CPDE: A methodology for the transparent distribution of centralized smart grid programs

Thi Thanh Quynh Nguyen, Christophe Bobineau, Vincent Debusschere, Quang Huy Giap, Nouredine Hadjsaid

CPDE: A methodology for the transparent distribution of centralized smart grid programs

Thi Thanh Quynh Nguyen, Christophe Bobineau, Vincent Debusschere, *Senior Member, IEEE,* Quang Huy Giap, and Nouredine Hadjsaid, *Senior Member, IEEE*

Abstract—Control and management in smart grids are facing many challenges such as scalability, heterogeneity and technology innovation. This requires a transformation from the traditional centralised paradigm into a distributed one. In this paper, a new distributed programming methodology, called Centralised Programming and Distributed Execution (CPDE), is proposed. CPDE relies on (i) the abstraction of the whole system as a distributed database, (ii) the use of the *Smartlog* declarative and reactive rule based language for expressing data manipulation, and (iii) the automatic *Smartlog* rule distribution according to data distribution. It thus provides a simple and straightforward mean for distributed programming. A centralised algorithm of fair over-voltage regulation of PV systems is used as a typical smart grids study case to validate the methodology and to compare it with centralized implementations. The experiments are implemented in a real-time simulation platform with a network of Raspberry Pis. In addition to showing its correctnes and ease of use, the performance of the CPDE implementation is studied, as well as its sensitivity to the increasing number of computing units and the data distribution. Results are promising and show the clear benefits of this methodology compared to more classical implementations.

Index Terms—Distributed programming, CPDE methodology, distributed database, declarative language, *Smartlog*, smart grid.

1 Introduction

The integration of renewable energy and storage devices into the traditional power systems requires immediate response to manage intermittent resources, through bidirectional energy flow controls and decentralized management of the renewable energy generation. The power grid needs a technology transformation to meet these challenges [1].

The cooperation of the electricity grid and the information network promises to bring to life an innovative grid [2]. Along with the development of smart devices such as smart meters, micro-computers and the use of available communication infrastructures (e.g., Wi-Fi [3], 3G, 4G, 5G, power-line communication [4], etc.), the electrical system could be better controlled and managed, up to real-time [5].

Nevertheless, the information network integrated into the electrical grid poses many problems, especially regarding the scalability of the infrastructure. The larger the system is, the more powerful the computing servers must be. That leads to a problem of costs, both from the investment and operation point of view. Moreover, the huge amount of data driven to servers in short time can cause a problem of velocity in data processing, particularly in heterogeneous and complex systems like smart grids. Thus, local processing should be encouraged to limit overload and bottleneck phenomenons in central servers [6], [7].

- *Dr. Nguyen, Dr. Debusschere and Prof. Hadjsaid are with Univ. Grenoble Alpes, CNRS, Grenoble INP*, G2Elab, 38000 Grenoble, France. E-mail: thi-thanh-quynh.nguyen@g2elab.grenoble-inp.fr.*
- *Dr. Bobineau is with Univ. Grenoble Alpes, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France.*
- *Dr. Giap is with the Department of Electrical Engineering, University of Science and Technology, University of Danang.*

**Institute of Engineering Univ. Grenoble Alpes.*

1.1 Implementing algorithms for smart grids

Numerous centralized algorithms have been developed to ensure parameter control and management in power grids (e.g. voltage or frequency control). However, with the development and scale-up of smart grids, these algorithms may no longer be sufficient. They show indeed many shortcomings, such as very high computation and communication costs, single points of failure, etc. [1].

Besides, along with the current development of the infrastructures, computing devices can be found largely scattered over the power grid. This huge available computing resources can fully participate in the management of the power system itself. Distributed algorithms in this context seem to be more efficient than the centralized ones, because they deal with the imminent problems of control and management while ensuring the replacement of the conventional centralized controller [8]. However, there are still some drawbacks in distributed programming that restrains their deployment in reality.

For instance, using generic distributed algorithms to achieve agreement on a common parameter value in power grid control, such as consensus algorithms is possible (e.g Metropolis [9], Finite-time Average Consensus algorithms [10] and Maximum Degree Weight [11]). But this kind of solution presents a convergence rate that depends on the grid's configuration and that is mostly not compatible with near real-time needs of smart grid control.

Specific distributed algorithms for power grid management, such as Dual Decomposition [12], Optimality Condition Decomposition (OCD) [13], Analytical Target Cascading (ATC) [14], Auxilliary Problem Principle (APP) [15], Alternating Direction of Multipliers Method (ADMM) algorithm with Proximal Message Passing (PMM) [16] have been pro-

https://www.doi.org/10.1109/TPDS.2020.3019759 © 2021 IEEE Published by the IEEE Computer Society

posed for the optimal power flow problem (OPF). But they need up to several thousand iterations to converge for nodal OPF in a small grid [17]. Furthermore, the convergence rate is shown to be linear with the number of power grid nodes [18]. The ALADIN algorithm [19] was proposed to reduce the number of iterations in comparison of the ADMM algorithm, but this improvement is not proven in the case where the number of nodes increases significantly. In addition to a slow convergence rate, neglecting the communication delay leads to non-reactive control and management of power systems, incompatible with a their evolution towards smart grids. Stability limits of the systems could be violated and the quality of the energy would be difficult to ensure.

To summarize, programming with such distributed algorithm is still challenging. A distributed program is a set of cooperating local programs, and all data exchange and process synchronization have to be taken into account. These challenges increase with the number of participating computing devices. A solution for reducing these challenges is to use higher abstraction levels, for instance, abstracting the whole system as a distributed database and using declarative data manipulation languages. Control and management of power grids consist in reacting to changes in the environment (i.e. change in the sensor data) to adjusts parameters (i.e. acting on actuators). Declarative and reactive languages are thus needed. Distributed datalog-based languages such as Netlog [20], Nlog [21] or *Smartlog* [22], the later one being dedicated to smart grid use, can be advantageously exploited. But, even with the use of such languages, data exchange among participating devices is still to be managed.

In this paper, the architecture of distributed systems such as multi-agent systems [23] as well as the protocols used for data exchange are not presented as they do not fit with the focus of the method presented in this paper. Instead, we focus on solving issues related to data exchanges management and processes synchronization.

1.2 Objectives of the work

We aim at simplifying distributed programming in smart grids by totally hiding data exchanges among computing devices. We propose to use the *Smartlog* rule-based data manipulation language to program centralized control and management algorithms and then to automatically transform these centralized programs into sets of cooperating programs (i.e. to distribute the rules of the programs) in order to transparently achieve a distributed execution. This distributed programming methodology is named CPDE, for Centralized Programming and Distributed Execution. The CPDE method has been first mentioned in [24], focusing on the basic principle of the method.

In this paper, we first recall the general structure and behavior of *Smartlog* programs, and the corresponding system architecture (section 2). We then detail our automatic rule distribution algorithm according to the distribution of data (Section 3). Section 4 and Section 5 present an experimental performance evaluation of distributed programs obtained using CPDE compared to both Smartlog and Java centralized implementations. This evaluation will show the advantages of CPDE (simplicity, correctness and efficiency).

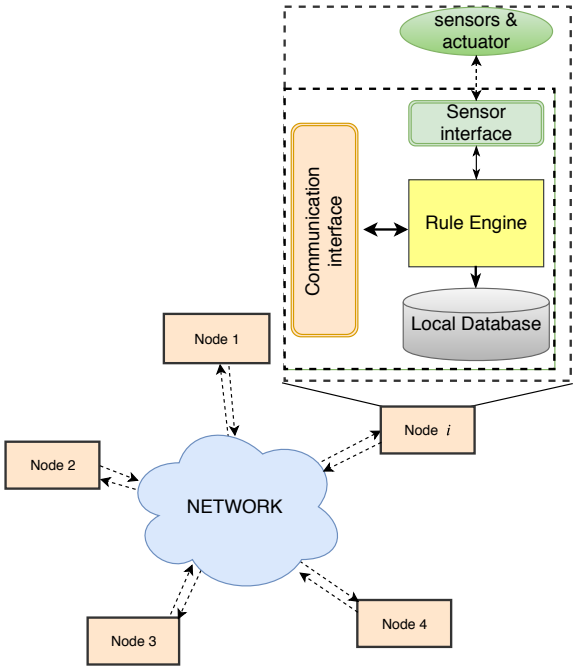


Fig. 1: Structure of each node in the grid.

The experiments are realized with a real-life voltage control algorithm (AAPC [25]) over real-time power grid simulation and actual computing devices, in a power hardware in the loop experimental validation. Section 7 concludes the paper on the main advantages of the CPDE methodology.

2 Smartlog Language and System

The major details of the architecture of the nodes in the considered smart grid as well as the *Smartlog* language can be found in [26]. The primary knowledge is reintroduced in this section before addressing the CPDE methodology.

Note that the work was developed based on a context in which internet of things components are more and more participating in the power system's infrastructure. Reusing electrical legacy system was considered to decrease investment, as cited in [26]. Besides, Smartlog is a high level abstract language aiming to simplify the distributed programming. Our first prototype is implemented based on an executable machine language (PostgreSQL and Java in our test-case) and as the rule distribution is performed statically (off-line), rule triggering and execution can be implemented in any language using code generation, on very basic hardware platforms. In this case, a small and cheap extension to any legacy component is possible, even if most probably not considered, due to the actual number of such components in traditional distribution grids

2.1 Architecture of a node in the network

Each computing node is an elementary cell in the network that has the ability to store data, communicate and perform computations. The architecture of each node is presented in Fig. 1. The architecture of each node contains three parts:

- The *local database* stores the node's information: Sensing data, parameters, intermediate results and so on.

- The *rule engine* decides which rules have to be triggered in the *Smartlog* program and executes them.
- The *sensor interface* is in charge of the interaction with the sensors and actuators.
- The *communication interface* manages data exchanges with others nodes in the network.

2.2 Structure of the Smartlog program

The structure of a *Smartlog* program is proposed in Listing 1.

```
01 Program(NameOfProgram){
02 Data_types{//define the data types
03 A(Value1: int key, Value2: int, Value3: float).
04 H(Value1: int key, Value2: int, Value3: float).
05 }
06 Initial_data{//set up initial data
07 H(1, 0, 0).
08 }
09 Module(A){//rules
10 H(m, k, t) :- A(m, n, t),n >5, k:= t+n;
11 }
12 ...
13 }
```

Listing 1: The structure of a *Smartlog* program.

- The `Data_types` (lines 2 to 5) defines the scheme of data stored in the local database.
- The `Initial_data` (lines 6 to 8) declares the data initialization. These insertion will not activate any rule in the `Module` blocks.

Smartlog programs contain sets of rules in the form `Head:-Body` (line 10). When a rule is triggered, the validation of the body $\mathcal{B}$ leads to the generation of the head $\mathcal{H}$. Data generated by the head part of rules can be locally stored (e.g. intermediate results or parameter activating an actuator) or sent to another node to pursue distributed computing. The body part of a rule is composed of one or more terms that can be atoms a_j (i.e. test of existence of data items in the local database), conditions c_j or assignment of variables s_j. All terms have to be satisfied in order to produce the head. Rules are triggered by a modification or insertion of data items corresponding to the first term of the body. To optimize the execution, rules that must be triggered by the same `data_type` are grouped in modules. Rules in the same module can be exclusive or executed in sequence.

2.3 Network communication and cooperation

Each node in the network has its own IP address, which helps identify the location of the node. Nodes communicate using the TCP/IP protocol in which data packets are sent and received as in sequential order. In *Smartlog*, the change in a specific `data_type` activates the corresponding calculations. Transferring data from one node to another can trigger calculations elsewhere as well. That is notably why *Smartlog* can fully support distributed programming.

As an illustration, let us consider two modules, A and B, in two different sites, i and j. After executing the rule of the module A in the site i, B is sent to the site j. Herein, it triggers the next rule validation in the module B in the site j.

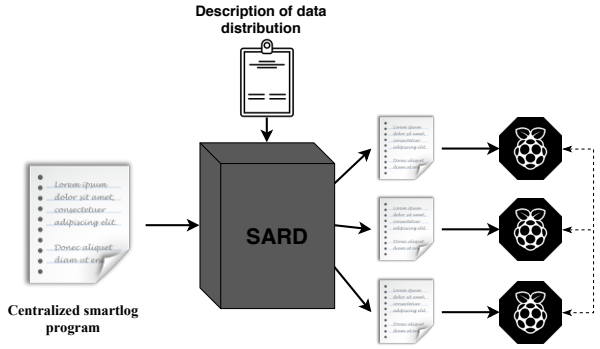


Fig. 2: Principle of *Smartlog* rule distribution with CPDE.

```
Module(A){
^B(i, t) : A(i, a, t),
    C(i, @j); }
```

Listing 2: Module A, site i.

```
Module(B){
E(i, c) : B(i, t), D(a,
    c), t>c; }
```

Listing 3: Module B, site j.

3 Semi-automatic rule distribution (SARD)

We propose a new programming methodology, called *Centralized Programming and Distributed Execution* (CPDE), to simplify the distributed programming in *SmartLog* by hiding synchronization and data exchange aspects.

The CPDE methodology approaches the distributed *Smartlog* programming based on the principle presented in Fig. 2. A centralized program (that can be executed on a single device) is transparently rewritten into multiples cooperating *Smartlog* programs according to a given data distribution schema. These programs will be instantiated on the distributed devices and will perform equivalent computations. The rewriting process exploits a *Semi-Automatic Rule Distribution* processor (SARD) developed to that purpose.

3.1 Principle

Based on the description of the data distribution, the SARD processor analyzes the set of rules of a centralized *Smartlog* program. It decomposes them (if necessary) into subrules, corresponding to each location, while ensuring that the distributed programs present the same behavior as the centralized one. Some critical hypotheses should first be satisfied in the input program.

- The centralized *Smartlog* program runs without error;
- The communication network is reliable.

In addition, the volume of communications and transferred data must be minimized, limiting the bandwidth usage. In the following sections, we focus on two main parts: the description of the data distribution and the rule transformation process.

3.2 Description of the data distribution

A big part of the smart grid data is naturally scattered over the network. For example, sensor data are stored where they are produced and parameter data are stored where they are exploited (actuators). The rest of the (intermediate) data can

be stored anywhere. The design of the distributed database allows each dataset scheme to be fragmented horizontally (groups of data items), vertically (groups of data attributes from all data items), or in a hybrid manner. Each part of the dataset is called a fragment.

For the horizontal fragmentation, each fragment holds all the attributes of the original `data_type`, but it only contains a part of the original data items, specified by a predicate [27]. The original dataset can be recomposed by the union of all horizontal fragments.

$$DT = \cup_i DT_i \tag{1}$$

For the vertical fragmentation, each fragment holds several attributes, mandatory including the key attributes, extracted from all data items of the dataset. The original dataset can be recomposed by joining all vertical fragments.

$$DT = \bowtie_i DT_i \tag{2}$$

The hybrid fragmentation of a dataset is a horizontal fragmentation followed by a vertical fragmentation of all horizontal fragments. The original dataset can be recomposed with (3).

$$DT = \cup_i (\bowtie_j DT_{ij}) \tag{3}$$

In the power grid, measurement and control data are normally fragmented horizontally according to the identifier of the corresponding sensor/actuator and located near them. Meanwhile, the intermediate data allocation depends on the optimal data placement process. The design of the optimal data placement is not the focus of this paper, assuming that it has been correctly specified beforehand. Each data fragment is allocated to one device in the network.

To describe the data distribution in the network, we propose to combine fragmentation aspects (horizontal and vertical) and allocation aspects. The proposed syntax is conceived as an extension of the syntax of the *Smartlog* `data_type` block (D). Each `data_type` definition contains the original `data_type` ($D_i | D_i \subset D$) with its attributes (A_i) and its data fragments($F_i | D_i = \{A_i, F_i\}$).

In each fragment definition ($F_{ij} | F_{ij} \subset F_i$), we define three parts: (i) the filtering condition in the key attributes C_{ij} (for the horizontal fragmentation); (ii) the set of attributes A_{ij} (for the vertical fragmentation) and (iii) the location in which the data items are stored $L_{ij}, F_{ij} = \{C_{ij}, A_{ij}, L_{ij}\}$. The location part is mandatory, while the condition and attributes parts are optional. As an illustration, the `data_type A`, horizontally fragmented is presented in Listing 4, lines 2 to 13.

```
01 Data_types{
02   A(Value1: int key, Value2: int, Value3: int){
03     Fragment(A1){
04       Condition: a>2;
05       -- Attributes:
06       Location: 'S1';
07     }
08     Fragment(A2){
09       Condition: a<=2;
10       -- Attributes:
11       Location: 'S2';
12     }
13   }
14 }
15 IPmap{
16   S1: 192.168.1.100;
17   S2: 192.168.1.101;
18 }
```

Listing 4: A sample *data_type* block.

Since all devices of the network should be known by name, locations are thus expressed as names. The correspondence between device names and IP addresses are defined in the IPmap block of the data distribution description, as shown in listing 4, lines 15 to 18.

the SARD processor follows three main steps: (i) rule rewriting, to take datasets fragmentation into account; (ii) rule distribution, to manage data exchanges and processes synchronization and (iii) the distributed program generation.

3.3 Rules transformations

Rule transformations are the first two steps (rewriting and distribution) of the SARD processor, presented below.

3.3.1 Rule rewriting

The original rule is developed based on the description of the data distribution. The idea is to first replace atoms in the body part of rules by their reconstruction from fragments, and then duplicate rules for each head fragment as described by Algorithm 1.

Algorithm 1 Rule rewriting.

procedure RULEREWRITING(rule $\mathcal{R}$)
  for $\mathcal{B}_i$ in $\mathcal{B}$ **do**
    if $\mathcal{B}_i$ instantOf Atom **then**
      get D_i from B_i
      $D_{ik} \leftarrow D_i, a_{ik} \leftarrow \{D_{ik}, B_i\}$
      if D_i is not horizontal fragmentation **then**
        Group of juncture D_{ik} of D_i
        Replace B_i by set of juncture a_{ik} ($SA | a_{ik} \subset SA$)
      else Replace B_i by $a_{ik} (a_{ik} \in SA)$
      if $i == 0$ **then**
        for a_{ik} in SA **do**
          Create a new Rule $\mathcal{R}_r$, add a_{ik} to $\mathcal{R}_r$
          add $\mathcal{R}_r$ to list of Rule ($LR | \mathcal{R}_r \subset LR$)
      else
        for a_{ik} in SA **do**
          for $\mathcal{R}_r$ in LR **do**
            if $C_{a_{ik}} \cap C_{\mathcal{R}_r} \neq \emptyset$ **then**
              add a_{ik} into $\mathcal{R}_r$
    else
      for $\mathcal{R}_r$ in LR **do**
        if $C_{\mathcal{B}_i} \cap C_{\mathcal{R}_r} \neq \emptyset$ **then**
          add $\mathcal{B}_i$ into $\mathcal{R}_r$
  for F_i in $\mathcal{H}$ **do**
    $D_i \leftarrow F_i, a_i \leftarrow \{D_i, \mathcal{H}\}$
    if $\exists c_i \in F_i$ **then**
      add c_i into $\mathcal{R}_r$
    Add a_i to $\mathcal{R}_r$

The body rewriting depends on the type of fragmentation (horizontal or vertical). Atoms vertically fragmented are simply replaced by the conjunction of fragments (join). For example, the rule R, presented in (4), where the `data_type` B is fragmented as $B(\underline{x}, y, z) = B_1(\underline{x}, y) \bowtie B_2(\underline{x}, z)$, is rewritten as rule R_1 in (5).

$$R : H(m,k,t) \text{ :- } A(m,n,t), B(m,\_,k). \tag{4}$$

$$\rightsquigarrow R_1 : H(m,k,t) \text{ :- } A(m,n,t), B_1(m,\_), B_2(m,k); \tag{5}$$

The horizontal fragmentation implies to duplicate the rule for each fragment. For example, if B is horizontally fragmented into B_3 holding B items where $x > 10$ and B_4 holding B items where $x \leq 10$, B can be recomposed with $B(\underline{x}, y, z) = B_3(\underline{x}, y, z) \cup B_4(\underline{x}, y, z)$. The rule R from (4) will be then rewritten as rules R_{21} and R_{22} in (6).

$$\begin{cases} R_{21} : H(m,k,t) \text{:-} A(m,n,t), B_3(m,\_,k); \\ R_{22} : H(m,k,t) \text{:-} A(m,n,t), B_4(m,\_,k); \end{cases} \tag{6}$$

After rewriting the body part, the fragmentation of the head has to be considered. The resulting rules of the previous step will be duplicated for each fragment of the head. Horizontal fragments imply adding the fragment condition to the body part. If H in rule R is vertically fragmented into $H_1(\underline{x}, y)$ and $H_2(\underline{x}, z)$ then the rule R_1 is finally rewritten into rules R_{11} and R_{12} as presented in (7).

$$\begin{cases} R_{11} : H(m,k) \text{ :- } A(m,n,t), B_1(m,\_), B_2(m,k); \\ R_{12} : H(m,t) \text{ :- } A(m,n,t), B_1(m,\_), B_2(m,k); \end{cases} \tag{7}$$

If H is horizontally fragmented into H_3 and H_4 with conditions $x > 12$ and $x \leq 12$ respectively, then the same rule R_1 is rewritten into rules R_{13} and R_{14}, (8).

$$\begin{cases} R_{13} : H(m,k,t) \text{ :- } A(m,n,t), \\ \qquad\qquad B_1(m,\_), B_2(m,k), m > 12; \\ R_{14} : H(m,k,t) \text{ :- } A(m,n,t), \\ \qquad\qquad B_1(m,\_), B_2(m,k), m <= 12; \end{cases} \tag{8}$$

Aggregate functions (`COUNT`, `MIN`, `MAX`, `SUM` and `AVG`) are supposed to be computed over the whole original dataset. There is no problem in case of a vertical fragmentation as the aggregation concerns only one attribute, but the case of a horizontal fragmentation is more tricky. The solution consists in computing (sub-)aggregates for each horizontal fragment and store the results in intermediate datasets which then contribute to compute the final aggregated values.

For example, the rule R, defined in (9), where B is horizontally fragmented into B_1 and B_2, is first rewritten into rules R_1 and R_2, as in (10), according to the dataset fragmentation.

$$R : H(m,k,t) \text{ :- } A(m,n,t), B(m,\_,l), k := sum(l); \tag{9}$$

$$\rightsquigarrow \begin{cases} R_{11} : H(m,k,t) \text{:-} A(m,n,t), \\ \qquad\qquad B_1(m,\_,l), k := sum(l); \\ R_{12} : H(m,k,t) \text{:-} A(m,n,t), \\ \qquad\qquad B_2(m,\_,l), k := sum(l); \end{cases} \tag{10}$$

Then, each resulting rules are rewritten to compute the aggregates (`SUM`) over the fragments. The final rules are shown in (11) where R_{21} and R_{22} partially compute the sum over the fragments B_1 and B_2 respectively, and the rules R_{23} recompose the final value of the sum while deleting now useless intermediate results. As the first atom in the rule's body triggers the rule, R_{23} is itself decomposed in two rules to ensure the final `SUM` computation whatever the order of intermediate results production is.

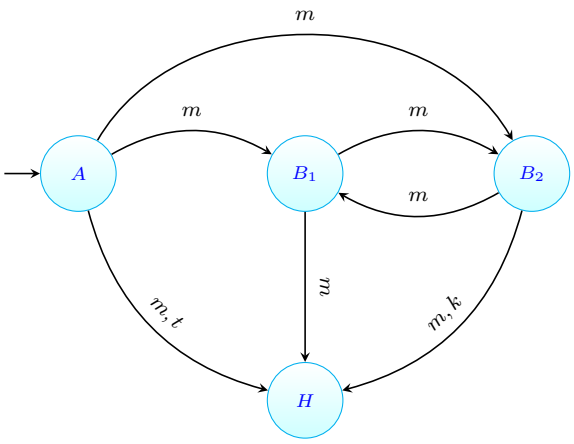


Fig. 3: Intial cyclic graph of Rule R_1, expressed in (5).

$$\begin{cases} R_{21} : Tmp_1(m,k_1,t) \text{:-} A(m,n,t), \\ \qquad\qquad B_1(m,\_,l), k_1 := sum(l); \\ R_{22} : Tmp_2(m,k_2,t) \text{:-} A(m,n,t), \\ \qquad\qquad B_2(m,\_,l), k_2 := sum(l); \\ \begin{cases} R_{23_1} : H(m,k,t) \text{:-} !Tmp_1(m,k_1,t), \\ \qquad\qquad !Tmp_2(m,k_2,t), k := k_1 + k_2; \\ R_{23_2} : H(m,k,t) \text{:-} !Tmp_2(m,k_2,t), \\ \qquad\qquad !Tmp_1(m,k_1,t), k := k_1 + k_2; \end{cases} \end{cases} \tag{11}$$

3.3.2 *Rules distribution*

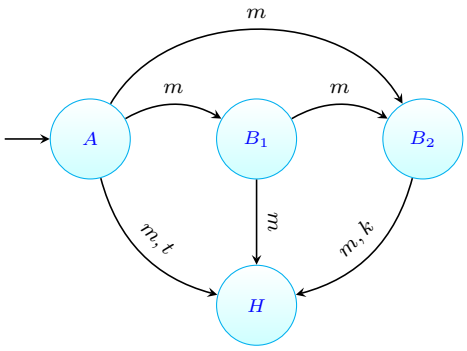
After determining which rules can be incurred from the original rule $\mathcal{R}$, the next step is to analyze the rewritten rules R_r according to the data allocation and to decide the data transferred among locations. Each rule is considered one after the other in the set of R_r.

3.3.2.1 Rule in the form of a cyclic graph

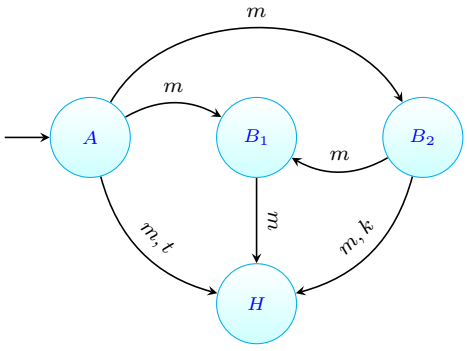
We use a directed (cyclic) graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $(\mathcal{V}, \mathcal{E})$ is a set of vertices, corresponding to terms of the rule, and labeled edges, representing links among terms, respectively. For example, in Fig. 3, the edge between A and B_1 represents the passing of the variable m from A to B_1, meaning that the term B_1 is evaluated after A.

The first term of the body part of a rule (A in Fig. 3) is the only one that possesses values assigned to all variables. Thus all paths in the graph start with this term and end with the head part of the rule. We analyze all variables of the terms in the rule in the reverse order (from $\mathcal{H}$ to $\mathfrak{a}_0$) to constitute the edges. Each term that can create new variables (except the condition one) can be a vertex of the graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.

As the body of a *Smartlog* rule is a conjunction of terms, all these terms are commutative, the commutativity of terms possibly leads to produce cycles in the graph. The whole graph $\mathcal{G}$ represents all possible execution orders of the terms

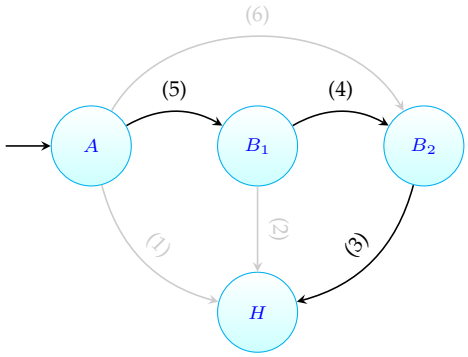


(a) First acyclic graph.

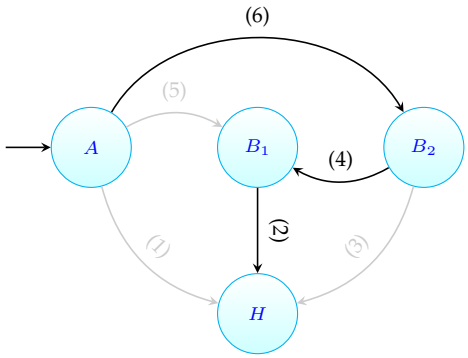


(b) Second acyclic graph.

Fig. 4: Possible acyclic graphs of Rule R_1.



(a) First covering acyclic graph.



(b) Second covering acyclic graph.

Fig. 5: Covering minimum acyclic graphs of Rule R_1.

of a rule. As an illustration, the initial directed cyclic graph constituted for Rule R_1, expressed in (5), is shown in Fig. 3.

3.3.2.2 Covering minimum acyclic directed graph

To select the best evaluation order of the rule, we have first to enumerate all covering minimum acyclic graphs representing actual possible evaluation orders (each vertice is attained only one time). This enumeration is performed in two steps: (i) eliminating edges participating in cycles to avoid infinite loops, and (ii) eliminating redundant edges to avoid unnecessary communications and computations. For example, by deleting one of the edges in cycles ($B_1 \rightarrow B_2$ or $B_2 \rightarrow B_1$ in the cyclic graph presented in Fig. 3, we generate two acyclic graphs (Fig. 4a and Fig. 4b respectively).

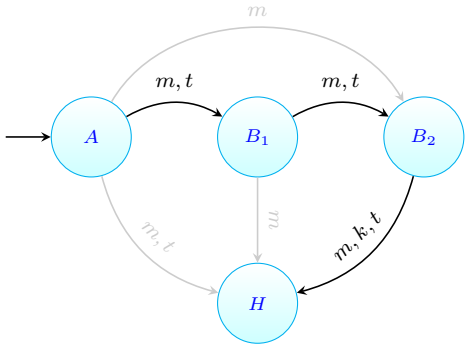
If there are two different paths between two vertices (A and B_2 in Fig. 4a), one direct and one passing through other vertice(s) (B_1 in the same figure), the direct one is then redundant and should be eliminated. The resulting covering minimum acyclic graphs are shown in Fig. 5.

3.3.2.3 Variable rerouting

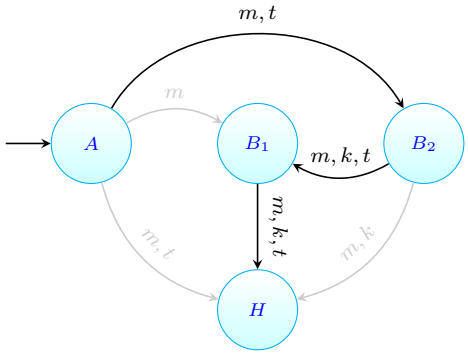
By eliminating redundant edges, the transportation of some variables may be forgotten and has to be compensated. For example, in Fig. 5a the transfer of variable t from vertex A to vertex H has been lost. This variable has to be rerouted using the retained path between A and H as shown in Fig. 6.

3.3.2.4 Data location and transfer

The location of each vertex (fragment `data_type`) of the minimum acyclic graph has to be considered. Location of atom vertices is known from the description of the data distribution. These locations allow constituting sub-graphs where all `data_types` are co-localized and thus identifying needed communications between locations. Non-atom



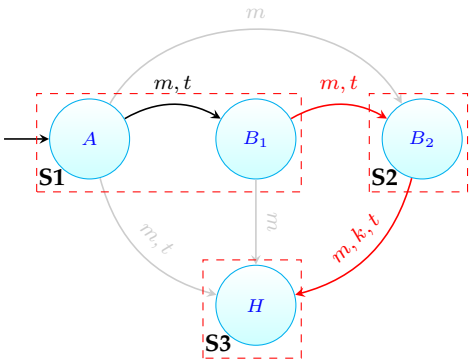
(a) Rerouting data items of Fig 4a.



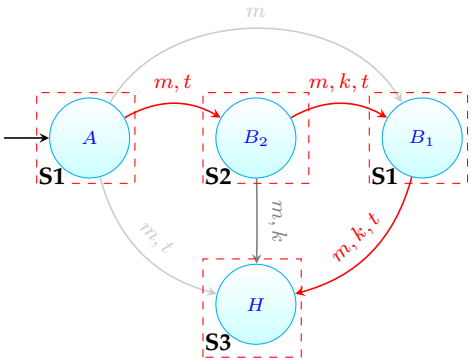
(b) Rerouting data items of Fig 4b.

Fig. 6: Rerouting variables of acyclic graphs of Rule R_1.

vertices (assignments or conditions), as they may filter out communications, have to be evaluated as soon as possible



(a) Data location in Fig. 4a.



(b) Data location in Fig. 4b.

Fig. 7: Data location in acyclic graphs of Rule R_1.

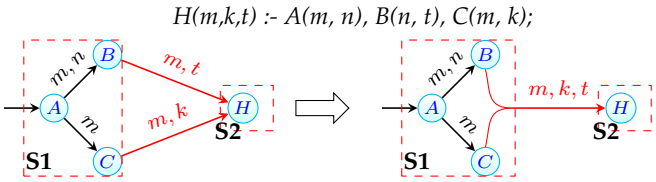


Fig. 8: Simplify connections between two locations in a rule.

on locations where all needed variables have known values.

For instance, in Rule R_1, from (5), we assume that A and B_1 are co-localized in site S_1, B_2 is in site S_2 and H in site S_3, as illustrated by red dashed rectangles in Fig. 7a and Fig. 7b (communcations are represented as red edges).

Messages between two locations are generated by local rules and include all data to be transferred. For example in Fig. 8, where two sets of data have to be transferred from S_1 to S_2 ((m,t) from B_1 to H, and (m,k) from B_2 to H respectively), only one message is transmitted, including the variables (m,t,k).

3.3.2.5 Process synchronization

It is always possible to build a minimal acyclic directed graph for a global rule comprising only one input (the first term of the body) and only one output (the head of the rule), and to partition according to the data locations. Each location evaluate the local rules that are supposed to be independent. These cooperating locations have thus to be synchronized to achieve equivalent evaluations as the original global rule. The synchronize locations ensure that messages contain sufficient information to match data included in messages coming from different locations.

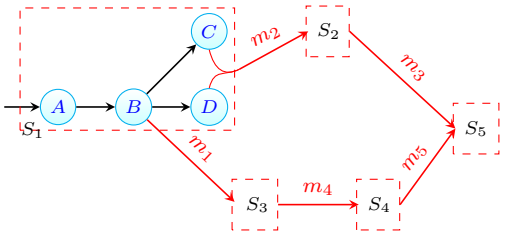


Fig. 9: An illustrated graph for synchronization process.

For example, in Fig. 9, S_5 receives two messages from S_2 and S_4 (m_2 and m_5 respectively). These two messages must be synchronized, so we need to find the location at the origin of m_2 and m_5 (here S_1). More precisely, we need to identify the term in S_1 at the origin of these two messages. m_2 is a consequence of m_1, generated from terms A, B, C and D. m_5 is a consequence of m_4 and m_3, the latest one is generated from terms A and B. The term causing the need of synchronization is thus B. So the key of B has to be propagated in all outgoing messages from location S_1 until location S_5.

The communication cost of each candidate minimum acyclic graph must now be computed in order to select the most efficient one.

3.3.2.6 Selection of the optimal graph

This step is critical to properly optimize the distributed computation. Each graph is examined and only the one minimizing the communication cost is retained. With the hardware architecture hypotheses of [22], the communication decision is subject to optimization with the objective functions defined as follows:

1) Minimize the number of communications;
2) Minimize the number of transferred data-items.

In fact, to determine precisely the influence of each variable on the objective function ($\mathcal{F}$), all the characteristics of the considered network such as bandwidth, error rate, communication costs, etc. must be taken into account. In this paper, we do not focus on the cost model and we use a simple one, for illustration, as shown in (12).

$$\min \mathcal{F} = x * y \tag{12}$$

Where x represents the number of communications and y the total number of transferred data-items.

The optimal evaluation graph is the one offering the minimum cost. The optimal minimum acyclic graph allows determining the optimal rule execution order in which the behavior of a rule is still transformed in the same way as in the distributed system. The transferred data items must be started from the root of the graph (the vertex holds the first term of the body part) and end at the destination which holds the head part of the rule.

The rule distribution complies with the optimal graph. The direction of communication between two vertices respects the direction of the edge, and the transferred data items are the label of the edge.

As an illustration, let us consider the objective function of the two labeled graphs presented in Fig. 7a and Fig. 7b, $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively. For Rule R_1, expressed in (5), the optimal graph is shown in Fig. 7a with $\mathcal{F}_1$. Indeed, in Fig. 7b,

there are three communications and eight transferred data items, but in Fig. 7a only two communications and five transferred data items, thus $\mathcal{F}_1 = 2 * 5 < \mathcal{F}_2 = 3 * 8$.

3.4 Generating the programs

The last step of the SARD processor is to construct the distributed *Smartlog* programs for each location. Firstly, the data communication is taken into account to create new communication rules and temporary `data_type` triggering the next evaluation in a different location.

3.4.1 Performing communication

Data are packed in the form of a temporary `data_type` and sent to another location by message passing. The name of the temporary `data_type`, which is generated automatically, allows triggering the next actions in another location. The new temporary `data_types` are added to the `data_types` block in both locations (the sender and the receiver). The `Data_type IPMap` is declared and added at the end of the body part to indicate the mapping IP of the target address. For example, data sent with the optimal graph of Fig. 7a is expressed by the rules declared in Listing 5 and Listing 6.

```
Module(A1){
^Tmp1(x,t) :- A1(x,y,t), B1(x,_), IPMap('S2',@ip); }
```

Listing 5: Sub-rules generated in the site $S1$.

```
Module(Tmp1){
^B1(x,k,t) :- Tmp1(x,t), B2(x,k), IPMap('S3',@ip); }
```

Listing 6: Sub-rules generated in the site $S2$.

After transforming the original rule into sub-rules in which all the term of the body part have the same location, each rule is assigned corresponding to its location.

3.4.2 Generating distributed Smartlog programs

A distributed *Smartlog* program is constructed in each location with the set of `data_types`, `initial_data`, and the assigned rules. In each program, rules are grouped by module, a set of `data_types` and `initial_data` are rewritten for the `data_type` block and `initial_data` block, respectively.

A *Smartlog* program comprises possibly many modules, and each one may contain many rules. The process of rule transformation into sub-rules can cause rule duplication in each location. In that case, the simplification of the program is performed.

4 Experimental Evaluation

In this section, we apply the SARD processor over a reference *Smartlog* program implementing an over-voltage protection for a photovoltaic power generation. We then compare the behavior of both implementations in order to highlight the correctness and performance of the resulting distributed programs. We then study the influence of the number of nodes and the data distribution design on the performance.

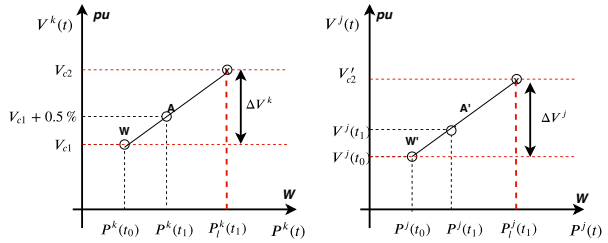


Fig. 10: Adaptive Active Power Capping method [25].

Note that the presented method has been successfully applied to other reference smart grids regulation processes like the frequency containment reserve for instance [22], [26]. Those results are not presented in this paper because the comparison of the implemented regulations is not relevant for the comprehension of the method and also not to extend the length of the paper.

4.1 AAPC: over-voltage control for smart grid

The rise of renewable energy, especially Photovoltaic (PV) sources, in traditional distribution grids causes risks of power unbalance as well as a decrease of the quality of the energy. One of the most notable issues, when this energy is injected to the grid, is over-voltage. There are many solutions approaching this problem, and one of the most effective methods with a centralized control is the Adaptive Active Power Capping (AAPC) [25]. The principle of the method is presented in Fig. 10.

The purpose of the AAPC method is to determine the limit of the produced active power of each PV source so that the nodes' voltage in the whole grid do not exceed the bound of over-voltage and ensure the fair sharing of active power for all participating sources. The AAPC method assumes that a small change in voltage is approximated with the power variation in a small period of time span (t_1-t_2), with V_{C_1} and V_{C_2} the two concerned bounds of the voltage. V_{c2} is the bound of over-voltage, in which the source is disconnected from the grid, $[V_{C_1} - V_{C_2}] = [1.042 - 1.058]$ (pu). The over-voltage occurs in one or several nodes of the grid, called critical nodes (Cri). In the AAPC method, $V_{c_2}^{'}$ corresponds to the voltage of other nodes j (not critical ones), and is estimated based on the ratio $PROV$.

$$PROV = \frac{V_{c_2} - V^k(t_1)}{V^k(t_1) - V_{c_1}} = \frac{V_{c_2}^{'} - V^j(t_1)}{V^j(t_1) - V^j(t_0)} \tag{13}$$

In principle, AAPC uses the linear regressive method for each photovoltaic source to predict the upper power limitation (Plim) expressed in (14).

$$P_l^j(t_1) = P^j(t_0) + (V_{c2}^{'} - V^j(t_o))/\xi^j \tag{14}$$

With ξ the linear coefficient (Slope) calculated by the ratio of voltage variation (Vvar) and power variation (Pvar).

$$\xi^j = \frac{V^j(t_1) - V^j(t_0)}{P^j(t_1) - P^j(t_0)} \tag{15}$$

The photovoltaic power production within its threshold prevents over-voltages in the grid. P_{ref}^j, the generated power of the j^{th} PV node in the next step, is its limit power

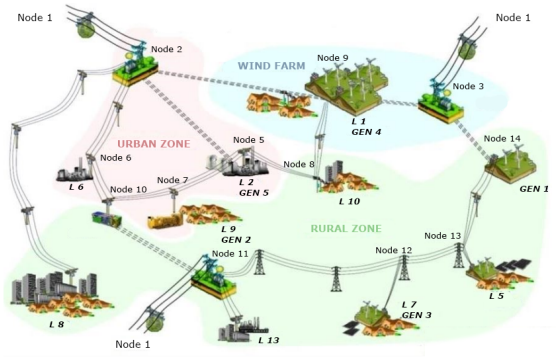


Fig. 11: PREDIS Test network configuration [28].

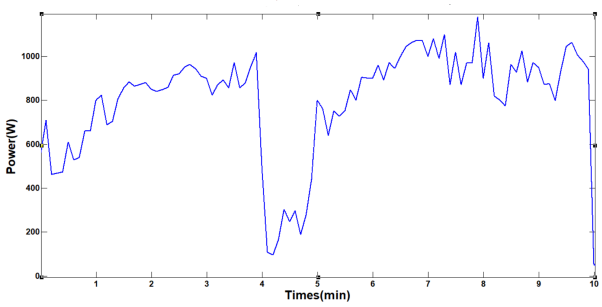


Fig. 12: The PV generation curve for 10 minutes.

(Plim) or maximal produced power (Pmax) predicted by a *maximum power point tracking* calculation, expressed in (16).

$$P^j_{ref}(t_1) = min(P^j_l(t_1), P^j_m(t_2)) \tag{16}$$

Each photovoltaic node has the same responsibility to participate in the stable operation of the grid. Thus, the curtailment of each PV panel must be identical. The power production of each node at next time t_2, called $P^j_a(t_2)$, is expressed in (17).

$$P^j_a(t_2) = \eta * P^j_m(t_2) \tag{17}$$

With η the power curtailment (Pcur), defined as:

$$\eta = \sum_{j=1}^{j=n} P^j_{ref}(t_1) / \sum_{j=1}^{j=n} P^j_m(t_2) \tag{18}$$

4.2 Experimental setup

The distribution grid PREDIS [28] is used as a test object. This grid has 14 nodes, with five distributed sources, three asynchronous machines and multiple static loads. The grid configuration is shown in Fig. 11. This grid is simulated in MATLAB/SIMULINK and executed in OPAL/RT for real-time simulation. Raspberry Pis play the role of local computing units installed at the nodes with distributed sources (PV), and are named following the node order.

The centralized *Smartlog* program has `data_types` declared as follows (identifiers are underlined):

- `Measure(ID, Timestamp, VOLT, POW, Pmax)`: Instantaneous measure at node `ID` of the voltage, active power and maximal produced power;
- `Warning(ID, Timestamp)`: Keeps the timestamp at which the node `ID` attains the warning status;
- `WarningMeasure(WarningID, ID, VOLT, POW)`: Measure the data of all nodes when the grid enters warning status;
- `Alert(ID, Timestamp, PROV)`: Stores the timestamp at which the critical node attains the alert status, and the corresponding value of the $PROV$ ratio;
- `Slope(Cri-ID, ID, Slope, Vvar)`: Slope coefficient and V_{var} for the linear regression;
- `Plimit(Cri-ID, ID, Pref)`: Computed injected power limit for each node;
- `Curtail(ID, Pcur)`: Computed power curtailment percentage of each Photovoltaic node;
- `Actuator(ID, Yield)`: Output curtailment values for each Photovoltaic node.

The dataset is generated automatically during the real-time simulation. The measured data at each distributed PV source is sent to each corresponding Raspberry Pi .

Regarding the scenario of voltage control, we consider the change of injected power at each node. To verify the methodology, the injected power has to change with time and sometimes cause over-voltage in the grid. We carry out the simulation for 10 minutes. During this time we keep the load constant and change the PV irradiation. In reality, the change of solar power depends mainly on climate and cloud conditions. The curve presented in Fig. 12 is used to evaluate the method as well as to assess the response time for this case study.

5 Exerimentation and Analysis

According to the objectives of the experimentation, we focus on evaluating the behavior of the algorithm implementations rather than the algorithm itself.

5.1 Correctness of the distributed execution

5.1.1 Description

To demonstrate the correctness of our method, we implement the centralized algorithm from [25] in Java (which represents the reference implementation in our work) and in *Smartlog*. The results of these two implementations are compared with the results of the distributed implementation using the CPDE methodology and the SARD processor.

Regarding the data distribution, there are eight `data_types` in the centralized *Smartlog* program. They are assumed to be fragmented horizontally and distributed over five Raspberry Pis, except three intermediate `data_type` (`Warning`, `Alert`, and `Curtail`), only located in two raspberry Pis at node N^{o}7 and N^{o}9, respectively.

In the Predis grid, the 7^{th} node, corresponding to Raspberry Pi N^{o}7, is designed as the critical node during the simulation (this node has the highest voltage in the grid). Therefore, we will discuss the result mainly for this node in the following sections.

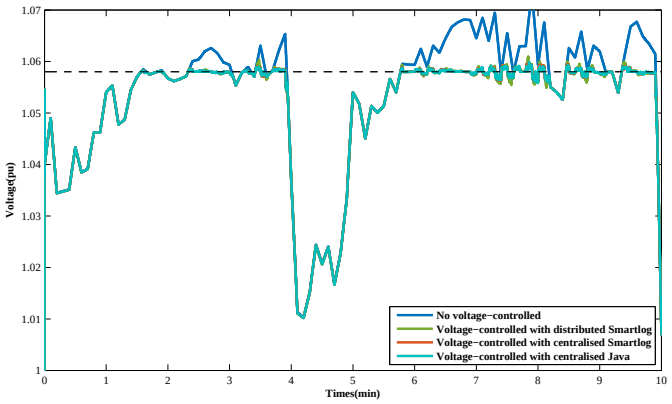


Fig. 13: 7^{th} node's voltages for all implementations.

TABLE 1: Response times for the major implementations.

	T_{comp}	T_{comm}
Centralized	$\sum_{j\in N}\sum_{i\in R_j} t_i$	0
Distributed	$\max(\sum_{i\in R_{No7}} t_i, \sum_{j\in R_{No9}} t_j)$	Nt_c

5.1.2 Results and discussion

Fig. 13 shows the voltage response at the 7^{th} node for both implementations (centralized and distributed). The voltage in all implementations is controlled and does not exceed the upper bound (1.058 pu). The results are identical, which confirms the correctness of the SARD algorithm.

At the second minute of the simulation, the active power of the PV increases linearly and causes an over-voltage in the grid. The AAPC algorithm is activated to restrain the percentage of power production. In continuous time, the curtailment is almost inversely proportional to the increase in active power. This means that the curtailment decreases linearly when the power production increases and exceeds the upper bound of power. In practice, the response time is defined as the interval between two consecutive updates of the power curtailment when the grid operates in over-voltage. We thus estimate the response time of each deployment in the experiment based on this definition.

We call t_i the local processing time of each rule, t_c the average delay time for each communication and N the number of computing units in the network. The response time of a node is estimated with (19).

$$T_{res} = T_{comp} + T_{comm} \tag{19}$$

With R_j the number of fired rules at the j^{th} node. The analysis of the response time is shown in Table 1.

The average response time of the three implementation (centralized *Smartlog* program, centralized java program and distributed *Smartlog* program) are shown in Fig. 14.

The response time of the distributed program is smaller than the response time of the centralized program: $\max(\sum_{i\in R_{No7}} t_i, \sum_{j\in R_{No9}} t_j) + Nt_c < \sum_{j\in N}\sum_{i\in R_j} t_i$. Indeed, in the real-time platform, the time delay t_c is not significant in comparison to the processing time. Besides, the data sharing over the network speeds up the query time. Moreover, computing units operate in parallel and the sharing of computing load will apparently reduce the total computing time.

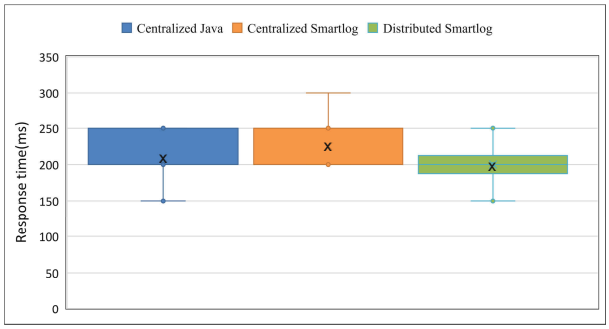


Fig. 14: Average response time for all implementations.

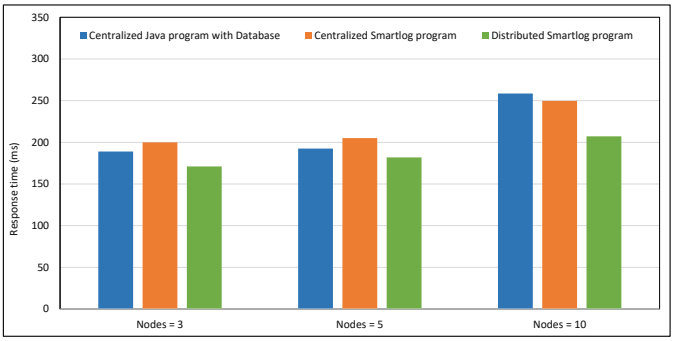


Fig. 15: Impact of the number of nodes on the response times.

Note that, the way the distributed programming is designed, in a reactive manner, will ensure the correctness of the method even when the network presents a bad communication delays.

5.2 Impact of the number of computing units

5.2.1 Description

In order to evaluate the influence of the number of computing units, we scale up the number of PV generator nodes as well as the number of computing units in the network. Since the experimental infrastructure is limited, the number of Raspberry Pis used is 3, 5 and 10 (the maximum number of Raspberry Pi on our real-time platform).

5.2.2 Results and discussion

The results of the experiment are proposed in Fig. 15. The average response time is considered in this case.

In all situations (3, 5 and 10 computing nodes), the response time of the distributed *Smartlog* execution is smaller than the two centralized ones. The gain in response time increases with the number of computing nodes. Also, as the number of nodes increases, the difference between the response times of the three implementations increases as well. The performance of the distributed *Smartlog* execution can be justified using the following arguments.

Calling $\bar{t_i}$ the average time needed to execute a rule, and $\bar{R_j}$ the average number of rules processed by a node. When N is big enough, so that $Nt_c \gg$

$(\max(\sum_{i\in R_{N^o7}} t_i, \sum_{j\in R_{N^o9}} t_j)$, the limit of the response time of the distributed programming is:

$$\lim_{N}\left(\max\left(\sum_{i\in R_{N^o7}} t_i; \sum_{j\in R_{N^o9}} t_j\right) + Nt_c\right) = Nt_c \quad (20)$$

Meanwhile, the limit of the response time of the centralized program is:

$$\lim_{N}\left(\sum_{j\in N}\sum_{i\in R_j} t_i\right) = N\bar{R}_j\bar{t}_i \quad (21)$$

With the same experiment architecture, $t_c < \bar{R}_j\bar{t}_i$, when the number of nodes in the network increases ($N \to \infty$), the response time of the centralized program increases much faster than the one of the distributed program.

5.3 Impact of the distributed data configuration

5.3.1 Description

The sensing data are naturally scattered, usually close to the sensors in smart grids. To show the impact of the distributed data configuration on the performance, we focus mainly on the impact of the fragmentation and allocation of the intermediate `data_types` that are used to perform the aggregate computation over the network. If all the intermediate `data_types` are stored on the same node, this one is in charge of the computation of aggregate functions. Otherwise, if we distribute these intermediate `data_types`, the aggregate computation is carried out in parallel by multiple nodes of the network. The data distribution schemes of the intermediate `data_types` are expressed below:

- Case 1: The intermediate `data_types` are stored in only one of the five Raspberry Pis (7^{th} node in our case).
- Case 2: Two intermediate `data_types` (`Warning` and `Alert`) are stored in the Raspberry Pi N^{o}7 and `Curtail` in the N^{o}9.
- Case 3: We add a new Raspberry Pi at the 6^{th} node of the grid to store the intermediate `data_types`.
- Case 4: Similar to Case 2, but `Alert` is fragmented vertically and stored in the Raspberry Pis N^{o}7 and N^{o}9.

The first activation of the voltage regulation is considered to analyze the impact of data configurations on the performances. We use again (19) to assess the response time of each data distribution configuration. Note that Cases 1, 2 and 4 present five computing units, while Case 3 six.

5.3.2 Results and discussion

The four presented configurations are tested on the same application. The two main purposes of this test-case are to show the impact of using multiple data distribution configurations on the performances of the implementation as well as the adaptation of the methodology to multiple data distribution designs.

The response of the four cases are presented in Fig. 16. For all cases, the objective of voltage regulation is reached: the voltage is controlled around the upper limit (1.058 pu)

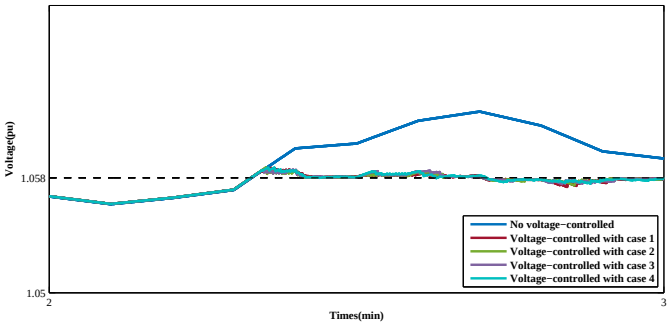


Fig. 16: 7^{th} node's voltages for all data distributions.

TABLE 2: Response times for all data distributions.

Case	T_{comp}	T_{comm}
1	$\sum_{i\in R_{N^o7}} t_i$	$(2N-2)t_c$
2	$\max(\sum_{i\in R_{N^o7}} t_i, \sum_{j\in R_{N^o9}} t_j)$	Nt_c
3	$\max(\sum_{i\in R_j(j<>N^o6)} t_i, \sum_{i\in R_{N^o6}} t_i)$	$2Nt_c$
4	$\max(\sum_{i\in R_{N^o7}} t_i, \sum_{j\in R_{N^o9}} t_j)$	Nt_c

when the PV production is over the limit, causing an over-voltage in the grid. The implementation can manage various designs of data distribution, even with `data_types` that are fragmented horizontally or vertically: The centralized algorithm is exactly transformed in the distributed one.

Table 2 presents the comparison of the four cases and the related response times expressions.

The communication times of Cases 2 and 4 are equal, but the number of fired rules in a node is not the same in each case. Case 4 presents a longer computing time than case 2 because it must deal with data integrity by regrouping vertical fragments. For Case 3, the aggregate function is dispersed on another node bus, which needs more communications than in Case 1. In fact, the comparison of their performance depends on the deviation of the average time delay of communications and the required time to execute a rule.

This analysis reinforces the results shown in Fig. 17. The response time in the experiment is collected based on the interval of consecutive changes of the power curtailment values, once the power production increases linearly and causes over-voltage. We proceed with a statistic study of the four cases for the first moments of the simulation, when the voltage regulation is activated during the experimentation. The results are placed next to the value of the centralized implementations for comparison.

As seen, Case 2 is the best one. That can be explained as follow. In Case 1, similar to the centralized implementation, all the intermediate data are stored on the same node, common computations are performed in only one node, which is why the response time is larger than in the other cases. In Cases 2, 3, and 4, the intermediate data are allocated in multiple nodes. For Case 3, another Raspberry Pi (N^{o}6) is in charge of the aggregated computation, but it takes more time than in Case 2 because it requires more communication to request data from/to the various locations. The vertical data fragmentation in Case 4 is also good for the average response time, which is smaller than Cases 1, 3 and equal

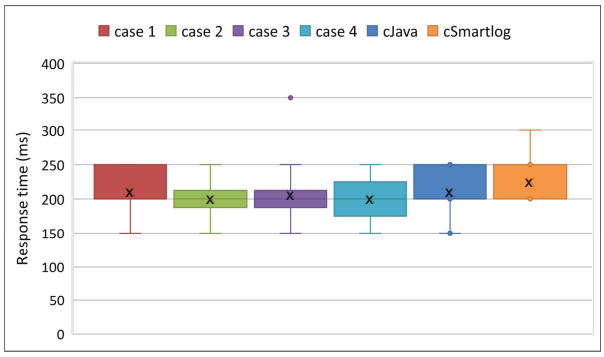


Fig. 17: Responses times of all considered study cases.

to Case 2 in the distributed implementation. However, the vertical fragmentation in Case 4 seems to not fit with this application as it present a deviation compared to Case 2.

The configuration of the data distribution has a strong influence on the performances of the execution. Thus, the problem of the optimal data distribution must be addressed in the future to reach the best performances with the upcoming *Smartlog* implementations.

6 Discussion

Applying the proposed method in distributed management for smart grids shows promising results. However, the approach of distributed programming by using the *Smartlog* language is still in early stages. A few drawbacks can be highlighted in this paper, in a form of prospective researches:

- The distributed execution can not be automatically configured once a node is disconnect from the network as the rules are currently statically distributed;
- The SARD processor is based on a semi-automatic rule distribution principle. The data fragments should be predefined;
- The data location is supposed to be unique in the network, but in fact, the data are usually replicated, so that the decision of communication can be optimized if the data replication is taken into account;
- The communication cost model is simplified in this paper. A more accurate model should be used for each studied network.

7 Conclusion

In this paper, a new distributed programming methodology is proposed, called Centralized Programming and Distributed Execution (CPDE). This methodology is developed to facilitate distributed programming by first implementing centralized algorithms with *Smartlog* (*Smartlog* being a declarative and reactive rule-based data manipulation language dedicated to smart grid management needs). This centralized program is then translated into a distributed one using a Semi-Automatic Rule Distribution (SARD) processor according to a given data distribution description. This methodology totally hides the difficulties of distributed programming (i.e. data transfer and process synchronization are now transparent).

An experimentation has been conducted to validate our approach and highlight its performances and scalability compared to traditional centralized implementations (in Smartlog and Java). We also studied the influence of the number of participating nodes and data distributions alternatives on the performances. All these experiments have been realized in real-time simulation environment with hardware in the loop (computing devices) using a real-world application of over-voltage control with Photovoltaic power sources.

Future work will include: (i) the optimization of the data distribution, regarding its strong influence on performance; (ii) a more precice communication cost model; and (iii) the improvement of the prototype, for example using dedicated software instead of PostgreSQL as data storage and rule execution engine.

Acknowledgments

The authors would like to thank the French Embassy in Vietnam and the Foundation Grenoble INP for funding the PhD leading to the presented results.

References

[1] A. Hirsch, Y. Parag, and J. Guerrero, "Microgrids: A review of technologies, key drivers, and outstanding issues," *Renewable and sustainable Energy reviews, Elsevier*, vol. 90, pp. 402–411, 2018.

[2] H.-P. Schwefel, Y.-J. Zhang, C. Wietfeld, and H. Mohsenian-Rad, "Emerging technologies initiative smart grid communications: Information technology for smart utility grids," in *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, 2018.

[3] J. Zhang, A. Hasandka, J. Wei, S. Alam, T. Elgindy, A. Florita, and B.-M. Hodge, "Hybrid communication architectures for distributed smart grid applications," *Energies*, vol. 11, no. 4, p. 871, 2018.

[4] F. Passerini and A. M. Tonello, "Smart grid monitoring using power line modems: Anomaly detection and localization," *IEEE Transactions on Smart Grid*, 2019.

[5] K. Sayed and H. Gabbar, "Chapter 18 - scada and smart energy grid control automation," in *Smart Energy Grid Engineering*, H. A. Gabbar, Ed. Academic Press, 2017, pp. 481–514. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128053430000188

[6] M. Ma, L. Fan, and Z. Miao, "Consensus ADMM and Proximal ADMM for economic dispatch and AC OPF with SOCP relaxation," in *2016 North American Power Symposium (NAPS)*, Sep. 2016.

[7] W. Liu, W. Gu, Y. Xu, Y. Wang, and K. Zhang, "General distributed secondary control for multi-microgrids with both pq-controlled and droop-controlled distributed generators," *IET Generation, Transmission & Distribution*, vol. 11, no. 3, pp. 707–718, 2017.

[8] P. Lin, C. Jin, J. Xiao, X. Li, D. Shi, Y. Tang, and P. Wang, "A distributed control architecture for global system economic operation in autonomous hybrid ac/dc microgrids," *IEEE Transactions on Smart Grid*, vol. 10, no. 3, pp. 2603–2617, Oct. 2018.

[9] Y. Hanqing, Y. Liangzhen, L. Qi, C. Weirong, and Z. Lijun, "Multiagent-based coordination consensus algorithm for state-of-charge balance of energy storage unit," *Computing in Science & Engineering*, vol. 20, no. 2, p. 64, 2018.

[10] F. Guo, C. Wen, J. Mao, J. Chen, and Y.-D. Song, "Distributed cooperative secondary control for voltage unbalance compensation in an islanded microgrid," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 5, pp. 1078–1088, 2015.

[11] Y. Xu and Z. Li, "Distributed optimal resource management based on the consensus algorithm in a microgrid," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 4, pp. 2584–2592, April 2015.

[12] A. Soares, O. De Somer, D. Ectors, F. Aben, J. Goyvaerts, M. Broekmans, F. Spiessens, D. van Goch, and K. Vanthournout, "Distributed optimization algorithm for residential flexibility activationresults from a field test," *IEEE Transactions on Power Systems*, vol. 34, no. 5, pp. 4119–4127, 2019.

[13] A. Kargarian, J. Mohammadi, J. Guo, S. Chakrabarti, M. Barati, G. Hug, S. Kar, and R. Baldick, "Toward distributed/decentralized dc optimal power flow implementation in future electric power systems," *IEEE Transactions on Smart Grid*, vol. 9, no. 4, pp. 2574–2594, 2018.

[14] A. R. Malekpour, A. Pahwa, and B. Natarajan, "Hierarchical architecture for integration of rooftop pv in smart distribution systems," *IEEE Transactions on Smart Grid*, vol. 9, no. 3, pp. 2019–2029, 2018.

[15] B. Korte and J. Vygen, "Spanning trees and arborescences," in *Combinatorial Optimization*, Springer, Ed. Springer, 2018, pp. 133 – 157.

[16] D. K. Molzahn, F. Dörfler, H. Sandberg, S. H. Low, S. Chakrabarti, R. Baldick, and J. Lavaei, "A survey of distributed optimization and control algorithms for electric power systems," *IEEE Transactions on Smart Grid*, vol. 8, no. 6, pp. 2941–2962, 2017.

[17] S. Mhanna, G. Verbič, and A. C. Chapman, "Adaptive admm for distributed ac optimal power flow," *IEEE Transactions on Power Systems*, vol. 34, no. 3, pp. 2025–2035, 2018.

[18] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine learning*, vol. 3, no. 1, pp. 1–122, 2011.

[19] A. Engelmann, T. Mühlpfordt, Y. Jiang, B. Houska, and T. Faulwasser, "Distributed ac optimal power flow using aladin," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5536–5541, 2017.

[20] S. Grumbach and F. Wang, "Netlog, a Rule-Based Language for Distributed Programming," in *Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jan. 2010, pp. 88–103. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-11503-5_9

[21] T. Li, J. Ma, Q. Pei, C. Ma, D. Wei, and C. Sun, "Privacy-preserving verification and root-cause tracing towards uav social networks," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[22] T. T. Q. Nguyen, C. Bobineau, V. Debusschere, Q. H. Giap, and N. Hadjsaid, "Using declarative programming for network data management in smart grids," in *International Database Engineering & Applications Symposium*. ACM, 2018, pp. 292–296.

[23] Y. Wang, T. L. Nguyen, M. H. Syed, Y. Xu, V. H. Nguyen, E. Guillo-Sansano, G. Burt, Q. T. Tran, and R. Caire, "A distributed control scheme of microgrids in energy internet and its multi-site implementation," *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2020.

[24] T. T. Q. Nguyen, V. Debusschere, C. Bobineau, A. Labonne, C. Boudinet, Q.-H. Giap, and N. Hadjsaid, "A new approach for the distributed deployment of centralized algorithms in smart grids," in *IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, 2019, pp. 1–5.

[25] S. Alyami, Y. Wang, C. Wang, J. Zhao, and B. Zhao, "Adaptive real power capping method for fair overvoltage regulation of distribution networks with high penetration of pv systems," *IEEE Transactions on Smart Grid*, vol. 5, no. 6, pp. 2729–2738, 2014.

[26] T. T. Q. NGuyen, V. Debusschere, C. Bobineau, Q. H. Giap, and N. Hadjsaid, "Smartlog–a declarative language for distributed programming in smart grids," *Computers & Electrical Engineering*, vol. 80, p. 106499, 2019.

[27] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[28] M. C. Alvarez-Herault, A. Labonne, S. Toure, T. Braconnier, V. Debusschere, R. Caire, and N. Hadjsaid, "An original smart-grids test bed to teach feeder automation functions in a distribution grid," *IEEE Transactions on Power Systems*, vol. 33, no. 1, pp. 373–385, 2017.

Thi-Thanh-Quynh Nguyen is currently a PhD student of Grenoble Institute of Technology. Her thesis is rolling in the collaboration of two laboratories: Grenoble Electrical Engineering Laboratory (G2Elab) and Grenoble Informatics Laboratory (LIG). Her research interests include Big data, distributed data management for smart grids and distributed control and management in microgrid.

Christophe Bobineau has obtain his PhD in computer science from the University of Versailles Saint-Quentin in 2002. He is working since then at the Grenoble Informatics Laboratory (Grenoble Institute of Technology). His topics cover transaction management, distributed query optimization and data storage from embedded systems to Big Data.

Vincent Debusschere has obtained his Ph.D. in ecodesign of electrical machines from the Ecole Normale Superieure de Cachan in 2009. He joined the Electrical Engineering Laboratory of the Grenoble Institute of Technology, in 2010 as an Associated Professor. His research interests include renewable energy integration, modeling of flexibility levers for smart grids, multi-criteria assessment, artificial intelligence and optimal design of complex systems.

Quang Huy Giap is a Lecturer-researcher at the University of Science and Technology, The University of Danang. He received his PhD in Automation-Production in 2011 from Grenoble INP, France. His research is situated in the field of Automation, with a special focus on fault detection and diagnostics technologies, renewable energies and energy management.

Nouredine Hadjsaid is a Professor at Grenoble Institute of Technology, Director of the Laboratory of Electrical Engineering of Grenoble (G2ELAB). He is also a visiting professor at Virginia Tech (USA) and NTU (Singapore). His research interests are in smart grids, which include distributed generation and power grids, information and communication technologies in power grids, and power grid safety, among others.