Formal Modeling and Validation of ReDy Architecture Intended for IoT Applications

Kaoutar Hafdi, Abdelaziz Kriouile, Abderahman Kriouile

Abstract — **Internet of Things (IoT) faces different architectural challenges to meet the large scale application issues, the heterogeneity, and the self-adaptivity.**

Many IoT applications require a dynamic construction of the system and should ensure a high degree of reliability. To this end, we propose the ReDy architecture [1], which is a reusable solution for reliable and dynamic distributed IoT applications.

In this paper we propose a formalization and validation of the ReDy architecture. For this end, we propose a formal model using LNT language [2]. We propose also a suitable algorithm to implement a reliable and dynamic membership management. Then we give a formal validation of this critical part based on formal modeling and model checking techniques [3].

Index Terms — **Internet of Things (IoT), Distributed systems, Formal methods.**

I. INTRODUCTION

Internet of Things (IoT) is the next wave of digital transformation: many objects that surround us will be connected. Sensors, actuators, and computing units will form networks for different applications. Those applications may concern home and personal uses, enterprises uses, public utilities, or transportation. The IoT is already ready for some specific uses as the majority of personal uses and some enterprise uses. Today, different challenges are still open. In particular architectural challenges to meet the large scale application issues, the heterogeneity, and the self-adaptivity of complex systems. A large part of current work of the IoT architecture have been inherited from the wireless sensor networks background [4]. Other architectures should be investigated for different application domains [5].

Kaoutar Hafdi, IMS Team, ADMIR Laboratory, Rabat IT Center, ENSIAS, Mohammed V University, Rabat, Morocco,

Abdelaziz Kriouile, IMS Team, ADMIR Laboratory, Rabat IT Center, ENSIAS, Mohammed V University, Rabat, Morocco,

Abderahman Kriouile, Farasha Systems, Rabat, Morocco and SUPMTI, Rabat, Morocco

Many IoT applications require a dynamic construction of the system and should ensure a high degree of reliability. For this end, we propose the *ReDy distributed systems* for Reliable and Dynamic distributed systems [1]. Those systems are designed using the *ReDy architecture*. The ReDy architecture is a reusable solution for a large spectrum of distributed systems. Our solution integrates two important requirements that are common to the concerned systems. The first one is to design the system in a highly dynamic environment, i.e, components can continuously join and leave the system network. The second requirement is about fault tolerance. The designed system should have a high resistance to faults, which permit to preserve the overall behavior of the system even in the presence of faulty components. A large family of systems needs to guarantee the above requirements. Besides proposing a common architecture for those systems, our solution gives general rules that should be respected and implemented during the design phase so as to construct reliable and dynamic distributed systems.

As application samples we have: large scale wireless sensor networks, deploy and forget networks, self-adaptive systems of systems, critical infrastructure monitoring, smart grid and household metering, autonomous vehicles, heterogeneous systems with interaction between other sub-networks, smart traffic, intelligent transportation and logistics [5].

We take the advantage of the formal methods, witch are a particular kind of mathematically based techniques for the specification and verification, to model the ReDy architecture and to validate complicated behaviors. The use of formal method allows us to ensure a good level of reliability and robustness of our proposed design.

Formalizing our system using a formal model let us to express the behavior of the system in an unambiguous way: the formal specification expresses a unique semantic. In addition to that, this formal model can be validated using automatic and exhaustive formal methods.

That is why we opted for using formal methods in modeling and validation of our proposed architecture.

Contributions: In this paper we propose a formalization and validation of the ReDy architecture [1] which is a reusable solution for the different IoT applications presented above. We propose a formal model for the ReDy architecture using LNT language [2]. Then we focus on the most critical part of this architecture which is the

membership management. We propose an algorithm to implement a reliable and dynamic membership management. Then we propose a formal validation of this critical part based on formal modeling and model checking techniques [3]. The formal validation uses the CADP toolbox [6].

Outline: The rest of this paper is organized as follows.

Section II presents the ReDy distributed systems architecture. Section III describes the proposed formal model of the ReDy architecture. Section IV details the enhanced shuffling algorithm used for membership management of ReDy systems. Section V exhibits the formal validation work of the proposed algorithm.
Section VI surveys related work.

II. REDY SYSTEMS

Many distributed systems have several common requirements, regardless of features relative to the field on which each system is applied. The common requirements can be gathered and analyzed in order to propose reusable solutions for each family of similar systems.

Our objective is to propose a solution for a family of distributed systems designed for highly dynamic applications in hazardous environments. We call those systems *ReDy distributed systems* for Reliable and Dynamic distributed systems. The ReDy solution focuses on the dynamic construction of the system and ensures interactions between the components of the system. Our proposition combines solutions proposed in the literature in a packaged solution to be reusable for distributed systems derived from different application fields and having several common general requirements.

A. ReDy Architecture

The ReDy architecture stipulates that the global system is composed of several subsystems. Each subsystem is a distributed system and consists of several components. There are three types of components:

1) *Detection units* are components in touch with the external environment and are responsible of detecting environmental changes. In general, those units are representing sensors. The nature of the sensor depends on the type of events that should be detected.

2) *Action units* are components in touch with the external environment and are responsible of executing actions affecting the environment. In general, those units are representing actuators. The nature of the actuator depends on the type of events that should be achieved.

3) *Governance units* are in charge of collecting information from detection units and taking adequate decisions according to the analysis of the received information. The decision is then sent to action units. We should notice that each subsystem has a unique governance unit, and that governance units of subsystems can communicate between each others. In order to strengthen the fault tolerance of the system, this vital unit is replicated. The replica takes over in case of the crash of the principal governance unit.

Each subsystem is composed of one governance unit and several detection and action units.

Example: Figure 1 illustrates an example of the global architecture of the ReDy distributed system. In this example, the global system is composed of three subsystems. Each subsystem contains one governance unit. The first subsystem is composed of three detection units and two action units. The second subsystem is composed of one detection unit and three action units. The third subsystem is composed of two detection units and two action units. This is a very simplified model used just for illustration purposes. The governance units are connected to each other, while the detection and action units of each subsystem are connected only to the governance unit of this subsystem.

ReDy systems are designed according a hybrid architecture which combines centralized solution, i.e, client-server solutions, for the communication between components inside a subsystem, and decentralized solutions for the communication between different governance units.

In our work, we focus on studying how the network of governance units is constructed and how the communication between governance units is handled, which corresponds to the decentralized part of the system.

B. Membership Management

In this part, we present how the components of our ReDy distributed system are organized.

The decentralized part of our distributed system is constructed following an *unstructured peer-to-peer architecture* [7]. In the remaining, the governance units represent the nodes of the unstructured peer-to-peer architecture. The principal reasons that motivate our choice for unstructured peer-to-peer architecture is that it is the most convenient architecture for highly dynamic environments, i.e, the performance is not deteriorated by nodes leaving and joining the system [7]. As a result, this architecture is adapted for systems with potentially major failures. In addition to that, the communication on such architectures is achieved by epidemic broadcast, which corresponds to our requirements, since the communication between governance units is realized by disseminating the information over all units.

The graph construction: Unstructured peer-to-peer systems are built using randomized algorithms. The main idea of such systems is that each node maintains a list of neighbors such that this list is constructed in a more or less random way [8]. This list is called *partial view*. There are many ways to construct such a partial view. In this paper, we propose to use the enhanced shuffling algorithm proposed by Voulgaris et al in [9] and formalized by Jelasity et al in [10], [11].

In the enhanced shuffling algorithm, each node maintains a list of c neighbors. The basic idea of this algorithm is that the nodes exchange their list of neighbors

periodically (shuffle operations). The exchanged list is of

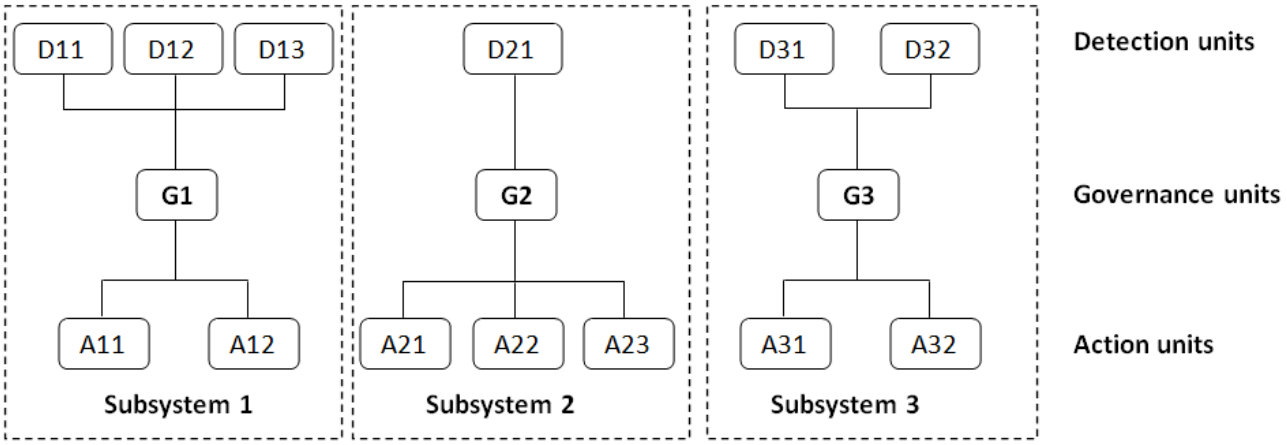


Figure 1: ReDy Archtecture Example

the same length for all nodes. This length is called *Shuffle Length* (*SL*) and it is always smaller than *c* (the number of neighbors). An other important parameter used in this algorithm is the neighbors *ages*. For a given node of the network, having a list of neighbors, we give for each neighbor a number that we call *Age*. The age of all neighbors is increased by one when shuffling (executing a shuffle operation). Such a parameter allows to know the oldest neighbor in the list.

Example: Figure 2 shows a shuffle iteration. The peer 4 initiates the shuffling. It chooses the peer 6 because it is the oldest neighbor. The length of the neighbors list is c = 5 and the shuffle list is SL = 3. Peer 4 sends the list {4,2,8} to peer 6, which updates its entries and sends back to peer 4 the list {2,5,3}. Peer 4 updates its entries and both peers 4 and 6 hold an updated neighbors list.

Joining or living the network: To join the network, a node has just to find one node in the overlay and to be tied to it. The joining node should construct its neighbors list and should be included in neighbors lists of other nodes. This is achieved by the periodic shuffling.

To leave the network, there is nothing to do: just leave. The system will adapt and ignore the node that has left when it is not responding. This feature provides a high failure resistance because a failed node can not inform the system when failing.

C. Communication Model

The communication in our system is achieved by disseminating the information over all communicating peers. In other words, the most convenient communication model in our case is the broadcast.

Our system should realize a high degree of reliability. This is achieved by tolerating failures and dealing with them in order not to deteriorate the overall functioning of the system. Selecting processes and links nature is of a great importance for reaching this objective. Practically speaking, we should ensure that a process that fails and recovers later can continue participating in the system. This is abstraction made about the process which is called: *crash-recovery process abstraction* [12], [13]. The communication channel abstraction defines the point-to-point link. This link should fit with the abstraction made in processes and ensures that even if a process crashes, it will be able to deliver sent messages over the network after recovering. This feature is ensured by using *stubborn link abstraction* [12], [13].

Taking into account those two abstractions, the devised algorithm used for the components communication is a *Uniform Reliable Broadcast* [13] and it is designed using four principal events: Initialization event, Recovery event, Broadcast event and Delivery event.

The reliable variant of this algorithm ensures that even if the sender crashes at the middle of a broadcast operation, all correct processes will deliver the message. The uniform variant of the algorithm ensures that if a faulty process delivers a message, then all correct processes deliver this message, i.e, the set of messages delivered by faulty processes is a subset of messages delivered by correct processes.

In our paper, we focus on the membership management of our system. For further information about the communication model, see our previous paper [1].

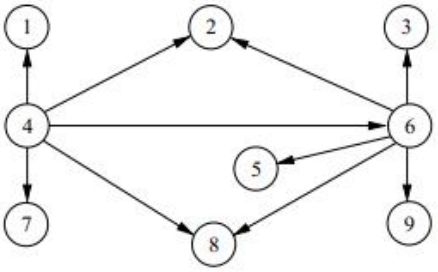
III. ReDy Architecture Formal Modeling

In this section, we start by presenting the formal language used to model our ReDy architecture. After that, we present the formal model of the ReDy architecture. We focus on giving the defined modules and processes that are used.

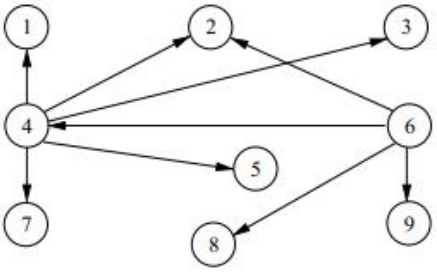
A. CADP toolbox and LNT language

CADP (Construction and Analysis of Distributed Processes) is a toolbox for the design of asynchronous concurrent systems [6]. CADP supports several process calculi specification languages and offers various tools for simulation and formal verification, including model checkers (temporal logics and modal μ-calculus). CADP is designed in a modular way and puts the emphasis on intermediate formats and programming interfaces,

enabling to combine CADP tools with other tools and adapting to various specification languages. Today, CADP contains around fifty tools and more than a dozen libraries.



(a) Before shuffling



(b) After shuffling

Figure 2: Shuffling algorithm

LNT [2] (a shorthand for "LOTOS New Technology") is a modern formal specification language that has been designed and implemented in the CADP toolbox since 2005. LNT is intended to be concise, expressive, easily readable, and user-friendly. LNT combines the best features of process calculi, functional programming languages, and imperative programming languages. The semantics of an LNT model is defined as a *Labeled Transition System* (LTS) [14], following a black box view of the system.

B. Fomally Modeling ReDy Systems with LNT Language

In this part, we present the formal modeling of the ReDy architecture using LNT language. To achieve that, we define six modules communicating between each other. The module *main* uses the module *sub_system*. The module *sub_system* uses three modules: *governance_unit*, *action_unit* and detection_unit. The former three modules use the module *types* (Figure 3).

The *types* module defines the different data types used in this model. First of all, we define the *index_sub_system* as an index for the subsystems composing our system. In our cases we limit the range of possible index to 4 subsystems because we define four subsystems in our global system. We instantiate different predefined functions for this type, which are the equality comparison "==", the inequality comparison "<>", the inferiority "<=", and the strict inferiority "<".

```
type index_sub_system is
  range 1 .. 5 of Nat
  with "==","<>","<=", "<"
end type
```

We define also the types *index_detection* and *index_action* which are indexes of action units and detection units. Those types are defined with the same predefined function as above.

```
type index_detection is
  range 1 .. 3 of Nat
  with "==","<>","<=", "<"
end type
```

```
type index_action is
  range 1 .. 3 of Nat
  with "==","<>","<=","<"
end type
```

In the *main* module we start by the system variables declaration, then we initialize these variables. The system variables are then transmitted to the subsystems as parameters. Each subsystem is identified by a subsystem index *index_sub_system*.

We define a parallel composition between several subsystems. In this case we have four communicating subsystems. This parallel composition is defined with communication on several gates.

In the following, we focus on the gates used for communications among system elements, corresponding to both centralized communication and decentralized communication.

The gates used for centralized communication inside a subsystem are DETECTION, DECISION, ACTION gates.

The gate used for decentralized communication between different subsystems is FORWARD gate, which is used to inform other subsystems that a problem is detected. In the following, we note *GATES1* the set of gates DETECTION, DECISION, ACTION, and FORWARD gates.

```
module main(sub_system) is
process MAIN [GATES1]
is
 var <<system variables>>
 in
  ... -- variables initialization
    par FORWARD in
      sub_system [GATES1]
           (<<system variables>>,index_sub_system(1))
    ||
      sub_system [GATES1]
           (<<system variables>>,index_sub_system(2))
```

```
    ||
      sub_system [GATES1]
          (<<system variables>>,index_sub_system(3))
    ||
      sub_system [GATES1]
          (<<system variables>>,index_sub_system(4))
    end par
  end var
 end process
 end module
```

In the *sub_system* module, we define the parallel composition between subsystem components. We can define different subsystem structures with different numbers of detection units and different numbers of action units. In this case, we present an example of a subsystem

```
      end par
    end par
end process end module
```

Governance_unit module consists of two processes. The *governance_unit* process initializes and calls the *membership_management* process.

This process is responsible of the construction of components network. In particular, this process is responsible of adding and removing subsystems to the neighbor list of the present subsystem. In the following *MM Gates* denotes *Membership Management Gates*.

```
module governance_unit (types) is

process governance_unit[DETECTION, ACTION,
```

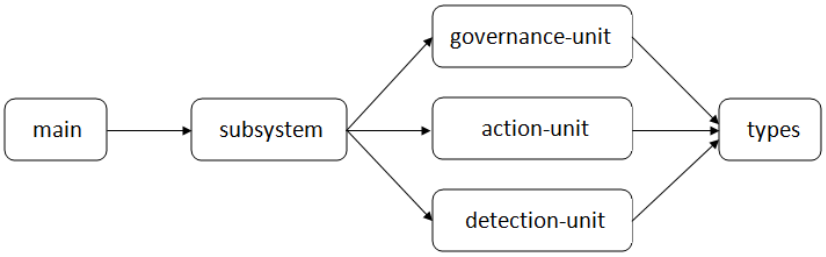


Figure 3: LNT Modules

with one governance unit communicating in parallel with two action units and three detection units.

```
module sub_system(governance_unit,action_unit,
  detection_unit)  is

process sub_system [GATES, SHUFFLING_TRANSFER]
  (nb_neighbors,SL:NAT,id_sub:index_sub_system)
is
 par DETECTION, ACTION in
  governance_unit[DETECTION,DECISION,ACTION,
     FW_DETECTION,SHUFFLING_TRANSFER]
     (nb_neighbors,SL,id_sub)
  ||
    par
       par
         action_unit[ACTION] (id_sub, index_action(1))
         ||
         action_unit[ACTION] (id_sub, index_action(2))
       end par
      ||
       par
          detection_unit[DETECTION]
                   (id_sub, index_detection(1))
          ||
          detection_unit[DETECTION]
                   (id_sub, index_detection(2))
          ||
          detection_unit[DETECTION]
                     (id_sub, index_detection(3))
        end par
```

```
 FORWARD_DETECTION,MM Gates]
   (<<membership management parameters>>,
     my_id_sub:INDEX_SUB_SYSTEM)
is
  var <<local variables definition>>
     new_detection : bool,
     action_to_do : bool
  in
    -- initialization
    <<membership list initialization>>
    new_detection := false;
    action_to_do := false;
    membership_management[DETECTION,ACTION,
        FORWARD_DETECTION,MM Gates]
      (<<MM parameters>>,
          my_id_sub, new_detection,action_to_do)
  end var
end process
-----------------------------------------
process membership_management[DETECTION,
 ACTION, FORWARD_DETECTION,MM GATES]
   (<<MM parameters>>
    my_id_sub:INDEX_SUB_SYSTEM,
    new_detection : bool,
    action_to_do : bool)
is
 var <<local variables definition>>
 in
    <<membership management code>>
    membership_management [DETECTION, ACTION,
    FORWARD_DETECTION,MM GATES]
```

```
   (<<MM parameters>>,
      my_id_sub, new_detection, action_to_do)

 end var
end process end module
```

Detection_unit module is implemented by a process composed by a permanent loop. This loop expresses permanent iterations of a non-deterministic choice between two branches. The first branch is a DETECTION action which means that the detection unit informs the corresponding governance unit that an event is detected. The second branch is an intern action *i* which expresses that the detection unit does not detect an event in this iteration.

```
module detection_unit (types) is

process detection_unit[DETECTION : any]
 (id_sub:index_sub_system,id:index_detection)
is
  loop
      select
        DETECTION(id_sub,id)
      []
        i
      end select
  end loop
end process
end module
```

Action_unit module is implemented by a process composed by a permanent loop. This loop expresses permanent iterations of a non-deterministic choice between two branches. The first branch is an ACTION action which means that the action unit receives an action order from the corresponding governance unit. The second branch is an intern action *i* which expresses that the action unit does not receive any action from the corresponding governance unit.

```
module action_unit (types) is

process action_unit[ACTION:any]
 (id_sub:index_sub_system,id:index_action)
is
  loop
      select
        ACTION(id_sub,id)
      []
        i
      end select
  end loop
end process
end module
```

IV. SHUFFLING ALGORITHM

In the following, we focus on the membership management part between subsystems in particular between the subsystem elements that communicate with the other subsystems which are the governance units. In this part, we have a decentralized mode where each subsystem is a node in a peer to peer communication with the other nodes.

In this chapter, we propose a formalization of the enhanced shuffling algorithm used for the membership management of our system components. We start by defining all variables and data types we need.

- *nb_neighbors* : an integer representing the number of neighbors of a given node.
- *SL* : an integer representing the number of elements in the shuffle list. This number is always the same in all shuffle iterations and it is always smaller than the number of neighbors (*SL* < *nb_neighbors*).
- *add_T* : is a data type to define the type *address*. Each node has a specific and unique address which is an integer such that $1 \leq add_T \leq nb_neighbors$.
- *neighbor_T* : a data type to define a neighbor. A neighbor could be *empty* which corresponds to an empty neighbor, or non empty and defined by an address and an age. The age of all neighbors is increased by one on each shuffle iteration.
- *neighbors_list* : the set of neighbors of the node
- *shuffle_list* : a set of neighbors of the initiating node contained on its neighbors list. This list is sent by the initiating node to the receiving node in a shuffle iteration.
- *shuffle_list_Q* : a set of neighbors of the receiving node contained on its neighbors list. This list is sent by the receiving node to the initiating node in a shuffle iteration.
- *neighbor_Q* : the oldest neighbor in the neighbors list of the initiating peer. *add_Q* and *age_Q* are the address and the age of *neighbor_Q*.
- *Myadd* : element of type *add_T*, used to define the address of the current communicating peer.

On each shuffle iteration, we have two interacting peers: the initiating peer and the receiving peer. In the following, we note *P* the initiating peer and *Q* the receiving peer.

- Case 1 : The initiating peer

The first step in a shuffle iteration is to increase by one the age of all neighbors of the initiating peer *P*.

The second step is to select a set of neighbors from the neighbors list, which corresponds to the shuffle list that will be exchanged in a shuffle iteration. We start by selecting *neighbor_Q* with the highest age among all neighbors. We achieve that by browsing the neighbors list of the initiating peer *P* looking for the highest age. *neighbor_Q* corresponds to the receiving peer in a shuffle iteration. Then we select *SL-1* other random neighbors that we put in the shuffle list. *Random_neighbor* is a function that returns a randomly chosen neighbor from *neighbors_list*.

```
add_Q = neighbors_list[1].add
for (i = 1, i < nb_neighbors, i++)
    if neighbors_list[i+1].age > neighbors_list[i].age
    then add_Q = neighbors_list[i+1].add;
         age_Q = neighbors_list[i+1].age;
    end if
end for
neighbor_Q = (add_Q, age_Q)

shuffle_list[1] = neighbor_Q
for (i = 2, i ≤ SL, i++)
    shuffle_list[i] = Random_neighbor;
end for
```

In the third step we replace Q's entry in *shuffle_list* with a new entry of age 0 and with *P*'s address.

After that we send the subset *shuffle_list* to peer *Q* using the function *Send()*. This function has three parameters: the address of the sending peer, a list of neighbors of the sending peer, and the address of the receiving peer.

Then peer *P* receives from *Q* a subset of no more than *SL* elements of its own neighbors. It is the list *shuffle_list_Q*. We use the function *Receive()* which has got two parameters: the address of the sending peer (peer *Q*) and a list of neighbors of peer *Q*.

When peer *P* receives *shuffle_list_Q*, we check whether there are peers in this list that have P's address or peers that are already contained in *neighbors_list*. Those peers are discarded. After that we start updating P's *neighbors_list* to include all remaining peers in *shuffle_list_Q* (except peers that have been already discarded). We start by using empty slots, then we replace entries among the ones sent to peer *Q* (peers contained in *shuffle_list*).

```
for (i = 1, i ≤ SL, i++)
    neighbor_i = shuffle_list_Q[i]
     if neighbor ≠ empty
     and (neighbor_i.add ≠ Myadd)
     and (forall (j = 1, j ≤ nb_neighbors, j++)
        neighbor_i.add ≠ neighbors_list[j].add
     end forall)
    then
        for (j = 1, j ≤ nb_neighbors, j++)
           neighbor_j = neighbors_list[j]
            if (neighbor_j = empty)
            then neighbor_j = neighbor_i;
               done = true;
            end if
        end for
        if (done = false)
        then
            for (j = 1, j ≤ nb_neighbors, j++)
               neighbor_j = neighbors_list[j];
               if (Is_in_shuffle_list(neighbor_j) = true)
               then neighbor_j = neighbor_i;
                   Break;
               end if
            end for
        end if
    end if
end for
```

We define the function *Is_in_shuffle_list()* to check weather a peer is included in *shuffle_list* and returns a boolean according to this result.

- Case 2 : The receiving peer

For the receiving peer Q algorithm, we have two executed steps. Once the receiving peer Q receives the shuffle list from the initiating peer P, peer Q executes the first step consisting of preparing a random list of at most *SL* neighbors, then peer Q sends this list to peer P.

In the second step, peer Q updates the list of its neighbors by taking into account the list sent by peer P. This step is executed the same as for the initiating peer P.

V. Formal Validation of the Shuffling Algorithm Formalization

In this chapter, we propose to validate our shuffling algorithm formalization using formal validation.

We propose to implement the algorithm in our formal model presented in Section IV.

The formal validation is limited by the state space explosion problem, that is why we focus on validating complicated behaviors such as the shuffling algorithm in our case. We start by eliminating all the non relevant details regarding the behavior that we want to validate.

In our case we propose a minimized model with only governance units.

This formal model is composed by three principal modules: the *main* module, the *types* module, and *governance_unit* module.

In the *types* module, we define new types related to the shuffling algorithm that we need in our program. In this module, we define five types:

- *INDEX_GU*: is an integer ranging from 1 to 4. This type expresses the index of the governance unit which is unique for each governance unit. For this type, we can use the predefined functions of equality, inequality, superiority and strict superiority.

```
type INDEX_GU is
  range 1 .. 4 of Nat
  with "==","<>","<=", "<"
end type
```

- *TAB_IS_NEIGHBOR*: a table of Boolean defining for each node whether other nodes are neighbors or not. The length of the table is the total number of nodes in the system. In this case, the total number of nodes in the system is four.

```
type TAB_IS_NEIGHBOR is
  array [1 .. 4 (*nb of nodes*)] of BOOL
end type
```

- *NEIGHBOR_T*: a constructor used to define the type *neighbor*. A neighbor can be either an *empty_neighbor* or a pair of two fields. The first field is the index of the node (*INDEX_GU*). The second field is an integer denoting the age of the node (*Age*). For this type we can use four

predefined functions: two functions to access the *NEIGHBOR_T* fields which are *get* to read and *set* to modify, and two other functions to compare two neighbors which are *equality* and *inequality*.

```
type NEIGHBOR_T is
  empty_neighbor,
  NEIGHBOR(ind:INDEX_GU,age:NAT)
  with "get", "set", "==", "<>"
end type
```

- *NEIGHBORS_LIST_T*: a constructor used to define the list of neighbors. It contains elements of type *NEIGHBOR_T*. The length of this list is the same for all nodes (*nb_neighbors*), which is a variable defined and initialized in the main module). In this case, the number of neighbors is three.

```
type NEIGHBORS_LIST_T is
  array [1 .. 3 (*nb_neighbors*)]
    of NEIGHBOR_T
end type
```

- *SHUFFLE_LIST_T*: a constructor used to define the shuffle list. It contains elements of type *NEIGHBOR_T*. The length of this list is *SL*, which is a variable defined and initialized in the main module. In this case, the shuffle list length is two.

```
type SHUFFLE_LIST_T is
  array [1 .. 2 (*SL*)] of NEIGHBOR_T
end type
```

In the *main* module, we define the process *MAIN* which is parametrized by two gates: *Membership_initiating* gate and *Shuffling_transfer* gate. In the body of the process, we start by defining two variables: *nb_neighbors* denoting the number of neighbors that each node can not exceed, and *SL* which denotes the length of the shuffle list of each node. Those two variables are initialized in the main process and remain the same during the execution of our program. In this case we suppose that the number of neighbors is three and the shuffle list length is two.

After that, we define a parallel composition between four processes instantiations. The global synchronization set is composed of two gates: *Membership_initiating* and *Shuffling_transfer*. The four processes in the parallel composition represent the governance units composing our distributed system and communicate on gates *Membership_initiating* and *Shuffling_transfer*. Those processes have two common parameters *nb_neighbors* and *SL* for shuffle length, and one specific parameter *index_GU* which is specific for each process. In this case we have four communicating governance units, each one is instantiated with a specific *Index_gu* from 1 to 4. The governance units are also instantiated with the value of *nb_neighbors* and the shuffling list length (*SL*). In the following, we note *GATES2* the set of gates composed by *Membership_initiating* and *Shuffling_transfer* gates.

```
process MAIN [GATES2]
is
var nb_neighbors : NAT,
   SL : NAT
in
  nb_neighbors := 3;
  SL := 2;
  par GATES2 in
    governance_unit [GATES2]
          (nb_neighbors,SL,index_GU(1))
  ||
    governance_unit [GATES2]
          (nb_neighbors,SL,index_GU(2))
  ||
    governance_unit [GATES2]
          (nb_neighbors,SL,index_GU(3))
  ||
    governance_unit [GATES2]
          (nb_neighbors,SL,index_GU(4))
  end par
end var
end process
```

In the *governance_unit* module, we declare two processes: *governance_unit* process and *membership_management* process.

The *governance_unit* process is an initialization process. It is executed one time when the program is launched. In this process, we define the initial connections between neighbors. In our case, the resulting connections from the initialization process are presented in Figure 4. A non deterministic choice structure is used to allow each governance unit to define its neighbors list.

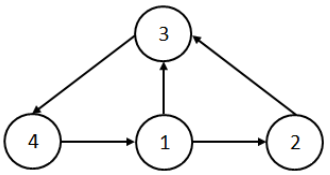


Figure 4: Nodes initialization

Then we call the *membership_management* process.

The *membership_management* process is parametrized by two gates for communication: *Membership_initiating* and *Shuffling_transfer*.

It has five formal parameters : the former *nb_neighbors* and *SL*, *neighbors_list* which define the list of neighbors of the actual governance unit, *GU_is_neighbor* a table of Boolean to check whether governance units of the system are neighbors of the actual governance unit, *my_id_GU* to

give the index of the actual governance unit.

Then we start by defining variables: index_NL an integer to browse the neighbors list, index_SL an integer to browse the shuffle list, *index_q* the index of the oldest neighbor, *index_1* and *index_2* used to identify other governance units, *q_age* the age of the oldest neighbor, *shuffle_list* the list that is sent during a shuffling operation, *q_shuffle_list* the received list during a shuffling operation. The membership management code is explained step by step in the following.

```
process membership_management[GATES2]
  (nb_neighbors:NAT,SL:NAT,
   neighbors_list : NEIGHBORS_LIST_T,
   node_is_neighbor: TAB_BOOL,
   my_id_GU:INDEX_GU)
is
 var index_NL, index_SL : NAT,
    index_q, index_1,index_2:INDEX_GU,
    q_age : NAT,
    shuffle_list,q_shuffle_list:SHUFFLE_LIST_T
  in
<<membership management code here>>
end process
```

In a shuffling operation, a governance unit is whether an initiating node, a receiving node, or a non concerned node. This behavior is modeled as a non deterministic choice with three cases. In the remaining, we note P the initiating node and Q the receiving node.

- Case 1: initiating node

We start by a rendezvous on *Membership_initiating* gate. During this communication, we have two exchanged data: my_id_GU the index of the initiating node which is sent to other nodes, *index_1* received parameter from the receiving node. This communication takes place only if *index_1* is different from *my_id_GU*.

```
Membership_initiating(my_id_GU, ?index_1)
where (index_1 <> my_id_GU);
```

The first step of the shuffling algorithm is to increase by one the age of all neighbors in the neighbors_list of the initiating node. This behavior is expressed in the *loop_ NL*.

```
neighbors_list[index_NL]:=
neighbors_list[index_NL].{age=>
neighbors_list[index_NL].age+1};
```

In the same loop, we check on each iteration the age of the node in order to find the oldest neighbor. By this way, we reduce the complexity of the algorithm.

```
if(neighbors_list[index_NL].age>q_age) then
  index_q := neighbors_list[index_NL].ind;
  q_age := neighbors_list[index_NL].age
end if
```

After that, we start filling the shuffle list. In the first cell, we put the node P with age 0.

```
shuffle_list[1]:= NEIGHBOR(my_id_GU,0);
```

Then we fill other cells of the shuffle list randomly from nodes in neighbors list.

Once the *shuffle_list* is ready, we achieve a communication by rendezvous on the gate *Shuffling_transfer* between governance units of the system. Four parameters are exchanged in this communication: *my_id_GU* the index of the initiating node P, sent from P to Q, *shuffle_list* the shuffle list of P, sent from P to Q, *index_q* the index of the receiving node Q, sent from P to Q, *q_shuffle_list* the shuffle list of Q, sent from Q to P.

```
Shuffling_transfer(my_id_GU, shuffle_list,
            index_q, ?q_shuffle_list);
```

Once *q_shuffle_list* is received from Q, we set the age of Q to 0.

After that, we start updating *neighbors_list* using the received q_shuffle_list. We discard nodes having the index of the initiating node P and nodes already contained in P's *neighbors_list*.

```
(q_shuffle_list[index_SL].ind<>my_id_GU)
and (node_is_neighbor[NAT
    (q_shuffle_list[index_SL].ind)]==false)
```

After that we start filling *empty_neighbor* cells then we replace nodes already sent to Q in *shuffle_list*.

```
loop NL in
index_NL:=index_NL+1;
if(neighbors_list[index_NL]==empty_neighbor)
then (neighbors_list[index_NL]:=
     q_shuffle_list[index_SL];
     break NL)
elsif (neighbors_list[index_NL]==shuffle_list[2])
then (neighbors_list[index_NL]:=
     q_shuffle_list[index_SL];
     break NL)
end if;
end loop
```

- Case 2: Receiving node

If the node is a receiving one, we start by a rendezvous on *Membership_initiating* gate. During this communication, we have two exchanged data: *index_1* a received parameter from the initiating node expressing its index and *my_id_GU* the index of the receiving node which is sent to other nodes. This communication takes place only if *index_1* is different from my_id_GU.

```
Membership_initiating(?index_1, my_id_GU)
 where (index_1 <> my_id_GU);
```

After that, the receiving node prepares randomly a shuffle list that will be sent to the initiating node.

Once the *shuffle_list* is ready, we achieve a communication by rendezvous on the gate *Shuffling_transfer* between governance units of the system. Four parameters are exchanged in this communication: *any INDEX_GU* the index of the initiating node P, sent from P to Q, *q_shuffle_list* the shuffle list of P, sent from P to Q, *my_id_GU* the index of the receiving node Q, sent from Q to P, *shuffle_list* the shuffle list of Q, sent from Q to P.

```
Shuffling_transfer(?any INDEX_GU,
 ?q_shuffle_list,my_id_GU,shuffle_list);
```

The receiving node updates the list of its neighbors by taking into account the received list from the initiating node. This update is executed the same as for the initiating node.

- Case 3: Non concerned node

This node is a passive node. All remaining nodes (except the initiating and receiving nodes) are non concerned nodes. Such a node communicates on both *Membership_initiating* and *Shuffling_transfer* gates. The non concerned node gets a knowledge about the handled communication and data transfer without taking part of the transfer.?

```
Membership_initiating(?index_1, ?index_2)
where (index_1 <> my_id_GU)
    and (index_2 <> my_id_GU) ;

Shuffling_transfer(?index_1,
  ?any SHUFFLE_LIST_T, ?index_2,
  ?any SHUFFLE_LIST_T)
where (index_1<>my_id_GU)
    and (index_2<>my_id_GU)
```

VI. Related Works

In addition to the unstructured peer to peer architecture that we have used in this paper for modeling the membership between the governance units, there exists a *structured peer-to-peer architecture* where the overall graph of nodes is constructed using a deterministic procedure such as the *distributed hash table* (DHT) [15]. As an example of a system implemented according to a structured peer-to-peer architecture and using the DHT procedure, we find the *Chord system* [16]. According to the survey proposed by Lua et al [7] comparing the structured and unstructured architectures, the most convenient one for highly dynamic systems exposed to failures is the unstructured architecture.

Several protocols are proposed at the MAC level of the sensing end of IoT systems such as TDMA (collision free), CSMA (low traffic efficiency) and FDMA (collision free but requires additional circuitry in nodes) schemes available to the user [17]. In our case we are interested on the system-level protocols either for the communication between sensors, actuators, and computing unit or for the communication between different subsystems.

The proposed formalization and validation approach is used by the industry to validate specific networking protocols to deal with complex behaviors that cannot be proven by classical test and simulation approaches [18], [19].

VII. Conclusion

In this paper, we present a formalization of the ReDy distributed systems architecture as well as the enhanced shuffling algorithm for the membership management of nodes of this architecture. Then we focus on the formal validation of this membership management part. This work proposes and validates formally a solution for IoT applications in large scale, highly hazardous, and dynamic environments.

The formal validation is limited to small configurations of the systems (i.e., number of nodes, number of governance/action/detection units) because of the state space explosion problem. In the other hand, it provides an exhaustive validation for those configurations. As a result, we can eliminate a big number of specification errors and, by construction, failures in an early stage.

This solution has to be implemented in larger configurations to be tested in real life conditions and to validate other aspects mainly the implementation aspects. The implementation of larger configurations in real life conditions have to deal with hardware aspects specific to each IoT application.

References

[1] K. Hafdi and A. Kriouile, “Designing redy distributed systems,” in Autonomic Computing (ICAC), 2015 IEEE International Conference on. IEEE, 2015, pp. 331–336.

[2] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding, “Reference manual of the lnt to lotos translator (version 6.1),” Inria/Vasy and Inria/Convecs, vol. 131, 2014.

[3] R. Mateescu and D. Thivolle, “A model checking language for con-current value-passing systems,” inInternational Symposium on Formal Methods. Springer, 2008, pp. 148–164.

[4] A. P. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi, “Architecture and protocols for the internet of things: A case study,” in Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on. IEEE, 2010, pp. 678–683.

[5] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," Future generation computer systems, vol. 29, no. 7, pp. 1645–1660, 2013.
[6] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes," in Proc. of STTT'13. Springer, 2013.
[7] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim et al., "A survey and comparison of peer-to-peer overlay network schemes."IEEE Communications Surveys and Tutorials, vol. 7, no. 1-4, pp. 72–93, 2005.
[8] A. Tanenbaum and M. Van Steen, Distributed systems. Pearson Prentice Hall, 2007.
[9] S. Voulgaris, D. Gavidia, and M. Van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays,"Journal of Network and Systems Management, vol. 13, no. 2, pp. 197–217, 2005.
[10] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "The peer sampling service: Experimental evaluation of unstructured gossip-based implementations," inProceedings of the 5th ACM/IFIP/USENIX international conference on Middleware. Springer-Verlag New York, Inc., 2004, pp. 79–98.
[11] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling,"ACM Transactions on Computer Systems (TOCS), vol. 25, no. 3, p. 8, 2007.
[12] R. Guerraoui, R. Oliveira, and A. Schiper, "Stubborn communication channels," Tech. Rep., 1998.
[13] C. Cachin, R. Guerraoui, and L. Rodrigues,Introduction to reliable and secure distributed programming. Springer, 2011.
[14] T. A. Henzinger, "The theory of hybrid automata," in Verification of Digital and Hybrid Systems. Springer, 2000, pp. 265–292.
[15] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in p2p systems,"Communications of the ACM, vol. 46, no. 2, pp. 43–48, 2003.
[16] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," Networking, IEEE/ACM Transac-tions on, vol. 11, no. 1, pp. 17–32, 2003.
[17] I. Demirkol, C. Ersoy, and F. Alagoz, "Mac protocols for wireless sensor networks: a survey,"IEEE Communications Magazine, vol. 44, no. 4, pp. 115–121, 2006.
[18] Kriouile and W. Serwe, "Formal analysis of the ace specification for cache coherent systems-on-chip," inInternational Workshop on Formal Methods for Industrial Critical Systems. Springer, 2013, pp. 108–122.
[19] A. Kriouile and W. Serwe, "Using a formal model to improve verification of a cache-coherent system-on-chip," inInternational Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2015, pp. 708–722.

Kaoutar Hafdi is a PhD Student at National Higher School for Computer Science and Systems Analysis (ENSIAS) , Mohammed V University and IMS team at ADMIR Lab in Rabat IT Center. She received a Master of Research in parallel, distributed, and embedded systems from Grenoble INP Institute in Grenoble, France obtained in 2013. She received also a Master of Engineering in software engineering from ENSIAS, Mohammed V University, Rabat, Morocco obtained in 2012.

Abdelaziz Kriouile is a Full Professor in the Software Engineering Department at National Higher School for Computer Science and Systems Analysis (ENSIAS), Mohammed V University, Rabat, Morocco. He is a member of IMS Team at ADMIR Lab in Rabat IT Center. He was the head of SIME Lab. (Mobile and Embedded Information Systems Laboratory) from 2010 to 2017. He received his PhD in Computer Science from the Nancy University, France in 1990. He received a State doctorate from the University of Mohammed V, Rabat, Morocco in 1995. His research activities focus on information systems, cloud computing, and context-aware service-oriented computing. He leads numerous projects related to the application of these domains.

Abderahman Kriouile is the founder and CEO of the start up 'Farasha Systems', which deals with the optimization of the output of solar energy power plants. He is also an associate professor at SUPMTI Engineering School in Rabat. Dr. Kriouile achieved a PhD in 2015 in "Formal Methods" applied to the verification of embedded systems on micro-electronic chips. His PhD was carried out in the framework of a collaboration between "STMicroelectronics" and the french national research organization "Inria". Prior to that, Dr. Kriouile gained engineering experience in the Avionics and Simulation department of Airbus in Toulouse, France. Dr. Kriouile holds a MSc. in Embedded System and Software Engineering from Telecom Nancy, France obtained in 2011.