Fault-recovery and Coherence in Internet of Things Choreographies

Sylvain Cherrier*, Yacine M. Ghamri-Doudane†, Stéphane Lohier* and Gilles Roussel*
* *Université Paris-Est , Laboratoire de l'Institut Gaspard Monge (LIGM)*
† *L3i Lab, University of La Rochelle, La Rochelle, France.*

***Abstract*—Facilitating the creation of Internet of Things (IoT) applications is a major concern to increase its development. D-LITe, our previous work, is a framework for that purpose. In D-LITe, Objects are considered as part of a whole application. They offer a REST web service that describes Object capabilities, receives the logic to be executed, and interacts with other stakeholders. Then, the complete application is seen as a choreography dynamically deployed on various objects. 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, leading to errors and failures. In this paper, we propose a solution to re-introduce coherence 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 links 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 Choreographies. This paper ends with a comparison between the checking cost and the reliability improvement.**

***Keywords*-Internet of Things; Choreography; Fault-tolerance; Fault-recovery**

I. Introduction

The Internet of Things 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[3], a framework for IoT applications creation and deployment, based on a *services choreography*. The main advantage of *services choreographies* stands in the distribution of the application logic across the network. The absence of central point leads to a better dissemination of the load and a more efficient use of energy, especially in constrained networks[4].

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 failures. For example, some stakeholders may miss steps in their choreography and the whole application becomes incoherent. IoT uses wireless links to make Objects communicate as it integrates WSAN (Wireless Sensors and Actuators Network). These wireless links are characterized by their unreliability. Tests on our testbed show that we still experience application failures, in spite of controls made at lower layers.

In this paper, we propose a new mechanism allowing the IoT programmer to introduce coherence controls in order to correct the logic of Objects involved in a global application. This coherence checking is deployed along with the services choreography. Some Objects will have to start a check to a list of '*to-be-checked*" Objects. They will check their own state, make correction if needed, then transmit in their turn the coherence control orders to their own list of "*to-be-checked*" Objects.

This paper is organized as follow: Section II presents related work while Section III describes our IoT solution D-LITe. The coherence checking architecture is described in Section IV. Section V contains our experimental study and our results. Finally, concluding remarks and future research directions are given.

II. Related work

Our approach of Internet of Things applications[3] is part of the Services Oriented Architecture(SOA) realm. SOA offers a distributed composition of programs in a "*loosely coupled platform-independent model*"[11]. G. Canfora and M. Di Penta present the specificity of SOA as "*radical changing [in] the development perspective*"[2]. 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 distributed software composition over heterogeneous hardware may lead to denial-of-service if too many controls are performed.

Testing Web services collaboration can be done at design time. H.Huang *et al.*[7] 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.*[11] propose the use of assertions that "*express the intention of the program by designers*". A simulator processes each web services and builds the complete interactions combinations. All path are checked and assertions are verified. This tool can detect design errors. However, we are more interested in adding global fault checking to running choreographies.

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[9] is a tool to test the reliability of a distributed solution. It aims at testing the behaviour of a choreography when unexpected errors occurs. KleeNet triggers network error on a running application, and has been able to detect design errors in ContikiOs, the open-source OS for IoT devices[6].

In this paper, we are interested in errors that happen while an application is in use. Our target is to offer a mechanism able to correct the impacts of malfunctions. 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, while KleeNet is used to re-design faulty applications. Our solution uses assertions to express control points[11]. 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[10], 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[1]. However, the authors obtain robustness by replication of the web services. On usual web, one can duplicate services, thanks to powerful hardware. In our case, IoT objects that interacts with the real world are often unique, and replication is not possible. On IoT, each object has a specific role (mainly because of its position and its unique point-of-view) and has often low computing capabilities.

III. Our platform: D-LITe

In previous works, we have presented our framework D-LITe[3] for designing Choreographed applications for IoT. By introducing hardware abstraction, D-LITe offers a easy way to create Iot applications. the Objects discovery and the application deployment add a real usability to our solution. The logic (the part of the application an Object must execute) is expressed with *Finite State Transducers* (FST)[5]. Automata are a simple way to express logical sequence of actions driven by events. FST are a good compromise between the need of programming Objects, the reduced set of their possible actions and the ease of understanding a programming language.

Each Object receives its own FST to be executed and a list of subscribers. The output of its Transducer becomes the input of subscribed Transducers. D-LITe is event-centric, an event being a received message or a change in the environment (for Objects with sensing capabilities). The event is treated, makes a Transition in the Transducer. Output messages are generated, that can be external ones (for subscribers) or intended to Object's hardware (for Object with actuating capabilities). D-LITe builds choreographies of logic running on Objects.

IV. Coherence checking overview

A. Motivation

Rather than trying to fix all errors that may have occurred at different levels and in different places, the idea of putting all the 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 programming language (i.e. C language[8]). He uses these predicates to bring back correctness in the global system. Errors causes can vary widely and may not be reproducible. A IoT choreography depends 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 mechanism did not manage it, seems to be out of reach.

To illustrate the need, 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 re-used to help people finding an emergency exit. Here, the flexibility of D-LITe can be used to deploy "*on the fly*" a new application on Objects, using their led 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.

In that scenario, the de-synchronization of the process leads to a non-understandable message. When an object goes out of synchronization, it will blink at the wrong time, and the whole message becomes incomprehensible. Errors occurrence cannot be avoided, but must be corrected.

B. Check-points overlay

In D-LITe, each logical element is a FST. Improving the coherence inside this application is based on the 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 another one. For example, in home automation, one can say that when the house door is closed, every light must be switched off. In our emergency scenario, we can assert that all the lights must be switched off before a new cascade of flashes begins.

Although very few 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, identified by a *Coherence Level* (an id he gives). The subset of Objects is not the same as its usual subscribers

Table I
A Check Point Overlay

Coherence Level	Object A	Object B	Object C	Object D
1	—	Up(1)	Close (S)	Dark (1)
2	In charge(3)	Stop(S)	Close(1)	Light(2)
3	Wake up (S)	—	Open(2)	Light(1)
4	Sleep(S)	Stop OR Close(1)		Light(2)

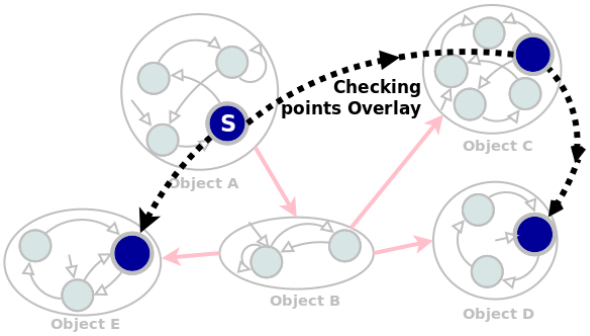


Figure 1. a D-LITe application is a choreography of FST. Each object follows its own logic. To improve coherence, combinations of stable states for the Objects are defined. Initial states are one of these groups, but there are probably some others. Programmers build an overlay of coherent states among all possible combinations. The root of each overlay starts a cascading check to ensure the coherence of each element.

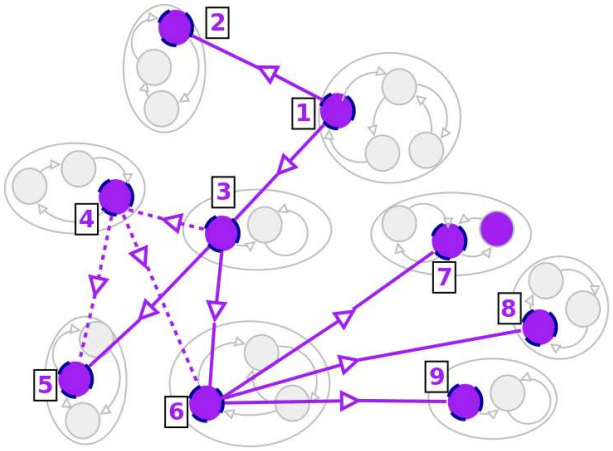


Figure 2. Checking points overlay is a tree that uses some of the Objects involved in the application. Only one *coherence level* is represented here. Object 1 (tree's root) is in charge of starting the check (when it reaches the indicated State). The check request follows the tree, each node checks itself, and spreads the check to followers. A "*OR*" is organized between node 3 and 4. If 3 is not valid, node 4 is checked. This *coherence level* accepts two valid States for node 7.

list because the coherence checking range is different (i.e. the house door and all the lights). The root Object throws the check (S in Table I and Fig 1).

For each *coherence level* defined by the IoT programmer, a root Object must be identified. It defines the tree of each specific *coherence level* (Table I) in order to spread this check in cascade. Each tree (S-1-2-3...) can be different from the tree used by the application Fig 1. Each *coherence level* has its own root and its specific dependences.

C. Coherence check spreading

Fig 2 shows a more complete *coherence level* overlay. It is the programmer's duty to organize his tree. A demonstration of a logical "*OR*" is made in Fig 2. If 3 is in a valid State when checked, the check is transmitted to following Objects

Algorithm 1 Reception of a Check message

Require: $ckpRs$ Array of $ckpRcv$, $currentState$,
Require: $RcvChkId, RcvChkNumber, LastChkNumber$
Ensure: Verify state, change if needed, and propagate
if $RcvChkNumber \neq LastChkNumber$ **then**
$LastChkNumber \leftarrow RcvChkNumber$
$i \leftarrow 1$
while $ckpRcv[RcvChkId].states[i] \neq null$ **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
$i \leftarrow i + 1$
end if
end while
{Coherence error: alternate list or pushing FST in any right state}
if $ckpRcv[RcvChkId].altTargets = null$ **then**
$newState \leftarrow 1 + (random() \bmod (i - 1))$
changeFSTto($ckpRcv[RcvChkId].states[newState]$)
for all ip in $ckpRcv[RcvChkId].targets[newState]$ **do**
sendChkMsg($ip, RcvChkId, chkNumber$)
end for
return true
else
for all ip in $ckpRcv[RcvChkId].altTargets[]$ **do**
sendChkMsg($ip, RcvChkId, chkNumber$)
end for
end if
end if
return false

(5 and 6). But if 3 is invalid, no change is done, and 4 is checked. That 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). The "*OR*" is implemented through an *alternate Objects list* (Algorithm 1).

The coherence checking tree (Fig 2) is described by the programmer. He organizes his checks in cascade to avoid too many dependences on a single Object ("*to-be-checked*" Objects list may not fit in the very small Object memory). Each time a FST moves to a new State, it scans its checking table (used only for tree's root node) to see if a check is required. In that case, the coherence mechanism throws a check message to the "*to-be-checked*" Objects list with the *coherence level* identifier and a *random number* (to avoid loopbacks).

The check request is received and managed inside each

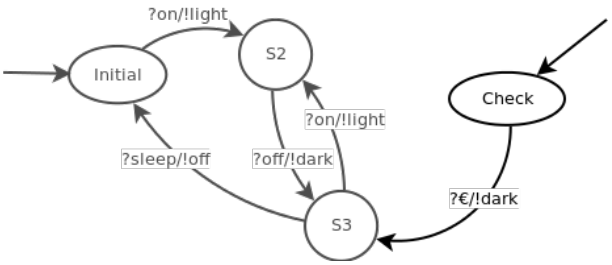


Figure 3. The FST running on an Object may describe what to do if a resynchronization is needed. Here, for the lamp receiving "*on*" and "*off*", and changing for "*Dark*" to "*Light*", an added State (*Check*) has a transition that switch off the light. If this added State is ever used, it is because we need to correct Object's State.

"*to-be-checked*" Object. The algorithm 1 shows the search of this check in the Object's table. When the *coherence level* is found, the current FST state is compared to the list of coherent States list. If it matches, the Object is safe, and the algorithm goes on by sending the check to its own list of "*to-be-checked*". If the State is not valid, two cases occurs:

- the alternate "*to-be-checked*" list 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 an alternate "*to-be-checked*" list. Alternate "*to-be-checked*" list is useful to express an "*OR*" in the checking logic. No correction is made here. The check is simply send to all Objects of the Alternate list.

The cascading coherence system makes each Object able to jump directly to a given state. 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 re-synchronization of some Objects. In that case, we also need to really switch off lights (jumping to the state "*dark*" remains the light on). That's why we introduce in Fig 3 a new transition that is triggered when the coherence mechanism forces the FST to be in a valid state. To respect automata formalism, it can be seen as adding a new initial state to the FST (Fig 3) that will be used by the coherence check mechanism.

V. Experimental study

We first have to determine whether faults may occur or not, because IoT applications are very varied. 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 not a serious impact. But some cases are more sensitive to de-synchronization. There is no self-recovery when counting informations (counting people entering or leaving a place, or sequencing lights as in our emergency scenario). If the event is lost, then the whole application is desynchronized without possibility of self-recovery.

A. Experiments details

To validate our proposition, we have designed an experiment based on counting received events. We study the impact of de-synchronization, and the cost and gains of our proposed mechanism.

1) Experiment scenario: We use a version of our framework D-LITe improved by checks, running on ContikiOs[6]. 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]. Our experiment uses a first group of 4 Objects: 1 generator and 3 counters. 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.

2) Metrics used: Because errors have very different sources (network, hardware, framework implementations or even user error), we 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 how many events are received out of the 12 send. This gives us the amount of useful corrections made in the re-synchronized group.

B. Results analysis

Our tests show that there is a difference between Cooja and the testbed. While we encounter nearly no error in Cooja (99% of checks were valid), 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). This difference reflects effect of the wireless channel un-reliability. Furthermore, the TelosB emulator in Cooja executes instructions faster, thus avoiding bottlenecks related to the real TelosB processor frequency and hardware reactivity.

Fig 4 displays the counter results for different nodes and error rates. It shows the difference between the number of send and received messages, due to the network poor reliability. With a 0% error rate, re-synchronization is useless

[1]A small smart-device designed for testing IoT http://www.memsic.com/

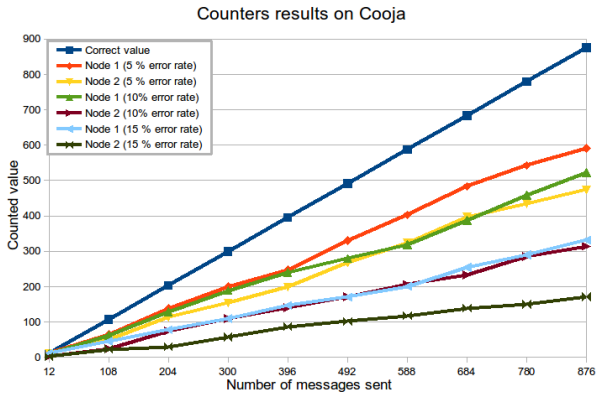


Figure 4. Depending on error rate and node's position, messages are lost, and counters results drift from the actual values.

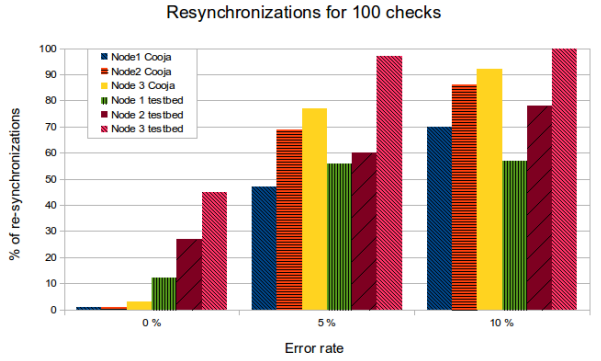


Figure 5. Every 12 events, a check is thrown. The higher the error rate, the more checks are needed to keep close to the correct value. When it is too important, even check requests are lost, and unreliability increases.

in Cooja, while failures already appear in the testbed. Fig 4 shows the impact of failures on a non self-repairing IoT applications.

Fig 5 gives a view of the rate of checks that actually led to a synchronization. The original code running on Cooja gives no error. For 100 checks, only 1% corrections are needed. Running on a testbed, the same application needs more than 10% re-synchronizations 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 experiment, the number of de-synchronization increases. The graph shows 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.

C. Application reliability versus check frequency

Fig 6 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

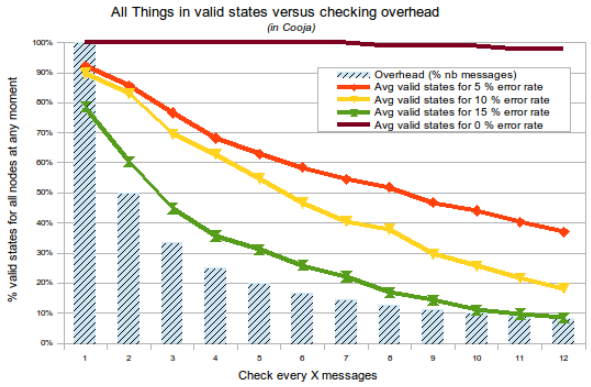


Figure 6. Depending on the check frequency and the error rate, the % of valid states vary. For example, to obtain 70% of valid states when a 10% error rate occurs, our application must be checked every 3 events. However, with our coherence system, results are very close to the actual value during the remaining 30%.

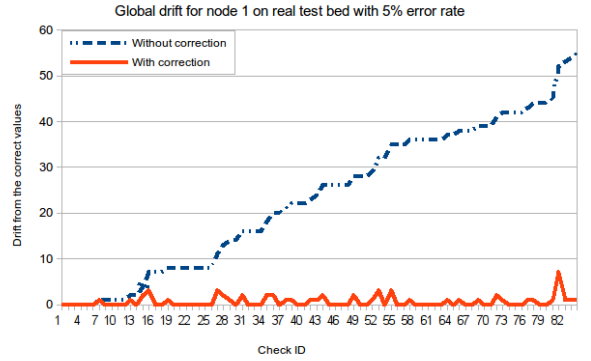


Figure 7. By resynchronizing the Object, our mechanism avoids an increasing shift from the real count of events. When checks are done, the gap remains very low.

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 inconsistent in Object's state when no checking mechanism is used. 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 6. 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.

In our experiment, an 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 of 1/8. Fig 6 shows this cost compared to the gain in precision.

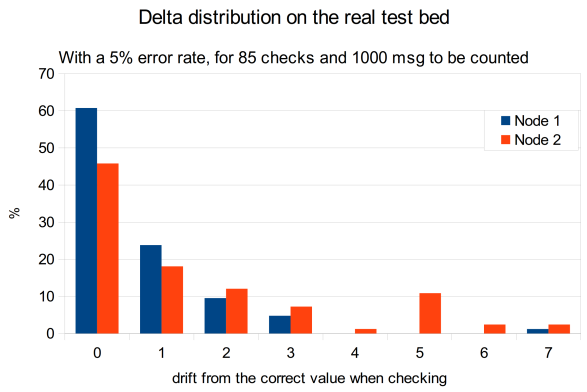


Figure 8. This distribution shows that the checking mechanism gives a good rate of valid results. Drift remains under control.

Fig 7 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 real value. Deviations due to various errors stay under control. Without checks, the value drifts more and more.

The checks frequency has an impact on the quality of the retrieved data. Thus, according to the desired accuracy and the amount of network errors, the programmer selects the range of allowed variability. The more controls he adds, the more overhead is induced. If he sets less check-points, the drift is widening. Fig 8 shows its distribution when checked every 12 events, with a 5% error rate. It never exceeds 7, even after more than 1200 events. Depending on the position of the Object, the precision varies. More than 60% of the counted values by Object 1 are valid. If you accept a maximum drift of 1, Object 1 has more than 90% correctness and Object 2 reaches 60%.

VI. CONCLUSION

In this paper, we presented one of the main issues encountered in IoT Choreographies: the lack of consistency due to lost messages. This paper proposes a mechanism to keep this de-synchronization under control.

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

In future work, we will extend the coherence overlay architecture and expressibility 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

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

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

[3] 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.

[4] 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.

[5] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. SALT: a simple application logic description using transducers for internet of things. In *IEEE International Conference on Communications - Communication Software and Services Symposium (ICC'13 CSS)*, Budapest, Hungary, June 2013.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] L. Zhou, J. Ping, H. Xiao, Z. Wang, G. Pu, and Z. Ding. Automatically testing web services choreography with assertions. *Formal Methods and Software Engineering*, pages 138–154, 2010.