysis of the protocol was a success because it allowed us to detect and correct several issues in the protocol.

1 Introduction

Fog Computing provides computing, storage and communication resources at the edge of the network, in proximity to the physical world. The Fog thus meets the latencies, privacy, QoS and geographical requirements of IoT applications that the cloud fails to resolve. For this reason, the deployment of IoT applications in the Fog has become increasingly popular and finds application in a wide range of sectors such as smart cities, agriculture, mining, healthcare or transportation.

There is a need of resilient IoT applications because devices in the Fog are unstable and prone to many types of failures like power failure (accidental unplugging, battery drain), hardware failures (due to external environmental conditions or wear-out), or software failures. These devices are also often connected through wireless networks which can be volatile and unstable because of connectivity issues. However, the design of a failure management protocols for IoT applications in the Fog is difficult because of the main characteristics of this environment, which is dynamic with entities joining and leaving the application, and heterogeneous in terms of hardware, software, network and communication models. Moreover, since IoT applications are geographically distributed and large-scale, a distributed management protocol is required.

This paper first introduces a failure management protocol for IoT applications. An IoT application consists of interacting appliances (i.e., sensors and actuators) and software elements hosted on Fog Nodes. This protocol provides a software solution to monitor the application and react to occurring failures in order to repair and restore a consistent pre-failure state of the application. It is worth noting that we focus on stateful IoT applications here, that is, applications involving devices and software that have internal states characterised by execution conditions, input parameters, environment variables and stored data resulting from computations and external interactions. These must be taken into account to restore the state of the application. Several managers are required to implement this protocol: a Stable Storage that keeps track of the state of the distributed elements of the application, a Global Manager which is notified in case of failure and that guides the recovery process, and Fog agents associated to Fog Nodes and appliances that are locally in charge of detecting and propagating failures as well as applying local recovery strategy to repair a failed entity.

Designing this protocol is very difficult particularly due to its distributed nature. Moreover, this protocol must respect important correctness properties (e.g., a failed software entity must eventually be restarted and restored, when a failure occurs every dependent entity must be paused until the failed entity has recovered, the state of a failed entity after restoration should be identical to its pre-failure state, etc). Thus, we decided to use formal techniques and tools to ensure that the protocol works as expected and respects the aforementioned properties. We chose to use LNT [5], a modern language for formally specifying concurrent systems, and model checking techniques [12] for analysing the specification with respect to the temporal properties to be preserved. These specification and verification steps were very useful for clarifying several questions as well as identifying and correcting some issues in the protocol. The main contribution of this paper is to report on the specification and analysis of the protocol using formal modelling languages and model checking techniques. The verification of the protocol allowed us to detect several issues and correct them in its implementation developed by Orange Labs.

The rest of this paper is organised as follows. Section 2 introduces the failure management protocol with a focus on the recovery phase. Section 3 presents the LNT specification of the protocol, the properties to be satisfied by the protocol, and the results of the verification including the detected issues. Section 4 compares our approach to related work and Section 5 concludes the paper.

2 Failure Management Protocol

This section presents successively the model for describing an IoT application, the managers implementing the failure management protocol, the four phases of the protocol in a nutshell, and the recovery phase in more details. Since the recovery phase is the most important and complex phase of the protocol, we decided to focus the specification and verification effort (Section 3) on that phase. The whole framework is called F^{3}ARIoT, and has been developed as a research prototype at Orange Labs. More details about the description of the protocol can be found in [20].

2.1 Application Model

An *IoT application* (application for short) is composed of the following constituents:

- *Software Elements* (SEs) are units of software to be executed. They participate in the execution of the application through their corresponding functions. A software element has an internal state and exposes a set of functional interfaces as well as non-functional interfaces for its administration.
- *Appliances* correspond to physical objects (e.g., switch, light, sensor, etc.) with no hosting capabilities. They are only accessible through their exposed API.
- *Fog Nodes* host the software elements, that is, they give access to physical resources and provide the runtime environment for the execution of software elements. Physical resources are usually restrained (e.g., Raspberry Pi or Arduino).
- *Logical Bindings* are abstractions of the communication models, which allow a couple of software elements or a software element and an appliance to interact together (two appliances cannot depend one of another).

An Application is modelled as a directed acyclic graph where each vertex represents a software element or an appliance, and each edge represents a binding. The direction of an edge gives the functional dependency between two vertices. If a vertex v_1 depends on a vertex v_2, then v_1 requires v_2 to be functional.

Note that we assume that all elements are uniquely identified and that applicative entities (software elements and appliances) are described using a behavioural model. Such a model defines the operations possible on that software or device as well as the order in which these operations must be triggered. As an example, a light can be turned on and off infinitely. In this work, we use Labelled Transition System for describing behavioural models. An LTS is a tuple (S, A, T, s_0), where S is a set of states, A is a set of operations associated to transitions, $T \subseteq S \times A \times D \times S$ is the transition relation, and $s_0 \in S$ is the initial state. A transition is defined by a tuple $(s_s, op, dir, s_t) \in T$ where s_s is the source state, op is the operation name, dir is the direction (either send operation with "!" or receive operation with "?"), and s_t is the target state.

Failures. As far as failures of the application are concerned, we consider in this work crash failures of software elements, appliances, and Fog Nodes. Therefore, a software element and an appliance can be in three different states in its lifecycle:

- Running: It executes its behaviour according to its behavioural model.
- Failed: It deviates from its correct behaviour and crashes. It stops sending and receiving messages as well as performing computations.
- Paused: The execution of its behaviour is temporarily stopped by the management protocol. It can later resume its activities when indicated.

The failure of a software element is different from that of an appliance in the sense that the former can always be restarted which is not the case for the latter. If the appliance has permanently failed and cannot be rebooted, it should be replaced by a functionally equivalent one. If no such appliance exists, the application can work in degraded mode, providing only a partial service. Note that the failure of a Fog Node induces the failure of all software elements hosted on the Fog Node.

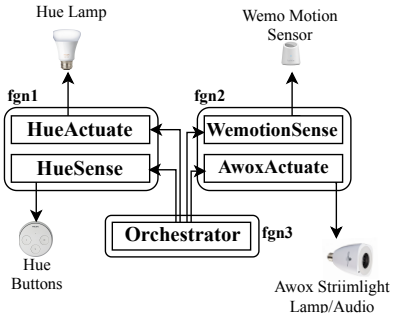

Fig. 1. Smart Home Application

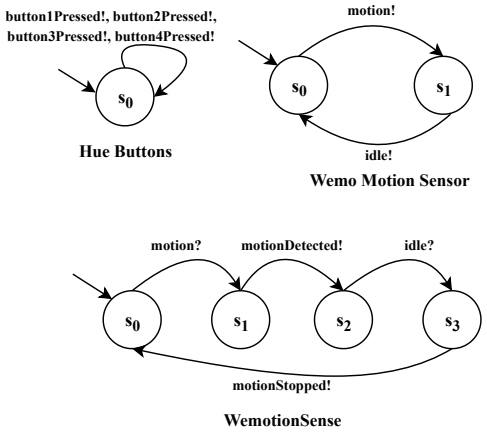

Fig. 2. Behavioural Models

Application example. Figure 1 introduces a simplified version of a smart home application for light automation and physical intrusion detection developed at Orange Labs. It is composed of four appliances, three Fog Nodes (fgn1, fgn2, fgn3) on which the software elements are executed, and eight bindings. The appliance Hue Buttons controls the Hue Lamp. The Hue Buttons are also used to set an alarm for intrusion detection. If a motion is detected by the Wemo Motion Sensor after setting the alarm, a warning sound is triggered on the Awox Striimlight and all the lamps are turned on. The Orchestrator triggers these scenarios based on the sensed events from the software elements HueSense and WemotionSense. Events are then pushed to the software elements AwoxActuate and HueActuate according to the defined scenarios.

Figure 2 shows the behavioural models of two appliances, Hue Buttons and Wemo Motion Sensor, and of the software element WemotionSense. The Hue Buttons device sends an event corresponding to the button pressed (four possible buttons). There is no specific state for that appliance and all four events are possible at any time. This explains why the behavioural model consists of a single state. In contrast, the Wemo Motion Sensor has two states to distinguish when motion is detected and then stopped. The software element WemotionSense has four states: It receives motion events and forwards them to the Orchestrator. Software elements may exhibit more complex behaviours. The behavioural model of the Orchestrator for instance consists of ten states and fifteen transitions as it synchronises with all the other software elements.

2.2 Protocol Managers

This section describes the managers that compose the failure management protocol. A *Stable Storage* (SS) is used for persisting data of all entities involved in an application. We assume that this storage is unaffected by applicative failures and its implementation can take various forms such as replicated or redundant file systems [17]. A *Global Manager* (GM) is a control and decision making entity that has a global view on the

application. It is notified when failures occur and it keeps a record of failed entities. It also guides the Fog Agents (see below) during the recovery process. The GM can also request available appliances from a device/object registry in order to replace a failed appliance. Each Fog Node and each appliance is managed by a *Fog Agent* (FA). This local manager provides an entry point for managing the lifecycle of the locally hosted software elements and appliances. A Fog Agent is in charge of local failure management. In particular, a Fog Agent detects failures, interacts with the GM for failure recovery and with the Stable Storage to retrieve the data necessary for the recovery phase.

Figure 3 gives an overview of the aforementioned managers and the way they interact together. We use a different syntax for distinguishing communication channels between Fog Agents and Global Manager / Stable Storage. In the rest of this paper, we assume all managers interact using synchronous communication.

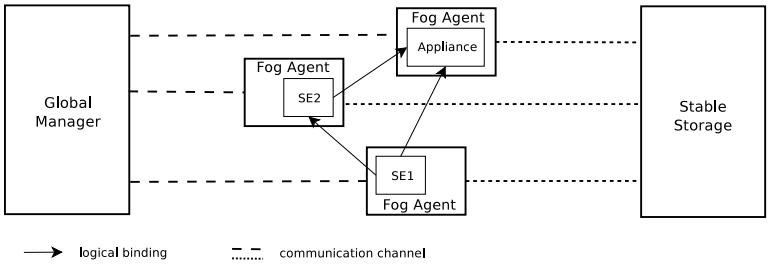

Fig. 3. Overview of Protocol Managers

2.3 Failure Management Protocol

The failure management protocol consists of four functional steps, which are carried out by the managers presented in the previous section: (i) state saving, (ii) monitoring and failure detection, (iii) failure notification, and (iv) decision and recovery. The first step aims at saving the local data that represents the state of execution of each entity of the application on the Stable Storage. The state saving step is policy-based to cope with the heterogeneous nature of the environment. A state saving policy defines for each applicative entity the data to be saved as well as the technique of saving. The techniques are based on checkpoint, message logging and function call record [8]. These techniques are chosen depending on the local storage and reliability assumptions made at the local Fog Node. The second step involves the monitoring of the application for failure detection. Monitoring can be achieved by Fog Agents through the following techniques: heartbeat, ping-acks, applicative message observation and local system observation. A failure detection triggers the third step in which failure notifications are propagated to Fog Agents of dependent entities so as to functionally pause the part of the application impacted by the failure. Concretely, when a failure occurs, the GM notifies the Fog Agents of the direct dependent entities of this failure. These Fog Agents pause the functional behaviour of these applicative entities and propagate the failure notification by sending similar messages to the Fog Agents of the entities depending on them. In

the worst case, all entities are paused. The final recovery phase aims at repairing and restoring the state of the application in order to bring back the application to a consistent state. In the case of a failed software element, it is restarted before restoring its state. If a Fog Node fails, a new placement for the hosted software elements is computed before redeploying them onto failure-free Fog Nodes and restoring their states. When an appliance fails, it is replaced by another functionally equivalent appliance. When recovery is completed, the GM initiates the sending of recovery notifications that are propagated by the Fog Agents to all dependent entities.

2.4 Recovery Phase

In this subsection, we present the final phase of the failure management protocol with more details. This recovery phase aims at repairing and restoring the application into a consistent state upon occurrence of a failure.

Global manager. When a failure occurs, the GM is notified by the Fog Agent that monitors the failed entity. The notification includes the identifier of the failed entity. The GM warns the Fog Agents of the entities having dependencies on the failed one that this failure has occurred. To do so, the GM relies on the architectural description of the application and sends a message to all Fog Agents of depending entities to let them know of the failure. We will see how Fog Agents react to those messages later on in this section. The GM then decides the steps for recovery depending on the type of entity that has failed. In the case of a software element or a Fog Node, it is assumed that there are enough resources to restart a software element or to find another Fog Node. If the failed entity is an appliance, the GM checks in the set of available appliances if there is an appliance that can replace the failed one. An appliance can act as a replacement if it has a similar behaviour to the failed one (checked using strong behavioural equivalence [19]). If there is one functionally equivalent appliance, the state of the failed appliance is restored on the replacement one. If this is not the case, the application cannot be restored and continues its execution with less features in a degraded mode.

The GM awaits for a notification from the Fog Agent confirming that the restoration has been completed (this notification is not necessary in degraded mode). Upon reception of this message, the GM sends a message to all Fog Agents of the depending entities (that have been previously warned of the failure) to let them know that the corresponding entities can resume their activity. The GM finally starts over and can handle a second failure. Note that in the current version of the protocol, the GM handles failures in sequence. In practice, the simultaneous occurrence of two failures hardly ever happens. Moreover, the recovery time is very low, less than a second according to the experiments we made with the protocol implementation. Figure 4 summarises the behaviour of the Global Manager.

Fog agent. A Fog Agent (FA) is in charge of handling software element failures on the local Fog Node and neighbouring appliances. When a failure occurs the FA first notifies the GM and waits for its decision with respect to that failure. If the failure is confirmed and restoration initiated, the FA retrieves the former state of the failed entity from the Stable Storage. It restores the state of the entity and resumes the execution of its functional behaviour from that state.

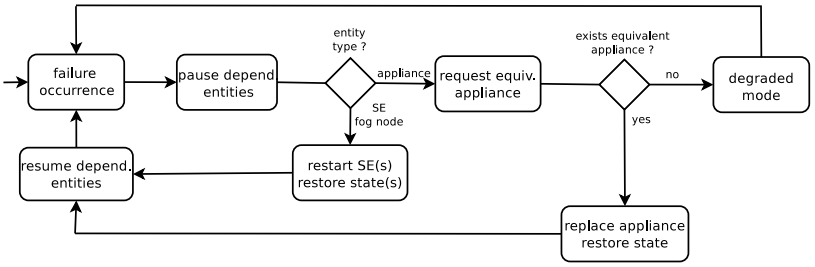

Fig. 4. Global Manager Behaviour

Another role of the FA is to propagate start / pause messages when a failure occurs in one of the neighbouring entities. In that case, the FA can receive a message directly from the GM or from one of the FAs of the neighbouring entities. Upon reception of this message, the FA interrupts the functional behaviour of its entities and propagates the message to all FAs of depending entities. In this way, all the entities that are impacted by the failure pause their functional behaviour. When the failure has been handled and the failed entity restored, each FA receives a message indicating it to start the paused entities again. This decision is propagated in the same way that failure notifications are propagated so that the application becomes fully functional again.

3 Model Checking

In this section, we first introduce the LNT specification for the failure notification and recovery phases of the failure management protocol (phases (iii) and (iv) in Section 2.3). Then, we present the properties of interest that must be preserved by the protocol. Finally, verification is described with some experimental results and a presentation of the issues detected and corrected in the protocol.

3.1 Specification

We chose LNT [5] as specification language because it is expressive enough and adequate for formally describing communication protocols as the failure management protocol presented beforehand in this paper. Moreover, it is equipped with CADP [12], a rich toolbox for analysing LNT specifications using model checking techniques.

LNT is an extension of LOTOS [14], an ISO standardised process algebra, which allows the definition of data types, functions, and processes. Table 1 provides an excerpt of the behavioural fragment of LNT syntax. B stands for a LNT term, A for an action, E for a Boolean expression, x for a variable, T for a type, and P for a process name. The syntax fragment presented in this table contains the termination construct (**stop**) and actions (A). LNT processes are then built using several operators: sequential composition (**;**), conditional statement (**if**), non-deterministic choice (**select**), parallel composition (**par**) where the communication between the involved processes is carried

out by rendezvous on a list of synchronised actions, looping behaviours described using process calls or explicit operators (**while**), and assignment (**:=**) where the variable should be defined beforehand (**var**). LNT is formally defined using operational semantics based on Labelled Transition Systems.

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

Table 1. Excerpt of LNT Syntax (Behaviour Part)

The specification for the failure management protocol consists of three parts: *data types* (~100 lines), *functions* (~600 lines), and *processes* (> 800 lines). A large part of the specification depends on the input application model (involved applicative entities and their dependencies), and is therefore automatically generated from a Python program (~1,500 lines) we implemented. For instance, the application used as example in Section 2.1 (consisting of nine software elements / appliances) results in about 2,500 lines of LNT specification that are generated automatically from the Python program.

Data types are used to describe mainly the application model (Fog Nodes, software elements, appliances, dependencies, behavioural models). As an example, an application consists of a set of Fog Nodes, a set of appliances and a set of logical bindings. *Functions* apply to data expressions and are necessary for several kinds of computation: extracting information from the application such as dependent entities, checking if two appliances have equivalent behavioural models, checking whether the application respects some specific invariants (absence of cycles, no disconnected entity, etc.), computing the target state in case of recovery of an entity, etc. *Processes* are used to specify the different managers of the failure management approach, namely the behaviour of the Stable Storage, of the Global Manager and of the Fog Agents. Note that we also use Fog Agents in this specification to model the functional part of the application, that is, the operations executed and exchanged among applicative entities. Another process, called simulator, is used to make the application execute functional operations and inject failures to the system. This simulator process is parameterised by the length of the execution (the max number of functional operations) and the max number of failures.

Figure 5 shows the four LNT modules used to specify the protocol where boxes correspond to LNT modules (generated code with dashed boxes). When a module is inside another one, it means that the external one includes the inner one. The DATATYPES module defines data types and functions. The APPLI module describes the application model. The GM module defines the behaviour of the Global Manager, which is independent of the application, so written once and for all. The MAIN module defines all other processes for the Stable Storage, Fog Agents and the main behaviour of the protocol.

For illustration purposes, we give in Figure 6 one instance of the simulator process, which is an example of processes. One can see that this process can either make the application evolved (top part) illustrating by the way why a part of the specification depends on the application, or can inject different types of failures to the application.

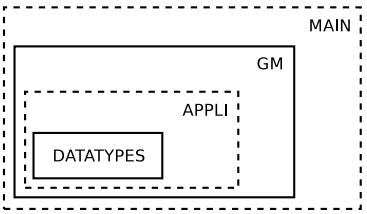

Fig. 5. Overview of the Specification Structure

The process keeps track of the number of occurred operations (functional or failure) in order to terminate (correctly) when the simulation has reached the max number of execution steps. As illustrated in Figure 6, we use actions for modelling functional operations (e.g., initiate, on, off) or for identifying the occurrence of specific events in the prococol (e.g., failureSE, failureAppliance, failureFogNode).

```
process simulator [ initiate:any, on:any, off:any, failureSE: any,
     failureAppliance: any, failureFogNode: any, finish: any]
     (nbFuncOperationsMax: Nat, nbFailureMax: Nat) is
var x, y: ID, sender: ID, receiver: ID, nbFuncOperations: Nat, senderTargetState:
     ID, receiverTargetState: ID, nbFailure: Nat in
  nbFuncOperations := nbFuncOperationsMax ;  -- to count the size of the trace
  nbFailure := 0 ;                -- to count the number of effective failures
  while (nbFuncOperations > 0) loop
    select  -- functional behaviour
       initiate (?sender of ID, ?receiver of ID, ?senderTargetState of ID,
            ?receiverTargetState of ID)
    []
       on (?sender of ID, ?receiver of ID, ?senderTargetState of ID,
            ?receiverTargetState of ID)
    []
       off (?sender of ID, ?receiver of ID, ?senderTargetState of ID,
            ?receiverTargetState of ID)
    [] -- failure injector (3 kinds of failures)
       if (nbFailure < nbFailureMax) then
          select
             failureSE (?x of ID, ?y of ID)
          []
             failureAppliance (?x of ID, ?y of ID)
          []
             failureFogNode (?x of ID)
          end select;
          nbFailure := nbFailure + 1          -- updating the number of failures
       end if
    end select;
    nbFuncOperations := nbFuncOperations - 1  -- updating the number of operations
  end loop;
  finish  -- correct termination
end var
end process
```

Fig. 6. Example of Simulator Process

Finally, the main process is generated and represents all processes (simulator, Stable Storage, Global Manager, Fog Agents for SEs and appliances) in parallel as well as the way they interact together as depicted previously in Figure 3.

3.2 Properties

We identified 12 properties that must be respected by the protocol. These properties can be organised in three different groups: (i) architectural invariants (prop. 1, 2, 3 below), (ii) final objective of the protocol (prop. 4 below), and (iii) additional functional properties (prop. 5-12 below). For some of these properties, we also give their formulation in the MCL language [18], the temporal logic used in CADP. MCL is an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators. Note that, since some of these properties depend on the functional actions used in the application (prop. 6, 7, 9), they are generated automatically using our Python program at the same time as the LNT specification. Note that we could verify additional properties, but the following 12 properties are the most important ones.

(i) Architectural invariants:
   1. There are no cycles in the application through dependencies.
   2. There are no disconnected entities in the application.
   3. Appliances have no dependencies on other appliances.

(ii) Final Objective:
   4. The state of a failed SE (or appliance if there is an equivalent appliance available) after restoration is identical to its state before the failure.

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

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

   This property is formalised making use of action CTL patterns [7].
   6. When a SE / Fog Node is paused, it eventually starts again. This is not always true for appliances because they cannot always be replaced.
   7. When a failure occurs, every dependent SE/appliance is paused.
   8. A sequence exists resulting in the application execution with no failure.
   9. A SE/appliance cannot execute its functional behaviour when paused or failed.
   10. The managers implementing the protocol (Global Manager, Stable Storage and Fog Agents) can always terminate correctly.

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

   11. The application is fully operational except when operating in degraded mode.

12. There is no sequence of two failures without a restore in between (illustrated on SEs below).

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

All the properties verified on the example introduced in Section 2.1 consists of about 600 lines of MCL. Half of the code corresponds to property 9, since in that case we generate one property for each possible functional operation for each entity in the application. Property 9 consists of the conjunction of all these atomic properties.

3.3 Experiments

To verify the protocol, we use as input a set of applications. For each application, we generate the part of the LNT specification depending on the application, and then we call CADP exploration tools for generating an LTS describing all the possible executions of the protocol for that application. In this LTS, transitions are labelled with the actions introduced previously in the specification, and we use these actions in the properties to check that the protocol works as expected.

The analysis of the protocol was run on a macOS Mojave machine with a 2.8 GHz Intel Core i7 processor, 16GB of DDR3 RAM and 256GB PCIe-based flash storage. In these experiments, we rely on a set of realistic smart home applications exhibiting various architectures, and involving different entities (Fog Node, SE, appliance) and logical dependencies. The two other parameters of these experiments are the length of the execution, that is, the max number of functional operations executed by the application, and the max number of failures. It is worth noting that since we use enumerative techniques here, there is no need to experiment with large applications, long executions or a high number of failures in order to find issues in the protocol. In contrast, most problems are usually detected on small applications and scenarios.

Table 2 summarises some of the experiments we carried out during the verification phase. The first column identifies the example. The next four columns characterise the size of the application (number of Fog Nodes, of software elements, of appliances and of dependencies). The following columns characterise the scenario in terms of execution length and number of failures, the size of the raw LTS (number of states and transitions), and the time in seconds for generating the LTS and for verifying all temporal properties. Those properties are analysed on a minimised version of the LTS (reduced wrt. strong bisimulation). It takes a few seconds to obtain that minimised version, which is computed using CADP reduction tools. The minimised version is about the half in average in terms of states and transitions compared to the raw version.

We now comment on the results presented in this table. Example 4 corresponds to the application introduced in Figure 1. These results show that the larger the application is (in terms of entities), the longer it takes to generate the corresponding LTS model (see

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

Table 2. Experimental Results

row 6 for example). Second, the main factor impacting the size of the LTS is the simulation parameters. Since we rely on enumerative techniques, one more functional action or one more failure generates many more executions since this action/failure may occur at any moment during the application execution and this results in additional interleavings of actions (see rows 1a-e and 2a-e in the table). The time for model checking all properties is much longer than the generation time for applications with up to 8 entities, but becomes smaller for larger applications (see rows 4 and 6). More generally, the verification time linearly increases with the size of the LTS whereas the generation time tends to explode when the application size increases. We use rather small applications for verification purposes (up to 12 entities for example 6 in the table) because this is not necessary to use large applications for finding issues, and contrarily, most problems are usually found on small yet pathological applications.

3.4 Detected Issues

The specification and verification helped to refine our understanding of the finer points of the procotol. In this section, we focus on three points of interest, which were identified or confirmed using model checking.

First, it is worth reminding that, although the protocol is always able to work and terminate correctly, this is not the case of the application. Indeed, in case of appliance failure, if there is no functionally equivalent appliance available, the application cannot be restored and the application keeps working in degraded mode. This was confirmed using verification techniques. During our experiments, when there were no additional equivalent appliance for replacing failed ones, the first part of property 11 in Section 3.2

"*the application is fully operational*" was violated because, in that case, an appliance has failed and cannot be replaced.

Another problem comes from the propagation of notifications in case of failure in an application with multiple dependencies. Suppose there is an application with four entities and dependencies among them looking like a diamond (for instance e_2 depends on e_1, e_3 depends on e_1 and e_4 depends on both e_2 and e_3). If e_1 fails, the Fog Agents of e_2 and e_3 are notified of that failure and send a notification to all Fog Agents depending on them. In that case, both e_2 and e_3 Fog Agents actually send a notification to e_4 Fog Agent, so the Fog Agent of e_4 is supposed to receive two notifications in that case. If those messages are not consumed, this can induce a deadlock (correct termination of the protocol is not possible, prop. 10) because one of the Fog Agents of e_2 and e_3 is not able to propagate its notification, resulting in an erroneous situation in the protocol. This can be corrected by either receiving as many notifications as supposed with respect to the failed entity and the structure of the application, or by accepting the first notification message and discarding the forthcoming similar ones.

Another interesting issue concerns multiple simultaneous failures. The protocol was originally supposed to support such failures, but the design of the protocol was more complicated and several properties were violated. This is due to the fact than when trying to handle several failures at the same time, some contradictory messages can be exchanged among Fog Agents (e.g., one message saying to restart an entity whereas another failure has occurred so a contradictory message asking to pause is also received). As a consequence, in the first version of the protocol, we decided to treat failures one after the other, which is reasonable since the occurrence of multiple failures is scarce and the implementation of the failure management protocol on a realistic testbed shows that the time taken to recover from failures is less than one second.

To conclude, it is worth saying that all properties were satified for all the examples of our dataset of applications used for verification purposes. In addition, the existing implementation of this industrial protocol developed by Orange Labs was modified in order to integrate all the feedback and issues discovered during the analysis of the protocol.

4 Related Work

In this section, we first compare the protocol presented in this paper with respect to similar protocols for failure management proposed in cloud computing or Fog computing / IoT. In a second step, we have a specific focus on similar protocols, which also make use of formal methods for verification purposes.

Failure management. [11] proposes a self-healing approach to handle exceptions in service-based processes and to repair the faulty activities with a model-based approach. More precisely, a set of repair actions is defined in the process model, and repairability of the process is assessed by analysing the process structure and the available repair actions. When an exception arises during execution, repair plans are generated by taking into account constraints coming from the process structure, dependencies among data, and available repair actions. In [4], the authors present fault-aware management

protocols, which permit to model the management behaviour of composite cloud applications, by taking into account the possible occurrence of faults suddenly occurring and misbehaving components. This approach also proposes to generate plans for changing the actual configuration of an application for, e.g., recovering an application that is stuck because of a faulted node. A few recent papers have focused on fault tolerance of IoT applications. [23] provides a fault tolerant approach through virtual service composition. Single service and single device failures are supported by using IoT devices of different modalities as fault tolerant backups for each other. [22] discusses the challenges of fault tolerance in IoT and proposes some potential solutions to consider. It suggests that natural redundancy of functionality across devices within the home, as well as usage scenarios, should be exploited to provide fault tolerance and also discusses the issues of this approach, like incorrect context sensing and actuating of devices. [1] proposes a fault tolerant platform for smart home applications. It provides fault tolerant delivery of sensor events and actuation commands in the presence of link loss and network partitions. [13] proposes an IoT-based architecture supporting fault tolerance for healthcare environment. The approach focuses on network fault tolerance which is achieved by backup routing between nodes and advanced service mechanisms to maintain connectivity in case of failing connections. Compared to these approaches, the main difference is that our approach focuses on stateful applications and allows the end-to-end management of a failure from detection to recovery of a consistent state of the application. The proposed protocol also takes into account the specificities of the environment such as heterogeneity and dynamicity.

Protocol verification. In [3,2], the authors present a reconfiguration protocol applying changes to a set of connected components for transforming a current assembly to a target one given as input. Reconfiguration steps aim at (dis)connecting ports and changing component states (stopped or started). This protocol supports failures and preserves a number of architectural invariants. This was proved using the Coq theorem prover. We preferred model checking techniques here, because they are convenient at design time in order to detect possible issues. Theorem proving is interesting when the developers have already at their disposal a stable version of a protocol, and they ultimately want to prove its correctness. In [21,9,10], the authors present a self-deployment protocol that was designed to automatically configure cloud applications consisting of a set of software elements to be deployed on different virtual machines. This protocol works in a decentralized way, i.e., there is no need for a centralized server. It also starts the software elements in a certain order, respecting important architectural invariants. This protocol supports virtual machine and network failures, and always succeeds in deploying an application when faced with a finite number of failures. A formal specification of the protocol allowed the successful verification of important properties to be preserved. [16,15] propose verification of IoT applications before deployment using model checking techniques. [6] applies infinite-state model checking to formally verify IoT protocols such as the Pub/Sub consistency protocol adopted by the REDIS distributed file system. The verification method is based on a combination of SMT solvers and overapproximations as those implemented in the Cubicle verification tool. Since these protocols involve infinite data structures, the authors chose to use analysis techniques capable of reasoning on infinite state spaces.

5 Concluding Remarks

In this paper, we focus on a failure management protocol, which allows the supervision of IoT applications and the management of failures. This protocol targets stateful IoT applications in the sense that those applications handle and store data during their execution. When a failure occurs, the protocol detects the failure and restores a consistent pre-failure state of the application to make it functional again. Since designing such distributed protocol is error-prone, we decided to rely on formal specification techniques and verification tools in order to ensure that the protocol respects some important properties. We used a process algebraic specification language and model checking techniques for verifying these properties. These analysis steps helped to detect several issues and clarify some subtle parts of the protocol. This information was used to correct and revise the implementation of the protocol developed by Orange Labs.

As for future work, the first perspective is to extend the protocol specification to support asynchronous communication (communication via message buffers). Another perspective is to revise the protocol in order to support simultaneous failures. An idea in that direction is to distribute the behaviour of the Global Manager locally on Fog Agents in order to have a fully decentralised version of the protocol, which will make it more amenable to support multiple failures.

References

1. M. S. Ardekani, R. P. Singh, N. Agrawal, D. B. Terry, and R. O. Suminto. Rivulet: A Fault-tolerant Platform for Smart-home Applications. In *Proc. of Middleware'17*, pages 41–54. ACM, 2017.
2. F. Boyer, O. Gruber, and D. Pous. Robust Reconfigurations of Component Assemblies. In *Proc. of ICSE'13*, pages 13–22. IEEE Press, 2013.
3. F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11*, volume 6664 of *LNCS*, pages 103–117. Springer-Verlag, 2011.
4. A. Brogi, A. Canciani, and J. Soldani. Fault-Aware Management Protocols for Multi-Component Applications. *Journal of Systems and Software*, 139:189–210, 2018.
5. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator, Version 6.7. Inria, 2018.
6. G. Delzanno. Formal Verification of Internet of Things Protocols. In *Proc. of FRIDA'18*, 2018.
7. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of ICSE'99*, pages 411–420. ACM, 1999.
8. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
9. X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. D. Palma. Reliable Self-deployment of Cloud Applications. In *Proc. of SAC'14*, pages 1331–1338. ACM, 2014.
10. X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. D. Palma. Reliable Self-deployment of Distributed Cloud Applications. *Softw., Pract. Exper.*, 47(1):3–20, 2017.
11. G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. Exception Handling for Repair in Service-Based Processes. *IEEE Trans. Software Eng.*, 36(2):198–215, 2010.

12. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 2(15):89–107, 2013.
13. T. N. Gia, A.-M. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen. Fault Tolerant and Scalable IoT-Based Architecture for Health Monitoring. In *Proc. of SAS'15*. IEEE, 2015.
14. ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, ISO, 1989.
15. A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün. IoT Composer: Composition and Deployment of IoT Applications. In *Proc. of ICSE'19*, pages 19–22. IEEE / ACM, 2019.
16. A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün. Rigorous Design and Deployment of IoT Applications. In *Proc. of FormaliSE'19*, pages 21–30, 2019.
17. B. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, Xerox Palo Alto Research Center, 1979.
18. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.
19. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
20. U. Ozeer, X. Etchevers, L. Letondeur, F.-G. Ottogalli, G. Salaün, and J.-M. Vincent. Resilience of Stateful IoT Applications in a Dynamic Fog Environment. In *Proc. of MobiQuitous'18*, pages 332–341. ACM, 2018.
21. G. Salaün, X. Etchevers, N. D. Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*, volume 7740 of *LNCS*, pages 60–79. Springer, 2013.
22. D. Terry. Toward a New Approach to IoT Fault Tolerance. *Computer*, 49(8):80–83, 2016.
23. S. Zhou, K.-J. Lin, J. Na, C.-C. Chuang, and C.-S. Shih. Supporting Service Adaptation in Fault Tolerant Internet of Things. In *Proc. of SOCA'15*, pages 65–72. IEEE, 2015.