Fault-recovery and Coherence in Internet of Things Choreographies

Sylvain Cherrier, Yacine Ghamri-Doudane

Fault-recovery and Coherence in Internet of Things Choreographies

Sylvain Cherrier[1], Yacine M. Ghamri-Doudane[2]

[1] *Université Paris-Est , Laboratoire de l'Institut Gaspard Monge (LIGM) CNRS : UMR8049*

[2] *L3i Lab, University of La Rochelle, La Rochelle, France.*

Facilitating the creation of applications for the Internet of Things (IoT) is a major concern to increase the development of the IoT. Due to devices heterogeneity, programmers have to find solutions that hide hardware diversity. D-LITe, our previous work, is a framework that introduces genericity by providing a universal programming interface for any Object. In D-LITe, Objects are dynamically configured to have their own behaviour, and their actions/reactions are considered as part of a whole application. D-LITeful Objects offer a REST web service that describes their capabilities, receives the application logic to be executed, and interacts with other stakeholders. Then, the complete application is seen as choreography of Object behaviours. But the main issue of choreographies is the loss of coherence. Because of their unreliability, some networks used in IoT may introduce de-synchronization between Objects behaviours, leading to inter-actions errors and failures. In this paper, we propose a solution to reintroduce coherence among the different stakeholders implied in the application, in order to keep the advantages of choreography while dealing with this main issue. An overlay of logical check-points at the application layer defines the dependences between the coherent states of a set of objects and triggers re-synchronization messages. Correcting statements are thus spread through the network, which enables fault recovery in the choreography. This paper ends with a discussion on the trade-off between the checking cost and the reliability improvement.

Index Terms—Internet of Things; Services Choreography; Fault-tolerance; Fault-recovery

INTRODUCTION

The Internet of Things (IoT) is a rising domain that gives Internet widespread connectivity to real world objects. IoT focuses research interest because it re-uses well-known protocols. The success of IoT will come from the ability to easily create applications. In our previous work, we have presented D-LITe [*Cherrier et al., 2011*], a framework for IoT applications creation and deployment, based on services choreographies. The main advantage of services choreographies is based on the distribution of the different parts of the application across the network. The absence of central point leads to a better dissemination of the uses of the resources, and a more efficient use of energy, especially in constrained networks [*Cherrier et al.,2012*].

However, services choreographies are subject to a major issue. Unlike orchestrations which are under the control of a unique central point, the spread of the logic may lead to desynchronisations. For example, some stakeholders may miss steps in their choreography and the whole application becomes incoherent. As it integrates WSAN (Wireless Sensors and Actuators Network), IoT uses wireless links to make Objects communicate. These wireless links are characterized by their unreliability. Indeed, tests on our testbed show that we are still facing desynchronisations, in spite of controls made at lower layers.

In this paper, we propose a new mechanism allowing IoT programmers to introduce coherence controls in order to correct the logical state of Objects involved in a global application. This coherence checking is deployed along with the services choreography. To respect the decentralized approach chosen for the framework, the coherence mechanism is organized in cascade. Some Objects start a check by sending an order to a list of "followers" Objects. Receiving this order, these "followers" check their own state, make correction if needed, then transmit in their turn the coherence control orders to their own list of "followers" Objects. By doing so, we show in this paper that we are able to reintroduce coherence. Finally, the cost of introducing such coherence is analyzed in terms of communication overhead.

This paper is organized as follow: Section II presents the background composed of related works and an introduction to our platform D-LITe. The coherence checking architecture is described in Section III. Section IV contains our experimental study and our results. Finally, concluding remarks and future research directions are given in Section V.

BACKGROUND

Related work

Our approach of Internet of Things applications [*Cherrier et al., 2011*] is part of the Services Oriented Architecture (SOA) realm. SOA offers a decentralized composition of programs in a "loosely coupled platform-independent model" [*Zhou et al., 2010*]. G. Canfora and M. Di Penta present the specificity of SOA as "radical changing [in] the development perspective" [*Canfora et al., 2009*]. Implication of that change leads to "lack of observability of the service code" and "lack of control" because of the infrastructure independence and the absence of access to the running code. The "cost of testing" for such a decentralized software composition over heterogeneous hardware may lead to denial-of-service if too many checks are performed.

Testing Web services collaboration can be done at design time. H.Huang et al. [*Huang et al., 2005*] use language translations from OWL-S (Ontology Web Language for Services) into a model checkers compatible one, in order to test some services composition. This translation is used to generate test cases in an a priori check. To specifically test web services choreographies, L.Zhou et al. [*Zhou et al., 2010*] propose the use of assertions that "express the intention of the program by designers". A simulator processes each web services and builds the complete interaction.

All paths are checked and assertions are verified. This tool can detect design errors. However, in our case, we are more concerned by global fault checking of running choreographies, which is not necessary and as such not addressed for web applications.

These a priori model checking approaches are efficient to detect design problems. Besides them, many other unpredictable failures can corrupt the running composition, especially if unreliable networks are involved. KleeNet [*Sasnauskas et al., 2010*] is a tool to test the reliability of a distributed solution. It aims at testing the behaviour of choreography when unexpected errors occur. KleeNet triggers network error on a running application. It has been able to detect errors in the design of ContikiOS, the open-source OS for IoT devices [*Dunkels et al., 2004*].

The issue of coherence and fault tolerance in distributed computing is the subject of numerous publications of the domain. E. Brewer has defined a theorem (the CAP theorem) about the consequences of the impossibility to "achieve both consistency and availability in an unreliable system" [*Gilbert et al., 2012*]. Its trade-off [*Brewer,2000*] between these 3 parameters is the base of research to retrieve consistency. Even if this theorem is rather of big-data and cloud domain, IoT is also concerned because of the "notorious unreliability"[*Gilbert et al., 2012*] of WSAN and mobile network. Checkpoint-based mechanisms is a solution that uses "snapshot of the state of[...] a particular point" [*Egwutuoha et al. 2013*] in order to "roll back" the application "to the most recent consistent state". In their survey [*Egwutuoha et al. 2013*], H.P. Egwutuoha et al. list multiple variations on the topic, as trade-off between global consistence, domino effect failures and the overhead generated.

IoT itself consists of several architecture propositions. Most of them are cloud-oriented and data-centric. In this case, solutions provided by the distributed computing community are useful. Some architectures are less centralized, and data are processed directly on a node, or on a gateway [*Cherrier et al., 2014*]. Such approaches leverage the network by sending only results, hiding the multiple sensors and the heterogeneity of hardware in use. This forbids the idea of retrieving lost information by average or multiple sources. But the capacity of using the processing capabilities in or near each object can also provide fault-tolerance mechanism, as in WuKong [*Hu et al., 2014*]. Service offered by an object can be replicated on others. A heartbeat monitor detects any object unreachable. In that case, another object is activated for the service missing.

In this paper, we are interested in errors that happen while an application is running. Our target is to offer a mechanism able to correct their effects. We assume that the application is a priori bug free. We aim to keep the running application in a correct state in spite of exogenous hardware/network/application errors. Our fault-tolerance mechanism extends to human cause failure [*Egwutuoha et al. 2013*] as part of IoT application failures. Assertions are used to express restrictions at control points [*Zhou et al., 2010*]. But it runs during the service execution phase, and provides a mechanism to correct logical de-synchronizations.

Trying to recover from the effects of exogenous errors has already been explored [*Singhal et al., 1995*], in which checking points and consistent set of states are presented. This approach offers fault-tolerance and recovery. It can be used for Web services in a similar way used in this paper [*Behl et al,2012*]. However, the authors obtain robustness by replication of the web services (as in [*Zhou et al., 2010*] for the IoT). On the web, one can duplicate services, thanks to powerful hardware. In our case, IoT objects that interact with the real world are often unique, and replication is not possible. On our vision of the IoT, each object has a specific role (*mainly because of its position and its unique point-of-view*) and has often low computing capabilities. For us, an Object is unique, because we have only a single copy of it, or because it represents a set of similar elements (*average of temp sensors, light, etc.*).

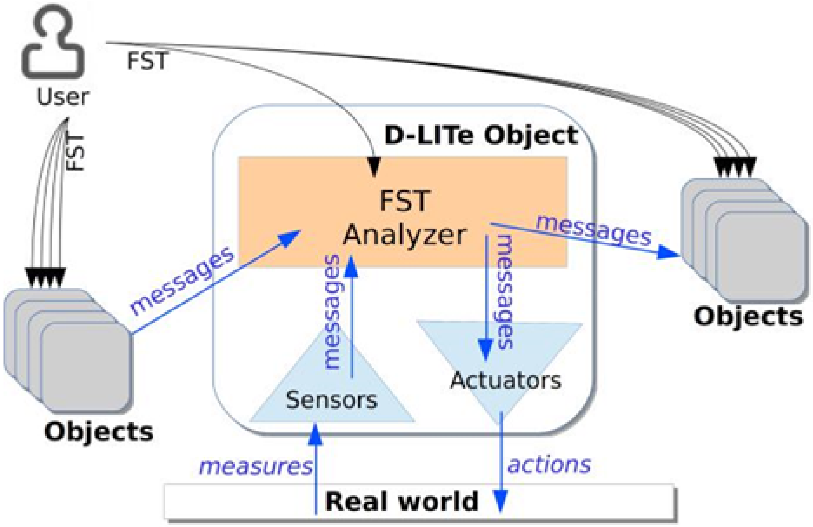

Figure 1. A D-LITeful Object gives a generic view of its sensing and actuating capabilities. Then, it is possible to use them through the description of a FST (Finite State Transducer, a Finite State Machine with an Output alphabet). A D-LITeful Object gets its FST input from other Objects or its own hardware (sensing). Its FST outputs are in destination of its own hardware (actuating) or other Objects. The FST that drives the whole logic is remotely and dynamically installed by the user through the network.

Our platform: D-LITe

In previous works, we have presented our framework D-LITe [*Cherrier et al. 2011*] for designing Choreographed applications for IoT. By introducing a Hardware Abstraction Layer, D-LITe hides Objects diversity and gives a universal representation of an Object. According to the features offered by the devices, D-LITe has a standardized set of orders to trigger actions or to sense the real world. These standardized orders are provided by the generic representation of the Object. This solution offers an easy way to create IoT applications over heterogeneous environments.

D-LITe uses standardized REST protocols to remotely configure the Objects for their integration in the application. The Object's discovery mechanism and the "Over the Air" deployment of an algorithm on each Object give a real usability to our solution. Each algorithm (the part of the application an Object must execute) is expressed with Finite State Transducers (FST) through a specific language called SALT [*Cherrier et al., 2013*]. SALT is designed to describe the behaviour an Object has to follow in order to participate to the global application (see Fig 1). SALT can drive the Hardware Abstraction Layer introduced by D-LITe and is based on FST (finite state machine, to which an output alphabet has been added). FST are a simple way to express logical sequence of actions driven by events. They are a good compromise between the needs of IoT programming, the reduced set of available actions and the ease of understanding a new programming language.

Each Object receives its own FST to be executed and a list of subscribers (*a selected list of Objects that will receive messages from this Object*). The output of a given FST becomes the input of the subscribers, as shown in Fig 1. D-LITe is event-centric, an event being the reception of a message or a change in the environment (*for Objects with sensing capabilities*). The event is treated by a Transition in the FST. Output messages are generated and sent, that can be for other Objects (*subscribers*) or intended to the Object's hardware (*Object with actuating capabilities*).

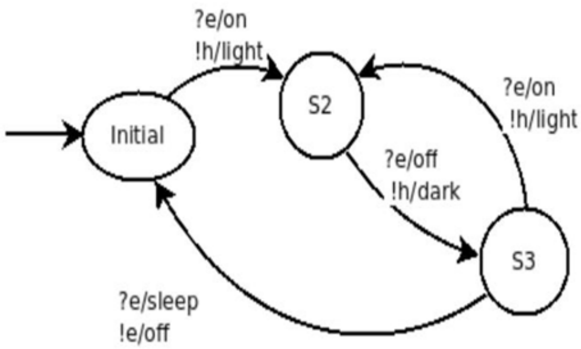

Figure 2. The FST presented above is usable on a D-LITeful smart light. Orders interacting with any light hardware are "light" and "dark". In this FST, external messages are identified by the "e/" prefix, while hardware messages use "h/". The question mark stands for input, the exclamation mark for output. This FST starts in the "Initial" state. When it receives "on" from another node (as indicated by "?e") , it switches on (going to state "S2" sends "light" to hardware, as indicated by the prefix "!h"). Then depending on "off" or "on" received messages, it switches off and on. When switched off (in state "S3"), it goes back to the "Initial" state when it receives "sleep".

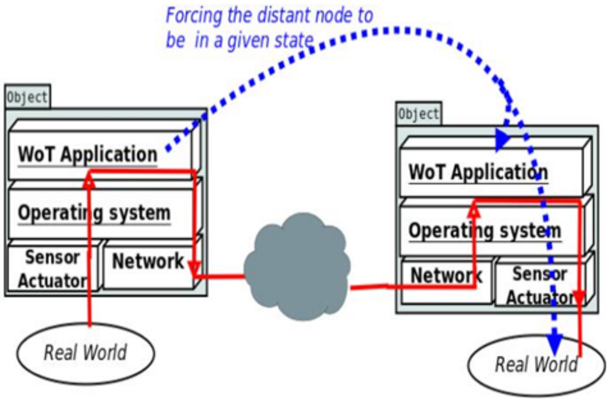

Figure 3. To reintroduce coherence between the different stakeholders, we need a direct access to object’s application and state. When a resynchronization is needed, an Object may have to go directly to a given state. This "break" in its own logic is useful to correct the effect of lost messages between objects, due to network unreliability or other

In the D-LITe’s point of view, an IoT application is based on events that make Objects react, thus sending information to other Objects that will react to them and send information, and so on (see Fig 1). Events handled by D-LITe are each message received from other stakeholders, or generated by its own hardware (*for example, pressing a button sends a "push" message to the FST input, or a temp sensor sends its value*). Fig 2 represents an FST example. This one corresponds to a smart light. The message indicates the origin or destination: other Objects or hardware (as in Fig 1). Input messages (*with a "?"*) are received from stakeholders ("*e" stands for external messages*). Following its transitions, the FST Fig 2 operates the light (*output messages, starting with a "!", are for the Hardware Abstraction Layer "h*"), and invokes the commands "dark" and "light". These commands are defined in SALT as generic ones for all Objects having the "led" feature.

To sum up, using D-LITe, the IoT application algorithm is decentralized. Each Object follows its own strategy, depending on its inputs and described by its FST. Each algorithm generates outputs, defined in its FST. Because Objects subscribe to some others, the followers Input is fed by the observed Outputs. The resulting composition of behaviours is a software design called a Services Choreography (See Fig 4-top).

COHERENCE CHECKING OVERVIEW

Motivation

Rather than trying to fix all errors that may have occurred at different levels and in different places, the idea of putting Objects back into a compliant coherent state makes sense. Our idea is to provide a mechanism to help a programmer to express assertions, as in traditional programming language, such as C language [*Rosemblum, 1995*]. The programmer uses these assertions to bring back correctness in the global system. In Fig 3, the Object on the left resynchronizes the one on the right by setting it directly in a specific state, no matter its own algorithm (it may be desynchronized because of the loss of some messages). It makes no sense to stop the execution of an IoT application or to find out the failure source as this may not be a straightforward task. Error causes can vary widely and may not be reproducible. IoT applications depend on the reliability of the weakest part of the whole structure. Trying to work out which part induced an error, and the reason why internal mechanisms did not manage it, seems to be out of reach because of the heterogeneity of networks, hardware and software involved in the global cooperation. A mechanism giving resynchronization capabilities is more valuable to keep the IoT application coherent despite errors and network unreliability.

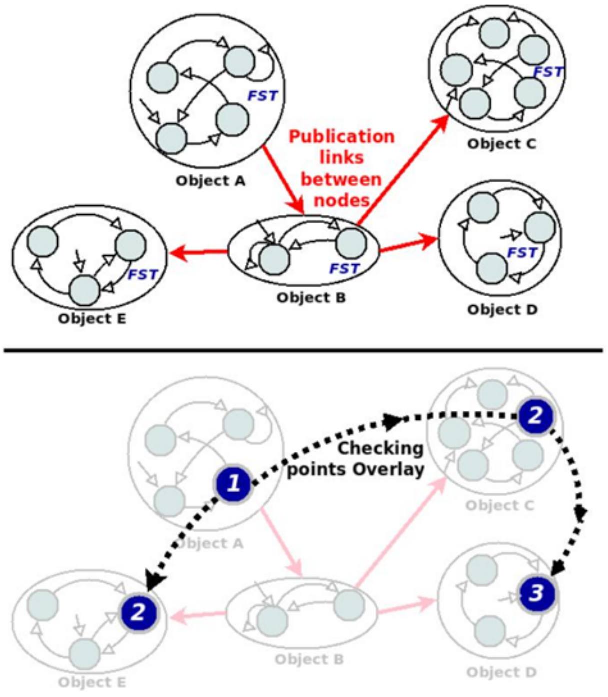

Figure 4. Top: a D-LITe application is choreography of FST. Each object follows its dedicated algorithm. Bottom: To improve coherence, combinations of stable states for the Objects are defined. Initial states are one of these groups, but there are probably some others. Programmers build an overlay of coherent states among all possible combinations. The root of each overlay starts a cascading check to ensure the coherence of each element.

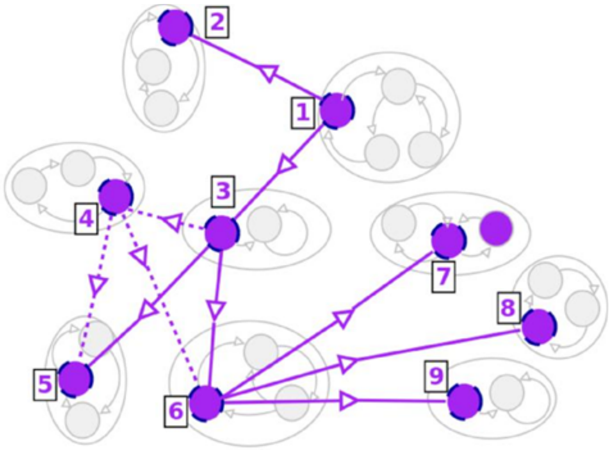

Figure 5. Checking points overlay is a tree that uses some of the Objects involved in the application. Only one coherence id is represented here. Object 1 (tree's root) is in charge of starting the check (when it reaches the indicated State). The check request follows the tree, triggered from Object 1 to Objects 2 and 3 in parallel. So, each node checks itself, and spreads the check to followers. An "OR" is organized between node 3 and 4. If 3 is not valid, node 4 is checked. This coherence id accepts two valid States

Check-points overlay

In D-LITe, each local algorithm is expressed as a FST. Improving the coherence inside the resulting composition (*the IoT application*) rests on a stable combination of States of the set of FST. In some cases, a programmer can assert that if a given Object is in a specific State, then some others must be in related states as well. For example, in home automation, one can say that when the house door is closed, the alarm must be switch on. If not, a message has been lost: it's a network error. Every light must be switched off. If not, it's a user error, but still, the application must correct it. In a smart building scenario, we can assert that when the security guard has checked every room and starts the alarm, then all sensors should be in a "clear" state.

Although very few states combinations are valid compared to all the possibilities, at least one exists: Initial State of each Object. Depending on his application, a programmer can detect some other valid states combinations for a subset of Objects. Because he has a global view of all Objects interactions, he can define coherent combinations of algorithm steps running on different Objects. Each combination is identified by a Coherence Id (*an id the programmer chooses*). This subset of Objects for that states combination is our Checkpoints overlay. It can differ from FST subscribers' list, because the coherence checking range is different (*i.e. the house door, the alarm and all the lights: The house door does not directly interact with all the house lights and is not an alarm remote controller, but there is a coherence between the door when it is locked, the alarm's state and each light's state*). The root Object (*here the house door*) throws the check during execution (referenced as 1 in Table I and Fig 4).

For each Checkpoints overlay defined by the IoT programmer, a root Object must be identified. It is because this Object has reached a given state that the programmer asserts that his application (*or some Objects part of it*) is in a coherent state. In such cases, he defines the tree of that specific Checkpoints overlay in order to spread this check in cascade. Each tree (1-2-3...) can be different from the tree used by the application (Fig 4). Each Checkpoint overlay has its own root, its own Coherence Id and its specific dependences

Table I : A CHECK POINT OVERLAY

Coherence id	Object A	Object B	Object C	Object D
1	—	Up(2)	Wait (1)	Dark (2)
2	Sleep(4)	Stop(1)	Close(2)	Light(3)
3	Wake up (1)	—	Open(3)	Light(2)
4	Up(1)	Stop OR	Close(2)	Dark(3)
5	Sleep(1)	—	—	Light(2)

Coherence check requests

Fig 5 shows a more complete Checkpoints overlay. The programmer's duty is to organize his tree. To avoid the domino effect, the cascade checking must limit the number of dependences on a single Object. Limitation of the list given to each Object to only its following nodes in the Checkpoints overlay relies on our vision of a distributed IoT. Indeed, the size of "followers" nodes list may not fit in every constrained Object's memory, depending on implementation and hardware capabilities. Also, as the main concept of D-LITe is based on the Choreography paradigm, the Fault recovery and Coherence mechanism we are proposing in this paper has to follow the same decentralized organization.

Algorithm 1 *Triggering a Check at the end of Changing State Procedure of our FST*

```
struct {
        int chkId; //coherence check id
        string state; // triggering state
        ip_addr[] targets; //prime followers list
} ckp;

Require: : chkp Array of ckp, currentState
Ensure: Throwing a check if needed
        i←1
        while chkp[i] ≠ null do
                if chkp[i]:state = currentState then
                        chkNumber←random()
                        for all ip in chkp[i]:targets do
                                sendChkMsg(ip; chkp[i]:chkId; chkNumber)
                        end for
                        return true
                else
                        i← i + 1
                end if
        end while
        return false
```

As described in Algorithm 1, each time a FST changes to a new state, the D-LITe virtual machine in charge of executing the FST scans its checking table to see if the programmer asked for a check at this state. If a check is requested, the coherence mechanism implemented in this root node throws a check message to the list of "followers" Objects. The check message contains the coherence id and a random number (*used to avoid loopbacks during checking*). Each "followers" will then react to this coherence check request, and will in turn spread it to its own "followers" list.

This distributed check has only one root, but it may have multiple checks inside the application. Each check has its own Checkpoints overlay, its own root, and follows its own tree. As each node is not locked *(it still reacts to incoming messages*), no deadlock can occur. However, a bad construction of checkpoints tree can entail a domino effect. For example, one tree asks an object to be in one state, leading another check tree to ask the first one to go back to another state. This is the result of a bad programming assertion, such as infinite loop errors in programs, and is due to a poor design of the programmer. The analysis of the circular dependences in Checkpoints overlays allows the detection of the problem so to forbid its deployment.

Logical operators

When the programmer starts to specify his coherence assertions for the application, he defines trees of dependences between specific states of FST running on several nodes of his application. Sometimes, he may have to combine states in a logical way. For example, the programmer wants to express that the given state of an Object implies a resynchronization on Object 1 AND Object 2, or on Object 1 OR Object 2 (see Table I). In fact, the AND is easy to realize by linking the two Objects to the first one. Having an OR needs a specific reasoning.

The "OR" is implemented with a break in the cascade in the case the node is not in the asserted state. In that case, the coherence check goes to a sibling (the next member of the OR expression). The programmer can have as many nodes as he needs in his expression. The OR algorithm makes no correction when it encounters a non valid state, it transfers the check to the sibling element of the OR expression. The only one than may change is the last sibling. In fact, the last element is the only one which follows the usual process (*correcting or not, and then spreads to followers*). This OR mechanism is implemented through an alternate "Sibling" node (See Algorithm 2).

A demonstration of a logical "OR" over a Checkpoints overlay is described in Fig 5. If Object 3 is in a valid state when checked, the check is transmitted to the following Objects (5 and 6). But if 3 is invalid, no change is done and no check is cascaded to nodes 5 and 6. Instead, the check request goes to the sibling Object, number 4. It indicates that we want Object 3 in a certain state OR Object 4 in a given one. 4 is checked in the usual way (*corrected if needed, and then followers are checked*). Table I presents 5 Checkpoints overlays that are readable from the tree or the node point of view. A logical "OR" appears for the tree Coherence id number 4 and Objects B and C. When Object A reaches the state "Up", it throws a resynchronization to B, which must be in state "Stop". In that case, the check follows to object D. If not, Object B is unchanged and the check goes to Object C. This one is checked, and corrected if necessary.

Coherence checks spreading

The coherence checking tree (Fig 5) is described by the programmer. He organizes his checks in cascade to prevent too numerous dependencies on a single Object *("followers" list of Objects may not fit in the very small memory of IoT devices*). Two algorithms are implied in the coherence mechanism. The first algorithm deals with the case of the root node. It makes no change but is useful to trigger the check. The second algorithm shows the reaction to a check request. Unlike the first, it can be triggered at any time, and may change the state of the object. These two algorithms are described in this part.

Algorithm 1 is executed each time a FST moves to a new State, on every Object. It scans its checking table (*present only if the Object is a tree's root node*) to see if a check is required. In that case, the coherence mechanism throws a check message to the "followers" Objects list with the coherence id and a random number. In the case there is a loopback, we must avoid a never ending loop. A random chkNumber combined with the Coherence Id will make this check unique.

Algorithm 2 is triggered when the Object receives a check request. This can happen at any moment, and it is not predictable. Contrary to the first algorithm that depends on the Object itself, the second one depends on the logic of other Objects implied in the application. The check request is received and managed inside each "follower" logic. Algorithm 2 starts by checking the chkNumber (*the anti-loopback mechanism*). Then, it searches this check id in the Object's table. When the Coherence Id is found, the current FST state is compared to the list of valid states list, as there are maybe multiple valid states (see node 7 in Fig 5). If it matches, the Object is safe, and the algorithm goes on by cascading the check to its own list of "followers". If the state is not valid, two cases occurs:

- the alternate "sibling" is empty. The algorithm randomly chooses one of the valid states, sets the FST to it, and then spreads the check as usual, now that the Object is correct.
- There is a "sibling". "Siblings" are useful to express an "OR" in the checking logic. No correction is made here. The check is simply sent to the "Sibling" Object.

The cascading coherence system makes each Object able to jump directly to a given state (see Fig 3). Doing this, it also has to make some physical changes if this Object is an actuator. In home automation for example, checking coherence when the door is closed eventually leads to a resynchronization of some Objects. In that case, because the coherence mechanism bypasses the algorithm executed by the object, we also need to really switch off lights (jumping to the state "dark" remains the light on). Our mechanism fulfills the logical state resynchronization. In the case of IoT distributed applications, it also need to ensure the consistency of physical state, according to the logical one (i.e. for actuators).

Algorithm 2 *Reception of a Check message*

```
struct {
    string[] states; //all valid states
    ip_addr[] targets; //followers list
    ip_addr altTarget; //sibling (for OR)

} ckpRcv;

Require: ckpRs Array of ckpRcv, currentState; Require: RcvChkId; RcvChkN umber; LastChkN umber
Ensure: Verify state, change if needed, and propagate
    if RcvChkNumber ≠ LastChkNumber then
        LastChkNumber ← RcvChkNumber
        i←1
        while ckpRcv[RcvChkId]:states[i] ≠ null do
            if ckpRcv[RcvChkId]:states[i] = currentState then
                for all ip in ckpRcv[RcvChkId]:targets[i] do
                    sendChkMsg(ip; RcvChkId; chkNumber)
                end for
                return true
            else
                i ← i + 1
            end if
        end while
                {Coherence error: sibling warned or pushing FST in any right state}
        if ckpRcv[RcvChkId]:altTargets = null then
            newState ← 1 + (random() mod (i 1))
            changeFSTto(ckpRcv[RcvChkId]:states[newState])
            for all ip in ckpRcv[RcvChkId]:targets[newState] do
                sendChkMsg(ip; RcvChkId; chkN umber)
            end for
            return true
        else
            sendChkMsg(ckpRcv[RcvChkId]:altTarget, RcvChkId; chkNumber)
        end if
    end if
    return false
```

Actuators resynchronization motivates the introduction of a new transition (see Fig 6) to the original FST (presented on Fig 2 and in previous papers). This new State is used when the coherence mechanism forces the FST to be in a valid state (i.e. it was in an incoherent one). In fact, the FST could have been in "S2" state while receiving the check. Forcing the FST to goes into "S3" state without that new Transition would let the light switched on, which is not correct now that we are in state "S3". To respect automata formalism, it can be seen as adding a new initial state to the FST (Fig 6) that will be used by the coherence check mechanism.

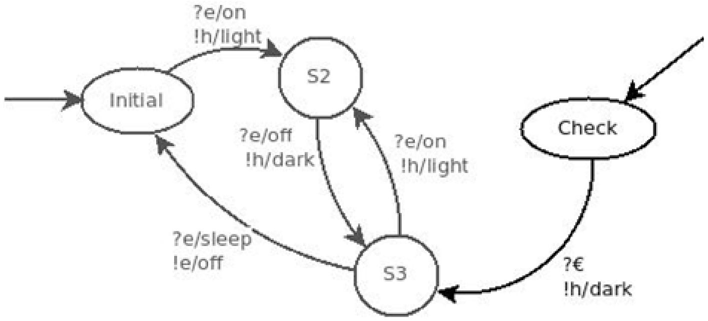

Figure 6. This FST is the same as in Fig 2 adapted for checking. An added State (Check) has a transition that switches off the light. If this added state is ever used, it is because we need to correct Object's state. If ever a check forces the FST to go into S3 state, it will actuate the hardware (switch off the light) to be coherent with Object's logic.

Checking an application

This section summarizes the whole resynchronization process. Once the decentralized logic is organized, the programmer starts to think to his resynchronization mechanism if he needs one. As seen above, for each stable combination, a specific Checkpoints overlay is defined, including a root node (*plus the state that throws the check) and the tree of nodes that depends on it (plus their valid states*). Each Checkpoints overlay may have a different root and a different tree with different valid states. Each node may be involved in different checkpoint overlays. That is the reason why each node receives a set of check configuration during the building of the resynchronization mechanism.

Table I shows the full view of the coherence checking defined by a programmer. Each Checkpoint overlay has its Coherence Id, lists the nodes involved, the root node, how the tree is organized (*according to the rank of each node*) and what states are valid. For example, the second Checkpoints overlay (Coherence Id number 2) is trigger when Object B goes to state "Stop", then the control is passed to Object C that must be in state "Close", then to Object D that must be in state "Light". Finally, verification is done on Object A which is valid if into "Sleep" state (*and resynchronized otherwise*). As a result, if Object C is incorrect during this verification and so forced into state "Sleep", a coherence check starts because "Sleep" is the root state of another Checkpoints overlay (Coherence Id number 5). In that case, the first level of verification tests Object D (*and corrects it if necessary*). Objects B and C are not tested in that Checkpoints overlay.

From the Object point of view, the participation to a Checkpoints overlay is defined in the Table I by the column corresponding to its name. For example, Object B receives the following commands:

1) Entering "Stop" state is the root of Checkpoint overlay id 2: Object C (*marked as the 2nd in the tree*) must be checked. It will receive a check order (Coherence Id 2). Its own table indicates that the valid state for this level is "Close".
2) When receiving coherence check id 1, Object B must be in (*or go into*) state "Up". No further transmission, because it ends the tree (*no Object under level 2*). Object D is also checked as a sibling, because Object C has sent a check message to D.
3) When receiving coherence check id 4, Object B must be in state "Stop" (*Or Object C must be into "Close" state, see the special case of "OR" above*). The check is then spread to Object D (*3rd in this tree*).

When all Checkpoint overlays are defined, each node receives the description of its participation in each tree. As seen above, the resynchronization is divided in small trees. In this example, Object B is the root of the tree (Coherence Id number 2). When Object B throws its check, it finishes on Object A (*through Object C and D*). This last Object must be in the state "Sleep". If not, D is resynchronized. In that case, Object D starts another check (*D is root of the tree (Coherence Id number 5*)). This resynchronization puts Object D in the state "Sleep", which is correct if you look to tree (Coherence Id number 2).

The resynchronization mechanism provides a better management of the errors we have encountered while using our platform, but must not add new risks of dysfunctions. The checkpoint overlay does not use locks. There is no blocking instruction; a FST is always listening to input messages. No deadlock can occur during the check of the Choreography. There is still the risk of an infinite loop created by the programmer, but this can be avoided by the analysis of the different trees. For example in Table I, a mistaken check for "Stop" of Object B in the tree Coherence Id number 5 would throw a new verification Coherence Id number 2 that could change the state of Object A, restarting the Coherence Id number 5. An acyclic validation algorithm is applied on Table I before deployment, to prevent this bad design.

EXPERIMENTAL STUDY

In order to correct any potential issue, we first have to determine whether faults may occur or not, because IoT applications have very varied forms. Application checking or fault recovery strategies can be highly variable. If an Object sends messages taken into account by subscribers in a short-cycling logic, fault recovery can be endogenous. For example, if it sends two messages (i.e. "on" and "off "), and if subscribers react with two states (i.e. "open" and "close"), losing one message has limited impact. Often, the consequences of a missed «on» message are automatically resolved by the next "off" message. But some cases are more sensitive to de-synchronization. A choreography that counts events could be an example of no self-recovery application (*counting people entering or leaving a place, or sequencing elements in cascade*). If an event is lost, then the whole application is desynchronized without possibility of self-recovery.

Experiment scenario

To illustrate the dynamicity and the needs for resynchronization, we propose an emergency scenario during a disaster in a public area. In a fully disorganized environment, self-powered objects with wireless connectivity and computing capabilities can be reused to help people finding an emergency exit. Based on disasters stories, it appears that the lack of communication infrastructure has a great impact on rescue efficiency. The quicker communications and services can be restored, the better the rescue.

So the ability to quickly use any remaining efficient wireless devices makes sense. Here, the flexibility of D-LITe can be used to deploy "on the fly" a new application over present Objects. In such a scenario, Objects are reprogrammed to lead people to the emergency exit. The first object flashes, then tells the next one to do so, and so on, showing the path to the nearest exit, for example the corridor. In the corridor, Objects will also flash to lead to the next point, etc. Even if this is a very simple example, we use this demo to show the interest of "On-The-Fly" reprogramming of an IoT Choreography.

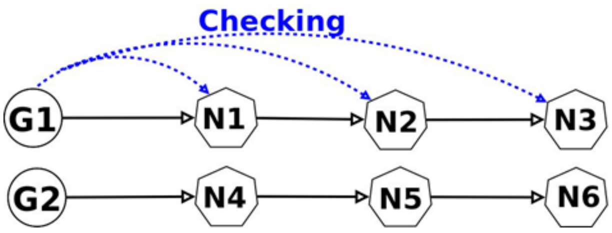

Figure 7. Our experiment uses 2 groups of 4 Objects. Objects Gx generate messages and Nx count them (in cascade). The first group uses checking to resynchronize while the second is here to load the network.

In this scenario, the de-synchronization of the flashing lights cascade leads to a non-understandable message. When an object goes out of synchronization, it will blink at a wrong time, and the whole emergency blinking path becomes incomprehensible to user, potentially leading to undesirable side effects (*even blinking in the wrong direction*). Errors occurrence cannot be avoided, but must be corrected in order to offer a real utility.

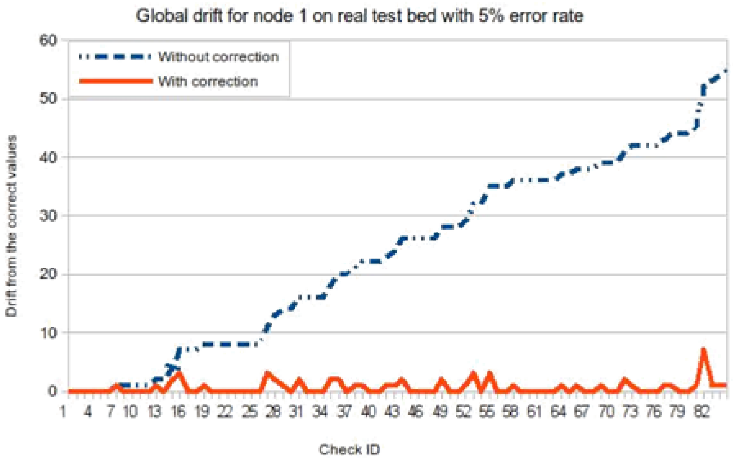

Figure 8. Depending on errors rate and node’s position, messages are lost, and counters results drift from the actual values.

Experiments details

To validate our solution, we have designed an experiment based on counting received events at any stage of the flashing lights cascade. We study the impact of de-synchronization, and the cost and gains of our proposed mechanism.

A version of our framework D-LITe has been improved with checks, running on ContikiOS [Dunkels et al., 2004]. ContikiOS comes with Cooja which emulates Objects, runs the code and simulates the network. We both run experiments on Cooja, and on a testbed using TelosB ([1]A small smart device designed for testing IoT: http://www.memsic.com/). Our experiment uses a first group of 4 Objects: 1 generator and 3 counters (see Fig 7). The generator sends 12 events (*1 per second*), and then waits 10 seconds, and so on. Counters are organized in cascade. The first counter receives the events, counts them, and sends them to the second one, which does exactly the same, and sends them to the third one. Each counter must receive the 12 events. The second group of 4 nodes is similar to the first one. The first group uses checking system while the second does not. In the first group, the generator sends a check during the 10 second pause. The 3 counters are resynchronized if needed. We collect the values given by each counter of each group.

Because errors have very different sources (*network, hardware or even user error*), we have added an error generator, randomly erasing messages following a settable error rate. Our tests are presented with 4 error rates: 0%, 5%, 10% and 15%. We collect experiment's data for at least 1000 initial events in each configuration, in both environment (Cooja and testbed). For each Object, we store the number of received events out of the 12 that are sent. This gives us the amount of useful corrections made in the resynchronized group.

Results analysis

Our tests show that there are differences between Cooja and the testbed. While we encounter nearly no error in Cooja (*99% of checks are ok*), errors appear more often on the real platform *(more than 10% of checks are useful for the first Object, causing more than 25% useful checks for the second*). These differences reflect the wireless network unreliability. In this study, the Cooja simulator, the testbed, and the error generator are compared in order to have a large panel of cases. IoT applications merge different technologies, including reliable and unreliable network, constrained hardware that sometimes support only UDP and not TCP, leading to various capabilities of recovering network errors. This study concentrates mainly of the effects of errors at the application layer, no matter their origin.

Fig 8 displays the counter results for different nodes and error rates. It shows the difference between the numbers of sent and received messages, due to the network poor reliability. With a 0% error rate, there is no need for resynchronization in Cooja, while failures already appear in the testbed. Fig 8 shows the impact of failures on a non self-repairing IoT application. The "correct value" line shows the event counter increasing as time goes by. The other lines show how the value drift from the correct value depending the position of the node and the error rate.

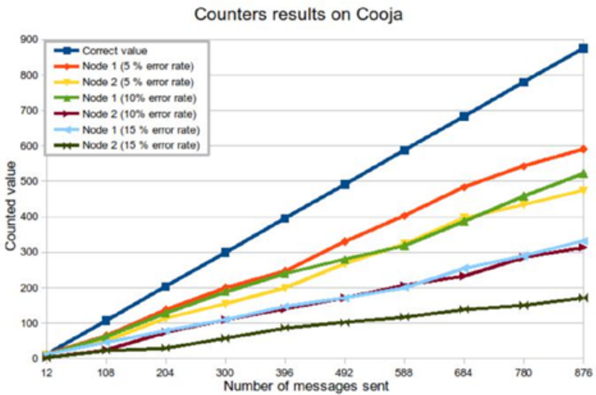

Figure 9. Every 12 events, a check is thrown. The higher the error rate, the more checks are needed to keep close to the correct value. When losses are too important, even check requests are lost, and unreliability increases.

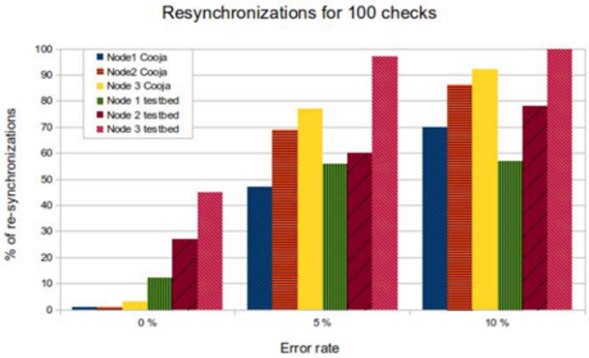

Figure 10. Depending on the check frequency and the error rate, the % of valid states varies. For example, to obtain 70% of valid states when a 10% error rate occurs, our application must be checked every 3 events.

Fig 9 gives a view of the percent of checks that actually led to synchronization. The original code running on Cooja gives nearly no error. For 100 checks, only 1% corrections are needed. Running on a testbed, the same application needs more than 10% resynchronizations for the first Object. Because the 2nd Object depends on the first one, more errors occur. With the introduction of an error rate in our Cooja-based simulation, the number of de-synchronization increases. The curves show how our correction mechanism is increasingly efficient to keep coherence. At a given error rate, the correction mechanism fails because even check messages are lost. A 10% error rate reaches that point for our experiment with the testbed. In that case, the programmer should reconsider his application.

Application reliability versus check frequency

Fig 10 describes the trade-off between application reliability and checks frequency. There is no global rule for tuning the checks frequency as this one depends on the application ability to self-stabilize. Our experiment is designed to be very sensitive to errors, because each Object counts events received from its stakeholder, in a cascade of dependences. Here, losing a single event causes an irrecoverable inconsistency in Object's state when no checking mechanism is running. The longer the application runs without checking, the larger the error. The average reliability of the first and second Objects of the experiment are displayed in Fig 10. The more often a check is made when the error rate increases, the better the valid states rate. When using our coherence mechanism, the application keeps very close to the actual value, avoiding an increasing shift.

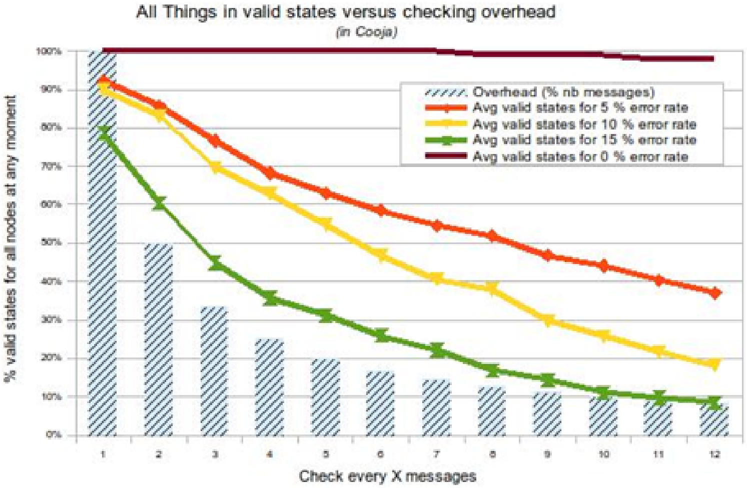

Figure 11. By resynchronizing the Object, our mechanism avoids an increasing shift from the real count of events. When checks are done, the gap remains very low.

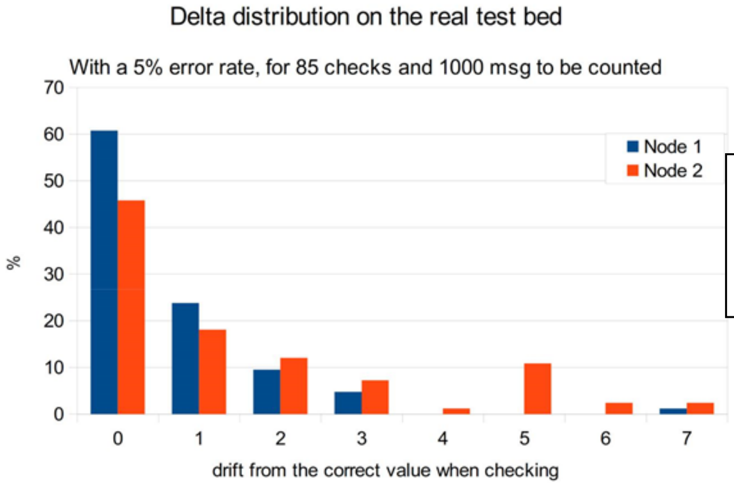

Figure 12. This distribution shows that the checking mechanism gives a good rate of valid results. Drift remains under control.

In our experiment, a 5% error rate gives an average of 50% valid states when checks are made every 8 events. Here, we say that the counter is false when it has missed one event. In case of error, the Object is quickly resynchronized. But the number of exchanged messages increases by 1/8. Fig 10 shows this cost compared to the gain in precision. Depending on the error rate, the curves give the percentage of valid states for a given check frequency. It also shows the communication overload for this level of checks. For example, with a 5% error rate, having all 4 nodes in a correct state with an accuracy of 70% will cost you a 25% communication overload (*with a check spread every 4 messages*). In the remaining 30%, some nodes have a small gap. This gap is resolved in less than 4 messages (*because of the check frequency*).

Fig 11 measures the drift from the actual value (1000 events) of our experiment on the testbed with a 5% error rate. When checking every 12 events, the drift is regularly deleted. The global trend remains very close to the correct value. Deviations due to various errors stay under control. Without checks, the counter drifts more and more.

Fig 12 shows the distribution of the gap between the correct value and what a node has counted. This distribution is based on 85 checks (*one check every 12 events*), with a 5% error rate. The error never exceeds 7, even after more than 1000 events. Depending on the position of the Object, the precision varies. More than 60% of the counted values by Object 1 are correct. If a maximum drift of 1 is tolerated, Object 1 has a correctness of 90% and Object 2 reaches 60%.

Discussion

IoT distributed applications may be self-stabilizing, because some of them are built as a never ending cycle, where one Object leads a global chain of reactions. But if they are not, or if the auto-stabilization does not happen quite regularly, the decentralized algorithm may drift more and more as time goes by. Introducing a resynchronization mechanism gives the programmer a solution to regularly remove this gap in order to keep the whole application under control. The checks frequency of this resynchronization has an impact on the quality of the retrieved data. Thus, according to the desired accuracy and the level of network errors, the programmer selects the range of allowed variability. The more controls he adds, the more overhead is induced. If there are less checkpoints, the drift increases.

The programmer then faces the choice between the correctness of the results given by his application and the impact of the overhead generated by his checks. Depending on the network constraints (in term of energy for example), a slight inaccuracy is tolerable, if errors effects remains under control and at low levels. To gain some precision in results, the programmer can choose to add the limited overload generated by more frequent resynchronizations.

CONCLUSION

In this paper, we presented one of the main issues encountered in IoT Choreographies: the lack of consistency due to the loss of messages. This paper proposes a mechanism adapted to distributed IoT applications in order to keep this de-synchronisation under control.

We have extended D-LITe, our existing IoT framework, to introduce coherence checking, by building an overlay of coherence check points over the different stakeholders of the Choreography. With this extension, the IoT programmer organizes resynchronization requests at given moments of running life cycle of the application, and avoids the appearance of wide inconsistencies due to the multiple potential hardware/software/network failures. Our checking mechanism keeps the Choreography in a tolerable margin of error for a slight message overhead. This margin of error is chosen by the programmer, and controlled by the frequency of his check requests.

As a future work, we think of extending the coherence overlay architecture and expressivity in order to deal with specific checks that IoT applications may require, such as multiple, logical or group checks in order to increase the expressiveness of the system.

REFERENCES

[Behl et al., 2012] J. Behl, T. Distler, F. Heisig, R. Kapitza, and M. Schunter. *Providing fault-tolerant execution of web-service-based workflows within cloud*s. In Proceedings of the 2nd International Workshop on Cloud Computing Platforms, page 7. ACM, 2012.

[Brewer, 2000] E. A. Brewer. *Towards robust distributed systems*. In PODC, volume 7, 2000.

[Canfora et al., 2009] G. Canfora and M. Di Penta*. Service-oriented architectures testing: A survey*. Software Engineering, pages 78–105, 2009.

[Cherrier et al.,2011] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. D-lite : Distributed logic for internet of things services. In IEEE International Conferences Internet of Things (iThings 2011), pages 16–24. IEEE, 2011.

[Cherrier et al., 2012] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. *Services Collaboration in Wireless Sensor and Actuator Networks: Orchestration versus Choreography.* In 17th IEEE Symposium on Computers and Communications (ISCC'12), page 8 pp, Cappadocia, Turquie, July 2012.

[Cherrier et al., 2013] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel, *SALT : a Simple Application Logic using Transducers for the Internet of Things* - Communication Software and Services Symposium (ICC'13 CSS), Budapest, Hungary, June 2013.

[Cherrier et al., 2014] S. Cherrier and Y. M. Ghamri-Doudane. *The "object-as-a-service" paradigm.* In Global Information Infrastructure and Networking Sym-posium (GIIS), 2014, pages 1–7. IEEE, 2014.

[Dunkels et al., 2004] A. Dunkels, B. Gronvall, and T. Voigt. *Contiki-a lightweight and flexible operating system for tiny networked sensors*. local computer networks. In Annual IEEE Conference on, 0, pages 455–462, 2004.

[Egwutuoha et al. 2013] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. *A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems*. The Journal of Supercomputing, 65(3):1302–1326, 2013.

[Gilbert et al., 2012] S. Gilbert and N. A. Lynch*. Perspectives on the cap theorem*. Institute of Electrical and Electronics Engineers, 2012.

[Huang et al., 2005] H. Huang, W.-T. Tsai, and R. Paul. *Automated model checking and testing for composite web services*. In Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on, pages 300–307. IEEE, 2005.

[Rosemblum, 1995] D. S. Rosenblum. *A practical approach to programming with assertions*.Software Engineering, IEEE Transactions on, 21(1):19–31, 1995.

[Sasnauskas et al., 2010] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. *Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment*. In Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, pages 186–196. ACM, 2010.

[Singhal et al., 1995] M. Singhal and F. Mattern. *An optimality proof for asynchronous recovery algorithms in distributed systems*. Information processing letters, 55(3):117–121, 1995.

[Hu et al., 2014] P. H. Su, C.-S. Shih, J. Y.-J. Hsu, K.-J. Lin, and Y.-C. Wang. *Decentral-ized fault tolerance mechanism for intelligent iot/m2m middleware*. In Internet of Things (WF-IoT), 2014 IEEE World Forum on, pages 45–50. IEEE, 2014.

[Zhou et al., 2010] L. Zhou, J. Ping, H. Xiao, Z. Wang, G. Pu, and Z. Ding*. Automatically testing web services choreography with assertions*. Formal Methods